Accelerating Genetic Programming Using Graphics Processing Units

Tony Lewis

Department of Computer Science
and Information Systems

Birkbeck, University of London

Submitted for the degree of Doctor of Philosophy

The work presented in this thesis is the candidate's own.

Tony Lewis

Abstract

Evolution through natural selection offers the possibility of automatically generating functionally complex solutions to a wide range of problems. Methods such as Genetic Programming (GP) show the promise of this approach but tend to stagnate after relatively few generations. To research this issue, execution speed must be substantially improved. This thesis presents work to accelerate the execution of such methods.

The work uses the Graphics Processing Unit (GPU) to target the evaluation of individuals since this is the most time-consuming part of the run. Two models have been emerging for this: dynamically compiling each new generation of individuals for the GPU or using a single GPU interpreter, to which successive groups of individuals can be sent.

Using the latter model, a GPU interpreter is constructed to implement cyclic GP, an advanced form of GP that imposes several challenging implementation issues which are addressed. Accelerating the evaluation using the GPU is only part of the story. The next part of the work interleaves CPU and GPU computation to keep both chips as busy as possible with the tasks to which they are best suited and then to recruit multiple GPUs and CPU cores to further accelerate the run.

Using the former model, a compiling system is constructed and this is used to investigate two methods to overcome the primary difficulty with the approach: long compilation times. That system implements Tweaking Mutation Behaviour Learning (TMBL), a form focused on long term fitness growth and overcoming the previously mentioned stagnation issues. Further work optimises two CPU tasks highlighted by profiling: tournament selection and individual copying.

These techniques are highly effective and permit much shorter run-times. This clears the way for research into stimulating long term fitness growth and hence for tackling new, complex problems.

Acknowledgements

Getting a PhD thesis to the point of submission is a tricky business. In my case, the support of my family, friends and colleagues has made it considerably less painful than it would otherwise have been. I offer my sincere thanks to everyone that has been a part of my life in the last few years (except the bloke that burgled our flat). I will use this space to express special thanks to those who have made contributions of particular note.

Thanks to my supervisor, George Magoulas. He gave me his trust by allowing me the independence of research that I needed, yet he gave me his support by providing help and advice whenever it was needed. He repeatedly showed a remarkable ability to identify at great speed how a piece of work should be improved; his comments often pinpointed an issue that I had not previously identified but which I immediately recognised to be the nub.

Thanks to those with whom I studied an Intelligent Systems MSc several years ago. The group's spirit continues to frame my enquiry into getting computers to evolve more interesting stuff.

Thanks to all those at Birkbeck's Computer Science department and at the London Knowledge Lab. They provided a great place in which to work. Thanks in particular to Long Chen. More than any other, he brightened up my daily work at the London Knowledge Lab and I shall miss our limited attempts at team-based juggling and our discussions of culture and language.

Thanks to all those in the Orengo-group. Before beginning the PhD, I spent several years working in Christine Orengo's group and have continued to work there for one afternoon a week during my research. Working there before my PhD was an excellent preparation; working there during my PhD meant that I had much more human contact and intellectual stimulation than I would have otherwise.

Thanks in particular to Ian Sillitoe and his family, to whom I owe much gratitude for many things. By getting me involved in his martial art, Ian ensured that I never became too preoccupied with my research as I regularly had to devote some of my concentration to avoiding getting punched in the head. (Thanks also to London's Jitsuka for providing the punches). He provided me with a form of therapy by pretending to be interested when I talked to him about my work. Towards the end, he made the huge sacrifice of reading through my work in more detail than anyone else (other than me), a truly arduous task which he performed with his trademark dedication and care. Perhaps his martial art has made him insensitive to pain.

Thanks to my family. They are always of the utmost importance to me and in particular, I have my mam to thank for a preposterous amount. Perhaps I will get to see my family more frequently after I have submitted. The younger of my two older sisters has been studying for a PhD in parallel with me and she showed me how it should be done by submitting her thesis many months earlier.

Thanks to the community of researchers in the GP/EC community. I have been lucky to be working in a fantastic community of researchers working on what I believe to be the most interesting area of research at present.

Thanks to the huge body of people that have cumulatively contributed to today's astonishing hardware and software, which is all too easily taken for granted. Particular thanks to those who have dedicated time and effort to open source projects and to their user communities. The number of projects used directly and indirectly in this work must be vast but the list of software and hardware worth a particular mention includes: C++, LaTeX, nVidia (cards and CUDA), GCC (and gdb, g++ and their standard library implementation), Valgrind (profiling, memory checking, memory profiling), Boost, Subversion, Trac, Gnuplot, Inkscape, Eclipse (with plugins such as Subclipse, Texclipse, CDT, Mylyn etc) and the Ubuntu distribution of Linux (from 7.04 "Gutsy Gibbon" to 10.10 "Marverick Meerkat" via a Heron, an Ibex, a Jackalope, a Koala and a Lynx).

Thanks to my wife. I am so grateful that we share our continuing adventure together. Her love and support have kept me going through this PhD. She has shared my joy when I have been childishly pleased with my work and she has put up with me when I have been childishly bad tempered about it (sorry love). With our flat in her hands, I could always rely on it being warm.

To all these people and many more: thanks.

Addendum: Thanks also to the two examiners of my thesis, Professor Yaochu Jin and Professor Qingfu Zhang. I am well aware of the huge commitment of time and effort that is required to read a thesis in sufficient detail to examine it, to conduct the examination and to produce all the associated reports and other paperwork. For this I am very grateful. Furthermore, both examiners raised many excellent points about the work and the discussion in the viva was interesting enough that I might go so far as to say I enjoyed the examination.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Definition

Natural selection is a remarkable algorithm, simple yet powerful. If we study nature closely, we find not just astonishing intricacy but also breathtaking functional complexity that appears designed for a purpose. Other processes direct biological evolution but natural selection alone explains this functional complexity, this illusion of design.

We implement the natural selection algorithm in our computers in an attempt to harness some of its creative power and we call this field Evolutionary Computation (EC). Various forms of EC have been proposed for tackling different sorts of problems. For example, a Genetic Algorithm (GA) entails evolving a string of characters.

For other application areas it may be unclear what structure of solution is required or it may be obvious that an algorithm or behaviour must be evolved. These problems may be better suited to Genetic Programming (GP) which entails evolving programs. If we could master the use of natural selection's creative power to generate complex functional behaviours automatically, this would give us a remarkable tool with which we could work on all manner of problems.

Although GP is often highly effective during the initial stages, it typically stagnates quickly. If a population of programs evolved not for a few hundred generations but for a few hundred thousand or more, could it generate more interesting behaviours and tackle more complex problems? Evolutionary biology vividly demonstrates the astounding amount of functional complexity that a cumulative process can build using nothing but small, unguided improvements and plenty of time.

Despite the ever faster computation being delivered by processor technology, GP remains unsuitable for problems that can only be solved through a long series of improvements. This suggests that computational power is not the main limit to the useful complexity that GP can evolve. If we could understand that limit, we might be able to alleviate it and hence attack new sorts of problems. This requires a form of EC that is similar to GP but that focuses on sustaining improvements in the long-term.

Unfortunately researching this limit on long term improvement is currently made practically difficult because of the amount of computation it requires. Although current Central Processing Unit (CPU) technology is powerful enough to show us that it is no longer the main obstacle, it is not yet powerful enough to fuel the many repetitions required to identify what is.

GP is typically quite slow because evaluating the individuals tends to be much more computationally expensive than is the case for other forms of EC. For this reason, GP researchers are typically eager for any possibility of speeding up their runs. The importance of speed is even greater when researching ways to stimulate long-term improvements in GP. There are several reasons for this. First, it requires long runs. Second, it requires many runs to investigate the effects of different properties over multiple

repetitions. Third, the data-set must typically be large for it to delineate a problem that can only be solved though a long series of gradual improvements. Fourth, the programs being evolved must be large to contain the complexity that the right technique should build up over many generations. Combined, these factors upgrade run-speed acceleration from an urgent issue to a *sine qua non*.

Problem *A standard implementation of program evolution (GP) is too slow on standard hardware. In particular, it will have to be a good deal faster if it is to be used for research into stimulating long term improvement so that more complex behaviours may be evolved.*

This thesis describes research into developing a range of techniques to accelerate GP with a focus on exploiting the power of the Graphics Processing Unit (GPU). Though the motivation for the research was to open up the possibility of research into long term improvements, the applicability of these GP-accelerating techniques is very wide. To explain further, it is necessary to give a brief overview of GP and its implementation on the GPU.

1.2 Overview of Genetic Programming (GP)

Evolutionary computation takes the basic notion of natural selection and uses it to construct an algorithm. There are many varieties of EC, but the basic recipe involves updating a population of candidate solutions through multiple iterations of three steps: generate, assess and select (as illustrated with pseudo-code in Figure 1).

- The first step involves generating new individuals. The first iteration's individuals are typically created randomly (although they may be seeded using successful results from previous experiments). Later individuals derive from the individuals selected in the previous iteration. The child of a selected individual may be modified by mutations (by analogy to biological mutation) and may also draw from one or more other parents (by analogy to sexual reproduction).

- The second step involves assessing the individuals according to some criterion. This typically involves measuring the individual's success in tackling the problem in question. The result of an individual's assessment can often be reduced to a single numerical value, called its *fitness*. Typically the fitness is calculated by evaluating the sum of the individual's performances over a set of examples of the problem, or *testcases*.

- The third step involves using the fitnesses of individuals to select those individuals that will be permitted to reproduce (i.e. that will be used in the next generating step).

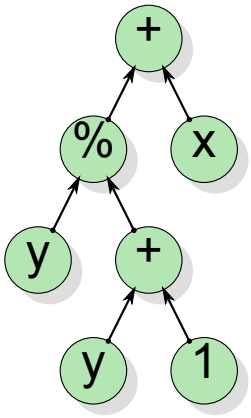
```
sub evolutionary_computation() {
  generation_ctr = 1;
  do {
    if (generation_ctr == 1) {
      population = initialise();
    }
    else {
      population = create_new_population(population, seln_indices);
      population = mutate_new_population(population);
    }
    fitnesses = assess(population, testcases);
    seln_indices = select(population, fitness);
    generation_ctr++;
  } until (termination_condition(population, generation_ctr));
  return population;
}
```

Figure 1: Pseudo-code illustrating the basic EC algorithm

The analogy between this process and biological evolution allows EC to draw from biological vocabulary quite naturally. For instance the individual's inherited structure may be referred to as its *genotype*, its behaviour resulting from this structure may be called its *phenotype* and sections of its genotype that have no (apparent) effect on its phenotype may be described as *introns*.

There are several types of EC and one of the key distinguishing features is the structure being evolved, the *representation*. The most commonly used is the GA, in which individuals are represented as strings of a finite alphabet of characters. This form is inspired by the sequences that make up biological genomes. Two less popular forms of EC are Evolutionary Programming (EP) and Evolution Strategies (ES), which typically evolve the parameters of fixed-structure programs and vectors of floating point numbers respectively. In all these cases, it is necessary for the implementer to choose some way in which individuals are to be interpreted as candidate solutions for the problem in question.

GP is the fourth major form of EC and is the focus of this thesis. This form of EC aims to evolve not just static individuals such as strings of characters or numbers but dynamic behaviours, i.e. programs. The standard representation of a GP program is a tree as depicted in Figure 2(a). More generally, most GP representations use either a node-based representation (such as the tree depicted in Figure 2(a)) or an instruction-based representation (such as the example depicted in Figure 2(b)). In either case, the individual is built out of instructions that take inputs and produce outputs. The number of inputs an instruction requires is known as its *arity*. Representation is discussed in more detail in Section 3.1.1.



(a) Tree-based representation

```
    ...
#label_23:
    r3 = r1 + r2;
    if (r3 > 0) goto #label_24;
    r8 = r9 - r4;
    ...
#label_24:
    r5 = r7 * r2;
    r5 = r1 + r2;
    goto #label_23;
    ...
```

(b) Linear representation

Figure 2: Individuals providing examples of representations commonly used in GP. Subfigure 2(a) shows an example of a tree that evaluates to $(y/(y+1))+x$. Subfigure 2(b) shows an example of a linear individual, which uses the form of a programming language.

The tree depicted in Figure 2(a) can be evaluated in a single, depth-first sweep through the nodes and its results do not vary over any further sweeps. By contrast *cyclic GP* (in which node-based structures are permitted to contain cycles) does not naturally suggest an order in which the nodes should be evaluated and different orders may give different results. This may be resolved by using an iterated flip-flop evaluation. Cyclic GP is a superset of non-cyclic GP since non-cyclic individuals (such as a tree) are valid cyclic individuals and give the same results under cyclic evaluation (with sufficient iterations). Cyclic GP is discussed in more detail in Section 3.1.3.

The selection step of an EC algorithm usually employs one of several standard selection schemes, which choose individuals with varying degrees of determinism and elitism. The selection scheme used in this work is tournament selection, which involves repeatedly choosing a random subset of the individuals and then selecting its member of highest fitness. This selection scheme is analysed and optimised in Chapter 7.

1.3 An Overview of Using the Graphics Processing Unit (GPU)

The GPU is at the heart of this work. The reasons for using this technology are outlined in Section 4.1.1. Some of the developments in the GPU's impact on EC and GP are discussed in Section 2.1 and the reason for its use in this research are outlined in Section 4.1.1.

The basic principle involved in using the GPU is to write a function to be run on the GPU, called a *kernel*. The kernel is then compiled, uploaded to the GPU and executed via calls from the CPU. As discussed in Section 2.1, this has been used in two ways for GP evaluation. In *data-parallel* approaches, each new batch of individuals is used to construct a new batch of kernels. This allows the compiler to produce highly optimised

code, which runs on the GPU at very high speed but considerable time is spent on the compilation for every new batch of code. In *population-parallel* approaches, a single interpreter kernel is constructed to process new batches of individuals as data. This interpreter only need be compiled and uploaded once, however it tends to run slower than the data-parallel equivalent.

1.4 Aims and Objectives

As described in Section 1.1, the motivation for the acceleration work described in this thesis was to enable research into stimulating long-term fitness growth in evolving behaviours. Hence the aim of this work was defined by the requirements of the long-term fitness growth research. Such research would need to investigate a range of problems and a range of forms of GP so the acceleration must allow for that. Such research would need to be performed without vast financial resources so the acceleration should focus on getting the most out of a single, reasonably-priced machine. Such research must allow comparison with standard GP so the core algorithm should be distorted as little as possible. By condensing these points, the aim of this acceleration work may be expressed as follows:

Aim *Accelerate program evolution (GP) as much as possible on a single, reasonably-priced machine with as little distortion of the algorithm as possible. This should be done flexibly to allow for a wide range of forms and problems.*

The requirement for a wide range of problems suggests that there may be differing data-set sizes, which in turn suggest that both population-parallel and data-parallel techniques should be deployed. The requirement for a wide range of forms suggests that both node-based and instruction-based GP should be implemented. Further, the node-based form should be cyclic because it is a superset of other non-cyclic, node-based forms as discussed in Section 1.2 and its structural complexity might be useful to stimulate long term improvement. For the work described in this thesis, a population-parallel approach is used to implement a cyclic node-based form and a data-parallel approach is used to implement an instruction-based form. Implementing all possible combinations of GP representations and GPU implementations is beyond the scope of this thesis. The other combinations are discussed as possible avenues for future work in Section 8.2.

Cyclic GP imposes additional constraints to make the population-parallel implementation more challenging. The challenge for the data-parallel implementation is to reduce compilation times so that the best evaluation speeds can be brought to bear on more moderately sized data-sets.

Accelerating the evaluation stage only tackles part of the problem because the other parts of the GP run also take non-trivial amounts of time. As will be seen in Section 2.1, much previous work on using the GPU to accelerate GP focuses on the evaluation only,

or else neglects to use the CPU at all during the run. Since the overall run speed is the ultimate aim, it is important to consider ways in which this can be reduced further once fast GPU evaluation is achieved. It is also important to ensure that time spent on CPU tasks is kept to a minimum through optimisation so that these tasks do not ruin the other achievements. These considerations allow the aim to be broken down into the following objectives:

Objective 1 *Use a population-parallel implementation to evaluate cyclic, node-based GP as fast as possible.*

Objective 2 *Take this further on one machine by improving the interaction between the CPU and the GPU and by using a second GPU.*

Objective 3 *Find ways to reduce compilation times of a data-parallel implementation of a form of EC with linear programs so that the best evaluation speeds can be brought to bear on more moderately sized data-sets.*

Objective 4 *Identify the worst and most avoidable bottlenecks within the CPU code and tackle them.*

1.5 The Validity, Scope and Assessment of the Research

As discussed in Section 1.4, the research in this thesis is concerned with objectives to speed up various forms of GP. This research was primarily motivated by the need to enable research into improving the long term fitness growth of GP. Once enabled, that research might still fail to deliver any interesting results, thus rendering this particular motivation fruitless. Nevertheless, the case for this work remains strong for two reasons: because the enabled research may well repay the investment and because the need for acceleration in the GP community is already strong enough to merit this being an independent research area. It is worth spending two paragraphs expanding on these two reasons.

Sometimes a potentially interesting area of research cannot be tackled until difficult preparatory work provides the necessary tools. This preparation must be conducted with the awareness that its motivation might prove hollow. Of course, one should not conclude from this that research should only be conducted if the benefits of the motivations can be confirmed in advance; such thinking would have prevented many of our most valuable scientific advances.

Further, although the particular ambition that happened to motivate this work may eventually prove fruitless, this is of little concern because the acceleration of EC, and particularly of GP, is of sufficient direct value to the research community that it has become an established research topic. This is clear from the wealth of high-quality research literature on the topic as discussed in Chapter 2. Although accelerating GP is only of real value if it can directly or indirectly improve the effectiveness of GP to solve

real problems, the research community has clearly indicated that they believe this is so.

The work in this thesis is concerned with accelerating EC, not with changing it. This conforms to the correct scientific approach of isolating a group of variables that are highly independent from the others so as to study them as clearly as possible. If acceleration had not been an established research question in its own right or if it had offered insufficient material to occupy a PhD, it would have been necessary for this research to muddle the highly independent questions of how to accelerate GP with how best to modify it to improve its results. Fortunately, the volume of literature on acceleration clearly demonstrates that this is not the case and so this thesis takes standard GP runs and finds ways to perform identical runs as fast as possible. It is very important to check code correctness by checking runs behave identically after acceleration; beyond that the run's results are of no interest and all that matters is the speed of the run.

Actually, outside of the acceleration research covered in the main chapters, the thesis does briefly touch on research conducted as part of the PhD to tackle GP's stagnation. Section 3.2 and Section 6.2 outline work that proposed Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble"), a new form of EC and a sister to GP. TMBL is described because it is the form that is accelerated in Chapter 6 and because it illustrates the sort of research that might follow on from this thesis. However this inclusion of TMBL should not detract from the central point that the core of the thesis is concerned with accelerating GP runs and not with the behaviour within them.

Similarly, the problems that are tackled in this research are not of interest. This thesis names the test problems used in this research and Chapter 2 mentions some of those used in the cited literature. However these are not discussed in great detail because the acceleration is typically independent of the problem being tackled. In this thesis, regression problems were often used but this choice is unimportant and largely arbitrary. For the purposes of assessing acceleration, it would have been equally good to test each individual on randomly generated testcases, discard the results of their evaluations and then randomly assign fitnesses to determine selection. The difficulty of the problem being tackled typically does not influence the difficulty of the acceleration; the magnitude of acceleration typically does not influence the quality of solution (except in that faster computation permits longer runs).

There are two ways in which this independence may be imperfect and it is worth discussing them here to demonstrate that they may reasonably be ignored in this thesis.

First, some problems involve large amounts of computation (and hence time) beyond what is required to evaluate the successive generations on the problems' testcases. These steps will still require large amounts of computation (and hence time) once the evaluation and other steps tackled in this thesis have been heavily accelerated. However this observation essentially amounts to the obviously true statement

that if a problem involves unavoidable, problem-specific time costs, then they will be unavoidable and it will not be possible to tackle them with general methods. It would not enhance experimental assessment of acceleration to consider such problems because this would just entail adding some constant amount of time to the evolutionary run both before and after acceleration. For this reason, the problems chosen for use in this thesis do not require much extra computation after the testcase evaluations.

Second, it is just possible that different problems may bias the distribution of instructions in the population and this may have some effect on the relative speed improvement of a GPU implementation over a CPU implementation. For example, one particular problem may tend to lead to a population that is very heavy on division instructions and it may be that the GPU's speed improvement over the CPU may be slightly greater for division than for other instructions. There is no particular reason to think that this effect is particularly likely or pronounced, especially compared to the many other issues that might affect the assessment.

1.6 Approach

Chapter 4 describes work to tackle Objective 1. This involves the design and construction of a population-parallel evaluator. As discussed in Chapter 2, other researchers had published work describing population-parallel implementations of GP before this research was conducted (and described in a 2009 paper [49]). However the work described in Chapter 4 makes a novel contribution by tackling cyclic GP. Since cyclic GP typically requires more memory for evaluation, this induces several challenging constraints on the architecture. The low memory requirements of tree evaluation mean that a single GPU thread is able to evaluate the entirety of a reasonably sized program (on a single testcase). The much greater memory requirements of cyclic evaluation mean that the evaluation of a single program (on a single testcase) must be split over multiple GPU threads. This adds considerable difficulty to the task. To get the best results, care must be taken to schedule the evaluation of multiple programs of differing sizes over groups of threads. Chapter 4 is implementation focused and describes several of the technical issues that underlie the chapters that follow it.

Chapter 5 describes work to tackle Objective 2. This involves three steps to improve the architecture's use of a GPU evaluator. Although these steps are discussed in the context of the population-parallel evaluator described in Chapter 4, they are equally applicable to other evaluators such as the one described in Chapter 6.

The first step of Chapter 5 involves interleaving the work of the GPU and the CPU so that both may perform their respective tasks simultaneously. This is achieved by splitting the population into sub-populations or *demes*. This technique of splitting the population is quite common in EC and is often found to be beneficial, even ignoring the additional speed benefits it confers in this context. Section 2.1.10 discusses the literature relating to this issue. By using demes, the CPU can asynchronously submit one deme

to the GPU and then return to preparing the next deme. This way, both processors can be kept busy simultaneously, meaning that the overall run can be completed sooner.

The second step of Chapter 5 involves recruiting a second GPU within a single machine to make the run faster still. At the time of the work (and of a 2009 paper describing the work), Compute Unified Device Architecture (CUDA) required that each GPU was accessed by a separate thread. Hence the second step requires effort to make the code manage the evolutionary run over multiple interacting threads. In May 2011, nVidia released CUDA v4.0, which permits a single thread to access all of a system's GPUs. Nevertheless, the work was still worthwhile because it makes runs faster by recruiting an additional CPU core.

The third step of Chapter 5 addressed a potential issue raised by the techniques used in the first two steps: that the introduction of demes might slow the run down through the need for transfers between demes. A deme transfer system is added that may be configured using a class that specifies where and when transfers are conducted and another class that specifies how. This system is enhanced with a "smart mode" that attempts to allow each step of the evolutionary run to be performed as soon as possible. This is made possible by splitting the deme transfers into separate donating and receiving tasks. This enables a deme to complete its deme transfer and return to normal processing even if its neighbours have not yet taken receipt of its donations. Experimental investigation shows that the time cost of deme transfers is small and that it can be mitigated through the use of the novel, smarter approach.

Chapter 6 describes work to tackle Objective 3. The linear representation used is TMBL. In this case, the basic data-parallel system is similar to those described in work by other researchers so the chapter does not dwell on describing this. Instead, the focus is on the major problem of data-parallel approaches: the time overhead required to compile code for the GPU. Chapter 6 describes two approaches to try to reduce these compilation times. The first approach involves writing the individuals in a lower level language (called Parallel Thread EXecution (PTX)) rather than C/C++. The CUDA compiler processes any C/C++ by first compiling it into PTX and then compiling this PTX code into GPU-ready binary code. Writing the individuals directly in PTX saves the compiler from having to undertake the first stage and it also allows a greater degree of control over the final binary file. The second approach involves pre-aligning individuals to reduce duplicated common code. Since the individuals being compiled are typically highly similar to each other, the compiler is often duplicating a great deal of the work. By pre-processing the individuals to identify and remove duplications, the load on the compiler can be reduced.

Chapter 7 describes work to tackle Objective 4. Profiling was used to identify those bottlenecks that were needlessly wasting CPU time. The two areas identified were tournament selection and the copying of individuals to construct new generations. In the case of tournament selection, investigation indicated that much of the time was

spent on generating random numbers. This is addressed with the help of a mathematical analysis of without-replacement tournament selection, which is used to construct a functionally equivalent algorithm that requires fewer random numbers. The mathematical analysis is also used to investigate the selection pressure of various tournament configurations. This is taken further by using an extension of the mathematical analysis that leads to a new measure of selection pressure. The new measure is then used to provide a selection pressure contour map of different tournament configurations for both with-replacement and without-replacement tournament selection.

The second optimisation of Chapter 7 tackles the time spent copying individuals to construct new generations. After consideration of the problem, it is clear that a major obstacle to its solution is the difficulty in understanding it clearly. To this end, a new way is proposed to depict the problem. This enables the proposal of a new heuristic to attempt to reduce the number of copies.

1.7 The Structure of the Thesis

Chapter 2 places the thesis in its context within the relevant literature on accelerating EC and more specifically GP. Section 2.1 occupies most of the chapter and describes previous approaches to this problem, with particular emphasis on GPU-based techniques. The use of assembly and machine code is discussed in Section 2.2 and the chapter closes with a discussion of tournament selection in Section 2.3.

Chapter 3 describes methods involved in the research. Section 3.1 discusses GP (including Cartesian Genetic Programming (CGP) and cyclic GP) in more detail. Section 3.2 describes a novel method, TMBL, which is used in some of the thesis. Section 3.3 describes CUDA. Section 3.4 outlines some conventions used in the thesis.

Chapter 4 describes work to construct an efficient population-parallel GPU evaluator that is capable of handling cyclic GP. The work is introduced in Section 4.1 and the issues pertaining to cyclic GP are reiterated in Section 4.2. Section 4.3 describes the relevant problems and the solutions adopted. Section 4.4 explores the system's novel features in more detail. Section 4.5 discusses issues relating to measuring GPU performance in Section 4.5.1 and describes the experiments and presenting the results in Section 4.5.2. Section 4.6 summarises the chapter and its contribution.

Chapter 5 describes work to improve run time further. Section 5.1 introduces the chapter. In Section 5.2, demes are used to allow the GPU and CPU to simultaneously operate on separate tasks to reduce run time. In Section 5.3, a second GPU is recruited in the same machine to further reduce run time. In Section 5.4, the transfers between demes are found to not unduly slow the system and then a refined system reduces this small cost further. The chapter and its contribution are summarised in Section 5.5.

Chapter 6 describes work on data-parallel acceleration. The chapter begins with an introduction in Section 6.1 and a reminder of TMBL in Section 6.2. Section 6.3 describes work to use the lower-level PTX language to reduce compilation times and Section 6.4

describes work to pre-align batches of similar code before compilation to identify similar code and reduce redundancy. The two techniques are combined on large TMBL individuals of 1000 instructions in Section 6.5. Section 6.6 summarises the chapter and outlines its contribution.

Chapter 7 describes work to reduce time wasted by the CPU by optimising two tasks, which were selected based on profiling results. The work is introduced in Section 7.1. Section 7.2 describes work to optimise tournament selection. Section 7.3 describes the second target for optimisation: the copying of individuals to construct new generations. A summary of the work is presented in Section 7.4 along with a description of the contribution that it makes.

The thesis closes in Chapter 8 with a discussion of the work and the potential avenues for future research that it suggests.

2 Literature Review

It is important to understand the position of this thesis within the body of relevant literature. Since the central aim of the thesis is to accelerate a form of Evolutionary Computation (EC), it relates to literature that describes other likewise attempts. Section 2.1 steps through differing approaches to this challenge, paying special attention to this thesis's choice of technology: the Graphics Processing Unit (GPU). Section 2.1 covers a good deal of the literature relevant to this thesis but leaves two gaps that are filled by Sections 2.2 and 2.3. Section 2.2 covers with literature relevant to to Chapter 6 on the use of assembly and machine code. Section 2.3 covers literature relevant to Chapter 7 on tournament selection and ways of measuring selection pressure.

This chapter mentions some of the application areas on which the described systems have been tested. However, as discussed in Section 1.5, these are not discussed in great detail because the acceleration is typically independent of the problem being tackled. In particular, the difficulty of the problem being tackled typically does not influence the difficulty of the acceleration and the magnitude of acceleration typically does not influence the quality of solution (except in that faster computation permits longer runs).

2.1 Acceleration

Much of this work is concerned with accelerating EC and in particular, forms of EC like Genetic Programming (GP). The GPU is used as the main approach. To understand this work, it is important to place it in its context within the literature on approaches to accelerating EC. Various approaches have been researched [73]. The following sections provide brief synopses of the major approaches. There are two broad categories of technique: techniques that adjust the algorithm so it works more quickly and techniques which leave the algorithm alone but implement it on some sort of parallel hardware. In practice, this distinction is not clear-cut since techniques in the latter category may adapt the algorithm to fit the parallel hardware.

The work on refining the algorithm involves either reducing the number of evaluations or making each evaluation do more. This reflects the dominance of evaluation in total run time, particularly for GP.

Presently, the work on using parallel hardware mostly fits into two categories: using some sort of network of machines or using some sort of parallel hardware device on one machine. The former tends to involve less technical difficulty and more scalability whereas using one parallel machine is often better with regards to dedicated access, volatility of processors, homogeneity of processors and potential memory sharing. El-Ghazali Talbi has discussed issues relating to these properties in his book on metaheuristics [85]. One of the pieces of work discussed in Section 2.1.9 fits into both

categories as it combines both types of parallelisation by using a network of machines equipped with GPUs [33].

2.1.1 Algorithm Design Approaches

The first area to be considered as a target for acceleration is the design of the algorithm itself. In some cases it is possible to reduce the GP run time by redesigning the algorithm to reduce the number of evaluations. The aim is to achieve this reduction with minimal impact on the algorithm's behaviour. There are two clear sources of evaluations which can be skipped without effect:

- **Skipping evaluations that do not affect the selection.** It is often possible to optimise away some of the testcase evaluations if they are not required to differentiate the fitter of two solutions. In general, these evaluations are often difficult to identify in advance. However it is helpful to observe that after testing a population on a large number of testcases, further changes to the fitness ranking of the individuals may be unlikely or even impossible. If the ranking is the only information needed by the selection scheme (the method of choosing the number of offspring for each individual), then the final testcases may be of little or no value. In this case, the evaluations over these testcases are good candidates for being skipped. It may be possible to use information from previous generations to estimate in advance a sensible number of testcases over which to evaluate.

  Gathercole et al proposed three different methods for selecting testcase subsets and compared the effects of each of them [22]. Dynamic Subset Selection (DSS) uses the information within a run to focus on those testcases which either haven't been used for a while or which are difficult (frequently misclassified); Historical Subset Selection (HSS) is less dynamic and focuses on those testcases which have been found difficult (frequently misclassified) in previous runs, padded out with a few easier testcases; Random Subset Selection (RSS) randomly selects a new subset of testcases at the start of each generation. The investigation using Thyroid problem data found that DSS allows GP to find better, more general results in less time and also found that neither HSS nor even RSS make GP much worse, despite reductions in run time.

  This work was extended three years later [24] with an investigation of another heuristic called Limited Error Fitness (LEF). In the LEF method, an error limit is applied to the testcases, which means that each individual is tested on successive testcases until it has misclassified that number of testcases. Once an individual has misclassified the error limit number of testcases, all further testcases are counted as failures. As the run proceeds, the error limit and the ordering of the testcases are updated according the performance of the Best Of Generation Individual (BOGI). This sort of approach might be more difficult to implement

for problems for which answers to each testcase cannot be so easily divided into right and wrong. The method was tested on the Boolean Even N Parity problem and the authors found that using LEF allowed GP to solve problem instances that it could not solve otherwise.

In another work, the same authors compared LEF against DSS and against standard GP on the TicTacToe problem and, again, the Thyroid problem [23]. They found that GP with DSS gets better answers with fewer evaluations. Intriguingly, they also found that smaller populations over fewer generations consistently produced a better answer using fewer tree evaluations. They suggested that "it is certainly worth an exploratory run or three with a small population size before assuming that a large population size is necessary."

Teller proposed another approach called the Rational Allocation of Trials (RAT) [87]. This approach is more theoretical and involves performing evaluations and then, based on the results, allocating more testcase evaluations to individuals according to the expected utility of those evaluations. Sample statistics are used to predict the chance that another testcase evaluation might make a given individual win some tournament it is currently losing or lose some tournament it is currently winning. "The key idea is that if an individual has no chance of winning a tournament, or if an individual is virtually guaranteed to win a tournament, no further fitness cases need to be evaluated" [87].

As EC is an inherently stochastic algorithm, it may be overkill to attempt to find the perfect fitness ranking of individuals. Provided that enough selection pressure is maintained in the correct direction, it may be reasonable to use a subset of the complete testcases for each evaluation and to set the size of that subset at the start of the run (in essence, the strategy called RSS by Gathercole et al [22]). In a vivid illustration of this point, Langdon has used a subset of $2^{11} = 2048$ testcases to solve the 20-mux problem which has $2^{20} = 1048576$ testcases and a subset of $2^{13} = 8192$ to solve the 37-mux which has $2^{37} = 137438953472$ testcases [44].

- **Skipping evaluations that have already been performed.** The approach to reap the benefits of these evaluations is to cache the previous results and then reuse them. For example, if a GP individual survives to the next generation unaltered and is to be tested on the same test set as before, it may be possible to reuse the previous results rather than recalculating them. Tree representations are particularly amenable to sophisticated caching techniques. This is because their substructures evaluate to the same result for a given testcase regardless of the context in the individual (as long as nodes with side effects are not being used). This makes it possible to cache results for sub-trees which may be of great benefit where there are sub-trees which are present in many individuals throughout a population. Unfortunately, this technique will not work for those representations

in which all parts of an individual may potentially interact.

Many frameworks use some mechanism to perform this sort of fitness caching, for instance the Evolutionary Computation in Java (ECJ) framework (`http://cs.gmu.edu/~eclab/projects/ecj/`) allows a fitness evaluation method to set an "evaluated" flag on an individual so that it need not be evaluated again until it is modified.

This technique is more difficult if the data-set changes between generations, perhaps because different generations are tested on different subsets of the full data-set or because the evolution is occurring in a dynamic environment.

More broadly, where fitness evaluations take an unreasonable amount of time it may be appropriate to use *fitness approximation* or *surrogate assisted* EC. These methods involve finding some faster model with which to approximate the full fitness evaluation. For instance, when attempting to evaluate the performance of a turbine blade, wind tunnel experiments can be approximated by computational fluid dynamics simulations and the full simulation using Navier-Stokes equations can be made even quicker by neglecting various aspects, leading to a simpler form of equations. This example is drawn from Jin's summary of the field [37]. More recent publications include work by Lim et al to develop a "generalization of surrogate assisted evolutionary frameworks" to deploy these techniques without the need to hand-craft a model for each problem [52].

2.1.2 Sub-Machine-Code Genetic Programming Approaches (SMCGP)

The alternative to reducing evaluations is to make each evaluation do more. When evolving GP individuals which operate on a native Boolean type and which only use the standard Boolean operators, several testcases may be calculated simultaneously using a standard Central Processing Unit (CPU) [70]. Poli achieved this by using the inherent parallelism available in the CPU which calculates Boolean operations on multiple bits at once. The 64 bit architecture CPU is now widely available and can be used to perform 64 Boolean operations at once. For application areas that require decimals or integers, Poli also showed in the same work how larger subsets of bits can be used. For instance, he used a 64 bit number to represent 8 numbers, each containing 8 bits. Decimal numbers were implemented by using a fixed point (rather than floating point) implementation although this requires some sophistication in the handling of some of the operations.

This method presents a trade-off: an increase in speed comes at the cost of a decrease in the range and resolution of the types on which it operates. Even at the limit when using Boolean types, the method can only ever deliver a speedup equal to the native number of bits for the CPU, which seems unlikely to be greater than 64 for some time.

2.1.3 Grid Computing Approaches

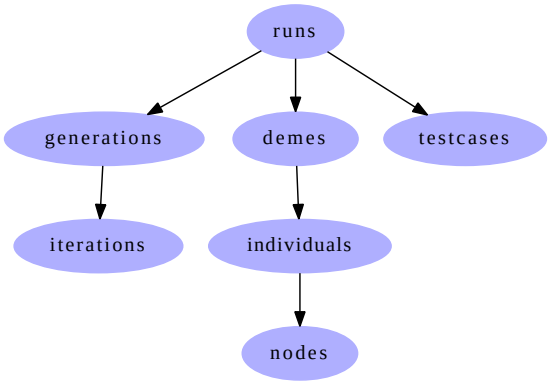
Once the algorithm has been fixed, the other clear way to improve the speed of EC is to provide more computing power. The GP community is fortunate to work on an algorithm that is "embarrassingly parallel" [1] which means that it can easily be divided into parallel tasks with few if any inter-dependencies. Figure 3 illustrates the many ways in which a GP run might be divided. The consequence of this is that GP tends to be an effective exploiter of parallel computing resources.

One way to approach this is to use a single many-processor machine: a supercomputer. In 1996, Turton et al used a Cray T3D 512 processor supercomputer at Edinburgh University to perform GP [90]. The focus of the research was on the GP results and the supercomputer's speedup was not reported. This approach is impressive but unfortunately, few people have access to a supercomputer.

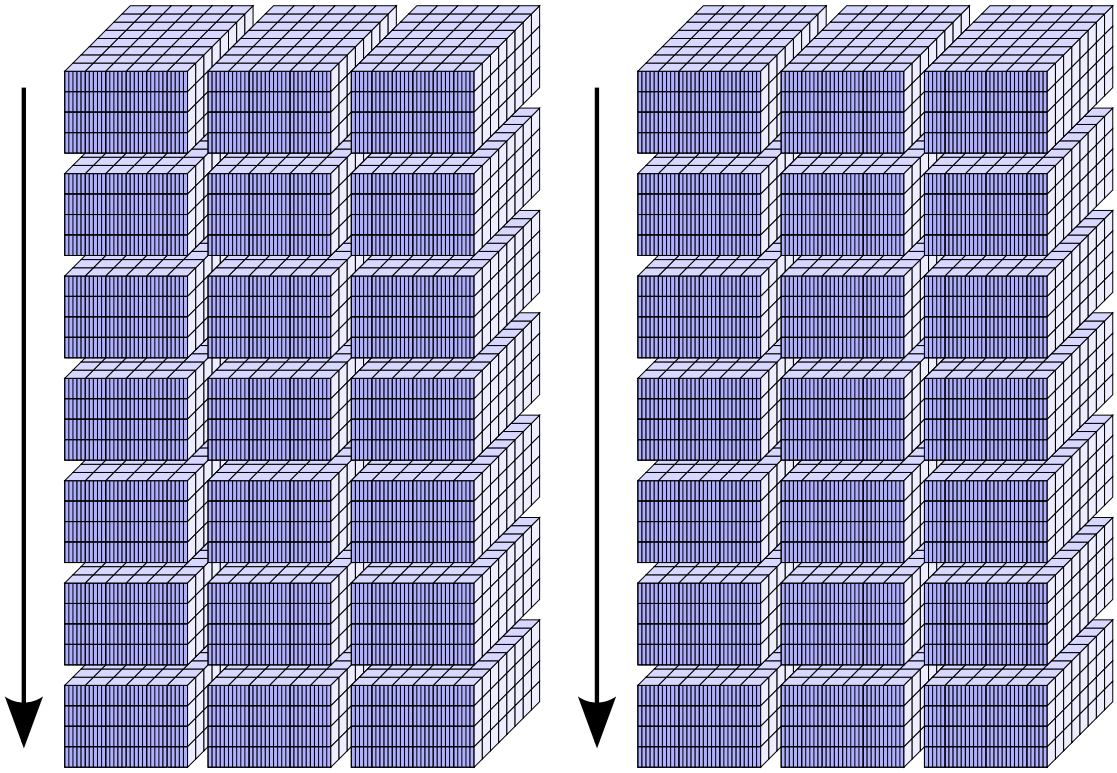
A more prosaic approach is to use a network of normal machines. In 1998, Andre et al described their work using a network of 66 transputers and one Pentium computer to accelerate GP [1]. The transputer was a microprocessor architecture designed specifically for parallel computing. The authors' system of transputers displayed a near-linear speedup in executing a fixed amount of code.

It is now quite common in many scientific computing fields to use open source operating systems and scheduling software to divide jobs over many computers and this has been used in some GP implementations. In 1999, Bennett et al described their system using 10 nodes, each running Linux on a 533 MHz Alpha processor [8]. They stated the total bill was $\$18,134$ and by connecting their computers in a "Beowolf-style" cluster, they were able to achieve about $0.5 * 10^{15}$ floating point operations per day (i.e. $5.787 * 10^{9}$ FLoating point OPerations per Second (FLOPS)). The disadvantage of this approach is that it requires large financial resources to obtain enough computers to achieve significant acceleration results. Furthermore, maintaining many machines and administrating computation over those machines typically requires a lot of human time.

In 1999, Chong described a GP system using Java Servlets to distribute GP over the Internet. This allowed the system to be executed "across the world over the Internet on heterogeneous platforms without any central coordination". In 2002, Groß et al [26] described their distributed system for learning chess. They started with an algorithm that could play chess and then used GP and Evolution Strategies (ES) to improve it. They constructed a distributed environment, called qoopy, and used this to perform their computation across the Internet. If the general public can be persuaded to donate some of their computational resources to such systems, they could prove to be very cost-effective . In cases for which the public's computation can not (or should not) be recruited, such schemes do not cheapen the computation, they merely distribute it.



(a) Relationships between GP components



(b) Computational cuboids suggest how GP may be parallelised

Figure 3: How embarrassingly parallel is GP? Let me count the ways: runs, demes (i.e. subpopulations), individuals, nodes, testcases, generations and iterations. Subfigure 3(a) shows a rough relationship diagram in which each arrow means "contains one or more". This structure applies to Subfigure 3(b), which shows two independent runs, using three demes of six individuals with five nodes, evaluated over eight testcases for seven generations of four iterations. The big arrows indicate the passage of time through the runs. There are several constraints on possible parallelisation. For a given testcase, all nodes in an individual must be complete before any node in the next iteration may be evaluated. All computation within a deme must be complete before moving on to the next generation. Deme transfers may require demes to complete certain generations before other demes can start other generations. Some representations may apply further dependencies between an individuals nodes within an iteration. This figure draws from ideas presented in a simpler version [46].

2.1.4 Field Programmable Gate Array (FPGA) Approaches

A Field Programmable Gate Array (FPGA) is a semi-conductor device which allows some of its logic to be programmed dynamically. The FPGA is then able to use the new configuration to process many entries using its highly parallel architecture. In 2000, Heywood et al investigated the use of the FPGA device to accelerate GP and to reduce the amount of source code it requires [35]. They concluded that the work they had completed was still "in the initial stages". In 2003, Eklund described a massively parallel linear GP model using VHSIC Hardware Description Language (VHDL), a language that could be implemented on devices such an FPGA [18]. VHSIC stands for "very-high-speed integrated circuits". In 2008, Vasicek implemented Cartesian Genetic Programming (CGP) on an FPGA and achieved a speedup of "30-40 times" [91].

2.1.5 Graphics Processing Unit (GPU) Approaches

Of the parallel processors that are currently widely available, perhaps the most interesting for EC researchers is the GPU. The GPU was developed for generating realistic three dimensional images at high speed, primarily for the games industry. The demands and vast financial resources of this industry led to a chip containing simple processors that are highly effective at floating point computation and at performing the same operations on multiple independent data.

As the GPU's rate of performing floating point calculations left that of the CPU trailing ever further behind, interest grew in the idea of exploiting the chips to tackle other parallel problems. During the same period, the processor manufacturers moved the hardware to a more generic model that could be adapted to new applications. In recent years there has been a rapid growth in interest in General-Purpose computing on Graphics Processing Units (GPGPU) techniques (note: GPGPU should not be confused with using GPUs for GP; GP is just one possible application of GPGPU). The GPU to be found on current graphics cards has remarkable potential computing power and the power of the technology appears to have been improving more quickly than for the standard CPU [15].

EC techniques such as GP are well suited to GPU computation because the task of evaluation is often easily broken up into independent evaluations for different individuals, testcases, population subgroups (or demes) and runs. As mentioned in Section 2.1.3, tasks which can be divided so easily are often referred to as "embarrassingly parallel" [1].

The most important framework in the context of this thesis is nVidia's Compute Unified Device Architecture (CUDA), which was used in this research and in many of the more recent papers implementing GP on the GPU (as summarised in Table 2 in Section 2.1.7). CUDA is a freely-available, proprietary framework for exploiting the compute capabilities of the nVidia GPU, and is available for Windows, Linux and

Mac OS. nVidia support CUDA users through substantial documentation and Internet discussion fora (`http://forums.nvidia.com/index.php?showforum=62`).

CUDA is based on the C programming language, although many C++ constructs are being added in newer releases. To use the CUDA framework, a user must write a function, called a *kernel*, to be run on the GPU. The kernel is compiled by the CUDA compiler and then uploaded to the GPU. It is then available for calls from CPU code. CPU code may dynamically upload new kernels to the GPU. Where this ability is not needed, the upload can be done behind the scenes instead by code generated automatically by the CUDA compiler.

To use a kernel, the CPU code performs a kernel *launch*, which means launching very many GPU threads to execute the kernel. Once the kernel is launched, the CPU code may perform other tasks whilst the kernel is executing. The GPU threads are capable of performing different computations from each other because the kernel's code can access the identity of the thread under which it is executing and use this to guide the computation. For example, a kernel may use the thread's identity to index the location of input data to the computation and then to index the location to which the results should be written. This way, kernel code has a high degree of flexibility in the way it divides up the work between threads.

CUDA GPU threads are grouped into thread *blocks*. A kernel launch specifies a *grid* of threads by specifying the number of threads per block and the number of blocks in the grid. A GPU thread may communicate with other threads in the same block via on-chip *shared memory* and may synchronise with them via a call to the CUDA function `__syncthreads()`. Kernel code should make no assumptions about the ordering and parallelisation of the execution of different thread blocks.

CUDA's functionality improves through releases of new hardware and new software. An nVidia card's compute capability denotes the functionality that it provides. Typically, a card provides all the functionality of cards with lower compute capabilities. The cards used in this work were of compute capability 1.3. Cards of compute capability 2.0 or higher offer several significant improvements to functionality as discussed in Sections 4.3, 4.3.3, 4.4.2, 4.4.3, 4.5.2 and 5.2.1. CUDA also improves through new releases of the software involved in the CUDA framework. At the time of writing, the latest stable release is v4.0, which was released in May 2011. Table 1 outlines the release history of this CUDA toolkit.

Current GPU architecture is essentially Single Instruction, Multiple Data (SIMD), which means that it involves many threads executing the same instructions on different data. nVidia describe their CUDA framework as Single Program, Multiple Data (SPMD) rather than SIMD. This means that all the threads in a given kernel launch must execute on the same program but do not necessarily have to follow the same execution paths through the code (and so do not have to all execute a single instruction simultaneously).

Month and year of release	CUDA version	Release highlights
May 2011	4.0	Share GPUs across multiple threads, use all GPUs in the system concurrently from a single host thread, no-copy pinning of system memory, C++ new/delete and support for virtual functions, support for inline PTX assembly
November 2010	3.2	Improved libraries
June 2010	3.1	Runtime/Driver interoperability, support for function pointers and recursion
March 2010	3.0	C++ class inheritance and template inheritance support, Driver/Runtime buffer interoperability
June 2009	2.3	Improved support for double-precision, improved handling of SLI GPUs, several Visual Profiler enhancements
May 2009	2.2	CUDA Visual Profiler reports memory transactions, improved OpenGL interoperability performance, new zero-copy feature allows kernel functions to read and write directly from pinned system memory
January 2009	2.1	Debugger support using gdb for CUDA, C++ templates are now supported in CUDA kernels
August 2008	2.0	Double precision support, improved device to array memcpy performance
December 2007	1.1	CUDA integrated into display driver, asynchronous API for memory copies and kernel launches, event API for querying status of CUDA calls
June 2007	1.0	Initial release

Table 1: A history of CUDA toolkit releases with highlights of each release's new features. This information was extracted from `http://developer.nvidia.com/cuda-toolkit-archive` and linked pages. Most releases also improve the supporting libraries and add support for new cards and new operating systems. Many releases are preceded by one or more release candidates which have been omitted from this table.

At present, tackling problems using the GPU requires dividing the problem appropriately and writing a suitable GPU program or "kernel" that can tackle each of these sub-problems. In practice this can be an intricate process. However the rewards are significant, with some applications having seen speed improvements of two or more orders of magnitude [28].

GP is particularly well suited to a GPU implementation because it often uses floating point numbers as the basic type in its evaluations and GPUs are particularly effective at floating point computation. Tackling GP with GPU technologies has been referred to as General Purpose Genetic Programming on Graphics Processing Units (GPGPGPU) and a website (`http://www.gpgpgpu.com`) is maintained by Simon Harding.

2.1.6 Uses of the GPU for EC Other than GP

Papers describing some of the earliest attempts to utilise the GPU for EC were published in 2005. Wong et al implemented the fitness evaluation, mutation and reproduction of an Evolutionary Programming (EP) algorithm on the GPU [104] [20]. The competition and selection were performed by the CPU. Their EP was tested on a set of benchmark optimisation problems using an nVidia 6800 Ultra. They observed speedups ranging from "about 1.25 to about 5.02".

Yu et al implemented a fine-grained genetic algorithm on the GPU [107]. They implemented both the evaluation function and the genetic operators on the GPU (along with a random number generator). They used the Cg framework with an nVidia 6800 GT and tested their system on the Colville minimisation problem. They observed an overall best speedup of "about 15 times" compared to a CPU implementation. They concluded by mentioning two key limitations: the "bottleneck of transferring data between system memory and video memory in each GA loop" and that they found the "commonly used binary encoding scheme of GAs seems hard to be implemented on the GPU because there is no bit-operator supported in current GPUs".

In 2007, Li et al implemented all steps of a fine-grained parallel Genetic Algorithm (GA) on the GPU [51]. The system was tested using three fitness formulae. Using an nVidia 6800 LE, they observed speedups of up to 73.6.

Wong attempted to combine these techniques with a Multi-Objective Evolutionary Algorithm (MOEA) in 2009 [103]. He had found that in addition to evaluation, "the non-dominance checking and the non-dominated selection procedures are also very time consuming" and so used a CUDA implementation to tackle this problem. He tested the system using an nVidia 9600 GT on a range of two-objective and three objective benchmark problems and observed speedups ranging from 5.62 to 10.75.

Munawar et al tackled the MAXimum Satisfiability (MAX-SAT problem) using the CUDA framework. [63]. They tested their system on an nVidia Tesla C1060 and observed speedups of up to 25 times.

2.1.7 Uses of the GPU for GP: Data-Parallel

Work on using the GPU to accelerate GP is of particular relevance to this thesis. The key publications in this area are summarised in Table 2.

Given the SIMD architecture of the GPU, perhaps the most intuitive approach is the "data-parallel" (or "fitness case parallel") approach. This uses the GPU's parallel threads to evaluate the different testcases. A separate GPU kernel is compiled for each individual and this is then used to evaluate the complete data set. For very large data sets, the time taken to compile the kernels is a small fraction of the total evaluation time and remarkable reductions in evaluation time have been observed as described below.

This approach was used by Chitty in one of the early pieces of work to accelerate GP with a GPU [15]. He used C for Graphics (Cg)—a language developed by nVidia—and an nVidia 6400 GO. This approach requires creating and compiling a fragment program for each individual and then evaluating it over the complete data set. The problems used to test the system were symbolic regression, the Fisher Iris classification data set and the 11-way multiplexer. The best acceleration compared to the CPU implementation was found for the 11-way multiplexer at around 29.98 times faster. The graphs in the paper indicate that for small data sets, the GPU implementation can be slower but that as the data set increases, the GPU implementation becomes relatively

Primary author	Year	Parallelism	Form	GPU	Framework	Problems
Chitty [15]	2007	Data	Tree	6400 GO	Cg	Symbolic regression, Fisher Iris data set, 11-way multiplexer
Harding [28]	2007	Data	CGP	7300 GO	Accelerator	Floating point, binary, regression, two spirals, protein classification
Harding [32]	2007	Data	CGP	7300 GO	Accelerator	Regression, two spirals, protein classification, cellular automata pattern
Harding [27]	2008	Data	CGP	7300	Accelerator	Image Filtering
Harding [29]	2008	Data	CGP	8800 GTX	Accelerator	Image Filtering
Langdon [45]	2008	Population	Tree	8800 GTX	RapidMind	Mackey-Glass time series prediction
Langdon [46]	2008	Population	Tree	8800 GTX	RapidMind	Bioinformatics: breast cancer with GeneChips
Robilliard [76]	2008	Population	Tree	8800 GTX	CUDA	Regression, Multiplexer, intertwined spirals
Wilson [98]	2008	Population	LGP	8800 GTX	XNA	UCI Ecoli classification, symbolic regression (sextic)
Harding [33]	2009	Data	CGP	8200 $\times$ many	CUDA	Intrusion detection, image filtering
Lewis [49]	*2009*	*Population*	*CCGP*	*8800 GT* $\times$ *2*	*CUDA*	*Symbolic Regression*
Robilliard [77]	2009	Population	Tree	8800 GTX	CUDA	Regression, Multiplexer, intertwined spirals
Wilson [101]	2009	Population	LGP	8800 GTX	XNA	Symbolic regression (sextic)
Langdon [44]	2010	Population	Tree	295 GTX	CUDA	Multiplexer (27-mux and 37-mux)
Maitre [55]	2010	Population	Tree	295 GTX $\times \frac{1}{2}$	CUDA	Regression
Wilson [99]	2010	Population	LGP	8800 GTX	XNA	Symbolic regression (sextic)

Table 2: A summary of some of the major studies into accelerating GP with the GPU. The entry in italics is a 2009 paper describing some of the work in this thesis. Here, CCGP refers to Cyclic Cartesian Genetic Programming.

better. Rough readings from the paper's graph of times on the regression problem indicate the GPU implementation is around twice as slow for 100 testcases but around 10 times as fast for 1600 testcases.

Around the same time, Harding et al also used a data-parallel approach [28]. Their work used the Accelerator package which is a .Net assembly, available only on Windows. Accelerator allows client code to use the GPU at a high level through applying mathematical operations to special arrays. The evaluation is performed "lazily", meaning that until the result is requested, Accelerator just stores the operations to be performed. When a result is requested, the necessary compilation of the instructions is performed, the code is executed on the GPU and the results returned. From the developer's point of view, this makes GPU access relatively simple. Indeed the authors report that "the total time required to reimplement an existing parser tree based GP parser was less than two hours [...]". The implementation used CGP and was tested on a trivial floating point problem and trivial binary problem as well as a symbolic regression problem, the two spirals problem and a protein classification problem. Again, the results indicated the GPU implementation was slower than the CPU implementation on small data-sets and short expressions but was faster as these grew. In one case, the GPU implementation was 7351.06 times faster when evaluating expressions

of length 10000 over 65536 testcases.

In a related paper, the authors extended this work to cover artificial developmental systems [32]. On testing a cellular automata system, the authors again found that the GPU implementation was slower than the CPU implementation for few cells and short expressions but much faster as these quantities increased.

This evidence suggests that for large data sets, data-parallel approaches are extremely effective but that for smaller data sets, they are less effective and can even be slower than a standard CPU implementation. This is because a data-parallel approach has a large overhead of compiling new individuals and transmitting them to the GPU to be evaluated. This time consuming overhead is worthwhile if and only if there are enough testcases to be evaluated. What does this mean for data-parallel methods? In the first two of these data-parallel papers, the authors stated the following:

> "Many typical GP problems do not have large sets of fitness cases for two reasons: First, evaluation has always been considered computationally expensive. Second we currently find it very difficult to evolve solutions to harder problems. With the ability to tackle larger problems in reasonable time we have to also find innovative approaches that let us solve these problems. [...] This leads to a gap between what we can realistically evaluate, and what we can evolve. The authors of this paper advocate developmental encodings, and use the evaluation approach introduced here we will be able to test this position." [28]

The authors included similar sentences in the conclusion of their paper covering developmental systems [32]. On the one hand, this statement can be seen as an encouraging call for the community to use the GPU to go further: to investigate if there are problems with vast data sets which allow GP to achieve new things or to develop new techniques to make use of such huge data sets. On the other hand, the statement can be seen as identifying a weakness of the data-parallel approach and it has been quoted in that light [76] [77].

2.1.8 Uses of the GPU for GP: Population-Parallel

For problems with smaller data sets, which are not as well disposed to data-parallel approaches, "population-parallel" approaches have been used [45] [76].

These involve using the GPU's parallel threads to evaluate the different individuals in the population (and potentially the different testcases too). The trick to achieve this is to write a single GPU interpreter kernel that treats programs as data. The advantage of these methods is that they do not require a new kernel to be compiled and launched for each new individual and so avoid the associated overhead. The difficulty is that each individual may have different behaviour. This is solved by using a GPU interpreter

kernel that handles the different individuals. This makes population-parallel methods more complicated to implement and slower at evaluating.

The names "data-parallel" and "population-parallel" reflect the idea that the former's kernels parallelise over the data set whereas latter's kernels also parallelise over the population. In practice, things are not that simple because data-parallel kernels may actually contain the code for several individuals and so may also parallelise over both the population and the data set.

The first papers on population-parallel approaches were published in 2008. Langdon et al used the RapidMind framework to "evaluate an entire population of a quarter of a million individual programs on a non-trivial problem in 4 seconds" [45]. They achieved this by constructing a population-parallel system to accept both the programs and the testcases as data. To allow GP trees to be evaluated efficiently, the trees were represented using linearised Reverse Polish Notation (RPN). This allows the GPU to interpret the tree as a list of instructions which use a stack. Rather than converting to linearised RPN for each evaluation, the system kept the individuals in this form throughout and used crossover and mutation operators that act on linearised RPN directly.

This paper described the GPU's rate of evaluation, not just by comparison to that of the CPU, but also by describing the absolute rate of performing GP operations (measured in million GP operations per second). Section 4.5.1 outlines several issues that warn against drawing hasty conclusions from direct comparisons of such rates. Nevertheless, such absolute rates of GP evaluation are the only way to make any such comparisons at all (and many of them will not be prone to the highlighted issues) so it was a big contribution to begin the trend of stating them.

Henceforth, the abbreviation Mgpop/s will be used to indicate "mega GP operations per second", where mega is used to mean 10^6 not 2^{20} (at least for the values generated in this work). The "gp" in these abbreviations highlights that the measurement only records the GP operations, which may be a small fraction of the total number of instructions, particularly in the case of a GP interpreter running on a GPU. In tables and figures, the term Mgpop/s may be shortened to Mo/s for brevity but this still refers to GP operations.

Langdon et al tested this on the problem of trying to predict the next value in the Mackey-Glass chaotic time-series using the previous 128. The system ran at 895 Mgpop/s for programs of size 11 and at 1056 Mgpop/s for programs of size 13. A comparison with a CPU implementation on the Mackey-Glass problem found that the CPU implementation took seven times longer. The system was also used to tackle the problem of classifying the sub-cellular location of proteins from their amino acid composition and to tackle a pair of problems that were labelled "Laser_a" and "Laser_b" in the results table. Larger programs were used for these problems. The authors noted that increased stack depth harmed the performance.

The system generated the correct behaviour for each instruction using a five way conditional statement. To investigate the effect of this, they compared a normal execution with one that had been altered to execute every possible condition (and discard none of the results). They found this only made the execution 2.89 times slower (rather than five times slower as might have been expected). They proposed a hypothesis that this was due to the time required by the terminals to load the data from global memory.

The paper does not give a precise description of the layout of work over the GPU threads. It may be that the RapidMind platform does not give the user control over this.

Another population-parallel paper was published in the same year (2008) detailing the work of Robilliard et al [76]. They built a population-parallel system within the ECJ library using the CUDA framework. The CPU translated the individuals to linearised RPN for each evaluation. Within each CUDA thread block, the threads are arranged in groups of consecutive threads called warps. At the time of writing, all CUDA devices use 32 threads per warp. The importance of this is that a warp's threads always execute in parallel. Divergence between threads within a warp ("warp divergence") costs time since it involves the whole warp following every branch taken by the warp's threads. Minimising warp divergence is one of the top priorities to maximise speed. Heeding this, the authors took care in how their system distributed work across threads. Their design ensured each warp evaluated the same GP program on different testcases. This approach reduces the difference in behaviour between threads in a warp and hence reduces warp divergence.

They tested their system on an nVidia 8800 GTX using a regression problem, two multiplexer problems and an intertwined spirals problem. Their best full-run speedups for these problems were around 50, 5 and 5 respectively. The paper identified that after acceleration, the CPU tasks had become the bottleneck in the system.

Connecting with standard EC frameworks—in this case ECJ—may be a useful way for the power of the GPU to be exposed to the typical EC user. This is discussed further in Section 2.1.13. In the same spirit, the authors made sample code available via the Internet.

In a later publication, the same authors extended their work with their ECJ system [77]. They investigated the effect of the layout of work across threads, the effect of using on-chip, shared memory and the effect of using a linearised RPN form on the CPU to avoid the costs of translation.

The first investigation compared the layout of work across threads as used in the previous work (now labelled "BlockGP") with a simpler approach, labelled "ThreadGP". The ThreadGP scheme evaluates each individual with its own thread over all testcases. The authors suggested that this sort of layout might be generated when using high-level kits that do not give access to thread management. It is not clear that this is correct. The move from BlockGP to ThreadGP involves changing from each thread

evaluating some of the testcases to each thread evaluating all of the testcases. By contrast, previous data-parallel approaches have divided work over the testcases so that no thread evaluates more than one testcase.

The experiments were again performed using an nVidia 8800 GTX on regression, multiplexer and spiral problems. The BlockGP arrangement was up to 12.94 times faster than ThreadGP and was faster in all tests performed.

The second investigation compared the system from the previous paper to one in which the individuals were kept in linearised RPN form throughout, saving the CPU the task of conversion. The authors acknowledged the assistance of W B Langdon in helping them achieve this. Experiments showed that this made the full run up to 10.193 times faster and that it made an improvement in all tests performed.

The third investigation looked into the effects of loading the programs into shared memory so that they do not need to be reloaded when evaluating successive testcases. This was found to reduce the run time considerably in most cases.

A later population-parallel based work combined the approach with Sub-Machine-Code Genetic Programming (SMCGP) techniques to tackle a vast Boolean problem [44]. This system allows the GPU to process 32 (or even 64) Boolean testcases in parallel. Furthermore, it uses four simple Boolean operators (and, or, nand, nor) that can typically be evaluated much faster than floating point operations such as division. Again, the individuals were represented in linearised RPN throughout. The system was tested using an nVidia 295 GTX, a graphics card containing two GPUs.

The problems tackled were the 20-way multiplexer and the 37-way multiplexer. These problems are vast, with $1,048,576$ and $137,438,953,472$ testcases respectively. The system's fitness function used a random sample of 2048 and 8192 testcases respectively. When a candidate was found that produced the correct answer on the full sample of testcases, it was tested on the full set. The author stated that the 20-mux problem had "never been solved by a tree GP before" and that the 37-mux had "never been attempted before" but had been solved by this system "in under a day".

The author reported a "sustain peak performance of 665 billion GP operations per second". Averaged over the whole run, the author found a rate of $254,000$ Mgpop/s. Even taking into account the SMCGP trick of performing 32 operations in one and the use of two GPUs, this is an impressive result for a population-parallel system.

2.1.9 Distributed Use of GPUs for GP

To achieve even more computational power, the data-parallel technique has been taken further through the use of a network of machines [33]. In this system, one machine acts as a root node, which sends work to additional client machines which are each equipped with a suitable GPU, a CUDA kernel compiler (`nvcc`) or both. At the beginning of a run, the root machine divides the testcases between the GPU equipped machines. For each evaluation, the root's first stage is to divide the population up and

send a group of individuals to each of the `nvcc` equipped clients to be compiled into CUDA kernels. The second stage is to send all of the compiled kernels to each of the GPU equipped clients to be executed on their testcases.

This setup allows the authors to harness spare computational power of a decent number of low power graphics cards available in a student laboratory. The cards used were nVidia GeForce 8200 and the total system used "between 14 and 16 computers". It is not stated which of the machines were equipped with a GPU, which with `nvcc` and which with both.

The problems on which the system was tested use a linear fitness measure. This means each client can compute a partial fitness over its subset of the testcases and return this to the root machine to be combined with the others. This avoids the need to transmit the full set of evaluation results back to the root in each generation.

The system was tested on an intrusion detection problem and an image filter problem that the authors had investigated in previous research [29]. These two problems have huge data sets: $4,898,431$ testcases and $10,023,300$ testcases respectively. The form of GP used in the experiments was CGP.

The authors give figures for "overall total performance" and "peak performance" in terms of the evaluation rate. Presumably the former is averaged over clients and the latter takes the results from the most efficient client but these measures' meanings are not explicitly described. In particular, it is not clear whether each is measured over the full run or over the evaluation stage only. For the network intrusion problem, the system achieved an average rate of 2250 Mgpop/s and a peak of 3440 Mgpop/s, both for a population size of 2048 and a CGP graph length of 2048. For the image filter problem, the system achieved an average rate of 7060 Mgpop/s and a peak of 10560 Mgpop/s, again both for a population size of 2048 and a CGP graph length of 2048.

The paper highlights interpreter divergence as a weakness of population-parallel approaches, stating "If there are 4 functions in the function set, we can expect that on average at least 3/4 of the shader processors are 'idle'. If the function set is larger, then more shaders will be 'idle' ". This may be an unduly pessimistic view of population-parallel methods since the system described in Chapter 4 (and in a 2009 paper [49] as discussed in Section 2.1.8) and the system designed by Robilliard et al (and described in a 2008 paper [76] and a 2009 paper [77]) both organise the work to ensure that each thread in a CUDA warp works simultaneously on the same program. There may be functions that require different behaviour depending on its inputs but this is nowhere near as inefficient as suggested.

This system performs the evolution in a single population and each stage of each generation is completed before any part of the next stage is begun. This is a trade-off: it allows the system to perform a pure single-population GP run but limits each stage to be as slow as the slowest client and prevents the system from simultaneously compiling kernels on some machines and GPU evaluating kernels on others.

2.1.10 The Use of Demes

The system described in this work (in particular as described in Chapter 5) uses the technique of dividing the population into subgroups and then implementing occasional transfers between the groups. These subgroups are known as *demes* (or sometimes the approach is called the *island model* and the demes are called *islands*). A single population which is not divided in this way is known as *panmictic*.

Section 5.4 examines the effect of deme transfers on run-time and investigates ways to mitigate their additional computational costs. However it is also important to consider the effect that splitting the population up has on the effectiveness of the EC algorithm. Fortunately, several EC researchers have already investigated demes, some inspired by an analogy with the effects of geographical dispersal of evolving species in the natural world and some motivated by a practical requirement to exploit a parallel processing architecture. Although the benefits and drawbacks are not experimentally investigated here, it is possible to find illumination from several empirical studies available in the literature.

Fernandez et al found some evidence to suggest that in some situations, it might be beneficial to the rate of fitness improvement to split the population into demes even in the absence of interactions between them [19]. In that case, the demes can be viewed as implementing entirely separate EC runs. Hence this finding suggests that it is, at least sometimes, better to deploy computational resources to perform several runs with smaller populations rather than one run with a big population.

The authors discovered this result by performing experiments comparing the behaviour of a fixed number of individuals under different population structures such as a single deme of 2500 or 10 demes of 250. In each case, the selection used was tournament selection where the tournament size is 10% of the deme size.

As discussed in Section 7.2.4, keeping the tournament size a constant fraction of population size does not guarantee a constant selection pressure as the population size is varied. The analysis performed in Section 7.2.2 shows that on tournament sizes of 10%, a single tournament selection from a deme of 250 is 95.0878% likely to come from the best 10.8% whereas a single tournament selection from a deme of 2500 is 95.3748% likely to come from the best 1.16% of individuals. This shows that 10% tournament selection provides very much stronger selection pressure in a deme of 2500 than in a deme of 250.

These differences in selection pressure may account for the observed differences in performance so caution is advisable in accepting the conclusion that split populations are better even without transfers.

Further experiments in the same study found that there is much more benefit to be had by also using transfers between the demes. This is perhaps what is typically thought of as a deme-based system and other work for GAs and GP have found similar results.

Punch had previously reported findings that multiple populations did not help in the artificial ant problem [75]. Fernandez et al performed more repetitions of the same experiment and found that Punch's result was not representative and that the evidence suggests a statistically significant advantage for the distributed case instead [19].

This evidence suggests it is reasonable to split the population up into demes, especially if it can be used to provide greater computational efficiency as is the case here. The evidence also suggests that deme transfers are necessary to be certain of the best results.

A more recent study on the use of demes performed a theoretical analysis of their effect on the time required to solve a particular problem [48]. The authors devised a fitness function called "LOLZ" which awards points to a genome of binary digits according to the number of leading zeroes or ones in each sub-block of a given length. The function is devised such that there is a maximum number of leading zeroes per block that can earn points and points are only awarded for a block if all previous blocks are complete (i.e. all ones). This means that the blocks must be solved consecutively and that solving a block means building up leading ones, even though building up leading zeroes will initially appear to improve fitness as effectively. Thus the function maps out a fitness tree with steady, easy-to-find fitness increases but also with several deceptive, dead-end branches. This is depicted in Figure 4. Given enough deceptive branches, a panmictic population is very likely to get stuck after taking one of the deceptive branches. In contrast, a deme-based population with the right configuration is highly likely to make it past deceptive branches by maintaining genomes representing both choices for long enough. By theoretical analysis, the authors derive the results that a deme-based population is capable of finding the "global optimum in polynomial time, while panmictic populations as well as island models without migration need exponential time, with very high probability."

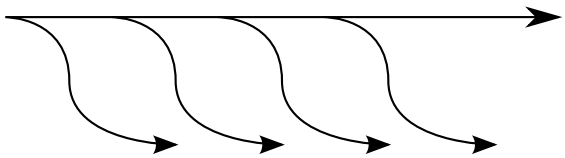


Figure 4: The fitness function described in the work by Lassig et al [48] provides easy improvements of fitness and several dead-end branches. The dead end branches initially reward fitness as much as the main path. Panmictic populations are likely to get stuck down one of the branches whereas deme-based populations (with the right parameters) are likely to sustain both choices long enough to make it past each dead-end branch.

It should be noted that this problem is engineered to possess exactly those properties with which we would intuitively expect a panmictic population to struggle relative to a deme-based population. The value of the work is that it demonstrates that this intuition is correct: deme-based populations can indeed do much better than panmictic populations on this showcase problem. However it remains unclear how widely these

properties exist in the sorts of problems to which we might wish to apply EC in practice.

There are several ways in which the use of demes can be mapped onto the GPU. An early use of demes on the GPU was performed by Garnica et al [21] in 2008. Their work implemented a GA on an nVidia FX 5950. Their system divided the population into two large demes and evolved one on the CPU and the other on the GPU. That approach involves making the CPU and GPU perform an identical set of tasks on their respective demes. An advantage of this approach is that the consistent behaviour on the two processors may well encourage rigorous testing that the two are both performing correctly. However this approach does not seem to be using the CPU-GPU architecture most effectively. The GPU and the CPU have different strengths and weaknesses so it seems more appropriate to divide up tasks accordingly. The GPU is good at performing parallel tasks very fast so it seems sensible to assign all of the time-consuming evaluation to the GPU rather than giving some to the CPU. On the other hand, writing and maintaining code for the GPU is laborious and intricate so it seems sensible to implement the less intensive tasks (selection, crossover and mutation) on the CPU only.

As the GPU performs work significantly quicker than the CPU, the authors allowed the GPU to continue performing more generations than the CPU until the GPU requested a transfer. This is a shame since it involves modifying the algorithm being implemented and the asynchronous approach will likely make reproducibility hard. An advantage of such an approach is that it facilitates keeping both the CPU and GPU well utilised. The best speedup achieved was 6.54 times faster than a CPU implementation.

A paper describing some of the work of this thesis was published in 2009 [49]. The paper covered the work described in Section 5.2 to exploit the parallelism between the two chips by having the GPU evaluate one deme whilst the CPU prepares another. It also covered the work described in Section 5.3 to use demes to keep multiple GPUs and multiple CPU cores busy. Since then, other researchers have also investigated the use of demes on GPU systems.

Pospichal et al implemented a GA with demes on the GPU using CUDA [74]. They implemented almost all of the algorithm on the GPU, with the CPU just initialising the individuals at the start of the run. They used one thread per individual and one block per deme. Transfers occurred between the demes asynchronously. They compared results from an nVidia 8800 GTX, an nVidia GTX 260 SP216 and an nVidia GTX 280. For low settings, they find all are about as fast as each other but all are faster than the CPU; for high settings, they find the best is around two times faster than the worst at around 7437 times as fast as the CPU (presumably using a single core, though this is not specified).

Again, this asynchronous approach to deme transfers modifies the EC algorithm and presumably makes reproducibility extremely challenging. The asynchronous trans-

fers occur at points defined by the intricacies of the hardware so the algorithm is changed. Asynchronous transfers between thread blocks suffer a further problem in that it may be risky. This is because the CUDA documentation indicates that client code should not make any assumptions about the ordering and parallelism of the execution of blocks in a grid launch. This means that although the code may work in practice for the setups investigated, there is potential for the behaviour to vary wildly. If nVidia choose to schedule the blocks sequentially in future CUDA releases, the result would not be an island model so much as a sequence of runs with some potential for the results of some to seed others.

Further, the paper does not describe efforts to avoid possible conflicts between reads and writes to global memory from simultaneously executing blocks. The paper may have just omitted the description of this part of the system. However, if no such efforts have been made, this could cause trouble if, for example, one block is writing the details of an individual to be transferred whilst another is reading. It is difficult to see how this could be avoided as CUDA is not designed for communication between blocks in a single launch.

An alternative approach was explored by Tsutsui et al in their implementation of a parallel GA to solve the Quadratic Assignment Problem (QAP) using CUDA [89]. Their system ran each deme on a separate multiprocessor without CPU intervention for multiple generations but then transferred all data to the CPU for the transfers to be performed. Their system was tested on an nVidia 285 GTX and the speedups ranged from 3 to 12 times.

Luong et al performed a comparison between different methods for implementing demes for an Evolutionary Algorithm (EA) on the GPU using CUDA [53]. They compared three setups: using the GPU for evaluation and the CPU for the other steps; using the GPU for all steps; and using the GPU for all steps with the benefit of the GPU's shared memory. In particular, the latter two steps involve using the blocks of threads as demes and implementing the deme transfer on the GPU itself. An advantage of this is that the parallelism of the GPU can be brought to bear on the entire EA and overheads of switching data and tasks between chips can be minimised. A disadvantage is that exploiting the thread blocks in this way imposes constraints. Examples of such constraints include difficulties in allowing differences between the configurations of the demes, limits on the number of individuals per deme (to 512 or 1024 depending on the architecture) and, in the case of the third setup which uses the shared memory, limits on the sizes of the individuals.

In the asynchronous mode of the second and third setups, the GPU code transfers individuals between demes as and when it reaches the appropriate point. This system presumably suffers the same problems as those described above for the work of Pospichal et al [74]. The authors also implemented a synchronous mode which wraps each generation in one kernel launch so that all demes complete a generation before any

deme begins the next. Another potential issue with these approaches is that the CPU sits idle whilst the GPU is evaluating and so the computational power is not maximally utilised.

The experiments were performed using an Intel Xeon with 8 cores and an nVidia GTX 280. The test problem used was the Weierstrass-Mandelbrot function. The results indicated that all setups were considerably faster than a single core CPU implementation and that each of the progressive steps (moving all computation to the GPU and then utilising shared memory) improved the speed further. The synchronous mode was found to be a little slower than the asynchronous mode. The most impressive speedup seen over a single core CPU implementation was 2074.

In the literature survey, the authors describe the configuration used in this thesis (and as described in the 2009 paper mentioned above [49]) and the configuration used in the work of Garnica et al [21] as: "CPU and GPU simultaneously evaluate one separate local population with basic two-directional exchange mechanisms". In fact, although this accurately describes the configuration used by Garnica et al, in this thesis and in the related 2009 paper [49], none of the configurations that use the GPU, also use the CPU for evaluation. Instead, the CPU is used for the other steps of the GP algorithm. In discussing the architecture, the authors state that "due to many data transfers between the CPU and the GPU and a non-optimal task distribution, the performance of such approach might be limited". The issue of optimal task distribution is relevant to the architecture described in this thesis but much less so than it would be if the evaluation tasks were divided between the GPU and CPU in the way implied. Furthermore, the 2009 paper describing the architecture [49] described the steps to mitigate the problems of data transfers and explained that the steps were rather superfluous because so little time was spent on data transfers. This is covered further in Section 5.2.

The results of their study suggest that data transfers are a considerable limitation, so how can it be that they were found to be such a minor problem here? The explanation may be that the architecture used in this work deals in cyclic GP which is considerably more computationally intensive than the typical EA so it may be that the data transfers are considerable only by comparison to an associated computation that is not so intensive.

Demes are not the only approach to localising evolution; a Cellular Evolutionary Algorithm (CEA) involves placing the individuals of the population on the points of a grid and the selection process occurs within local neighbourhoods on the grid. A CEA is like a deme-based system in which the demes are of size one and the deme transfers involve a selection process between several neighbouring demes.

PUGACE is a framework for implementing CEAs on GPUs [81]. The authors of PUGACE have attempted to define separate module files which the user can provide although they concede that complete separation was not possible due to technical constraints. The framework was tested on the QAP using an nVidia 9800 GTX. The best

speedup seen over the CPU implementation was a factor of 18.53.

2.1.11 Uses of Related Technologies

Other GPU-like technology has also been used to accelerate EC. Wilson and Banzhaf have investigated using the "XNA's Not Acronymed" (XNA) framework to implement Linear Genetic Programming (LGP) on the PC, the XBox 360 (a video game console) and the Zune (a portable media device) [98] [101] [99]. For both the PC and the XBox 360, their framework allowed them to use either the CPU of the machine or the GPU. They used a linear form of GP which used the four standard operands ("ADD, SUB, MUL or DIV" [99]) and four registers. Flow control (the use of conditional statements) was not implemented as the authors observed that it was not required for the regression problem being attempted.

Although the XNA framework aims to provide a common framework to access the hardware of these various platforms, the authors found non-trivial variations were required in their code. The mutation was implemented using textures (one with potential mutations and one with a mask to determine what is mutated) to allow the mutation to be performed on the GPU. The problems attempted were a UCI Ecoli classification problem and symbolic regression of a sextic formula. The GPU used on the PC was a GeForce 8800 GTX.

The framework's highest observed computation rate was 1.695 Mgpop/s on the PC CPU, 19.074 Mgpop/s on the PC GPU, 0.158 Mgpop/s on the XBox 360 CPU and 0.533 Mgpop/s on the XBox 360 GPU. The greatest speedup over the PC CPU was 11.254 times which was achieved by the PC GPU. The best Zune run took around 3.924 times longer than the equivalent XBox 360 CPU run and the figures suggest a computation rate of 0.040 Mgpop/s.

An attempt has been made to implement an Estimation of Distribution Algorithm (EDA) (a type of EA) on the Cell Broadband Engine of a Playstation 3 (a video game console) [68]. The authors expressed disappointment at the best speedup achieved, which was between five and six times. Attempts to exploit the technology in other areas of scientific computing have had more success. For instance one investigation used the Playstation 3's Cell Broadband Engine to implement the Smith-Waterman algorithm [102], an algorithm widely used for sequence comparison in bioinformatics. They found the peak performance to be 3 times that of a CUDA implementation of the same algorithm running on a GeForce 8800 GTX.

2.1.12 Applications

It is not clear how widely useful it is to be able to perform GP on huge data sets. Some studies have demonstrated possible uses of this capability by finding problems with suitably large data sets.

Langdon et al [46] used a GPU accelerated form of GP on a problem of finding patterns associated with breast cancer on GeneChips. With a population-parallel framework using RapidMind and a GeForce 8800 GTX, the authors were able to perform runs with $5,000,000$ individuals over a huge amount of data. An interesting aspect of this work is that size of the data set arises from the number of features ($1,013,888$) rather than the number of testcases (251). To deal with this many features, the authors performed runs in multiple passes, selecting the most successful terminals from each pass to be used in the next. The system achieved more than 535 Mgpop/s which was 7.59 times faster than the CPU implementation.

Harding used a framework for performing CGP using Accelerator (as described in various papers [28] [32] discussed above) to tackle an image processing problem [27]. Each individual was evaluated on the pixels of a large grayscale image. For each pixel, the inputs were the values of the nine pixels in the three by three grid that had the pixel in question at the centre. The input image was the same as the target image but with 5% of the pixels damaged with noise. The fitness of an individual was the average error per pixel between the individual's output and the equivalent pixel in the target image. In other words: individuals were assessed on their ability to remove noise to reconstruct the original image.

Two images were used: one of 512×512 and another of 1024×1024. Each image was a tiled grid of 256×256 pixel images. A mask was used to ensure that evaluations of pixels at the edge of one image were not affected by pixels in the neighbours. The runs were allowed to proceed for 50000 evaluations.

The paper quotes the performance results in FLOPS. The performance using a 7300 GTX on the 512×512 image was 300 mega FLOPS; on the 1024×1024 image it was 620 mega FLOPS. Using a CPU-bound reference driver for the Accelerator implementation, an average performance of 1.82 mega FLOPS was observed for the larger image.

Harding et al extended this work to evolve individuals which duplicate the behaviour of various graphic filters available in the GNU Image Manipulation Program (GIMP) [29]. This work used a more powerful graphics card (a 8800 GTX), tackled a wider range of filters and introduced a separation between training data and validation data.

This time the results are quoted in Mgpop/s. The authors state "it is unclear of the relationship of this figure to Floating Point Operations Per Second". All images in this work are 1024×1024 pixels. The performance obtained was approximately 145 Mgpop/s and the peak performance was 324 Mgpop/s. The authors explain that "the processing rate is dependent on the length of the evolved programs". The CPU-bound reference driver achieved 1.2 Mgpop/s.

2.1.13 Possible Future Directions

The benefits of using the GPU to accelerate EC have become very clear. Unfortunately, the work required to incorporate GPU evaluation into an EC system is non-trivial, even for those following in the footsteps of the field's pioneers. Perhaps many of those using the GPU for EC in coming years will do so through frameworks provided by others. The work of Robilliard et al to incorporate GPU evaluation into the standard ECJ framework [76] [77] has been discussed above. The work by Soca et al on the PUGACE framework for implementing CEAs [81] has also been mentioned.

Work by Maitre et al has also moved in this direction by providing a GPU interface for EAsy Specification for Evolutionary Algorithms (EASEA, pronounced [i:zi:]) [54]. EASEA is a language that grew out of a collaborative project and aimed to help non-expert programmers try out evolutionary algorithms. The language lets users specify an evolutionary algorithm using as little code as possible to specify the problem. The work by Maitre et al expanded this by providing a new `-cuda` option for the EASEA compiler, which specifies that the evaluation should be performed on a CUDA-capable GPU. The authors tested their system on a Weierstrass benchmark problem using an nVidia 8800 and observed a speedup of 33.3 times. Using a borrowed nVidia 260, they observed a speedup of "about 105" times.

The authors wished to see how their platform fared with a non-expert so they worked with a chemist with little programming experience. The first attempt caused a crash but this was fixed by removing around 19,600 lines of largely superfluous library code from the 20,000 or so lines of code in the chemist's evaluation function. The second attempt did not crash but ran slower than a CPU implementation. This was addressed by linearising a data structure containing four pointers to arrays of three floating point numbers into one array of 12 floating point numbers. The third attempt provided a speedup of "nearly $\times 60$".

Later work published in 2010 by a subset of these authors investigated implementing a population-parallel GP system on the GPU [55]. As with previous population-parallel systems such as those described in Section 2.1.8, the individuals were represented with RPN. The system was tested using one of the two GPUs on an nVidia 295 and speedups of "around $\times 250$" were observed.

In their conclusions the authors stated: "However, this work is to our knowledge the first to focus on hardware scheduling of GPGPU cards in order to efficiently evaluate different individuals with as few as 32 fitness cases ". Earlier, they state: "Threads within a bundle execute the same GP individual on different fitness cases, but different bundles can evaluate different individuals. This technique allows to make improved use of the underlying hardware for as few as 32 fitness cases, which is the main contribution of this work[...]". It is not entirely clear what these two statements are saying. The work of Robilliard et al published in 2008 described dividing the work such that neighbouring threads evaluate the same individual on the different testcases [76]. Sim-

ilarly the 2009 paper covering some of this work described the adoption of the same technique [49].

The authors state their intentions to "make parallel GP programming over CUDA available in a language such as EASEA, making GPGPU-based GP (GPGPGPU?) available to all researchers who would be interested in trying them out without needing to program the cards themselves." Perhaps this is the sort of endeavour through which most GP researchers will be accessing the GPU's power in coming years.

2.2 Assembly and Machine Code

The main disadvantage of data-parallel techniques is the overhead of time spent compiling GPU kernels. Chapter 6 discusses attempts to tackle this problem. One of the approaches involves coding individuals in a lower-level, assembly-like language to reduce the workload on the compiler. This relates to two lineages of previous research: research on using GPUs for EC evaluation as discussed in Section 2.1.5 (and onwards) and the use of CPU assembly or even machine code to encode individuals. In the latter case—as in the former—researchers have been motivated by wanting to feed the computational hunger of GP with fast fitness evaluations. In contrast to the situation with the GPU, interpreting methods on the CPU are perhaps simpler than compiling methods.

CPU implementations of GP adopt one of three approaches to evaluating individuals: interpreting them, dynamically compiling them or directly encoding them in machine code. The last option is probably the most technically daunting and yet—perhaps surprisingly—was substantially investigated whilst GP was relatively young. Nordin and his collaborators were responsible for much of this work and the focus was a system originally called Compiling Genetic Programming System (CGPS) [65]. CGPS was later renamed to Automatic Induction of Machine code with Genetic Programming (AIMGP) to avoid the word "compiling" giving the false impression that the system dynamically compiles from source code in each generation [66]. The approach managed to incorporate such functionality as "arithmetic operators, large indexed memory, automatic decomposition into subfunctions and subroutines (ADFs), conditional constructs i.e. if-then-else, jumps, loop structures, recursion, protected functions, string and list functions". The later system was found to be 60 times faster than the interpreting system on average.

Langdon et al applied AIMGP to evolve the hand-eye coordination system to control a 60cm humanoid robot called Elvis [47]. The software architecture used three layers: a reactive layer, a model building layer and a reasoning layer. The model building layer utilised the high speed of AIMGP, stated to be around 40 times greater than that of conventional GP. The system used version 2.0 of Discipulus.

Rather than using constrained genetic operators to ensure program safety, Kuhling et al used the exception handling system of the host machine to provide the required

protection [43]. For those that prefer to prevent invalid solutions by assigning them poor fitnesses rather than by forbidding evolution from constructing them, this is presumably an efficient way of doing it because the exception handling system is designed for trapping such problems at a low level.

Squillero's motivation for evolving machine code programs in his μGP system [82] was not to make the GP faster but to use it to generate tests for the processor on which it runs. A μGP individual can be executed directly on the target processor or can be tested on a simulation of the processor and assessed for characteristics such as instruction coverage. The GP is used to develop test programs with suitable properties for effectively testing processors.

More recently, Siebel et al encoded the neural networks that they were evolving into machine code [80]. One of the nice features of their approach was to represent the weights of the neural network in an external data structure so they could be modified by the evolutionary process without having to recompute the machine code. Consequently, they found the time spent on compilation at the start of the run to be a negligible part of run time when many generations were evolved. They found that their technique performed around 5-10 times faster than a standard interpreted approach.

Two investigations into evolving GPU shaders [17] [56] provide a link between the two research lineages. Shaders are the programs that the GPU uses for rendering. In both cases, the fitness was provided through interactive user selection of objects, dynamically rendered by the GPU with the use of the shader. In one of these cases, the language used was quite low-level [56], in the other it was the high-level C-like language of nVidia's Cg framework [17].

2.3 Tournament Selection

Chapter 7 describes two pieces of work to optimise CPU-side computation, one of which investigated tournament selection. This involved a mathematical analysis of tournament selection that meant it could be implemented without requiring as many random number generations. The analysis also gained new insight into tournament selection. Hence it is appropriate to look at this work's context in the literature.

A selection scheme is a method to select which individuals are used to build new generations. Selection schemes are discussed in more detail in Section 7.1 but it is worth outlining the basic principles of tournament selection here to provide context for the discussion of related literature.

One tournament selection picks one individual from a population by identifying a group of individuals to compete in a tournament and then picking the member with the highest fitness as the winner. Larger tournament sizes lead to a higher probability that the best individuals will be selected. Here, the population's size will be denoted N and the tournament size m. Repeated tournament selections may be used to fill the next generation and additional tournament selections may be used to choose sec-

ond parents for crossover. When choosing individuals to compete in tournaments, a tournament selection scheme must choose whether to select individuals from the population without replacement or with replacement. In the former case, an individual may only be entered into each tournament once; in the latter case, multiple copies of an individual may be entered into a tournament.

Outside literature specifically analysing selection schemes, tournament selection is usually described without specific mention of whether the selection is performed with or without replacement [6] [73]. In that context of describing tournament selection, it makes sense to skim over such minor implementation details; in the context of this work, the distinction matters because contributions are made to the understanding of without-replacement tournament selection and its relationship to with-replacement tournament selection. Furthermore, it will be argued that this property should be mentioned whenever reporting experiments using tournament selection. The literature contains several studies examining tournament selection, and apparently all of these restrict themselves to with-replacement tournament selection, perhaps because it is considerably simpler to analyse. In this work, the code targeted for optimisation was initially using without-replacement tournament selection so this was used for the analysis and the optimisation work. This is an achievement since the without-replacement analysis is considerably harder as is seen in Section 7.2.2.

The literature contains several measures of selection pressure, outlined below. These are of interest and will be highlighted because Section 7.2.7 proposes a new measure of selection pressure, called the many-from-few measure.

Goldberg and Deb performed an important comparative analysis of many selection schemes [25]. As with several other studies, it considered selection schemes in the context of GAs rather than GP but since the type of entity being evolved in both cases is typically completely independent of the selection scheme, this may be ignored. Goldberg and Deb looked at proportionate selection, ranking selection, with-replacement tournament selection, and Genitor (a "steady-state" selection scheme in which new individuals are created one at a time rather than in generations [97]). For each, they assessed the time complexity of the algorithm with respect to the population size and the *takeover time*. The "takeover time" is the expected number of generations it takes for the best individual to fill the population. They found the takeover time of with-replacement tournament selection is $\frac{1}{\ln m}[\ln(N) + \ln(\ln(N))]$ and its complexity is $O(N)$.

Bäck investigated proportional selection, linear ranking, with-replacement tournament selection and (μ, λ) selection [3]. He found them to increase in selection pressure in that order. For with-replacement tournament selection, he derived the selection probability of the i^{th} best individual as $N^{-m}((N-i+1)^m - (N-i)^m)$. He performed an experiment to find the best fitness of an evolutionary run on a simple problem using the various selection configurations.

Later work from Bäck extended this to derive *selection intensity* and to carry out two

more empirical investigations [4]. Selection intensity is a biological concept that was imported into EC [61] [62] and is defined as the expected average fitness value after selection of a population with fitnesses from the normalised Gaussian distribution.

Equivalent results were found in two papers by Blickle and Thiele from the same year [10] [11]. One of those papers [10] derives other results for with-replacement tournament selection. Of particular note are the *reproduction rate*, "the ratio of the number of individuals with a certain fitness value f after and before selection", and the *loss of diversity*, "the proportion of individuals of a population that is not selected during the selection phase". The other paper [11] expands this analysis to also cover truncation selection, ranking selection and exponential ranking selection.

Motoki studied loss of diversity in with-replacement tournament selection in more detail [60] and, in particular, examined it over varying population size. The paper stated "from numerical results, we observe that in tournament selection, many more individuals are expected to be lost than with Blickle and Thiele's static estimate."

Perhaps the most relevant work is a more recent study into with-replacement tournament selection by Xie et al [106]. They plotted the selection probabilities in various ways to provide a visualisation of tournament selection. They then considered the effect of some programs having the same fitness values and visualised this. They proposed a variation of tournament selection that treats such groups of individuals with equal fitness as one cluster. They found that this "clustering tournament selection" increases the selection probabilities of the middle fitness individuals at the expense of the selection probabilities of the best individuals.

Enhancing the "backward chaining" of Poli and Langdon [71] [72], Xie et al suggested another scheme that avoids evaluations for individuals that are not selected in any of the tournaments [106]. The work described in Chapter 7 does not permit this as it requires all the fitnesses to be present so that the population can be sorted.

This is not seen to be a problem since the notion of saving significant computation this way seems questionable. Such savings would require that, with considerable regularity, there are a considerable number of individuals that never get a chance to be selected, even though the population best might be among them. Rather than being an opportunity for optimisation, this would seem to suggest that there is a problem with the selection scheme configuration.

Xie et al noted in their conclusion that the selection probabilities are not dependent on the population size.

2.3.1 Measures of Strength of Selection Pressure

Several possible measures of selection pressure have been mentioned and they are summarised in Table 3. Of these, the reproduction rate does not provide a single number but gives values for members of the population for a given configuration. This is useful but does not fulfil the role of giving a single value indicating selection pressure.

Selection intensity delivers a single number but relies on the assumption that the population having a Gaussian distribution of fitnesses. Takeover time and loss of diversity are more intuitive measures. Still, neither of these quite captures the notion of selection pressure, i.e. how much does the selection scheme favour the fittest individuals. In Section 7.2.7, a new measure of selection pressure strength, the many-from-few measure, is proposed in an attempt to address these issues.

Name	Primary author	Year	Description and notes
Takeover time	Goldberg [25]	1991	The number of generations for the best individual to take over the entire population
Selection Intensity	Mülenbein [61] [62]	1993	"The expected average fitness value of the population after applying the selection method Ω to the normalised Gaussian distribution G(0,1)[...]" [10]. This concept is drawn from biology.
Loss of diversity	Blickle [10]	1995	"The proportion of individuals of a population that is not selected during the selection phase" [10]
Reproduction rate	Blickle [10]	1995	"The ratio of the number of individuals with a certain fitness value f after and before selection" [10]

Table 3: A summary of selection pressure measures from the literature.

2.3.2 Tournament Selection With or Without Replacement

The work in Chapter 7 makes a novel contribution by analysing without-replacement tournament selection. To make this contribution clear, this review has emphasised the fact that analytical literature appears to be restricted to with-replacement tournament selection. Although the analytical literature apparently ignores without-replacement, there is no apparent reason to think it is used less widely. Indeed there are reasons to suggest it might be preferable:

- It seems more intuitive that if a tournament of individuals is selected to compete from a population, then the tournament should not include duplicates.
- Without-replacement has the nice feature of scaling from no selection pressure with tournament size one to completely deterministic selection of the fittest individual with tournament size N.
- There is a sense of wasting random numbers using with-replacement because a random number might be spent adding an individual to a tournament that has already been added by a previous random number. This matters, since generating high-quality random numbers typically requires non-trivial amounts of computation time.

With respect to the last point, random number generation might be viewed as the price one has to pay to achieve higher selection pressure. Under this view, without-

replacement represents better value. This is particularly relevant to this work as very high selection pressure is used.

An example, generated using the tools derived in Chapter 7, helps illustrate the point. To achieve the same selection pressure as a without-replacement tournament of size 99 in a population of 100, a with-replacement would need a tournament size of 459. Using standard implementations, this would require 4.59 as many random number generations. This selection pressure is strong but perhaps not so strong as to make the example ridiculous: it is approximately the equivalent of without-replacement tournament selection using a tournament size of 8.76% of a population of 5000.

The only advantages of with-replacement tournament selection that come to mind are:

- It is easier to analyse and
- It might be slightly easier to implement.

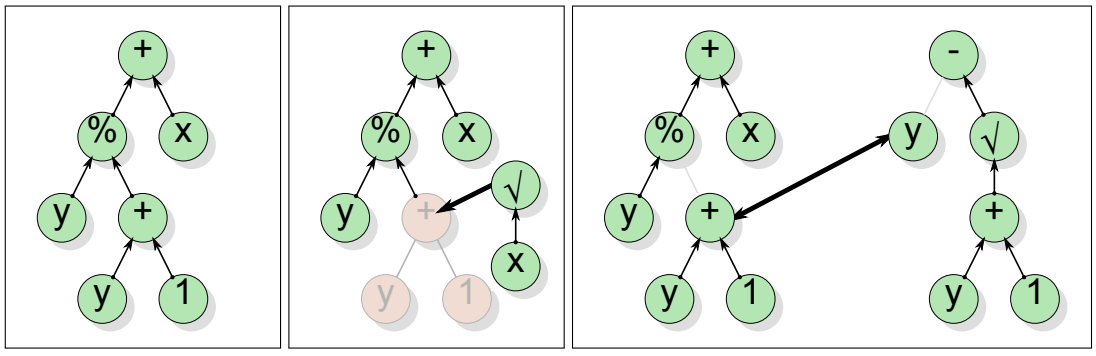
The latter point is minor because most modern programming languages provide tools that make without-replacement sampling trivial to implement (such as `random_sample()`, provided with most modern C++ compilers).

3 Methods

3.1 Genetic Programming (GP) Representation

3.1.1 Nodes and Instructions

Attempting to evolve a program presents considerable challenges. Not least of these is finding a good way to represent programs so that they are susceptible to evolution. The standard approach to this issue is to use the tree representation. An illustrative example of a tree-based Genetic Programming (GP) individual is shown in Subfigure 5(a). One of the advantages of this representation is that it naturally suggests methods of implementing mutation and crossover as indicated in Subfigure 5(b) and Subfigure 5(c) respectively. The representations of GP are often categorised into tree-based, graph-based and linear (as illustrated in Figure 6(a)). Graph-based representations are those representations that are depicted by graphs not constrained to be trees. Graph-based representations may or may not constrain individuals to be acyclic.

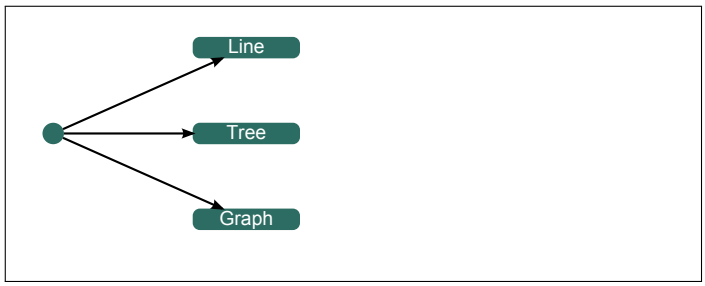


(a) Tree evaluation (b) Tree mutation (c) Tree crossover

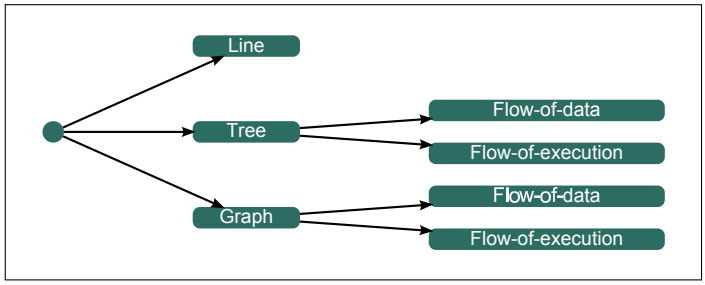
Figure 5: The tree representation naturally suggests methods for evaluation (5(a)), mutation (5(b)) and crossover (5(c)). Subfigure 5(a) depicts a tree that evaluates to $(y/(y+1))+x$. Subfigure 5(b) depicts a natural method for mutating trees, in which a randomly selected subtree is randomly regenerated; here, $y+1$ is replaced by $\sqrt{x}$. Subfigure 5(c) depicts a natural method for performing crossover, in which randomly selected subtrees are exchanged; here, one tree's $y+1$ is exchanged for another tree's y.

Perhaps this tree/graph/linear approach to classifying GP representations might be improved. It has previously been observed that a further distinction can be made between several of the non-linear representations based on what the diagram used to depict the individual represents [86]. In some cases, the diagram represents the flow of data from one computing node to another (as in Figure 5(a)) whereas in others, it represents the flow of execution from one node to another (as in Figure 7(a)). For instance in representations such as the standard tree-based representation, each point on the graph represents a node, which sends the output of its calculation as input to the next nodes on the graph. There are other graph-based representations (such as linear-tree [38], linear-graph [39], Parallel Algorithm Discovery and Orchestration (PADO) [88], Genetic Network Programming (GNP) [40] and GRAph structured Program Evolution

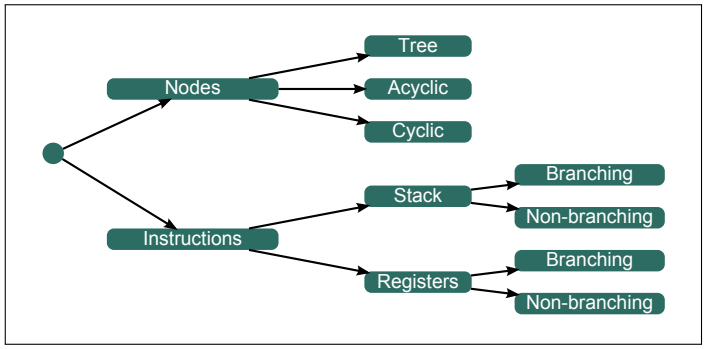
(GRAPE) [79]) for which the graph represents a network of instructions to be traversed whilst carrying some memory structure on which the instructions may operate. Hence the classification might be divided into "flow-of-data" representations and "flow-of-execution" representations [86]. This first refinement to the classification is shown in the first two subfigures of Figure 6.



(a) The standard classification



(b) A refinement to distinguish flow-of-data/flow-of-execution

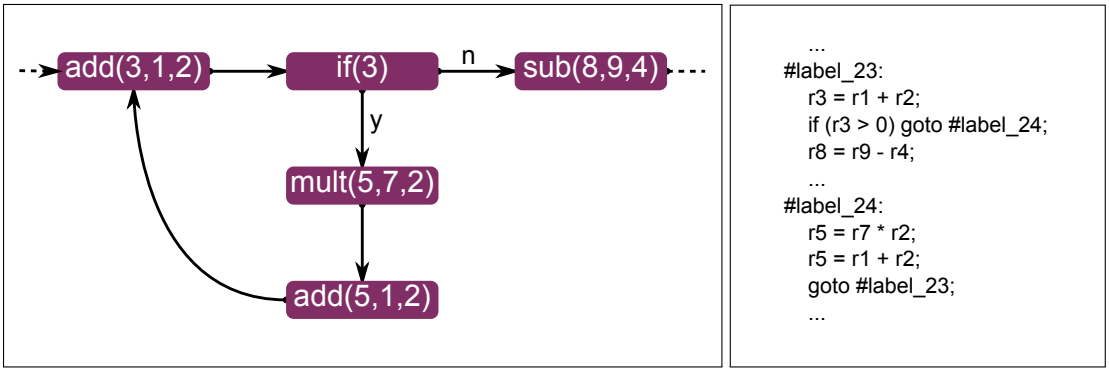


(c) A different approach

Figure 6: Steps to attempt improving the standard classification of GP representations. Subfigure 6(a) shows the standard approach of splitting representations by their layout: line, tree or graph. In Subfigure 6(b), this is refined to distinguish between flow-of-data and flow-of-execution within tree-based and graph-based representations. However, this still fails to keep Figure 7's individuals in the same classification. Subfigure 6(c) shows a different approach in which the primary distinction is made on whether the representation is node-based or instruction-based. Node-based representations are further divided according to their layout (tree, acyclic or cyclic) and instruction-based representations are further divided according to whether they are use a stack or registers and then according to whether they are branching or not.

In fact, it might be possible to improve the traditional classification further. What properties should a good classification have? It should help distinguish between items

based on their objective properties, placing items in different categories if and only if their objective properties are different. Yet under this stipulation, the graph/tree/linear classification scheme fails, even with the flow-of-data/flow-of-execution refinement, because it separates out some functionally equivalent representations because of the way in which they are depicted. In other words, it distinguishes representations based on subjective criteria and consequently places representations with the same objective properties in different categories. To see this, consider the two examples illustrated below in Figure 7. The first shows a graph-based representation and the second shows a linear representation; yet these two individuals are the same and have functionally identical execution. They are merely depicted in different styles.



(a) Graph-based representation (b) Linear representation

Figure 7: Two functionally equivalent individuals that are classified separately because they are depicted differently. The individual in Figure 7(a) is laid out as a graph whereas the individual in Figure 7(b) is linear. Ideally, a classification should place these representations together.

Since there is no functional difference between these two examples, they should be classified together. Nevertheless a classification is still useful because there remain real differences between the way, say, a tree is evaluated and the way the two examples in Figure 7 are evaluated. To refine the classification, then, it is necessary to describe this difference in objective terms regarding their functional evaluation rather than in terms of their depiction.

To achieve this, it helps to think of how these representations utilise the two basic ingredients that must make up any computation: instructions and memory. The following classification is proposed: representations like tree-based GP should be distinguished by the way they bind instructions and memory slots together in pairs, which we call nodes. In trees (such as the one depicted in Figure 5(a)), the nodes are normally thought of as indivisible units but it helps to consider the instruction and memory slot under the node's bonnet. In this light, a node may be interpreted as meaning "perform this node's single instruction on its inputs and store the result in a single piece of memory within the node to be made available as an input to other nodes". Under this scheme, each instruction writes its output to precisely one piece of memory,

to which it has exclusive write access. (Of course, an efficient implementation might reuse memory locations for distinct nodes but that is immaterial here because such an implementation would be functionally equivalent.)

By contrast, other representations such as the one depicted in Figure 7 do not bind instructions and memory in this way and a slot may be written to by any number of the instructions (including none of them). Node-based systems constrain instructions together into exclusive pairs called nodes; other systems do not. It is suggested that *this* is what distinguishes tree-based GP from linear GP. The constraints induced by a node-based representation naturally lead to one way of depicting individuals; the flexibility of a node-free representation allows both graphs and lists to depict individuals effectively. The style in which individuals are depicted is not important; what matters is the presence or absence of the set of functional constraints that define nodes.

If the classification is accordingly altered to use this distinction of whether or not representations are node-based, it no longer places functionally identical representations in different categories. This change in the classification is depicted in Figure 6(c). Under the new approach, node-based representations are further divided according constraints on their to their layout (tree, acyclic or cyclic) and instruction-based representations are further divided according to whether they use a stack or registers and then according to whether they are branching or not.

3.1.2 Cartesian Genetic Programming (CGP)

The GP used in Chapter 4 is a form of Cartesian Genetic Programming (CGP). CGP is a node-based representation that drops standard GP's requirement that individuals must be trees. This allows the results of nodes to be used by multiple other nodes. The standard presentation of a CGP individual involves laying the nodes out in a two-dimensional grid (whence the "Cartesian" in the name). Each node's inputs may each be drawn from the output of any node within the last few columns of nodes. The "levels back" parameter—one of several—defines the number of columns back that a node's input may seek for the node to which it will connect. The result of this prescription is that an individual node's output may be used by zero, one or several other nodes. Figure 8 shows an example CGP individual, illustrating these points and Figure 9 presents some CGP pseudo-code.

Issues such as crossover are dealt with elegantly through the use of a genotype that describes the individual using a string of integers. This permits the representation to import all of the mechanisms, such as crossover, from the field of the Genetic Algorithm (GA). Recent CGP work has tended to use one row and a parameter setting that allows node inputs to be connected to any previous node in the row [27] [57]. CGP is a very well studied representation [14] [16] [27] [30] [31] [34] [41] [57] [58] [91] [92] [93] [94] [95] [96] [100] [105].

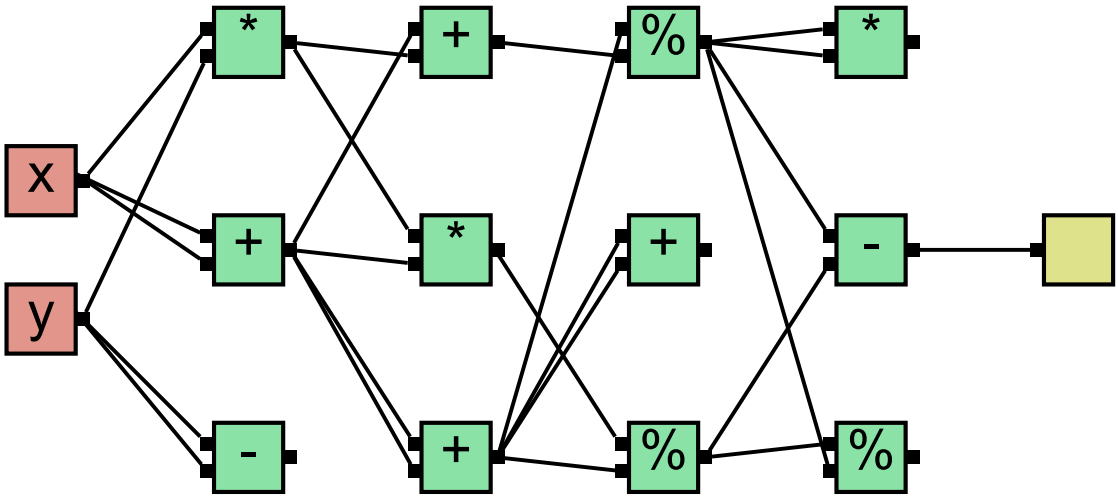


Figure 8: An example Cartesian Genetic Programming individual. The input nodes are in red, the normal nodes are in green and the output node is in yellow. This example has two inputs, one output, a layout with three rows by four columns and a "levels back" parameter of one (meaning that each node's input must be connected to a node in the previous column). Each input on each node is connected to exactly one other node, whereas a node's output may be connected to zero, one or many nodes.

3.1.3 Cyclic Genetic Programming

The form of CGP used in Chapter 4 is cyclic CGP. Cyclic GP involves allowing the connections between nodes to form cycles. In the case of standard CGP, cycles do not form because the nodes may only draw their inputs from nodes in previous columns. Cyclic CGP allows the nodes to receive their inputs not only from nodes in previous columns but also from nodes in later columns (and possibly from nodes in the same column). A "levels forward" parameter is used to specify the number of columns forward from which a node may draw its inputs.

Permitting cycles in a GP structure raises a question about how individuals are to be evaluated. For standard, tree-based GP, the order of execution of the instructions is implied by the structure and each instruction need only be executed once to derive the final answer for a single evaluation. By contrast, the evaluation for cyclic GP is iterated, with successive iterations performing new computations based on the values of the previous iteration. Much of this thesis deals with iterated forms such as cyclic GP (Chapter 4) and Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble") (Chapter 6). Hence the runs described in this thesis typically involve multiple generations, iterations, sub-populations (or *demes*), individuals, instructions (or nodes) and testcases.

3.2 Tweaking Mutation Behaviour Learning (TMBL)

The form of Evolutionary Computation (EC) used in Chapter 6 is TMBL, a form of EC that has been developed as part of this work's investigation into long term fitness

```
sub cartesian_evaluation(individual, testcase) {
  nodes = individual.get_nodes();
  node_values = [];
  foreach node (nodes) {
    inputs = [];
    input_node_indices = node.get_node_indices_of_inputs();
    foreach input_node_index (input_node_indices) {
      push inputs, node_values[input_node_index];
    }
    push node_values, node.perform_node_operation(inputs);
  }
  return individual.get_output_node_value();
}
```

Figure 9: Pseudo-code illustrating the principles of Cartesian Genetic Programming

growth. TMBL has been proposed as a sister to GP in research conducted as part of this PhD research and published in a 2010 conference paper [50]. Like GP, it entails evolving programs; unlike GP, it prioritises the long term growth of fitness above all else. This may be at the expense of efficiency in the initial generations if necessary. It is built on the following hypothesis: *long term fitness growth is dependent on the ease with which mutations can affect an individual's behaviour without (necessarily) ruining its existing functionality*. Such changes are known as *tweaks*, a term coined as part of this PhD research and in a 2010 conference paper [50].

Of course, the quest to develop long term fitness growth is superfluous for problems that are simple enough to be solved before GP's stagnation sets in.

Before developing the motivation for TMBL, it is worth outlining how its view fits with the dominant perspective for EC argumentation: the fitness landscape.

The fitness landscapes of forms like GP and TMBL can be difficult to ponder because they can have very unintuitive topologies due to genetic operators that have some non-zero probability of mutating any individual into any other. This leads to all points of the fitness landscape being connected to each other, which complicates the standard views regarding local/global optima and requires notions of distance to be probabilistic. Further, given a non-zero probability of any individual being mutated into any other, the probability of finding the global optimum within n generations tends to 1 as n tends to infinity. An initial, naive reading of this might be that the answer to all problems is simply to increase the number of generations. However we know that in most non-trivial problems, the theoretical possibility of mutating directly to the global optimum is not of much practical use.

Despite these complications, the fitness landscape abstraction remains useful. Much discussion of fitness landscapes focuses on how getting trapped in local optima prevents reaching the global optimum. Certainly, successful EC must incorporate the exploration required to escape small-scale local optima. However it is not clear that the

correct diagnosis of what prevents GP from the sort of long term fitness growth seen in biology is that it is not able to escape sufficiently large local optima. The guidance should come from examining why biological evolution does not suffer the stagnation of GP.

True, biological evolution has been able to deploy vast population sizes and time-scales to perform vast searches of a population's neighbouring genomes. However, since populations do not have foresight, populations do not typically make long-term movements into less well adapted areas of genome space in order to move to escape a local optimum. The remarkable observation about biological evolution is not that it manages to find the globally optimal genome, but that the local optima that it achieves through its cumulative improvement are so functionally complex and so well adapted. Hence, our aim should not be to change our search strategies to ensure that we always get to the highest point of our fitness landscapes but to change our fitness landscapes to raise the larger-scale optima so that most points are on the (possibly bumpy) slopes of very high peaks.

In practice, few hills within GP fitness landscapes are very high. GP's stagnation tends to cap them at a certain height because as the individuals get fitter, they quickly become less evolvable. To understand why this is so, we must leave the abstraction of the fitness landscape and delve into the details of what is actually happening within GP individuals.

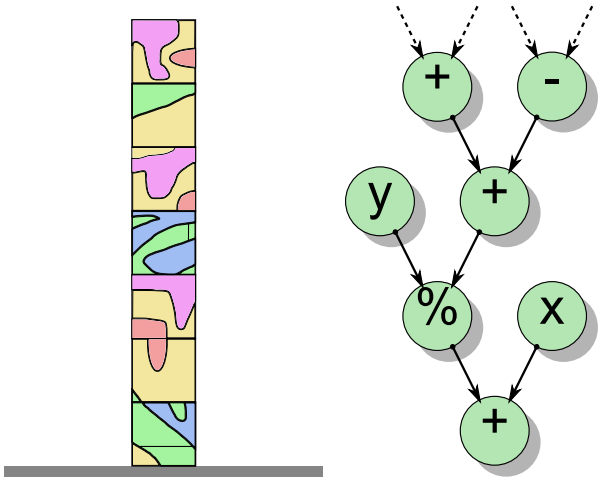
3.2.1 An Analysis of the Problem

In order to understand TMBL's approach, it is important to understand the related analysis of what currently limits long term fitness growth.

Consider a toy puzzle consisting of many cube-shaped blocks that must be lined up in some specific order according to the patterns on their surfaces. Assume that it is quite possible to stack the blocks into a tall tower without them toppling. Further assume that the puzzle is sufficiently tricky to require a good deal of trial and error but that planning is forbidden. Now imagine attempting to solve the puzzle by stacking the blocks vertically in a single, free-standing column as depicted for a small example in Figure 10(a). It is intuitively clear that this single-stack, trial-and-error approach is doomed; given a puzzle with enough blocks, the strategy will stagnate.

Why must this be so? This is because once initial progress has been made, it becomes difficult to make changes without ruining previous achievements. Once successful regions are formed in the stack, it becomes extremely difficult to adjust any blocks below without the successful region falling over. Hence as progress is made and as successful regions are formed, the cost of meddling increases for more and more of the blocks and so each next step becomes harder. There might be some easy improvements that can be made, particularly near the top of the stack, but once these get used up, the same problems remain. Eventually the attempt grinds to a halt.

Note that this strategy's attempts may consistently start well and may consistently make moderate progress. Initially, it would be easy to put one block on top of another or to substitute this block for that. This should not mislead; the strategy is limited.



(a) A small tower of blocks (b) An inverted GP tree

Figure 10: Illustrations of a small tower of blocks (a) and an inverted GP tree representing $x + (((?-?) + (?+?))\%y)$ (b). It is claimed that these two challenges face similar limitations. In both cases, progress may initially be good but it stagnates later on as initial achievements get buried beneath dependent material. It becomes ever harder to make further improvements without damaging previous results.

This analysis also illuminates the stagnation seen in GP. Compare the tower of blocks to an upside-down GP tree as depicted in Figure 10(b). In both cases, it becomes increasingly difficult to improve the structure by a process of trial and error because the more successful material that is built on top of an item, the harder it becomes to change that item without doing more harm than good. In the tower, the lower blocks provide physical support for the blocks above; in the inverted GP tree, the support is functional.

Then how are these processes able to make any progress at all? The early solutions are unremarkable so progress can be made by additions and occasional lucky random alterations. This process continues and the solution collects components built with the best luck seen so far. Consequently it becomes increasingly rare that a new randomly trialed component is better than the component it would replace, which represents the best luck seen so far in that area. The closer to the structure's core that the candidate change would occur, the more damage it is likely to do to the prior achievements and so the more exceptional good luck is needed to succeed.

These arguments emphasise the problem with building a single structure out of such highly interdependent units: as the structure becomes increasingly elaborate, it becomes increasingly difficult to modify the structure without ruining prior achievements. Imagine if biology had somehow been constrained to allow only one gene (translated to one protein chain) per organism. As evolution added ever more func-

tions to this Swiss Army knife protein, new mutations would face ever more formidable constraints to maintain the precise structural configurations required to maintain all the previous functions.

Given plentiful blocks or nodes, why can't the process just keep improving by adding at the fringes until the resources dry up? For one thing, passable components congeal at the heart of the structure and get buried until there is no practical way to improve them. Each new layer of passable component that establishes itself, adds new limitations to the material deeper down. That said, developing at the top of a tower may continue to improve its value at the same rate until the blocks are used up. For the GP tree, the outlook is worse because most components will tend to restrict the influence and scope of their neighbours further out from the core.

As with the tower, consistent moderate progress in tree-based GP should not mislead; all attention should remain on the situation after the initial progress and on overcoming the obstacles that arise then.

In reality, few players of the blocks game would persist with building a single vertical tower for long before switching to a strategy of assembling the puzzle horizontally. Once the puzzle is laid out flat, changes can easily be made without ruining previous achievements, making better results easier. This concept of a change which affects without ruining is at the heart of this work and is given the name *tweak*. For example, a sub-tree replacement mutation is not a tweak because it completely removes the previous sub-tree and thus requires the new random sub-tree to do a better job in that position than the sub-tree it replaces.

When tweaks are prevented, a candidate alteration must do a better job than the "best luck" component it would replace or damage. When tweaks are encouraged, a candidate alteration may alternatively succeed by making a new contribution to an existing component whilst still allowing it to persist and to function as before.

The aims then, are to focus on the situation later on in the run and to find a form of program evolution that is "laid out flat" to encourage tweaks.

3.2.2 A TMBL Form to Avoid Limitations

TMBL focuses on the situation later on in the evolutionary process. The path taken to that position is secondary; the primary concern is what can be done to encourage further development once there. Consider what this situation tends to look like. At this point, the population fitness has typically made substantial progress and the best individuals have a lot to lose from a bad mutation. The initial flurry of improvements has waned and few generations see the population best improve. The individual that most recently improved the population best is likely to be dominating the population through its descendants. Other lineages that do not match the fitness of this top individual disappear quickly and descendants with deleterious mutations rarely last more than a few generations.

For these reasons, there is unlikely to be much diversity in the enduring core of the population, just minor variations on one form of solution. This means there is little for a crossover operator to work with, so although it may or may not confer some additional benefit, it isn't the significant source of functional innovation. Instead, the evolutionary process must rely on mutation to provide most adaptive steps [1].

It was argued in Section 3.2.1 that to improve long term fitness growth, the focus should be on finding a structure that encourages tweaks (changes which affect behaviour without ruining existing functionality).

What sorts of properties of a program representation might encourage tweaks? Firstly, a change to one component of the program should be able to affect the behaviour of another part of the program without it being necessary to also change that other part. Compare this to the way that a newly evolved gene's product can interact with a pre-existing biological process carried out by other genes' products. This suggests that the program should be built out of actions that affect other entities rather than static components that present their results for use by another part of the program. Instead of the overall behaviour arising from a single structure, it should arise from many parts which evolve to make their own contributions. Secondly, each component should have as few ties with functionally unrelated components as possible. This suggests that the design should not force components of the program to share aspects globally.

This analysis can be used to design a standard form for TMBL. Note that this is just one of many possibilities and other researchers are encouraged to propose their alternative suggestions for achieving the aims of TMBL. The methodology used here is to review the properties that divide the various GP representations and, at each stage, use a fresh focus on tweaks to guide a choice.

3.2.3 Choosing whether to use nodes

Using nodes makes it hard to modify the behaviour of a program without damaging existing functional behaviour. In other words, using nodes hinders tweaks. This is because the changes affecting the behaviour of a node-based program will involve changes to active nodes (nodes which currently affect the output) but this involves disrupting the contribution that node and its active children were already making to the output.

Changes at the boundary between active and inactive nodes may minimise the number of useful nodes that are disrupted. Unfortunately, such changes at the fringes of the functional structure tend to have restricted influence and tend to create a new fringe with even less influence. As argued in Section 3.2.1, building at the fringes does not solve the problem.

Nodes undermine the stated aim of building programs out of actions which can

[1] Where *mutation* should be construed in the broad sense of any heritable change that does not draw material from other individuals. The broadness of this definition is discussed further in Section 3.4.

directly affect the behaviour of other parts of the program. Without nodes, it is relatively easy to change an instruction to modify some register without disrupting other instructions that are already using it. For these reasons, nodes will not be used in this representation.

3.2.4 Choosing the structure of memory

What structure of memory seems most likely to encourage tweaks? Until now, the word "register" has been used to refer to any part of an EC program's memory. From now on, it is worth being more precise because in addition to plain registers, GP systems may alternatively arrange the memory into a stack or indexed memory.

In register-based memory, each instruction is tied to specific registers that it uses for its output and inputs. Unlike when using nodes, each register may be read from or written to by multiple instructions. This means that the instructions must be placed in some order to ensure consistency and to avoid access clashes. Non–node-based GP systems more commonly use stack-based memory. This involves each instruction popping enough data off the stack for its inputs, performing its calculation and then pushing the result back onto the stack. Indexed memory involves providing read/write functions that allow a program to use a run time argument to indicate which slot of memory to access.

The stack approach seems likely to be the most brittle. As a stack-based program develops, it will become increasingly tricky for mutations to affect behaviour (in any way which involves the stack) whilst still preserving the state of the stack well enough to avoid damaging already functioning parts. A stack is too global in the sense that all separate computations in a program are forced to share the same stack.

Indexed memory is a potentially suitable approach which appears less brittle than a stack because it is relatively easy for newly mutated parts of a program to access one area of indexed memory without affecting already functioning parts of the program which access another area. However, since indexed memory is more complicated than register-based memory and requires more of its instructions, it is not included in the scope of this work.

Plain registers encourage tweaks because they make it easy for changes to one set of instructions to affect those registers being used by another set of instructions without ruining the actions of those other instructions. Furthermore, different parts of a program can easily avoid sharing resources by using separate registers.

Systems using registers often use relatively few, and an analysis of Linear Genetic Programming (LGP) found that 16 registers was suitable [12]. The consequence of this is to force growth to be vertical in the sense that programs develop their fitness by extending the list of instructions that cooperate in sequence on a small set of registers. That sort of growth is important and may be essential for developing some of the complex parts of an algorithm, but it involves building a complex network of interactions

and so makes tweaks increasingly difficult. For this reason it should be complemented by horizontal growth in which programs can develop their fitness by developing new groups of instructions and registers which make (fairly) independent contributions. This suggests that TMBL should use substantially more registers than are normally used in LGP.

3.2.5 Choosing the type of flow control

Programs that require conditional behaviour should permit some form of flow control in the representation. The most common forms of flow control in non–node-based GP use a single point of execution which flows through the program and jumps to different locations in the program depending on the result of an evaluation each time it reaches certain branch points. This approach is too global for TMBL because it requires that all components of a program must collaborate on a shared flow of execution. As programs develop complexity, it becomes increasingly hard for new parts to exploit their programs' flow control systems without damaging other parts already relying on them.

To encourage tweaks, the TMBL representation should instead use a more local system in which each instruction can determine its own execution status. This is achieved by allowing each instruction the potential to have its own if-condition test. An instruction with an active if-condition is only executed when the value at the if-socket is positive. This system can still be used to generate sophisticated behaviours by repeating the execution through the program for multiple iterations. Similar systems have been discussed for LGP that allow multiple, nested if-conditions [12].

3.2.6 Choosing the type of instructions

In a final step to encourage tweaks, the instructions are constrained to always have the target register as the first input register. Whereas instructions are normally of the form "overwrite register C with the result of adding register A and B", this constraint restricts them to the form "add register A to register B". This encourages instructions to modify the values in registers without destroying any information that they previously hold and so encourages changes that affect without ruining.

3.2.7 A Summary of TMBL's Standard Form

The resulting representation is somewhere between a linear (node-free) representation and a cyclic graph-based (node-based) representation. Figure 43 contains representative code implementing TMBL, which might help the reader get a feel for this and which might help clarify the following text. Like a linear representation, the instructions and registers are not paired together in nodes. A stack is not used as is often the case in linear representations and more registers are used than is normally the case (for

those linear representations that use them). Like a cyclic graph-based representation, the evaluation is iterated and all nodes are evaluated each iteration (except those that opt out via their if-conditions) rather than there being a single point of execution as is often the case in linear genetic programming. The instructions are constrained to be of the form "add the value in register A to the value in register B". The implementation can be summarised as follows:

- Each individual consists of an ordered list of instructions and two numbers indicating the number of registers and iterations to be used when evaluating the individual.
- Each instruction contains an if-switch, an if-socket, an input-socket, an output-socket and an operation.
- The if-switch is a Boolean value indicating whether the if-condition is to be used.
- Each of the sockets contains the index of a register or of a dimension of the test-case. The output-socket may only refer to a register (because instructions should not write to testcases).
- Before evaluation, the registers are all initialised to zero.
- In each iteration, each instruction is evaluated in turn.
- If an instruction has an active if-condition, the instruction is skipped whenever the value pointed to by the if-socket is negative.
- Executing an instruction involves reading the value pointed to by the input-socket and using the operation to apply that value to the register indicated by the output-socket.
- After the last iteration is complete, the output is taken from the last register.

In addition to the standard functions, a TMBL program has the functions SetValue and Copy. The SetValue function sets the target register to some floating point number held within the instruction (which is open to mutation). The Copy function copies the input to the output. It would be simple to modify this representation to allow for operations with arity other than two (although some thought may be required to construct these operators such that they act on a register rather than overwriting it).

Crossover could easily be applied but was not used in these experiments. The mutation operator varies each component of each instruction with some small probability and moves an instruction to some other location in the execution list with some small probability. The probabilities are set such that each individual has a 0.95 probability of having at least one mutation.

3.3 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) is a framework provided by nVidia to access an nVidia Graphics Processing Unit (GPU) for general purpose computing. The model involves writing a function for the GPU in a language like C with a few extras (and an increasing number of C++ constructs in newer versions). This function, known as a *kernel*, is compiled using the `nvcc` compiler and is then uploaded to the GPU, ready for execution. Executing the kernel involves specifying the number and layout of the threads that will execute the kernel. Threads are grouped into thread blocks, with each thread block in a launch having the same number of threads. The kernel code may access the index of the block and the thread under which it is executing. This offers a great deal of flexibility in the way a computation is divided amongst the threads. Code should make no assumptions about the order or parallelism with which the different thread blocks execute but interaction between threads within a thread block is possible with the use of the barrier synchronisation function `__syncthreads()`. This function ensures that all threads reach the call before any threads pass it. The GPU may read and write to a large quantity of slow, off-chip global memory, which is also accessible by the Central Processing Unit (CPU) and so allows data to be sent to and from the GPU. Threads within a block may also communicate with each other using a limited amount of faster, on-chip shared memory. Each thread also has a limited number of very fast registers with which to perform computations. These registers are not indexable.

Although all threads must execute the same kernel in a single launch it is possible for threads to take different paths of execution through this code. Thread blocks are grouped into groups of consecutive threads called warps, which at the time of writing always contain 32 threads. If threads within the same warp diverge from each other, this is implemented in the hardware by the entire group of 32 threads taking all paths through the code. Hence minimising such "warp divergence" is an important part of maximising execution speed.

3.3.1 PTX

The CUDA compiler, `nvcc`, compiles CUDA C into a GPU-ready binary in two stages via an intermediate, assembly-like language called Parallel Thread EXecution (PTX). PTX is well documented and is supported by nVidia. It is possible to instruct `nvcc` to retain its intermediate PTX files. By inspecting the code in these files, it is possible to see how the compiler implements any given CUDA C file in PTX.

The main drawback of programming the GPU in PTX rather than in CUDA C is that it is considerably more complex. On the other hand, directly writing PTX means that the time spent compiling from CUDA C to PTX is completely removed. For most CUDA developers, this time will be of little import because it will only be performed once; for data-parallel GP, compiling from CUDA C to PTX is performed many times

within a single run so it can waste a considerable amount of time as described in Chapter 6.

3.4 Conventions Used in the Thesis

Throughout the thesis, a few conventions are repeatedly used. Many of the graphs indicate the estimated standard error of the values of a line by using a paler bar of the same colour behind it (indicating the sample mean plus and minus one standard error). Many of the tables include the standard error of a sample within square brackets after the mean and use the $\pm$ symbol within the square brackets to indicate this.

Strictly speaking, the random number generators used in this work were not "true" random number generators (such as based on quantum events) but were pseudo-random number generators (PRNGs). However for the sake of brevity and simplicity, the distinction is ignored in this thesis and the "pseudo" is omitted.

In several places, this thesis refers to the protected division function, notated %, in place of the standard division function (notated /). The protected division function always returns some fixed constant value (usually zero) whenever the denominator is zero but acts like the standard division function otherwise. This function is often used in GP research but there is some debate regarding its merits. That debate is not addressed here because this thesis seeks to replicate runs identically but at higher speed, not to assess the effect of changes to the run on its internal behaviour.

In this thesis, the word *mutation* will be used in a broad sense to mean any change to the genetic material of an individual that does not involve the insertion of genetic material from another individual.

This is consistent with the definition of mutation widely used in biology. For instance in their undergraduate and graduate level textbook "Evolution", Barton et al state that "Mutation, formally defined as a heritable change in the genetic material (DNA or RNA) of an organism, is the ultimate source of all variation" [7]. The definitions used in GP are often similarly broad. For instance in "A Field Guide to Genetic Programming", Poli et al define mutation as "The creation of a new child program by randomly altering a randomly chosen part of a selected parent program" [73] and in "Genetic Programming: An Introduction", Banzhaf et al include a table of mutation operators applied in tree-based GP, which includes a wide range of operators such as subtree mutation and gene duplication [6].

By contrast, some of the EC literature (perhaps the GA literature in particular) uses the word *mutation* more narrowly to refer to a specific type of change, in which a new value is substituted into a single point of a genome. For instance in "An Introduction to Genetic Algorithms", Mitchell describes mutation by saying that "This operator randomly flips some of the bits in a chromosome" [59].

4 A Population-Parallel Implementation of Cyclic GP

4.1 Introduction

This chapter tackles the first objective outlined in Section 1.4: to use a population-parallel implementation to evaluate cyclic, node-based Genetic Programming (GP) as fast as possible. This beginning to the acceleration work uses a population-parallel implementation and a cyclic GP form, which encompasses a wide range of node-based GP forms. The chapter introduces many of the issues involved in Graphics Processing Unit (GPU) acceleration and so, by comparison with the chapters that follow, it is quite focused on implementation. This chapter's objective poses a range of new challenges for which the solutions are described. The resulting architecture is experimentally assessed and is found to execute cyclic GP up to 175.703 times faster than an implementation using a single core of a Central Processing Unit (CPU).

4.1.1 Motivation for Using Graphics Processing Unit (GPU) Approaches

Why use the GPU in this research? As discussed in Section 2.1.5, the GPU has recently become an extremely powerful way of accelerating GP computation. GPU approaches offer several advantages for this research:

- A GPU card is relatively cheap and easy to obtain.
- In the best cases, GPU acceleration has been shown to achieve impressive acceleration for a range of scientific computing problems.
- There is good reason to hope that this research is well suited to the GPU approaches. This is because GP is "embarrassingly parallel" [1] (as mentioned in Section 2.1) and because much of this research will involve many iterations and many testcases per evaluation which suggests a high ratio of computation to data transfer.
- GPU acceleration should be able to achieve valuable results without the use of multiple computers meaning fewer of the administrative overheads and complexities of maintaining machines and splitting jobs amongst them.
- A single computer can be extended with a motherboard that accepts multiple graphics cards if more computing power is required than can be provided by one GPU.
- The computing cores of a GPU are homogeneous, persistently present and available and can easily be dedicated to the required task. This is in contrast to the computing cores available in, say, a peer-to-peer networks of computers.

- GPU acceleration of GP is an active area of research and so work in this area can make a useful contribution to knowledge as well as being of direct practical value.

In combination, these advantages indicate that GPU approaches are a very interesting avenue for investigation. nVidia have released Compute Unified Device Architecture (CUDA), a technology which allows developers to write kernels for a range of nVidia GPU cards in C [67]. CUDA's extensive resources (such as documentation, libraries and tutorials) combined with the simplicity of writing kernels in C make it an appealing technology and it is the framework used for this work.

4.2 Cyclic Cartesian Genetic Programming

This investigation of cyclic genetic programming uses Cartesian Genetic Programming (CGP). It is important to distinguish between these two types of GP because a GP system may be Cartesian, cyclic, both or neither. The acronym CGP will only be used here to refer to Cartesian Genetic Programming.

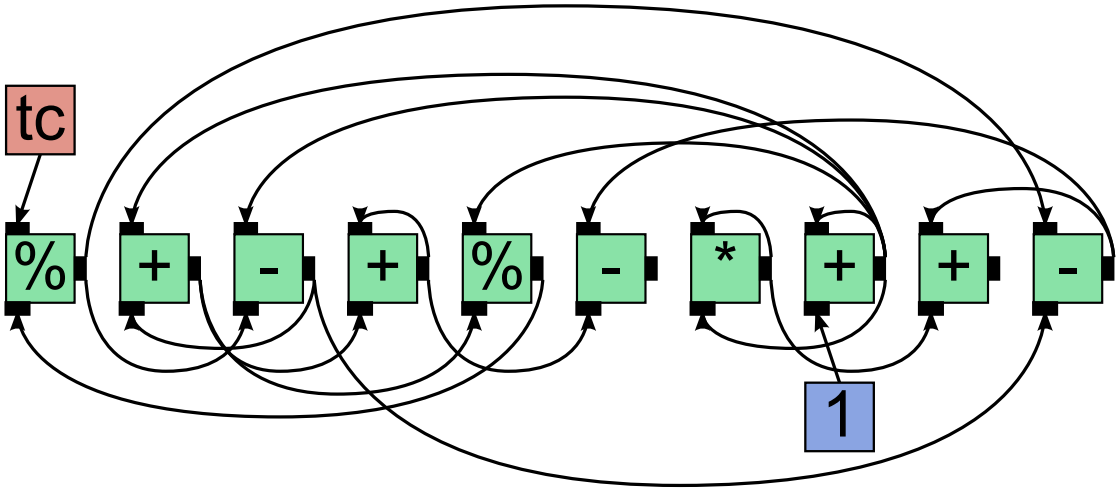
CGP is a form of graph-based GP introduced by Miller and Thomson [58]. The standard form of CGP is constrained to be acyclic, however the paper that introduced it explicitly stated the possibility of adjusting a parameter to allow cyclic individuals.

There are several cyclic forms of existing representations such as cyclic Parallel Distributed Genetic Programming (PDGP) [69] and Neural Programming (NP) [86], however CGP was chosen for this work because, as discussed in Section 3.1.2, it has been the subject of numerous papers and tutorials in recent years.

CGP was originally developed "for the purpose of evolving digital circuits" [30] and it approached the problem of graph crossover by using an elegant mapping from a genotype consisting of a string of integers to a graph-based phenotype.

In a CGP phenotype, the nodes are laid out in a two dimensional ("Cartesian") grid with the input nodes at the left and the output nodes at the right. The connections are constrained such that nodes' inputs are connected to nodes in the previous l columns where l is an adjustable parameter called "levels back". In a CGP individual, each node input has exactly one connection but nodes' outputs may be connected to no nodes, one node or many nodes. Recent CGP work has tended to use one row and a parameter setting that allows node inputs to be connected to any previous node in the row [27] [57]. This leaves few constraints remaining. Further constraints were removed for this work by allowing function nodes' inputs to be connected to the outputs of any other input node or function node (including their own outputs). This move means graphs may potentially be cyclic and so takes them into the territory of cyclic GP.

The evaluation of a normal CGP individual is much like that of any normal GP individual: each node connected to an input of another node is evaluated before it. The evaluation of a cyclic individual is trickier because the cycles make it unclear how



0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	2	0	0
0	3	2	1	2	0	0	3	0	-1
-2	5	3	4	1	-2	0	4	-1	-2
-4	7	6	9	.8	-6	0	5	-2	-5

Figure 11: An example of the iterated evaluation of a cyclic CGP individual. The line of green boxes is the collection of main nodes, the red "tc" box is the testcase and the blue "1" box represents a constant of one. The first input socket of a node is at its top, the second input is at its bottom and the output is at its right. The two division operators are protected so that they return zero when the denominator is zero, as described in Section 3.4. The first row of numbers represents the nodes initialised to zero and the following rows of numbers represent the successive evaluation iterations. Here, the testcase value is -4. In each row, each input is taken from the previous row's outputs. In this example, the eighth node happens to act as a counter by adding a constant one to the previous iteration's value.

to order the node evaluations and the ordering can dramatically affect the result. The standard approach to this is to evaluate in an iterated flip-flop fashion. Before the first iteration, all of the nodes (except the input nodes) are set to a value of 0. For each iteration after that, the nodes' results from the previous iteration are used as their outputs. In this way, the order of node evaluation in each iteration does not affect the result. This iterated evaluation is illustrated in Figure 11 and in pseudo-code in Figure 12. The experiments in this work varied the number of iterations per evaluation.

These representations bear some resemblance to neural networks. Research on using these powerful structures in GP could connect with research on the evolution of neural networks and might allow new problems to be tackled. However little research has been done on cyclic graph-based GP because the evaluations are time consuming. The evaluations require the whole individual to be evaluated over multiple iterations.

```
sub cyclic_cartesian_evaluation(individual, testcase, no_of_iters) {
  nodes = individual.get_nodes();
  node_values = array_of(0.0, nodes.size());
  prev_node_values = array_of(0.0, nodes.size());
  for (iteration = 0; iteration < no_of_iters; ++iteration) {
    for (node_index = 0; node_index < nodes.size(); ++node_index) {
      node = nodes[node_index];
      inputs = [];
      input_node_indices = node.get_node_indices_of_inputs();
      foreach input_node_index (input_node_indices) {
        push inputs, prev_node_values[input_node_index];
      }
      node_values[node_index] = node.perform_node_operation(inputs);
    }
    prev_node_values = node_values;
  }
  return individual.get_output_node_value();
}
```

Figure 12: Pseudo-code illustrating the principles of Cyclic Cartesian Genetic Programming

Furthermore, cyclic graph evaluation has larger memory requirements as will be explained in Section 4.4.3. This makes it much more difficult to design a system in which the processor accesses memory efficiently. This poses a new challenge in designing an appropriate CUDA evaluator. It also means that there may be greater improvements to be had over a standard CPU implementation which also faces the same problems and which presumably uses many slow accesses to memory off the processor's cache.

4.3 Overall CUDA Architecture

In order to use the CUDA framework to access the GPU, client code must launch one or more kernels. Each kernel is a piece of code which is compiled and sent to the GPU and then executed in parallel on multiple multiprocessors. A kernel is written as a C function marked with the `__global` qualifier. Newer releases of CUDA permit more C++ constructs in kernel code. A kernel must be declared `void`, i.e. it cannot return a value and must communicate any results back to the CPU by copying results to an appropriate section of memory. A kernel may call other functions and use their return values; functions to be solely used in this way are marked with the `__device` qualifier. One kernel launch can invoke a very large number of threads. Threads are grouped into thread blocks and thread blocks are in turn grouped into grids. Each thread block within a grid must contain the same number of threads. The code for a kernel is able to identify the block and thread in which it is executing and act accordingly. This idea is demonstrated in the following pseudo-code.

```
__device__ void doSomethingUsingThreadIndex(unsigned int fullThreadIndex) {
    /* Process the fullThreadIndex-th part of the required computation */
    ...
}

__global__ void kernelIllustratingUseOfBlockAndThreadIds() {
  const unsigned int threadIdxInBlock=threadIdx.x;
  const unsigned int numThreadsInBlock=blockDim.x;
  const unsigned int blockIdxInGrid=blockIdx.x;
  const unsigned int numThreadsInPrevBlocks=blockIdxInGrid*numThreadsInBlock;
  const unsigned int fullThreadIndex=numThreadsInPrevBlocks+threadIdxInBlock;
  doSomethingUsingThreadIndex(fullThreadIndex);
}
```

A variety of memory types are available for use by the kernel. Different memory types have very different properties and designing kernels that use them well can have a profound effect on computation speed. Some of the types of memory are designed for sequential access from sequential threads and so their performance degrades as access patterns deviate from this. In particular, the thread blocks are grouped into warps of threads and the memory performance depends on the access pattern within each warp (or half warp). For all current devices, a warp contains 32 consecutive threads.

Registers are very fast units of memory local to a specific thread and they are the default type of memory used by the compiler for local variables. There are a total of between 8192 and 32768 registers available per block, depending on the compute capability of the GPU device. Registers are completely separate from each other and are not addressable so they cannot be used in an array (unless the size and all the access indices of the array are compile-time constants so that the compiler can implement them as completely separate variables).

Shared memory is accessible by all threads within a given block and can be similarly fast although access speed depends on the access pattern used. There is a total of 16384–49152 bytes per block depending on the compute capability of the GPU device.

Registers and shared memory are both very useful types of memory but neither persists between thread block executions and neither is accessible from the CPU. To allow data transfers to and from the device, CUDA also provides global memory and constant memory. Accessing these types of memory is very much slower than accessing registers or shared memory (roughly in the order of 100 times slower) but there is a much larger amount of them available and it is possible to access them from the CPU. Since global memory access is so slow, it can often be quicker to recalculate derived data on the GPU than to load it from global memory. The amount of global memory varies between cards (and even between cards that use the same GPU) but, as examples, a GTX260 card might come with 896 MB and a GTX480 card might come with 1536 MB. Constant memory can offer improvements over global memory due to caching but at the cost of only providing read-only access to the GPU, as its name suggests.

Global memory is accessible from both the kernel code and the CPU code that initiates the kernel and so is typically used as a means of transferring data to and from the kernel. Kernel access to the global memory should ideally be *coalesced*, that is arranged into a single contiguous, aligned memory access across the threads in a half warp (and across the threads in a full warp for CUDA devices of compute capability greater than or equal to 2.0). The slowness of global memory accesses is made considerably worse if the access is not coalesced.

The typical way a kernel might use this memory model is to read data from global memory into shared memory and registers, use registers to perform calculations on the data in the shared memory and then copy results to some other area of global memory.

In cases where there are insufficient registers available, the compiler uses local variables as a substitute. Local variables have the same functional behaviour as registers but they are implemented in the same off-chip device memory as global memory and so have the same poor access performance. It is possible to force the compiler to use this mechanism to constrain the number of registers, for example to allow more threads per block.

A CUDA kernel launch is asynchronous so the host code is free to return to other tasks whilst the kernel is being executed and may query the readiness of the results at any time. The host code may intermittently check for the results whilst performing other tasks or may block until the results are ready if there is no other work that can be usefully performed.

4.3.1 Decisions and Constraints

The CUDA architecture allows code to launch many threads on the GPU. The user determines the size and number of the thread blocks and hence the total number of threads. CUDA makes no guarantees about the order of execution of the thread blocks so threads in different blocks cannot interact. However, threads within a block are executed in a batch in such a way that it is possible for them to communicate. These constraints give applications a lot of freedom about how to divide work amongst threads.

The task imposes some additional constraints. The evaluation of a population of cyclic GP individuals involves evaluating the many nodes of many programs over many testcases for many iterations. For a given program–testcase pair, all the work of one iteration must be complete before any work for the next iteration commences and each node must be able to access the results of the others. Apart from these constraints, there remains a lot of freedom regarding how to divide the work up over the threads. However there are further constraints and recommendations for achieving the best results from the CUDA technology.

If the work is divided up too much into tiny packets of work then the overheads of each thread may drown out time spent on real computation. If the work is divided up too little into huge packets of work then it may fail to saturate the GPU's computing

resources with enough threads per block and blocks per grid. Either of these extremes may yield disappointing results from the GPU.

Much of the design of the architecture involves making choices and using up freedoms to satisfy constraints and performance recommendations. The CUDA documentation provides detailed information on the architecture of the CUDA capable GPU and how it affects the design of good CUDA applications. The following text will introduce constraints and efficiency guidelines and explain how they influenced the design of the architecture in this work.

4.3.2 Kernel Details

As mentioned in Section 4.3, the execution of a typical kernel involves loading data into shared memory and registers, processing the values in shared memory and registers and then sending it back again.

The method for accessing shared memory arrays for which the size is not known at compile-time is rather involved. It requires calculating pointers to the memory by using offsets from the start of the shared memory, which requires knowledge of the size of each of the required arrays.

CUDA attempts to organise the execution of warps of threads to maximise efficiency which means that developers may not make many assumptions about the order of execution of threads within a block. However, it is possible to synchronise a block's threads at any point in the kernel's execution using the CUDA function `__syncthreads()`.

The kernel code may assume that the threads within a warp are always synchronised but must use `__syncthreads()` to synchronise threads in different warps. The result of this call is that afterwards it may be assumed that all threads in the block have reached that call and all effects of statements before the call are complete. Understandably, kernel code is forbidden from calling `__syncthreads()` from sections of code which may involve divergent thread behaviour because it is meaningless to call `__syncthreads()` if some threads cannot reach the point of the call.

CUDA is capable of loading each multiprocessor with multiple warps of threads. The multiprocessor keeps all information about the execution state of each warp on the chip and so is able to switch between executing different warps extremely quickly. The multiprocessor uses this to attempt to cover latencies so that, for example, if one warp requests a read from global memory, the multiprocessor can initiate the read and then get busy executing another warp for the several hundred clock cycles that it takes for the read to complete. Unfortunately, each multiprocessor only has a limited quantity of registers and shared memory so the number of warps with which it can be loaded depends on how many registers and how much shared memory a given kernel launch requires for its warps.

4.3.3 Textures

The code arranges the data to allow coalesced access as far as possible. However this is less relevant for data accessed via textures. Textures provide another method of read-only access to global-memory and allow efficient access that does not need to be coalesced. An early experiment during the development of the architecture indicated that textures gave improved speeds so they were incorporated into the system. Since the publication of a 2009 paper describing some of this work [49], newer nVidia documentation has stated that the future direction of CUDA devices is away from this use of textures to provide access to global memory (because devices of compute capability 2.0 and above provide an explicit cache). With this in mind, the code has been migrated to perform fewer of the memory accesses through textures. Nevertheless, the devices used in this work are of compute capability 1.3 and textures remains a key part of the architecture at the time of writing so must be described here.

On the CPU side, the texture objects can unfortunately only be handled by `nvcc` compiled code which means C++ code is not currently able to refer to the texture objects directly. There is good reason to want to be able to have the C++ directly refer to such resources that must be obtained and returned. It is considered to be good C++ practice in cases such as these to use the Resource Acquisition Is Initialisation (RAII) idiom to ensure that resources are automatically cleaned up [83]. This practice was widely followed for the several different types of CUDA resources required. This issue will be discussed further in the description of later stages of the research.

The code was written such that the actual textures are hidden from the C++ code whilst still allowing the RAII idiom to be used to ensure the textures are automatically unbound (and unbound before the appropriate memory is deallocated). In more detail: the `TextureBinding` class is designed using the RAII idiom so that it automatically unbinds the texture on destruction (and does this before the associated memory is freed). A further problem arises because it is not possible to declare arrays of textures. This was circumvented by using some hard-coding of specific textures and a range of C macro tricks to access them.

4.4 Implementation Details

4.4.1 Minimising Divergent Warps and Optimising Memory Access With Testcase-Groups

As discussed in Section 2.1.5, the CUDA architecture uses a Single Program, Multiple Data (SPMD) approach rather than a Single Instruction, Multiple Data (SIMD) approach. This means that all the threads in a given kernel launch must execute on the same program but do not necessarily have to follow the same paths.

As described in Section 4.3, the CUDA threads are grouped into warps and at the time of writing, all CUDA devices have 32 threads per warp. Threads within a warp

should always execute the same instruction in parallel for maximum efficiency. Neighbouring threads may follow different execution paths but this is implemented by all the threads in the warp executing all of the paths. Hence any flow control instruction (such as `if`, `switch` or `for`) that causes threads of the same warp to diverge from each other (i.e. to follow different execution paths) causes all threads in the warp to pay the full time penalty of executing all threads' branches.

Minimising warp divergence is one of the top priority optimisation aims in kernel design (along with minimising global memory accesses and designing the necessary global memory accesses to be coalesced wherever possible). This presents a choice of how to distribute work between neighbouring threads in a warp to minimise the warp divergence. It would be simpler to divide work using only one aspect of the computation. Referring back to the computational cuboid in Subfigure 3(b), the choice is how to divide up a single blue cuboid between neighbouring threads. This leaves three choices: to divide up the work over a warp by iteration, by testcase or by node (either within the same individual or not). As mentioned earlier, successive iterations of the same node–testcase pair must be evaluated sequentially. This leaves a choice between dividing up the work by testcase or by node.

Different nodes may have quite different behaviour; one node may have a different operation and a different arity to another. This makes dividing the work by node quite unappealing. On the other hand, the computation involved for different testcases is almost identical; typically the only difference to the computation is the set of values involved. This parallelisation over data is precisely the sort of division of work to which GPU computation is ideally suited.

This leads to the conclusion that contiguous threads in the warp should evaluate the same nodes of the same programs on different testcases. Indeed, the natural arrangement is to use contiguous threads to evaluate contiguous testcases and this makes good memory access patterns easier to achieve.

This approach has previously been proposed and used with success [76] [77] and it was adopted for this work.

If there are exactly 32 testcases, then it is very easy to assign one testcase per thread in each warp. When there are fewer than 32 testcases, they are padded out so that their number is a power of two in an attempt to minimise unnecessary warp divergence. For example, if there are 13 real testcases, they are padded out to 16 testcases so that each half of each warp evaluates these 16 testcases. In that case, there will still be some divergence within warps because each half of each warp will evaluate the testcases over a different sets of nodes. However, at least this padding avoids groups of testcases being unnecessarily split from the end of one warp over to the start of another.

Similarly, when there are more than 32 testcases, the testcases are padded out so that their number is a multiple of 32. In this case, the threads will evaluate the testcases in consecutive batches, with each batch being executed simultaneously by 32 consecutive

threads. For particularly large programs, it may be necessary to break up testcases into artificially smaller groups as explained later in Section 4.4.4.

This deals with the division of the work of the testcases. Since the group of threads evaluating a group of testcases is evaluating the same iterations of the same nodes of the same individuals at the same time, it is helpful to think of the group acting as one. For this reason, the phrase "testcase-group" will be used to describe a group of threads evaluating a group of testcases (and occasionally to describe the group of testcases themselves). When possible, the size of the testcase-group will be the same as the size of the warp (32 threads) but as discussed above, this may not always be true. Hence it will often be more useful to refer to the size of the testcase-group rather than the size of the warp. Figure 13 illustrates the division of a thread block into testcase-groups of threads.

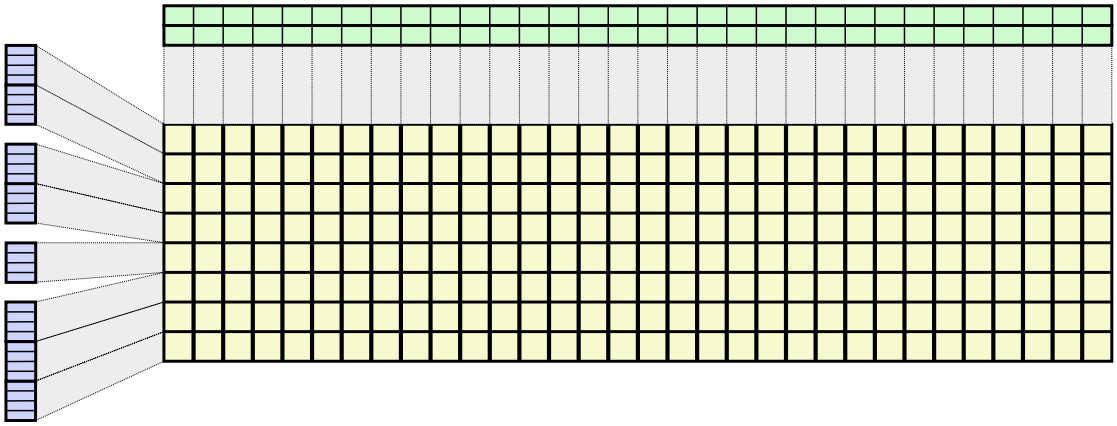


Figure 13: The design organises each thread block into *testcase-groups* of threads, each evaluating a *node-set* of nodes over all the testcases. Here, the block of 256 threads (in yellow) contains eight testcase-groups of 32 threads. Each testcase-group evaluates a specific node-set of nodes (in blue) over consecutive testcases (in green). When the testcase-group of threads completes one row of testcases, it moves onto the next. Here, each node-set contains four nodes and some programs require multiple node-sets. The thread block synchronises after each iteration so that threads working on the same testcase and the same program can use each other's results. This example is covered again in Figure 15.

4.4.2 Limits on Registers Constrain the Number of Threads

The documentation encourages the use of many threads in each thread block and many thread blocks in each grid. The CUDA device used in this work is of compute capability 1.3 and allows a maximum of 512 threads per block and 16384 registers per thread block as mentioned in Section 4.3. These registers provide very fast local memory and they are automatically used by the compiler for local variables in the kernel code. It is possible to force the compiler to limit the number of registers used per thread which it achieves through the use of shared memory. The code used for the kernel in this work is compiled with a limit of 64 registers per thread without any noticeable problems.

With 16384 registers available per block, this means that the kernel can be launched with a maximum of 256 threads.

More recent GPUs of compute capability 2.0 or above offer 32768 registers per thread block. If such processors were available for this work, the number of threads might still be kept at 256 and the compile limit on registers per thread might even be reduced slightly. The reason for this is that the GPU's resources are available to multiple units of work and so if the resource requirements are sufficiently low, the GPU is able to hold more units of work at the same time. This does not allow the processor to compute any faster but, as mentioned in Section 4.3.2, it does allow it to attempt to cover memory access latencies. This means that if one unit of work requests a memory access (which might take several hundred clock cycles), the processor can stay productive by switching to performing computations on another unit of work. The term "occupancy" is used to describe the number of units of work being processed as a fraction of the maximum possible units of work. nVidia provide a "CUDA Occupancy Calculator" spreadsheet for calculating the occupancy that will be achieved with different numbers of threads per block, registers per thread and bytes of shared memory per block. The spreadsheet uses graphs to indicate the effect on occupancy of varying any of these three parameters whilst holding the other two constant. These tools indicate that the best use of a device of compute capability 2.0 or above might be to constrain register use and shared memory use to improve latency covering.

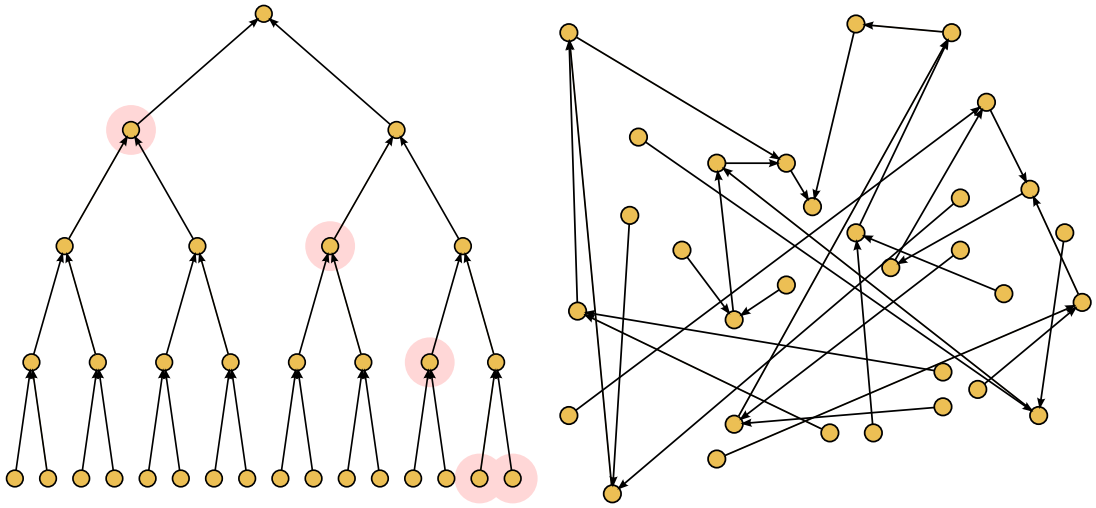
4.4.3 Shared Memory Limits

As discussed in Section 4.4.1, it is preferable to have neighbouring threads evaluating the same thing on neighbouring testcases. This naturally suggests a policy of "one thread evaluates one individual on one testcase". This policy was adopted with success for tree-based GP by Langdon [45] and Robilliard [76]. Unfortunately it turns out that the greater memory requirements of cyclic GP prevent this from working and require a more complicated solution.

As the kernel evaluates, it requires somewhere to store the values being computed for the nodes. Ideally, it would be best to store this local data in registers. Unfortunately, this is not possible for two reasons: firstly because there are no spare registers (as explained above) and secondly because the size and all access indices of any arrays of registers must be constant. This second rule in particular would induce unacceptable constraints on the rest of the architecture.

The next best memory after registers is shared memory, which can be slower depending on how it is accessed. As explained in Section 4.3, it is common for kernels to copy input data into shared memory, carry out processing in registers and shared memory and then copy results back to global memory. Shared memory is shared between the threads of a block and so allows threads to use each other's intermediate results which is useful for this architecture as will be seen.

The biggest limitation of shared memory is its small size. CUDA provides each block with 16384 bytes of shared memory (which can be increased to 49152 bytes in devices of compute capability 2.0 or higher but at the expense of some of the L1 cache offered by such devices). This is sufficient for many applications but unfortunately memory requirements tend to be greater when evaluating cyclic graph-based GP individuals than when evaluating tree-based GP individuals as illustrated in Figure 14.



(a) Tree-based GP (b) Cyclic, graph-based GP

Figure 14: Cyclic, graph-based GP requires more memory for evaluation than tree-based GP. Both example individuals have 31 nodes and 30 connectors, yet the tree-based example in Subfigure 14(a) uses a stack of 5 memory slots whilst the cyclic example in Subfigure 14(b) uses 31 memory slots, one per node. To see how the tree in Subfigure 14(a) may be calculated with 5 memory slots, trace a depth-first, left-to-right traversal of the tree, disposing values when they are no longer needed. Doing this shows that the most memory slots are required near the end of this traversal, when the values of the highlighted nodes must all be held in memory. It might be possible to evaluate the particular individual in Subfigure 14(b) with fewer slots but, in general, 31 memory slots must be available for 31-node, cyclic individuals.

This is because partial results are not reused in trees and so can be discarded after first use. A full tree of depth d and made up of function nodes with arity a will contain $\frac{a^d-1}{a-1}$ nodes but a stack-based tree evaluator may only require $(d-1)*(a-1)+1$ memory slots to evaluate it. To illustrate the significance of the difference, a full tree with $a = 4$ and $d = 10$ would have 349525 nodes but could be evaluated with only 28 memory slots. Evaluating 349525 nodes in a cyclic graph-based individual might require up to 349525 memory slots.

Worse, each combination of a node and a testcase requires two floating point numbers, or eight bytes, of this memory for the iterated flip-flop evaluation. Since there are 16384 bytes of share memory per block, this means that each thread block is restricted to evaluating 2048 node–testcase combinations at once. Furthermore, each individual must be evaluated within one thread block in order to use this shared memory.

Indeed, the situation is slightly worse still because the kernel is written to also use

some shared memory to store local copies of data on testcases and nodes because it improves speed to use fast memory for these regularly used data.

The decision described in Section 4.4.1 to divide consecutive testcases over up to 32 testcases means that it will often be impossible to evaluate any more than $2048/32 = 64$ distinct nodes in one thread block. Limiting all individuals to around 64 nodes would be rather severe so the code deploys tactics to evaluate larger programs (containing more than $\approx$64 nodes but fewer than $\approx$2048 nodes) whilst still allowing smaller programs in the same evaluation job to be evaluated as efficiently as possible.

Since there are 256 threads and since there will often be 32 threads per testcase-group, the $\approx$64 nodes will often need to be divided amongst the $256/32 = 8$ testcase-groups meaning that each testcase-group evaluates a maximum of 8 nodes. For any programs larger than 8 nodes, this requires dividing up the evaluation of a single program over multiple threads. As mentioned in Section 4.3.2, the CUDA function `__syncthreads()` is used by the kernel to synchronise the different parts of the evaluation.

Just as it is helpful to give the name "testcase-group" to a group of threads evaluating a single group of testcases, so it is helpful to give the name "node-set" to a set of nodes to be evaluated by one thread. Indeed it is much simpler to consider the division of work over threads according to the testcase-groups and the node-sets. Figure 13 shows this principle with a simple example. To help illustrate this further, Figure 15 shows the same example in more detail.

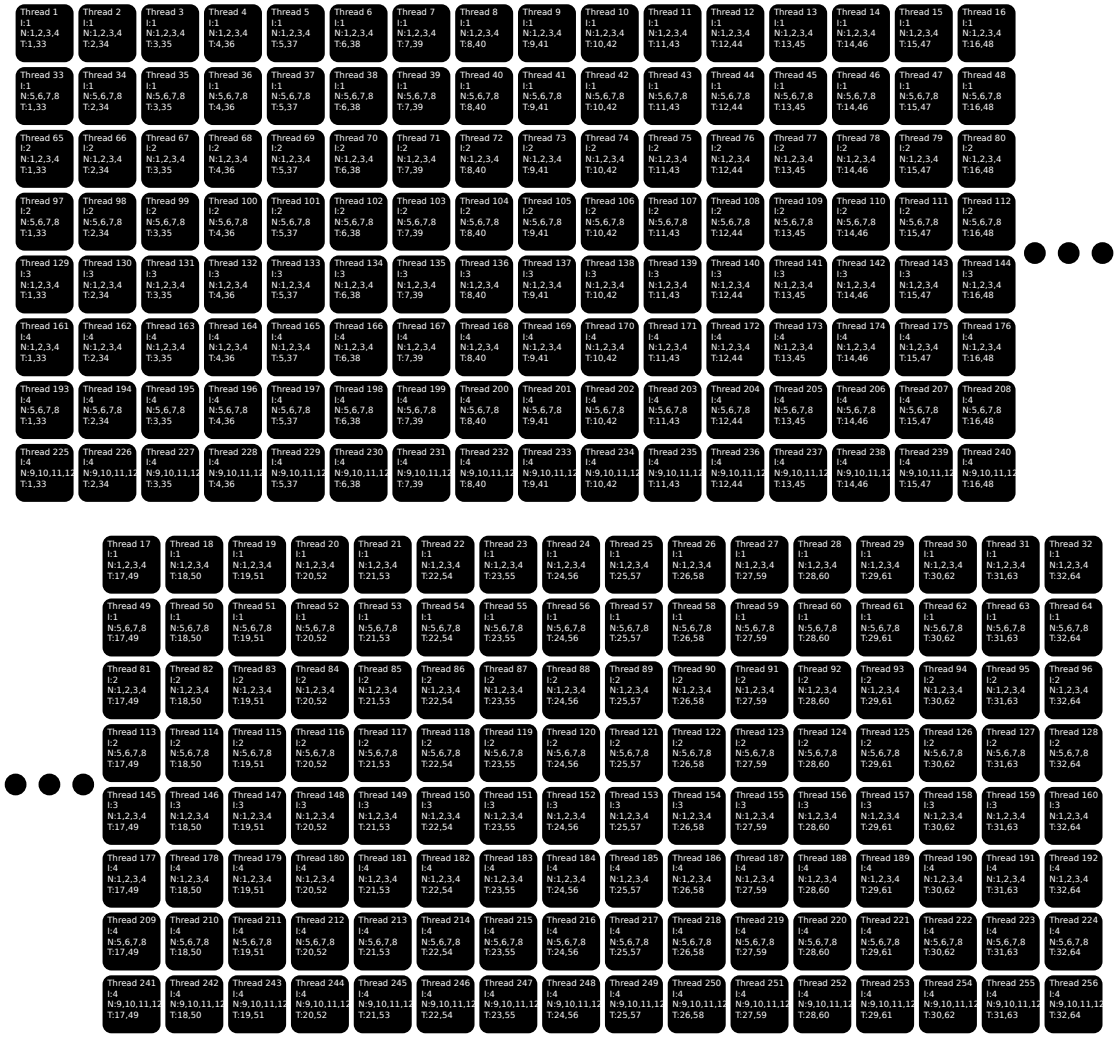
Figure 15(a) depicts every thread in a thread block of 256 threads arranged into 8 rows of 32 threads. Each row of threads evaluates one node-set on two consecutive testcase-groups and the 32 testcases of the each testcase-group are distributed over the 32 threads. In this example, each node-set contains 4 nodes which are to be evaluated by all the threads in the corresponding row. Figure 15(b) represents the same thread block and some more information. This shows that such descriptions are much simpler when expressed in terms of testcase-groups and node-sets.

Shared memory is arranged into banks with neighbouring memory locations in different banks. To achieve maximum efficiency, the threads in a half warp must all access different memory banks, which might commonly be achieved by each of a contiguous sequence of memory locations being accessed by any one of the threads. On reading, there is a special "broadcast" mechanism, which means it is just as efficient if all threads simultaneously read from one location.

4.4.4 ThreadPolicy and ThreadPlan

The above design decisions and method of viewing things in terms of testcase-groups and node-sets go some way to determining how jobs are processed. However there remains considerable flexibility about how each specific job is performed.

The policy for making the remaining decisions will be referred to as a `ThreadPolicy`



(a) The `ThreadPlan` details of a single block of threads

```
ThreadPlan[
   Testcase padding: 3
   No of threads per block: 256
   No of blocks: 1
   No of node-sets: 8
   Block 1 (index 0) {
      Number of testcases per testcase-group: 32
      Number of nodes per thread: 4
      Program and nodes by node-set:
         [P1,N1-4] [P1,N5-8] [P2,N1-4] [P2,N5-8] [P3,N1-4] [P4,N1-4] [P4,N5-8] [P4,N9-12]
   }
]
```

(b) A full `ThreadPlan` with one block of threads, equivalent to that in Subfigure 15(a)

Figure 15: Subfigure 15(a) shows an example of the division of work in a block of 256 CUDA threads. The letter I precedes the individual's number, N precedes the nodes' numbers and T precedes the testcases' numbers. The block has been separated into two sets of threads to allow the figure to fit on the page and the suspension points indicate the join. Subfigure 15(b) shows a full `ThreadPlan` (see Section 4.4.4), which contains the same thread block. In this subfigure, P precedes the program's number (equivalent to I preceding the individual's number) and N again precedes the nodes' numbers. These two subfigures show how viewing the division of work in terms of testcase-groups and node-sets simplifies matters considerably. Given the framework of `ThreadPlans`, Subfigure 15(b) conveys more information than does Subfigure 15(a).

and a specific set of such decisions for a given evaluation job will be referred to as a `ThreadPlan`. These terms correspond to two classes used in the code.

A `ThreadPolicy` receives the relevant details about an evaluation submission as input and produces a `ThreadPlan` as output. What are these details? The required details are the individuals' sizes and number of testcases. The `ThreadPolicy` is also able to call upon various values of the environment:

- The maximum number of threads allowed per block,
- The number of registers available per block,
- The number of threads per warp and
- The amount of shared memory available per block.

Although some of these values have already been used in the decisions described above, it is preferable to use dynamically retrieved values so that the code will automatically benefit from being run on newer hardware with improved abilities.

The `ThreadPolicy` takes all of this information and uses it to produce a `ThreadPlan`. What information must a `ThreadPlan` contain? It must contain the number of threads per block. For each thread block, it must specify the number of threads (or testcases) in each testcase-group, the number of nodes in each node-set and a reference to information about the node-sets in the block. For each node-set (of each block), the `ThreadPlan` must specify the index of the individual to be evaluated and the index (within the individual) of the first node in the node-set.

Combined with the decisions described above, this `ThreadPlan` gives all the information required to divide up the work amongst blocks of threads. Figure 15(b) shows an example `ThreadPlan`.

Of course, the `ThreadPlan` that a `ThreadPolicy` generates must match several constraints. For example, every node in every individual must be evaluated by at least one node-set (and in fact must be evaluated by exactly one node-set because otherwise threads can conflict with each other when writing to the shared memory). Furthermore, the nodes of a given individual must all be evaluated within the same thread block. For each block, the number of node-sets multiplied by the size of the testcase-group must equal the `ThreadPlan`'s number of threads per block. Finally, the amount of shared memory the `ThreadPlan` requires must be less than or equal to the amount available.

The code represents `ThreadPolicy` as an abstract base class so that different thread policies may be plugged in. However, only one `ThreadPolicy` was used for this work and there follows a description of how that `ThreadPolicy` goes about determining `ThreadPlans`.

The first steps are to evaluate the number of threads per block and the amount of padding to be applied to the testcases. Both of these decisions are described above and the calculations are fairly routine. From that point onwards, the layout of the testcases

is fairly obvious and can largely be disregarded (apart from for memory calculations etc). The remaining problem is the distribution of the nodes over the various node-sets. The default size of the testcase-group can be easily calculated as the lesser of either the warp size or the number of appropriately padded testcases.

This gives enough information to calculate the maximum number of nodes each node-set can contain within the shared memory constraints. This, in turn, makes it possible to calculate the number of nodes a block can evaluate. The next stage is to take all the individuals that have that number of nodes or fewer and pack them into blocks.

This is an instance of the well studied "bin-packing problem": given a list L of items, with sizes of varying fractions of the capacity of identical bins, find the optimal distribution of items between the bins so as to pack all the items using as few bins as possible. Here, each block is a bin with capacity in terms of the number of node-sets it can compute and the list of programs is the list of items with sizes in terms of the number of node-sets each program requires. Finding a solution to the bin-packing problem using the optimal number of bins, $OPT(L)$, is an NP–hard problem. However it has been shown that the number of bins, $FFD(L)$, used by a packing given by the First Fit Decreasing (FFD) heuristic satisfies the condition $FFD(L) \leq \frac{11}{9}OPT(L) + 1, \forall L$ [108].

The FFD heuristic simply sorts the items in decreasing order of size and then works though the items, packing each into the first bin in which it fits. The FFD approach is used here for packing the programs in the thread blocks.

Of course, there may be individuals too big to fit into any of the thread blocks. So when all possible packing is complete, the size of the testcase-group is halved and the process repeated to allow more packing. Halving the size of the testcase-group halves the number of program–testcase pairs the block is expected to compute and so frees up more shared memory per program allowing larger programs to be packed. This comes at the expense of increased warp divergence and so is only done for those programs that cannot be packed otherwise. This process is repeated until all the individuals are packed.

Consider the following worked example, illustrated in Figure 16. 20 programs are to be packed into blocks of 256 threads. When using 32 testcases per testcase-group, the shared-memory constraints allow 7 nodes for each of the 8 node-sets; when using 16 testcases per testcase-group, the constraints allow 6 nodes for each of the 16 node-sets; no further dividing of the warp is necessary in this example. The sizes of programs are 51, 48, 67, 49, 48, 20, 13, 54, 96, 55, 6, 68, 60, 55, 15, 81, 5, 37, 82 and 4 nodes respectively. Blocks using 8 node-sets of 7 nodes offer enough space to contain any programs of 56 nodes or fewer which means programs 1, 2, 4, 5, 6, 7, 8, 10, 11, 14, 15, 17, 18 and 20. These programs are sorted into descending order of the number of node-sets they require and then packed using the FFD heuristic to get the following blocks: 1, 8, 10,

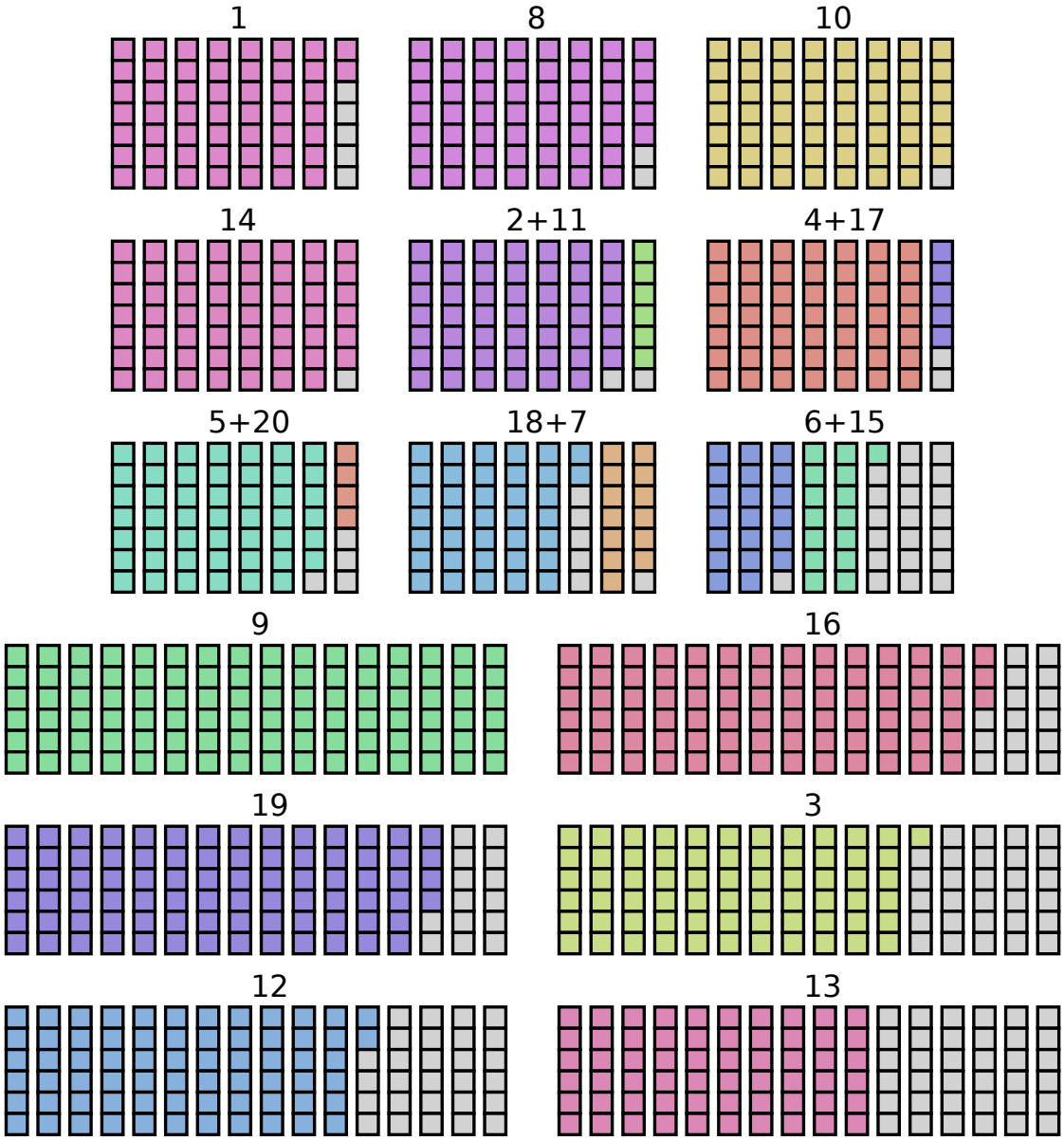


Figure 16: An example packing of 20 programs. Each rectangle of squares represents one thread block. However the depiction differs from that in Figures 13 and 15(a) in that squares do not represent threads. Here, the focus is on the node-sets not the testcases in the testcase-groups so each column of squares represents a node-set (a group of nodes to be executed in parallel across a testcase-group). Each square represents one node and the colours of the squares indicate the different programs to which the nodes belong. The hues have no significance other than that they highlight the different programs. Grey squares represent padding nodes. The upper nine blocks are evaluating 32 testcases in each testcase-group and this provides enough memory for seven nodes per node-set; the lower six blocks are evaluating 16 testcases in each testcase-group and have six nodes per node-set. In either case, the number of node-sets multiplied by the number of testcases per testcase-group is 256, the number of threads per block. The programs have been reordered so that their respective sizes are now of sizes 51, 54, 55, 55, 48, 6, 49, 5, 48, 4, 37, 13, 20, 15, 96, 81, 82, 67, 68, 60. See the main text for more information on the steps that lead to this packing.

14, 2+11, 4+17, 5+20, 18+7, 6+15. Packing the remaining programs into blocks using 16 testcases per testcase-group produces: 9, 16, 19, 3, 12 and 13.

At present the `ThreadPolicy` is unable to reduce the number of threads per block to allow it to handle extremely large individuals. If the halving of the testcase-group reaches one and there remain individuals too large to be packed, the `ThreadPolicy` must give up and output an error message. It may be possible to adjust the code to handle this sort of situation but this would significantly increase the complexity and is not necessary for this work.

When constructing a `ThreadPlan`, it is likely that there are some blocks, that cannot be completely filled with useful node-sets. In this case, the `ThreadPlan` is padded out with repeated copies of a padding node-set which extends the program referred to by the last useful node-set in the block. This padding node-set will refer to nodes beyond the end of the real program and indeed some of the other node-sets at the end of each program may contain some nodes beyond the end to fill out the complete node-set. The inputs and weights are padded out accordingly so that there are dummy values for the evaluator to use for these nodes. However these nodes will never be referred to by the other real nodes in the program and none of them will ever be set as the output node so they will have no effect.

As mentioned earlier in this section, the `ThreadPlan` must avoid evaluating any real nodes with more than one thread to prevent potential memory clashes. This constraint does not apply to the padding nodes because their results are not used.

The `CudaEvaluator` code that receives the `ThreadPlan` from the `ThreadPolicy` must ensure that the data which is copied to the GPU device is appropriately padded. This includes padding out the testcases with false testcases, padding out the programs with false nodes and padding out node data to the maximum arity used by any of the nodes.

4.4.5 Time Recording and Timeline Generation

In order to understand the performance of the system, it is helpful to get an indication of the timing of events. This is particularly valuable for the work described in Chapter 5 In order to obtain the times on the GPU, CUDA provides a mechanism to insert a point, called an event, into the stream of GPU requests and then later query when this event occurred. The time is retrieved relative to some earlier GPU event. To obtain an overall view of the system's timings, it is important to be able to relate these timings with CPU timings. To achieve this, the system precedes a run by inserting an anchor event into the GPU stream with no preceding requests so that the event occurs on the GPU immediately. Using this event, it is possible to map from GPU event timings to absolute clock times on the CPU. These techniques were used to obtain the times for the timelines, an example of which is shown in Figure 17.

Henceforth, all such timelines will follow the same rules. The time is shown on the x-axis in seconds. Different rows are shown in different regions marked as either

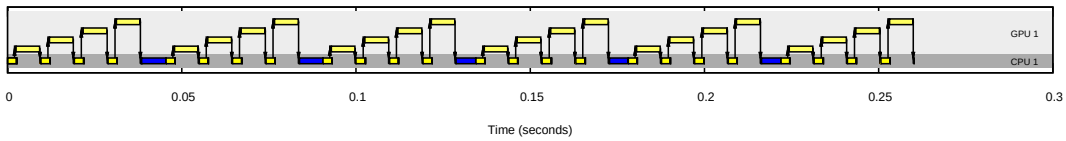


Figure 17: An example of the sort of timeline that can be automatically generated from runs. See the main text below for a guide. The times used to construct the timeline are real and are recorded throughout the run. The CPU times and the GPU times are recorded with different offsets so an anchor event is performed before the run and this permits the GPU times to be translated into their equivalent CPU times.

GPU or CPU according to the type of processor on which the line's events occurred. On later timelines, there may be multiple GPU and CPU regions to indicate the use of multiple processors. Within a GPU region, different rows represent GPU processing of different demes. On the CPU side, light yellow bars mean submission (and CPU evaluation where appropriate), orange bars mean gathering and dark blue bars mean preparation. On the GPU side, light yellow bars mean computation, red bars mean memory transfer, thin black bars indicate a deme waiting to be evaluated or gathered and arrows mean submission or retrieval. For both CPU and GPU, the absence of bars means idle (although on the GPU side, this is often because the GPU is working on another deme. This is summarised in Table 4.

CPU side, light yellow bars	Submission (and CPU evaluation where appropriate)
CPU side, orange bars	Gathering
CPU side, dark blue bars	Preparation
GPU side, light yellow bars	Computation
GPU side, red bars	Memory transfer
GPU side, thin black bars	A deme waiting to be evaluated or gathered
Arrows	Submission or retrieval

Table 4: A key to the timeline figures

The initial submissions are sometimes slower than all following submissions, presumably because of time spent on one-off setup tasks such as resource allocation. To avoid this distracting from the point being illustrated by the timelines, their runs each used an earlier warm-up generation, which is not shown.

4.5 Assessment of Performance

4.5.1 Performance Measurement Issues

Before describing the experiments, it is worth explaining that it is difficult to compare directly between results in this area and it is not clear what measurement is most useful. An ideal measurement would allow direct comparison between different pieces of re-

search and would give other Evolutionary Computation (EC) researchers an indication of the potential benefits.

Perhaps the most obvious measurement is the speed improvement over a CPU implementation. Unfortunately different systems will have different configurations in many areas such as the CPU, the GPU, the compiler and the compiler options. One test indicated that simply turning the compiler optimisations off makes the CPU evaluator used here run 3.32 times slower. Worse, each researcher may focus on a different type of EC system and so may have their own CPU implementation with a different level of efficiency. Those GPU researchers who persevere to craft the best CPU implementations are then punished by facing the harshest comparisons.

An alternative comparison might be sought by using the device emulation mode of the GPU technology. However, this may fail to discriminate the quality of the GPU code as any inefficiencies will be present on both sides of the comparison. This mode is also likely to be much less efficient than a good CPU implementation so it may give an artificially positive impression of GPUs. It was not used here because a test showed the device emulation runs to be 9.60 times slower than the CPU evaluator runs.

Another possibility is to measure the rate of evaluation of GP primitives. This has the advantage of being meaningful without a comparison to the CPU performance. The disadvantage is that it may be unfair on attempts to tackle GP systems that are inherently difficult to run efficiently (on either the CPU or the GPU). The greater memory requirements of cyclic GP discussed in Section 4.4.3 give cause to suspect that cyclic GP may be just such a system.

A further issue is the importance of distinguishing whether results are measured only over the period of evaluation or over the whole run. Work in this area often reports results measured only over the evaluation to indicate the full scale of the improvement. However only measuring during the evaluation may give an artificially positive impression of the speed improvements to be gained with GPUs.

The work described in Section 5.2 aims to reduce the overall run time by performing other tasks in parallel with the evaluation, so measurements over the whole run are important.

4.5.2 Experimental Setup and Results

The details of the system setup used in this work are provided in Table 5. Each result was averaged over ten runs. The times measured over the whole run did not include time spent creating and destroying resources at the start and end of the run.

The equivalence of the results from the GPU evaluator and CPU evaluator was verified but, as discussed in Section 1.5, the experiments in this thesis are not concerned with the behaviour of the runs, only accelerating them. For this reason, these results are not considered here. The experiments did not use additional optimisations (such as reusing results for identical individuals, removing nodes not connected to the out-

CPU	Intel Core2 Quad Q6600, 2.40GHz
Graphics card 1	BFG GTX260 OC MAXCORE 55nm 896MB
Graphics card 2	BFG GTX260 OC MAXCORE 55nm 896MB
Cores per card	216
Shader clock speed	1296MHz
Operating system	Ubuntu 10.10 (Linux 2.6.35-28-generic-pae)
Compiler	GCC v4.4.5
GCC options	-O3 (optimisation level 3)
CUDA	v3.2 (`nvcc` v0.2.1221)
nVidia driver	v270. 41.06
Fitness caching	None between generations
Cyclic evaluation	Iterated (no early convergence checks)

Table 5: A table summarising the technical details of the system used for the experiments.

put and terminating the iterated evaluation early on convergence) because these might have obscured the performance gain achieved by the GPU.

The CPU evaluator and the GPU evaluator were written as subclasses of a common abstract base class. The approach of using equivalent `Evaluator` classes is advocated because it facilitates adding support for future parallel technologies. This architecture also made it easy to compare the speeds and cross-validate the results. Validation was complicated by the fact that the CPU and GPU have slightly different floating point implementations. In the past this could be dealt with using CUDA's device emulation mode which attempted to emulate the GPU using CPU threads and which could be used for direct comparison with the CPU evaluator. Unfortunately, nVidia are changing the debugging functionality to allow debugging software to directly interact with kernels running on the GPU using facilities provided by the hardware of later cards. For this
reason, the emulation mode was deprecated in release 3.0 of the CUDA toolkit in March 2010 and removed in release 3.1 in June 2010. This sort of CPU emulation can now be achieved with third-party tools such as Ocelot (`http://code.google.com/p/gpuocelot/`). Anyway, there is less need for CPU emulation in the newer cards of compute capability 2.0 and higher since their floating point implementations have far fewer deviations from the IEEE 754-2008 [36] floating point standard.

When the code is built with the debug mode switched on, many checks in the code are executed. This mode also turns on such things as compiler debug symbols (for GCC and `nvcc`) and `GLIBCXX_DEBUG` which performs bounds checking on C++ standard STL containers. However the release build was used for the experiments described here and the release build has the debug mode switched off and the compiler optimisations switched on.

The problem tackled and the CGP parameters are summarised in Table 6. The experiments followed much of the CGP research in not using crossover [27] [57]. Acyclic CGP systems may use very large individuals because few of the nodes typically get

Objective	Match $x^2 + x + 1$
Testcases	Evenly spaced points from -4 to 4
Number of testcases	Varied in experiments
Function set	+, -, *, % (protected division)
Terminal set	x
Evaluation type	32-bit floating point numbers
Fitness	Sum of absolute errors
Selection	Tournament selection (size 30)
Initialisation	Standard CGP initialisation
Population	2160 individuals (4 demes of 540)
Mutation	All genes with probability 0.05
Crossover	None
Termination	30 generations
Inputs per individual	1
Arity of functions	2
Nodes per row	30
Nodes per column	1
CGP levels back	∞
CGP levels forward	∞
CGP self loops allowed	True
Iterations in evaluation	Varied in experiments

Table 6: A table summarising the parameters of the symbolic regression GP runs.

connected to the output and most of the nodes can be disregarded during evaluation. Unfortunately cyclic CGP individuals appear to be more prone to use a high proportion of their nodes, so smaller individuals of 30 nodes were used.

Since the first generation of the GPU run is likely to be relatively slower than that of the CPU, increasing the number of generations would be likely to improve the acceleration demonstrated by the GPU. A run of 30 generations was chosen to balance showcasing the GPU's abilities with ensuring that the experiment could be run in a sensible amount of time.

The aim of this investigation was to determine the ability of a population-parallel GPU implementation to accelerate cyclic graph evaluation. Table 7 shows the results of runs using the CPU architecture and these values are depicted in Figures 18 and 19. Table 8 shows the equivalents for the GPU architecture and these values are depicted in Figures 20 and 21.

With 512 testcases and 160 iterations, the GPU implementation is able to perform the whole run 175.703 times faster than is a CPU implementation. With only 30 nodes, 32 testcases, 10 iterations, 30 generations and 540 individuals per deme, the GPU evaluator ran 8.247 times faster than the CPU evaluator.

The rates for the CPU evaluator seem rather low despite it being coded as a set of tightly nested loops. This might support the hypothesis (indicated in Section 4.2 and Section 4.4.3) that the memory requirements of cyclic GP make it hard to implement

efficiently without specifically designing the architecture to allow efficient memory access.

4.6 Summary and Contribution

The most successful previous attempts to accelerate GP with a GPU have used a data-parallel approach on very large data sets but they have proved less effective on more modest configurations.

This work proposed cyclic genetic programming as another candidate system for acceleration. A cyclic CGP system was accelerated using a population-parallel evaluator. On a realistic configuration (30 nodes per individual, 400 individuals, 512 testcases and 160 iterations per evaluation) the GPU architecture ran 175.703 times faster than the single–core CPU equivalent when measured over the whole run. Execution time has previously been a major obstacle to research on cyclic GP but this work demonstrated that this no longer need be the case.

Three factors are likely to have played a significant part in this result. Firstly, the GPU is particularly powerful at manipulating floating point numbers, making each core very efficient at this sort of task. Secondly, the GPU is highly optimised for parallel processing and care was taken when designing the thread layout to follow the guidelines on how to allow these optimisations to be most effective. Thirdly, as described in Section 4.4.3, cyclic GP has extra memory constraints which are likely to have caused access inefficiencies in the CPU implementation but which should have been tackled more effectively in the GPU implementation by exploiting the various types of on-chip memory. It should be noted that these suggestions are only speculation and it is extremely difficult to be certain of the precise contributions of these factors.

This work made a novel contribution in tackling cyclic GP for a population-parallel GPU evaluation. This introduced strong constraints on the use of shared memory and required a new architecture capable of dividing the evaluation of a single individual and single testcase over multiple threads. This required a kernel capable of handling this (through appropriate use of memory and synchronisations). It also required a `ThreadPolicy` to construct a `ThreadPlan` for specific batches of programs to be evaluated.

As the system has been extended and made more robust since the publication of the 2009 paper [49], the results appear to have diminished slightly despite the installation of slightly more powerful cards. Future work may investigate this to see if the former results can be restored.

The use of CUDA on Linux can pose problems due to an unfortunate interaction with the X windows system. If a CUDA job runs for longer than five seconds, the X server may try to kill it. This only occurs when the X server is attached to the card in question. To circumvent these issues, a cheap graphics card is used to drive the

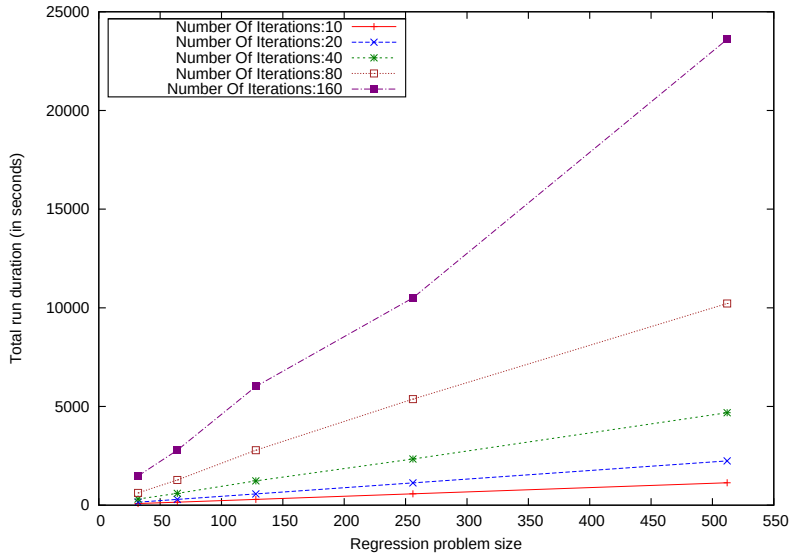


Figure 18: Total run duration in seconds for the CPU implementation over varying number of testcases and number of iterations.

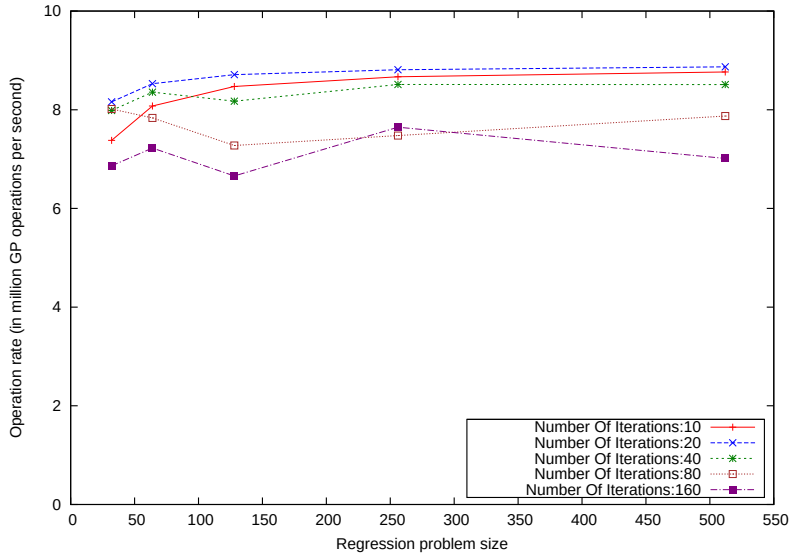


Figure 19: Operation rate (in million GP operations per second) for the CPU implementation over varying number of testcases and number of iterations.

Iterations	Number of testcases				
	32	64	128	256	512
10	7.380 mo/s [±0.016] 84.293 s [±0.185]	8.075 mo/s [±0.015] 154.073 s [±0.278]	8.472 mo/s [±0.018] 293.741 s [±0.622]	8.667 mo/s [±0.010] 574.186 s [±0.690]	8.766 mo/s [±0.013] 1135.480 s [±1.729]
20	8.160 mo/s [±0.016] 152.485 s [±0.299]	8.528 mo/s [±0.020] 291.805 s [±0.687]	8.710 mo/s [±0.018] 571.364 s [±1.208]	8.810 mo/s [±0.018] 1129.851 s [±2.273]	8.870 mo/s [±0.017] 2244.392 s [±4.178]
40	7.985 mo/s [±0.134] 312.553 s [±5.476]	8.357 mo/s [±0.156] 597.956 s [±13.015]	8.172 mo/s [±0.251] 1231.947 s [±45.730]	8.511 mo/s [±0.087] 2341.337 s [±24.547]	8.510 mo/s [±0.091] 4683.970 s [±51.079]
80	8.015 mo/s [±0.157] 623.668 s [±14.185]	7.834 mo/s [±0.179] 1277.677 s [±31.434]	7.276 mo/s [±0.279] 2784.065 s [±125.774]	7.477 mo/s [±0.216] 5377.222 s [±181.440]	7.872 mo/s [±0.240] 10220.635 s [±345.618]
160	6.864 mo/s [±0.300] 1480.404 s [±69.943]	7.224 mo/s [±0.245] 2787.479 s [±94.238]	6.658 mo/s [±0.162] 6016.100 s [±147.782]	7.646 mo/s [±0.209] 10504.058 s [±332.263]	7.014 mo/s [±0.379] 23601.420 s [±1684.645]

Table 7: The results for the CPU implementation in terms of operation rate (in million GP operations per second) and total run duration (in seconds) over varying number of testcases and number of iterations.

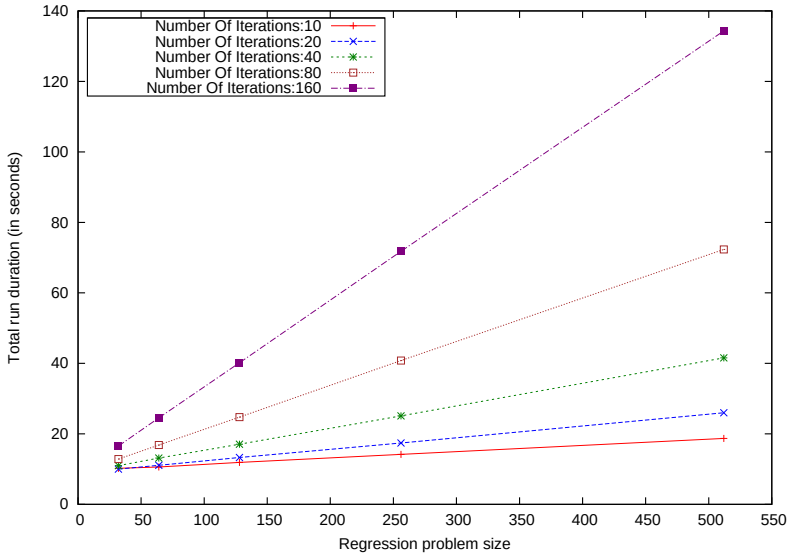


Figure 20: Total run duration in seconds for the GPU implementation over varying number of testcases and number of iterations.

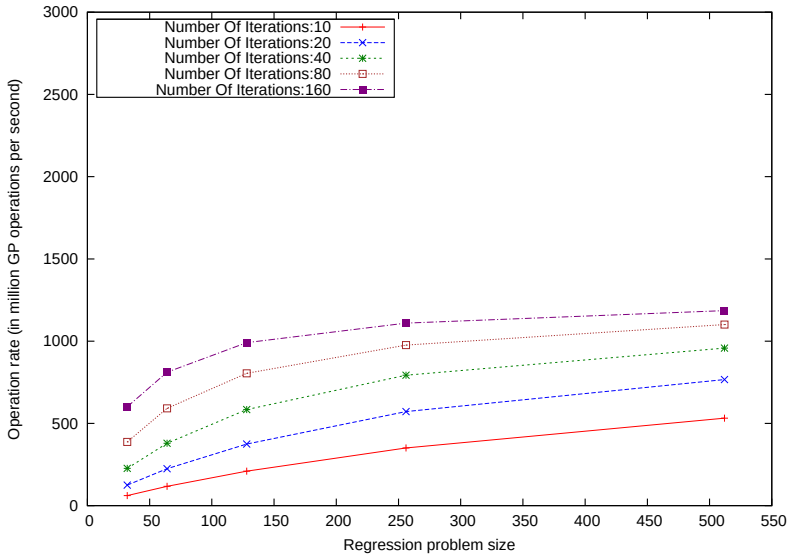


Figure 21: Operation rate (in million GP operations per second) for the GPU implementation over varying number of testcases and number of iterations.

Iterations	Number of testcases				
	32	64	128	256	512
10	60.915 mo/s [±0.552] 10.221 s [±0.092] 8.247 ×	117.237 mo/s [±0.663] 10.616 s [±0.060] 14.513 ×	209.402 mo/s [±1.154] 11.887 s [±0.065] 24.711 ×	350.799 mo/s [±0.665] 14.187 s [±0.027] 40.473 ×	531.932 mo/s [±1.019] 18.712 s [±0.036] 60.682 ×
20	124.572 mo/s [±1.217] 9.997 s [±0.097] 15.253 ×	224.756 mo/s [±1.660] 11.077 s [±0.081] 26.343 ×	374.142 mo/s [±1.975] 13.305 s [±0.070] 42.944 ×	571.876 mo/s [±2.495] 17.408 s [±0.076] 64.904 ×	766.200 mo/s [±2.453] 25.984 s [±0.082] 86.376 ×
40	226.496 mo/s [±1.203] 10.989 s [±0.058] 28.442 ×	379.469 mo/s [±2.939] 13.123 s [±0.100] 45.565 ×	583.930 mo/s [±3.983] 17.053 s [±0.118] 72.242 ×	793.110 mo/s [±2.710] 25.102 s [±0.085] 93.273 ×	957.914 mo/s [±2.639] 41.565 s [±0.115] 112.690 ×
80	387.249 mo/s [±2.628] 12.857 s [±0.087] 48.508 ×	591.747 mo/s [±3.894] 16.828 s [±0.112] 75.926 ×	804.421 mo/s [±1.981] 24.748 s [±0.061] 112.497 ×	976.014 mo/s [±2.437] 40.794 s [±0.101] 131.814 ×	1100.948 mo/s [±1.974] 72.327 s [±0.130] 141.311 ×
160	601.118 mo/s [±3.203] 16.563 s [±0.086] 89.380 ×	811.468 mo/s [±3.486] 24.536 s [±0.105] 113.608 ×	991.219 mo/s [±2.177] 40.168 s [±0.088] 149.773 ×	1109.425 mo/s [±1.890] 71.775 s [±0.122] 146.347 ×	1185.576 mo/s [±1.206] 134.326 s [±0.137] 175.703 ×

Table 8: The results for the GPU implementation in terms of operation rate (in million GP operations per second) and total run duration (in seconds) over varying number of testcases and number of iterations. Each entry is followed by the speedup the result represents over the equivalent CPU result in Table 7.

two displays so that the other two cards can be devoted to compute tasks and are not subject to attacks from the X server.

The evaluator can also process directed acyclic graphs and trees but is not expected to be as efficient as GPU evaluators specifically written to handle those structures. This is not a priority as cyclic graphs require much more computational resources.

The architecture described cannot handle individuals of more than 2048 nodes (and in practice might begin to encounter problems at approximately 1500 nodes due to other use of the memory). This can be alleviated with code that strips out any nodes which cannot affect the output.

It is expected that as parallel computing develops, other evaluators for different technologies may be added to the current repertoire. Perhaps this will become a common practice amongst EC researchers.

5 Improving GPU Usage

5.1 Introduction

This chapter tackles the second objective outlined in Section 1.4: to take the acceleration of one machine further by improving the interaction between the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU) and by using a second GPU. This is important because, as will be seen, the work in the preceding chapter can leave the CPU and the GPU idle for much of the run, thus wasting valuable computational resources. This chapter conducts an investigation into whether this can be done better. The investigation is performed in incremental steps:

- **Step one**: Section 5.2 investigates accelerating the run further by carrying out GPU execution, CPU execution and data transfer in parallel.
- **Step two**: Section 5.3 investigates accelerating the run further by using a second GPU and a second CPU core.
- **Step three**: The first two steps use sub-populations, or *demes* to achieve their speed improvements. Section 5.4 investigates the time cost incurred in implementing transfers between demes and the extent to which this can be reduced with a more sophisticated approach.

It is important to emphasise that although the techniques described in this section will be examined in the context of the population-parallel system described in Chapter 4, they should be applicable to any GPU evaluation system for Evolutionary Computation (EC).

Work in this area has often concentrated on the evaluation stage because this typically accounts for the majority of the execution time of a full Genetic Programming (GP) run. However, once the evaluation is accelerated, the time spent on the other parts of the run becomes more significant. For this reason, very impressive reductions in evaluation time may lead to much less impressive reductions in overall run time.

It is possible to execute GPU kernels in parallel with CPU threads: when a CPU thread launches a GPU kernel, execution returns to it almost immediately whilst the GPU is just starting work. The GPU and the CPU are thought to be the critical resources so they should work in parallel to minimise the time that they spend waiting for new work. This parallelism has previously been exploited to further accelerate the evaluation of an EC system by having the CPU and GPU simultaneously evaluating separate demes [21]. Here it is used to allow the GPU to evaluate one deme at the same time as the CPU is performing the other tasks on another. This approach is experimentally assessed and is found to further improve the overall acceleration by a further 1.806 times.

The second step builds on the first by adding the use of a second GPU. This in turn requires the use of a second CPU core so the technique requires work to allow the evolutionary algorithm to be safely distributed over multiple CPU threads. The technique is experimentally assessed and found to reduce the overall run time by up to 1.982 times. In combination, steps one and two are found to reduce the overall run time by up to 3.292 times.

The use of demes to achieve these techniques might raise concerns about the cost of the deme transfers that they introduce. The third step involves building a deme transfer system that can operate in the multi-threaded environment and then extends it with a "smart" mode for carrying out the deme transfers. The effects of using this system on various topologies are explored through the use of timelines. Experimental assessment reveals that, even with a particularly high deme-transfer frequency, the time cost of the deme transfers is small and that this small cost can be mitigated through the use of the "smart" deme transfers. The largest increase seen is 13.449% and this falls to an increase of 1.742% through the use of smart transfers.

5.2 Step One: Parallel GPU Execution, CPU Execution and Memory Transfer

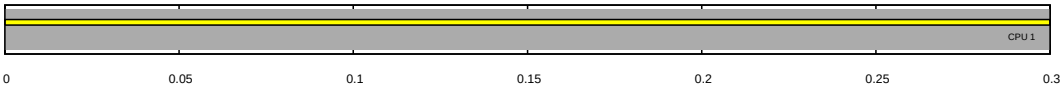
This step further enhances the acceleration by exploiting the parallelism between the CPU and GPU to allow each to work on separate tasks simultaneously. This is achieved through the use of demes.

5.2.1 Description of the Step

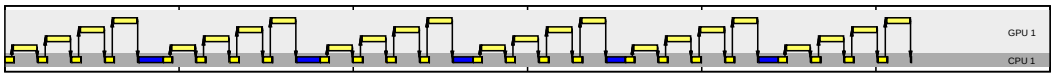
In Chapter 4, the GPU greatly accelerated GP evaluation (to 175.703 times faster than a single–core CPU equivalent when measured over the whole run). Figure 22(a) depicts a timeline associated with a standard CPU implementation and Figure 22(b) depicts a timeline for a standard GPU implementation. The extent of the acceleration means that the scale used for the other timelines only allows Figure 22(a) to depict the first CPU evaluation partially.

Section 5.1 explains that accelerating the evaluation in this way only tackles part of the problem because the other tasks of the run become more significant in the total run time. The same section also explains that synchronous access to the GPU and to the data transfer mechanism only utilises part of the available parallel computing power. This step aimed to use the remaining part of the power to help attack the remaining part of the problem.

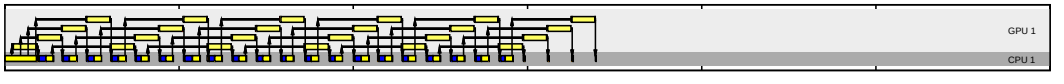
This is possible by dividing the population into demes. When one deme is sent for evaluation by the GPU, execution returns to the CPU which can then simultaneously process another deme. This work involves the CPU updating this other deme in preparation for its next evaluation, based on the results of its last evaluation. An example



(a) Timeline for standard CPU implementation



(b) Timeline for GPU accelerated implementation



(c) Timeline for GPU–CPU parallel implementation (step one)

Figure 22: By using the CPU and GPU in parallel—as in Subfigure 22(c)—run times can be reduced even further. The timelines' runs all used 6 generations, 256 testcases and 20 iterations. See Section 4.4.5 for a description of timelines.

timeline depicting this approach is shown in Figure 22(c). For most generations, the demes do not interact so they can be handled independently. Every few generations, this extra parallelism is halted so that evaluation results can be collected for the whole population and used to conduct migration between the demes. The extra parallel procedure is then restarted. Section 5.4 analyses the cost of these transfers and investigates a strategy to mitigate it.

Section 2.1.10 discusses some of the context of this work. Before the work was first described in a 2009 paper [49], Garnica et al had investigated having a GPU and a CPU each completely process their own demes. This has the disadvantage of having the demes running at different rates because the GPU's evaluation is much faster than the CPU's. It has the further disadvantage of failing to play to the relative strengths and weaknesses of the two processors. It requires writing complicated code for the GPU for tasks that could be performed as effectively on the CPU whilst also having the CPU perform tasks that are performed much faster by the GPU. In contrast, the architecture constructed for this step uses each processor to perform those tasks to which it is best suited.

Since this work was first described in the 2009 paper [49], others have explored running a distributed (deme-based or cellular) model entirely on the GPU [53] [74]. Again, this fails to exploit the strengths of the CPU and now wastes the CPU's computational power by having it sit idle. These approaches also suffer from complications arising from the difficulty of synchronising between parallel GPU tasks. These systems tend to use an asynchronous approach to transfers between sub-populations and this means that the algorithm is changed and that reproducibility is made extremely

difficult. Worse, Compute Unified Device Architecture (CUDA) implementations of this type are highly vulnerable to changes in future CUDA software and hardware because the nVidia documentation urges users to make no assumptions about the ordering and parallelisation involved in the execution of multiple blocks of threads in a single launch.

The architecture used in this work not only avoids these problems but also goes further by also performing data transfers to and from the graphics card in parallel with the GPU computation and the CPU computation. This is achieved by associating each deme with a CUDA stream and then inserting the sequence of requests for GPU data transfers and GPU executions in the appropriate stream. This information can then be used by CUDA to determine which data transfers and kernel executions must be run sequentially and which can be run in parallel. Before the CPU code uses any results from the GPU, it calls a CUDA function to ensure that all preceding operations in the relevant stream are complete.

Note that the CUDA stream handling is not perfect: if a call is blocked by preceding tasks, then it also blocks all calls of the same type (kernel execution or memory copy) that were launched behind it, even if they were launched in different streams. For example, consider the scenario depicted in Figure 23: client code initiates a kernel and then an asynchronous memory copy in stream one and then an asynchronous memory copy in stream two. In this case, the memory copy in stream two will not begin until both the kernel and the memory copy in stream one have completed because the memory copy in stream one will block whilst the kernel is running and this will block the copy in stream two. There is no algorithmic reason for this constraint; it is an artifact of the current nVidia design.

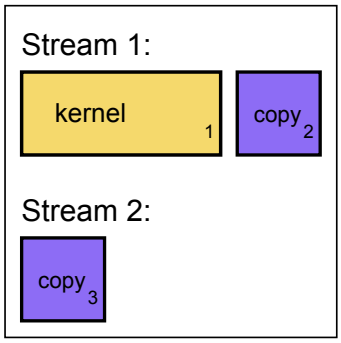


Figure 23: CUDA's handling of streams is not perfect. If these items are placed in these streams in the order of their numbers, the copy in stream 2 does not begin until the copy in stream 1 has completed. This is because the copy in stream 1 is blocked by the preceding kernel and this blocks all operations of the same type until it is complete.

GPUs of compute capability 2.0 or higher allow a copy from page-locked host memory to device memory (i.e. CPU memory to GPU memory) to run concurrently with a copy from device memory to page-locked host memory (i.e. they allow copies to and from the graphics card to run concurrently).

5.2.2 Implementation

The most natural scheduling is the one depicted in Figure 22(b): submit and then immediately wait to gather each deme in turn and repeat for the required number of generations. As Figure 22(b) shows, this means the CPU and GPU never work in parallel. Even repeatedly submitting and then gathering all of the demes, would not maximise parallelisation. It is better to submit all of the demes and then for each generation, gather and immediately resubmit each deme and finally gather each deme at the end of the run. In this way, demes are submitted to the GPU as soon as possible to keep the GPU as busy as possible and also the CPU does not sit idle waiting for the GPU to return results if it has submitting work it can do. In fact, though this strategy is an improvement, it does not handle the deme transfers as well as it should and a more advanced mechanism is explored in Section 5.4.2.

The code which implements evaluation is kept as independent as possible from the evolution code so that each instance of an `Evaluator` class just performs evaluation of individuals and does not need to keep track of things such as generations, demes and deme transfers. Similarly, the evolution code keeps track of these things but as a client of an `Evaluator` class does not need to concern itself with any of the details of the evaluation. However, since the evaluation submission may be asynchronous, the evolution code must be able to query its `Evaluator` about a specific submission. For this reason, a submission call to an `Evaluator` returns an `EvaluatorSubmissionHandle` object which the client may use to query the readiness of results and then to request them.

Within the `CudaEvaluator` class, the resources and parameters associated with performing a single submission are held within a `CudaResourceSet`. Each such `CudaResourceSet` contains the number of programs, the number of testcases, the CUDA stream being used, a set of pairs of memory handles and the index of the CUDA texture set to use.

Each of the CUDA resource objects held within a `CudaResourceSet` (a `CudaTextureSetIndex`, a `CudaStream` and several `CudaMemoryPairs`) is designed to hold a resource under the C++ Resource Acquisition Is Initialisation (RAII) idiom [83] so that the resources will be automatically freed (and freed in the correct order) when the `CudaResourceSet` goes out of scope. The steps involved in this are shown in a Unified Modelling Language (UML) sequence diagram in Figure 24.

A `CudaMemoryPair` holds a piece of page-locked host memory (i.e. CPU memory) and an equally sized piece of CUDA device memory (i.e. GPU global memory). The host memory must be page-locked in order to allow the data-transfer to happen in parallel with the execution of CUDA kernels. Using one class to hold both the page-locked host memory and the device memory makes it very easy to copy data between them. From the perspective of code which uses the `CudaMemoryPair` class, the allocation of each type of memory is done in parallel and it can be viewed as a special type

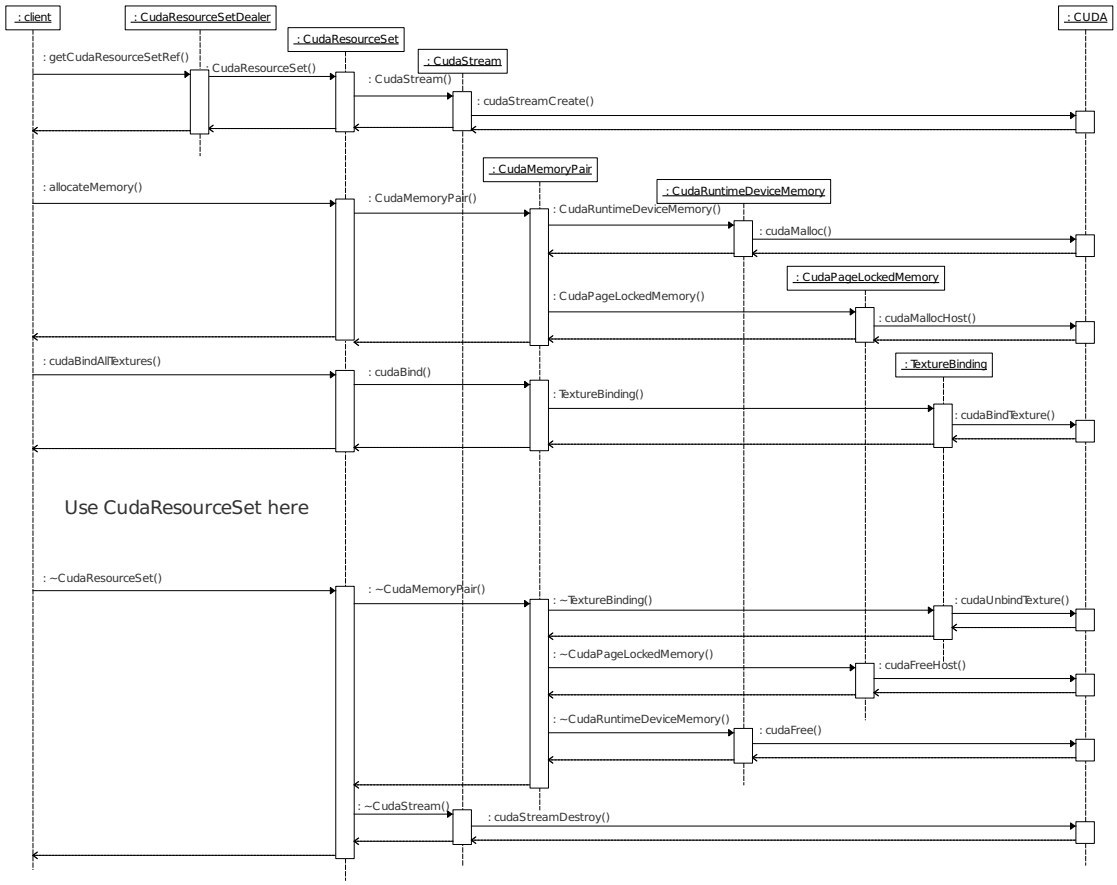


Figure 24: The `CudaResourceSet` and `CudaResourceSetDealer` classes encapsulate much of the work in allocating and freeing CUDA resources. The client requests a `CudaResourceSet` from the `CudaResourceSetDealer`, uses its resources via simple methods and then lets it go out of scope. The `CudaResourceSet` does the work of acquiring the CUDA resources (the stream, the device memory, the page-locked host memory and the texture binding) and uses its destructor to ensure that these resources are returned and in the correct order.

of double-memory.

A `CudaMemoryPair` also attempts to reuse memory in order to reduce the time that is wasted on allocations and deallocations. This works in the following way. In addition to holding the memory itself, a `CudaMemoryPair` object records two numbers: the amount of memory that was most recently requested and the size of the block of memory that was actually allocated. When client code requests a `CudaMemoryPair` to allocate memory for the first time, it actually allocates some multiple of that requested amount.

`CudaResourceSets` are reused so if the `CudaMemoryPair` is later reused by another job (as happens many times in a normal evolutionary run) and its client requests another memory allocation, the `CudaMemoryPair` can check if it already holds a sufficient amount of memory. If so, it will just reuse that previously allocated memory. If a deallocation and reallocation is necessary, the amount of memory requested will again be

multiplied to determine the amount of memory that should actually be allocated. This means that in practice, allocations or deallocations are rarely required except at the very start and end of the evolutionary run.

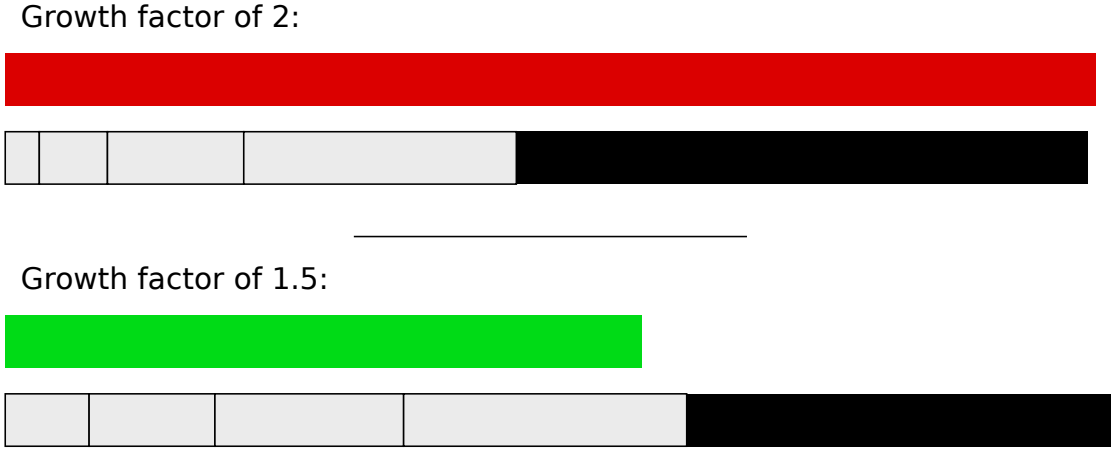


Figure 25: Figure showing the effects of two different reallocation growth factors on the reuse of memory when using a first-fit allocator. The grey blocks represent previously used memory, the black blocks indicate the currently active memory and the coloured blocks indicate the amount memory to be allocated in the next step, coloured according to whether the block will fit in the previously used space (green) or not (red). The upper part illustrates the problem of using a growth factor of 2: after several reallocations, the previously used space is not large enough to contain the next allocation and further reallocations will never solve this problem. In the lower part, a growth factor of 1.5 is used and memory gets reused after five allocations.

This exponential growth strategy is used in other areas of computer science such as in many implementations of the C++ STL vector container [84]. In the worst case of a sequence of requests for incrementally increasing sizes of memory up to size N, the exponential growth strategy performs O(N) allocations and wastes O(N) space [84]. The present code uses a multiplying factor of 1.5 as recommended in an article on reallocation growth strategies [42]. The article explains that, depending on the memory allocator's strategy, a multiplying factor of 2 can cause problems. The argument is as follows.

Consider a first-fit allocation strategy and say the first allocation requires a bytes, then after n further allocations the contiguous stretch of n previously allocated memory blocks will occupy $a(2^n - 1)$ bytes but the next allocation will require a larger stretch of $a.2^{n+1}$ bytes and so will not fit into this empty space. Since this argument holds for all values of n, the earlier memory is never reused and the sequence of allocated blocks moves endlessly up the memory space. This scenario is depicted in the upper part of Figure 25.

How low must the factor be reduced to avoid this problem? It turns out that the

cut-off is the golden ratio ($\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$) [2] so any value between 1 and φ suffices [42]. A value of 1.5 is widely used and is used here. The lower part of Figure 25 shows that the fifth reallocation then fits into the memory left from the first four.

In this context, the reallocations are often associated with refreshing the data in memory. This might make it possible to deallocate the current block of memory (depicted as a black block in each part of Figure 25) and hence potentially include it in the reallocation. This would mean that the reallocation multiplier could be higher than φ (provided it were kept below two). This current strategy was not found to be a problem (and reallocations are rarely needed anyway) so this possibility was not investigated further.

5.2.3 Experiments and Results

The aim of this first step was to investigate further accelerating the run by carrying out GPU execution, CPU execution and data transfer in parallel. The effect of this technique was assessed by using the technique whilst repeating the experiments as described in Section 4.5.2.

Table 9 shows the results for the parallel GPU–CPU architecture over the full run and compares them to the equivalent CPU results. The durations and operation rates as measured over the full run are not only included in Table 9 but are also shown in Figures 26 and 27 respectively.

The best overall acceleration was observed with 128 testcases and 160 iterations: the total run time was reduced by 191.248 times over the CPU implementation.

The best improvements over the original acceleration occurred with 64 testcases and 80 iterations: the total run time was reduced by a further 1.806 times. Figure 22(c) (repeated in Figure 28(b)) illustrates how this increased parallelism works using times recorded from a real run. The figure shows that data transfer is a very small factor in the total run time. This is not surprising as cyclic GP performs multiple iterations on each individual that is transferred.

[2] An argument why this might be true (rather than a rigorous proof) proceeds as follows. Observe that if the initial allocation is $a > 0$, the factor is $k > 1$ and there have been $n + 1$ allocations, then the condition that the next allocation will fit into the contiguous block of previous allocations is equivalent to:

$$\begin{aligned} a + ak + ak^2 + \ldots + ak^{n-1} &\geq ak^{n+1} \\ \Rightarrow \frac{a(k^n - 1)}{k - 1} &\geq ak^{n+1} \\ \Rightarrow k^2 - k - 1 + \frac{1}{k^n} &\leq 0 \\ \Rightarrow \left(k - \frac{1+\sqrt{5}}{2}\right)\left(k - \frac{1-\sqrt{5}}{2}\right) + \frac{1}{k^n} &\leq 0 \end{aligned}$$

The limit of the left hand side of this equation as $n \to \infty$ is $\left(k - \frac{1+\sqrt{5}}{2}\right)\left(k - \frac{1-\sqrt{5}}{2}\right)$ and the solutions of equating this to zero are $k = \frac{1-\sqrt{5}}{2}$ and $k = \frac{1+\sqrt{5}}{2}$, and only the latter of these meets the initial assumption that $k > 1$.

Iterations	Number of testcases				
	32	64	128	256	512
10	64.888 mo/s [±0.475] 9.592 s [±0.070] 8.788 ×	127.165 mo/s [±0.789] 9.788 s [±0.060] 15.741 ×	250.309 mo/s [±1.233] 9.943 s [±0.049] 29.542 ×	478.616 mo/s [±2.989] 10.402 s [±0.065] 55.200 ×	900.238 mo/s [±7.303] 11.063 s [±0.085] 102.638 ×
20	136.977 mo/s [±1.556] 9.095 s [±0.103] 16.766 ×	264.161 mo/s [±1.640] 9.423 s [±0.059] 30.967 ×	523.919 mo/s [±2.396] 9.501 s [±0.044] 60.137 ×	1000.557 mo/s [±12.694] 9.964 s [±0.127] 113.393 ×	1230.926 mo/s [±1.013] 16.172 s [±0.013] 138.783 ×
40	271.465 mo/s [±3.537] 9.183 s [±0.128] 34.036 ×	539.110 mo/s [±5.441] 9.241 s [±0.093] 64.707 ×	1036.073 mo/s [±7.491] 9.612 s [±0.069] 128.168 ×	1245.314 mo/s [±0.883] 15.985 s [±0.011] 146.471 ×	1256.684 mo/s [±0.683] 31.681 s [±0.017] 147.848 ×
80	554.578 mo/s [±3.419] 8.977 s [±0.055] 69.474 ×	1069.559 mo/s [±13.048] 9.320 s [±0.114] 137.090 ×	1250.971 mo/s [±1.198] 15.913 s [±0.015] 174.955 ×	1263.611 mo/s [±0.898] 31.508 s [±0.022] 170.662 ×	1268.945 mo/s [±1.408] 62.751 s [±0.070] 162.876 ×
160	1076.508 mo/s [±7.528] 9.250 s [±0.065] 160.044 ×	1252.781 mo/s [±1.061] 15.890 s [±0.013] 175.423 ×	1265.630 mo/s [±0.731] 31.457 s [±0.018] 191.248 ×	1274.696 mo/s [±0.614] 62.467 s [±0.030] 168.154 ×	1276.595 mo/s [±1.323] 124.749 s [±0.129] 189.191 ×

Table 9: The results for the CPU-GPU parallel implementation in terms of operation rate (in million GP operations per second) and total run duration (in seconds) over varying number of testcases and number of iterations. Each entry is followed by the speedup the result represents over the equivalent CPU result in Table 7.

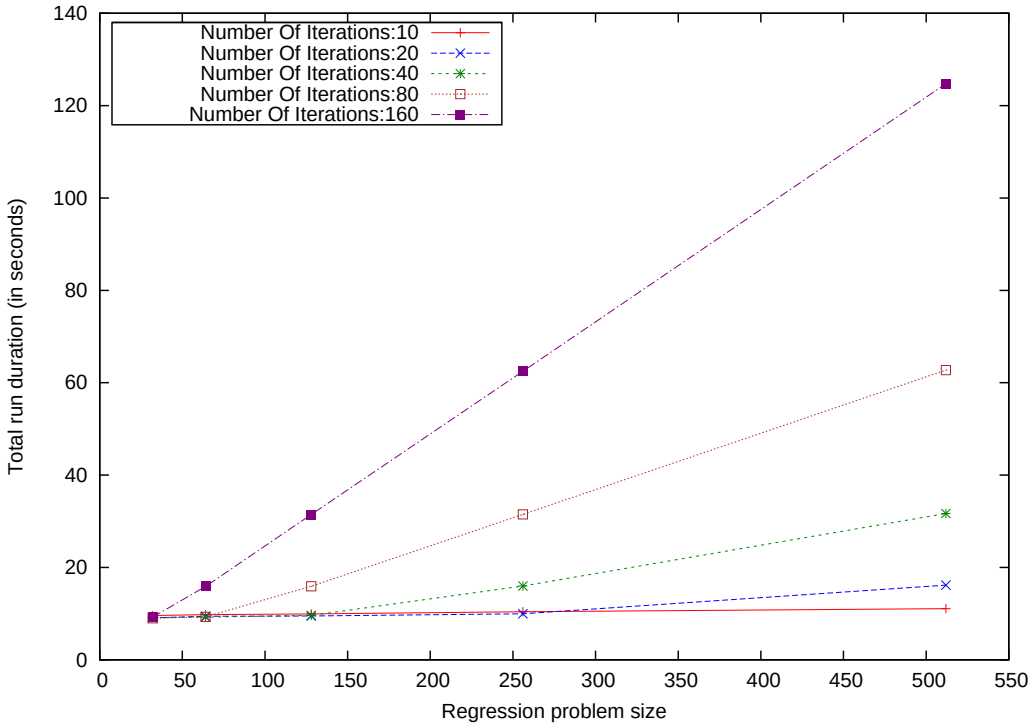


Figure 26: Total run duration (in seconds) for the parallel CPU-GPU implementation over varying number of testcases and number of iterations.

The best reductions are expected to occur when the CPU and GPU take similar amounts of time on each deme. Through optimisation of whichever step is taking the longest, it should then be possible to get close to halving the run time.

5.3 Step Two: Using Two GPUs

5.3.1 Description of the Step

Chapter 4 describes the use of a GPU to accelerate cyclic GP evaluation, as depicted in Figure 28(a). Section 5.2 describes the use of the parallelism between the CPU and GPU to improve this acceleration further, as depicted in Figure 28(b). This step makes the final addition to the architecture: a second GPU. The system of demes already described in Section 5.2.1 provides a natural way to divide evaluation work between the GPUs. This approach is depicted in Figure 28(c).

5.3.2 Implementation

The use of multiple GPUs is complicated by the fact that CUDA requires a different thread to access each GPU. This constraint has been relaxed in v4.0 of the CUDA toolkit, released in May 2011. This is too late to be used here; a paper describing the basics of this work was published in 2009 [49]. Furthermore, although the use of multiple threads adds some complications in a few areas of code, it also makes the architecture more powerful because it exploits further CPU cores. The complications are not extensive because the demes are largely independent and because each thread has its own CUDA resources. Nevertheless, it is worth outlining the software design that addresses these issues.

As discussed in Section 5.2.2, the `CudaEvaluator` associates each job with a `CudaResourceSet` which contains the details of the CUDA resources that are required to perform the job. A job's `CudaResourceSet` is referenced by its `EvaluatorSubmissionHandle`. These handles are returned to the `CudaEvaluator`'s client whenever it submits a new job. To acquire the `CudaResourceSet` for a given `EvaluatorSubmissionHandle`, the `CudaEvaluator` passes the handle to the `getCudaResourceSetRef()` method of the singleton `CudaResourceSetDealer`. This method determines whether a `CudaResourceSet` has previously been assigned to the job and if so returns it, otherwise it allots and returns a new one.

When all processing is complete for the job, the relevant `EvaluatorSubmissionHandle` is passed to the `returnCudaResourceSet()` method of the singleton `CudaResourceSetDealer`. This `CudaResourceSet` will then be freed up and kept by the `CudaResourceSetDealer` for future reuse.

An index is required to identify the set of textures to be used for a given job and this index is stored in a `CudaTextureSetIndex` object in the relevant `CudaResourceSet`. When a particular job is assigned its `CudaResourceSet`, a `CudaTextureSetIndex` is cho-

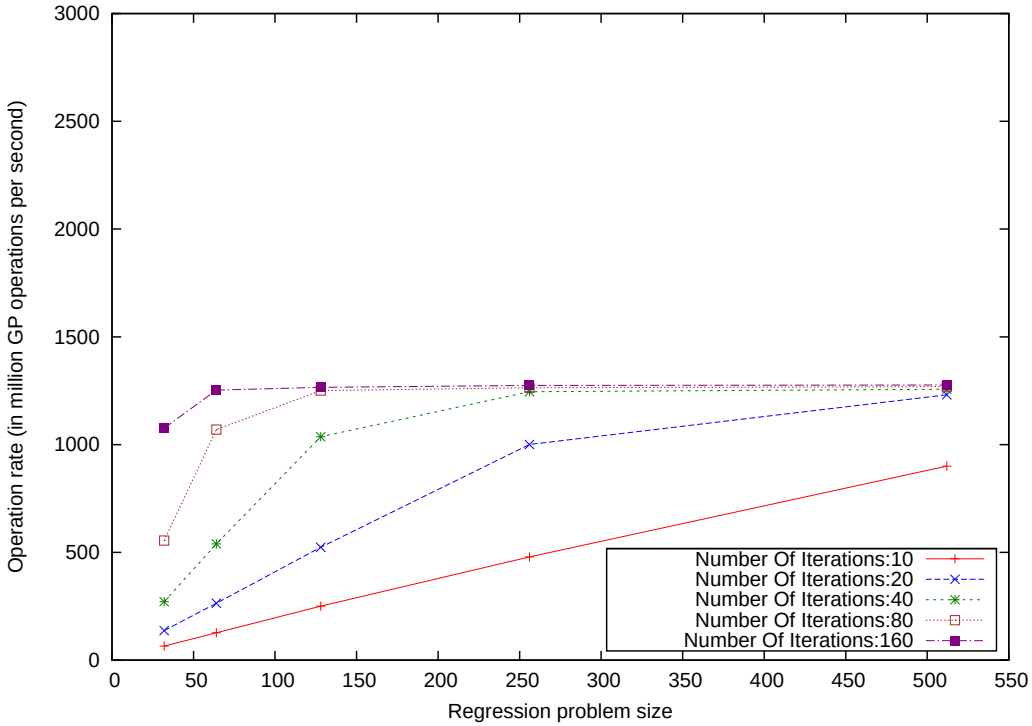
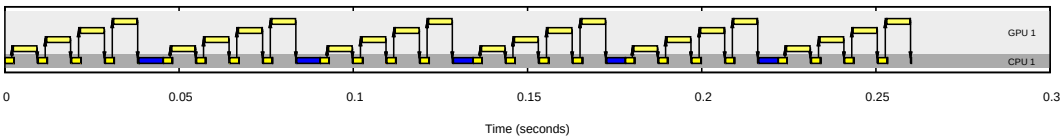
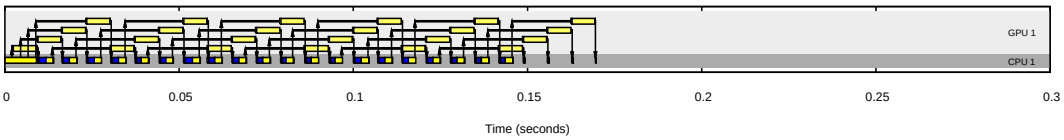


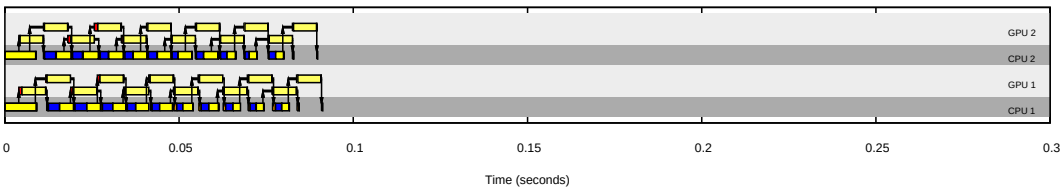
Figure 27: Operation rate (in million GP operations per second) for the parallel CPU-GPU implementation over varying number of testcases and number of iterations.



(a) Timeline for GPU accelerated implementation



(b) Timeline for GPU–CPU parallel implementation (step one)



(c) Timeline for GPU–CPU implementation using two GPUs and two CPU cores (step two)

Figure 28: By using multiple CPU cores and multiple GPUs—as in Subfigure 28(c)—run times can be reduced even further. Subfigures 28(a) and 28(b) depict the previously described GPU acceleration (Chapter 4) and CPU-GPU parallelism (Section 5.2). Subfigure 28(c) shows the additional effect of using two CPU cores and two GPUs. The timelines' runs all used 6 generations, 256 testcases and 20 iterations. See Section 4.4.5 for a description of timelines.

sen for it from the `CudaTextureSetIndexRepository`. A `CudaTextureSetIndex` object automatically returns itself to the `CudaTextureSetIndexRepository` on destruction (but not until the corresponding `CudaMemoryPair` in the `CudaResourceSet` has been destroyed). Figure 29 adds this process into the previous UML sequence diagram of Figure 24. Figure 30 shows the relationships between these various classes.

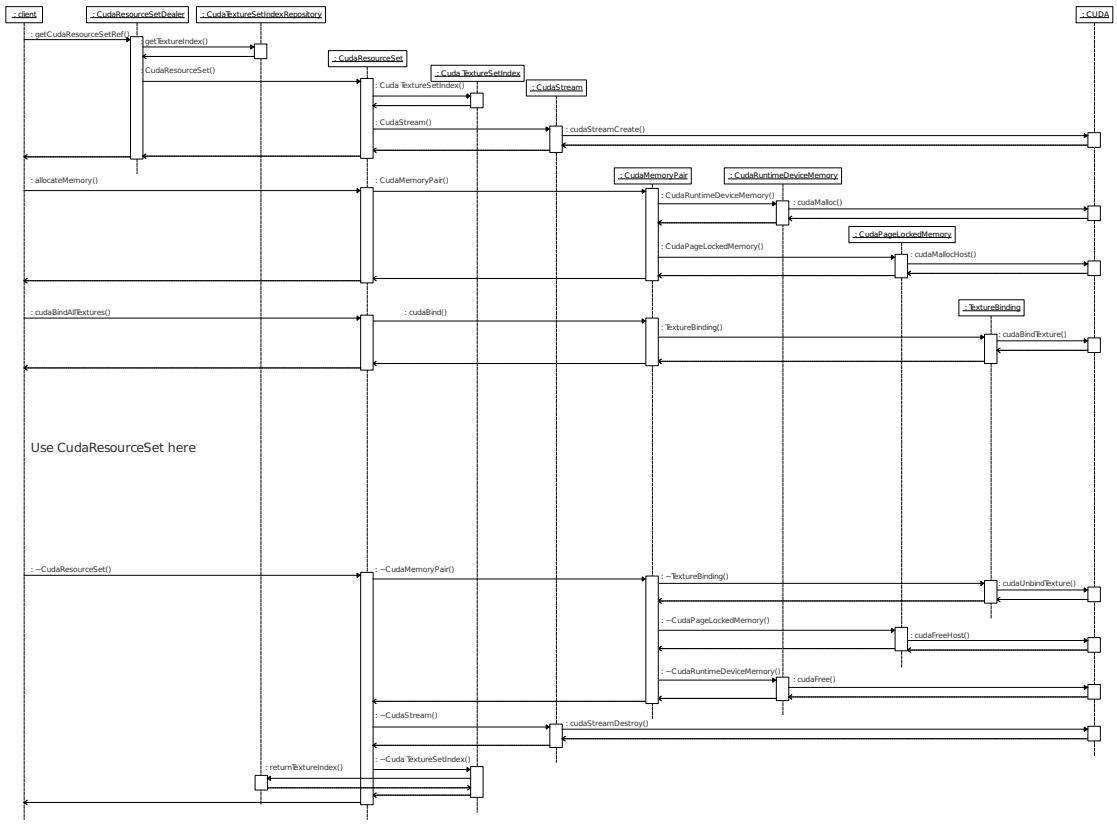


Figure 29: The `CudaResourceSet` setup (previously illustrated in Figure 24) may be extended to also handle the `CudaTextureSetIndex`. Whilst the `CudaResourceSetDealer` is constructing a new `CudaResourceSet`, it acquires a `CudaTextureSetIndex` from the `CudaTextureSetIndexRepository` and uses this to construct the new `CudaResourceSet`. Later on, when the `CudaResourceSet` executes its destructor, it can ensure that the destructor of the `CudaTextureSetIndex` is executed at the correct point, which returns it to the `CudaTextureSetIndexRepository` so that its index may be reused.

A subtle problem arises with regard to the freeing up of resources. On the one hand, performing consecutive runs in one thread using multiple `CudaEvaluators` leads to CUDA resources running out if they are not freed up. On the other hand, simply cleaning up CUDA resources when the `CudaEvaluator` goes out of scope creates a new problem because if its CUDA resources have been used by other threads which have since terminated, they will have already been cleaned up and an attempt to clean them up again will result in an error.

To solve this issue properly, it is necessary to keep track of the different CUDA resources held by the `CudaEvaluator` according to the thread that used them. Further-

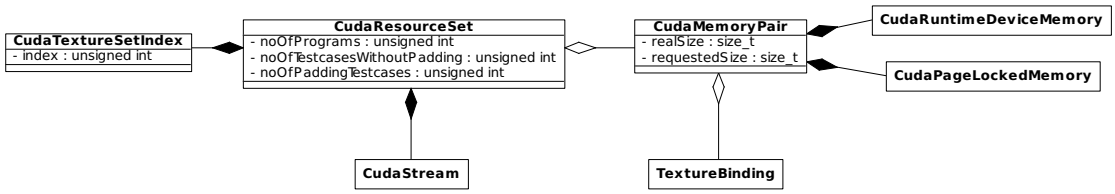


Figure 30: A UML class diagram showing the various resources managed by the `CudaResourceSet` class. A `CudaResourceSet` owns a `CudaTextureSetIndex` and a `CudaStream` and it also has a `CudaMemoryPair` which in turn has a `TextureBinding` and owns a `CudaRuntimeDeviceMemory` and a `CudaPageLockedMemory`. The `CudaResourceSet` has three integer attributes to keep track of its work: `noOfPrograms`, `noOfTestcasesWithoutPadding` and `noOfPaddingTestcases`. The `CudaTextureSetIndex` has an integer attribute for the index. The `CudaMemoryPair` tracks the requests and allocations using two integer attributes: `realSize` and `requestedSize`.

more, it is highly desirable to do this in such a way that they will be automatically cleaned up when they are no longer required. This need is met by the `AccessHandle` class. In informal terms, this can be viewed as a deal offered by the evaluator to its client threads: "I will perform evaluations for you but to do that I have to hold resources specifically for you, so my condition is that you must hold onto this handle throughout, show it to me with every request and let go of it when your last request is complete". Figure 31 is a UML sequence diagram showing how this works in practice, shielding the client from the complexity of multi-threaded CUDA resource management.

More formally, the specifications are as follows:

- There is one `AccessHandle` for each thread and evaluator combination.
- A thread wishing to use a given `Evaluator` must first obtain and hold an `AccessHandle` for it.
- Internally, the `Evaluator` contains an `AccessHandleRegistrar` to keep track of the `AccessHandle`s held by its various client threads.
- The client thread must pass the correct `AccessHandle` with every call to an `Evaluator` (and the correctness of this `AccessHandle` is validated).
- Each client thread may only obtain one `AccessHandle` from each `Evaluator` (and an attempt to obtain any further `AccessHandle`s results in an error).
- When the `AccessHandle` goes out of scope (which will happen when the thread is complete if not before), it automatically signals the `Evaluator` to free any resources being held for that thread.
- A thread that has relinquished its `AccessHandle` for an `Evaluator`, may acquire another one if required.

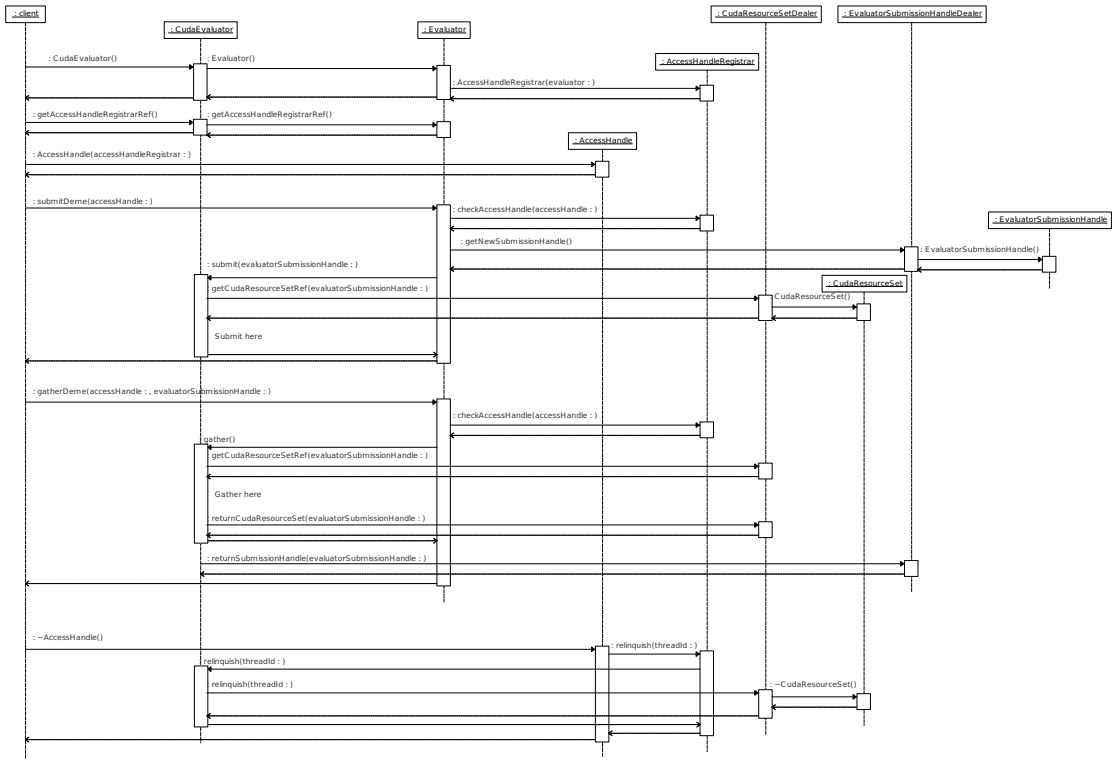


Figure 31: The client code is insulated from much of the complexity of managing CUDA resources. The client constructs (or is passed) a `CudaEvaluator`, uses this to acquire a reference to the `AccessHandleRegistrar`, and then uses this to construct an `AccessHandle`. It may then use this `AccessHandle` for numerous submit and gather calls on the `CudaEvaluator`. When finished, the client can clean up the resources by simply allowing the `AccessHandle` to go out of scope.

The multi-threaded design requires several other classes to be thread-safe. Three of the thread-safe classes have already been described above: `AccessHandleRegistrar`, `CudaResourceSetDealer` and `CudaTextureSetIndexRepository`. The thread-safety of the `DemeTransferManager` is discussed in Section 5.4.2. Four other thread-safe classes (`FitnessRecorder`, `NodeCounter`, `ProgressDisplay` and `Timer`) are used to record and display information about the run but do not directly affect the run itself. The `CpuEvaluator` has its own thread-safety mechanism to allow it to be used in a multi-threaded run.

The two remaining thread-safe classes are the `EvaluatorSubmissionHandleDealer` and the `RandomNumGenDealer`. The `EvaluatorSubmissionHandleDealer` class deals out `EvaluatorSubmissionHandles` in response to requests from multiple threads. The `RandomNumGenDealer` class handles the random number seeds for different threads. Each thread can acquire access to its own random number seed by a call to the singleton `RandomNumGenDealer`. When a new group of threads are created, their random number generators are each seeded in a deterministic way, based on the parent thread's random number generator. This greatly improves reproducibility since it means that the sequence of random numbers in the child threads can be reproduced by simply

setting the right seed in the parent thread before those threads are spawned.

5.3.3 Experiments and Results

Iterations	Number of testcases 32	64	128	256	512
10	124.322 mo/s [±1.048] 5.007 s [±0.043] 16.835 ×	245.357 mo/s [±2.172] 5.075 s [±0.045] 30.359 ×	481.108 mo/s [±4.035] 5.176 s [±0.043] 56.751 ×	914.344 mo/s [±8.175] 5.447 s [±0.049] 105.413 ×	1735.707 mo/s [±13.243] 5.738 s [±0.043] 197.888 ×
20	251.983 mo/s [±0.738] 4.938 s [±0.014] 30.880 ×	492.162 mo/s [±5.115] 5.061 s [±0.052] 57.658 ×	1016.231 mo/s [±9.806] 4.901 s [±0.045] 116.581 ×	1831.333 mo/s [±20.005] 5.441 s [±0.058] 207.655 ×	2376.598 mo/s [±3.042] 8.376 s [±0.011] 267.955 ×
40	493.114 mo/s [±6.237] 5.054 s [±0.062] 61.843 ×	1031.007 mo/s [±11.750] 4.833 s [±0.055] 123.724 ×	1922.307 mo/s [±12.429] 5.180 s [±0.034] 237.828 ×	2404.669 mo/s [±2.653] 8.278 s [±0.009] 282.838 ×	2466.436 mo/s [±2.868] 16.142 s [±0.019] 290.173 ×
80	1001.312 mo/s [±10.650] 4.976 s [±0.053] 125.335 ×	1942.620 mo/s [±13.181] 5.126 s [±0.035] 249.254 ×	2420.000 mo/s [±3.191] 8.226 s [±0.011] 338.447 ×	2475.459 mo/s [±1.600] 16.083 s [±0.010] 334.342 ×	2508.680 mo/s [±2.566] 31.741 s [±0.033] 322.001 ×
160	1936.591 mo/s [±19.929] 5.145 s [±0.052] 287.736 ×	2429.562 mo/s [±2.905] 8.194 s [±0.010] 340.185 ×	2485.024 mo/s [±2.776] 16.021 s [±0.018] 375.513 ×	2520.197 mo/s [±2.920] 31.596 s [±0.037] 332.449 ×	2529.779 mo/s [±3.295] 62.952 s [±0.082] 374.911 ×

Table 10: The results of the multiple GPU and multiple CPU core implementation in terms of operation rate (in million GP operations per second) and total run duration (in seconds) over varying number of testcases and number of iterations. Each entry is followed by the speedup the result represents over the equivalent CPU result in Table 7.

The aim of this first step was to investigate further accelerating the run by carrying out GPU execution, CPU execution and data transfer in parallel. The aim of this second step was to investigate accelerating the run yet further by using a second GPU and a second CPU core. Figure 28(c) illustrates how this increased parallelism works using times recorded from a real run.

It should be noted that from this point to the end of the chapter, many of the results represent an architecture that uses multiple CPU cores and are compared to CPU results representing an architecture with only one. Hence, the comparison should be seen as illustrating how powerful the whole architecture can make one machine rather than how much better GPUs are than CPUs (which this comparison cannot fairly illustrate).

For assessment, the experiments described in Section 4.5.2 were repeated again, this time using both the technique from step one and this technique from step two. The results of this change are shown in Table 10 along with comparisons to the equivalent CPU results. The durations and operation rates as measured over the full run are not only included in Table 10 but are also shown in Figures 32 and 33 respectively.

The best improvement over the original GPU acceleration occurred with 128 testcases and 40 iterations: the total run time was reduced by 3.292 times. The best improvement over the results in step one occurred with 512 testcases and 160 iterations: the total run time was reduced by a further 1.982 times. Perhaps most importantly, the best improvement over the CPU results occurred with 128 testcases and 160 iterations: the total run time was reduced by 375.513 times and the evaluation rate over the full run was 2529.779 million GP operations per second.

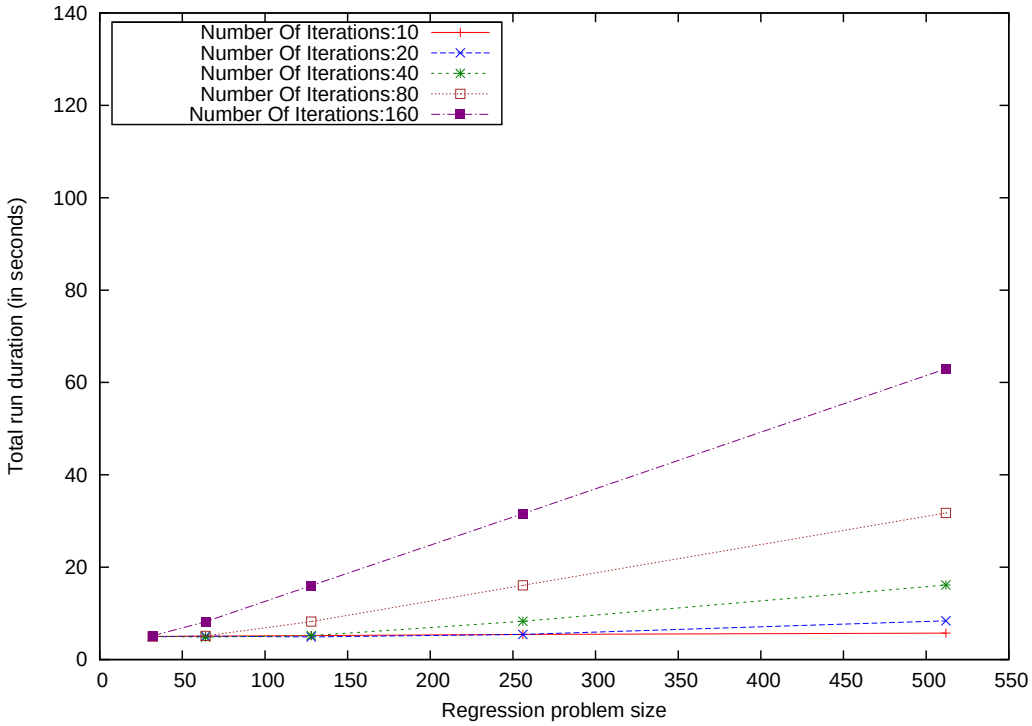


Figure 32: Total run duration (in seconds) for the multiple GPU and multiple CPU core implementation over varying number of testcases and number of iterations.

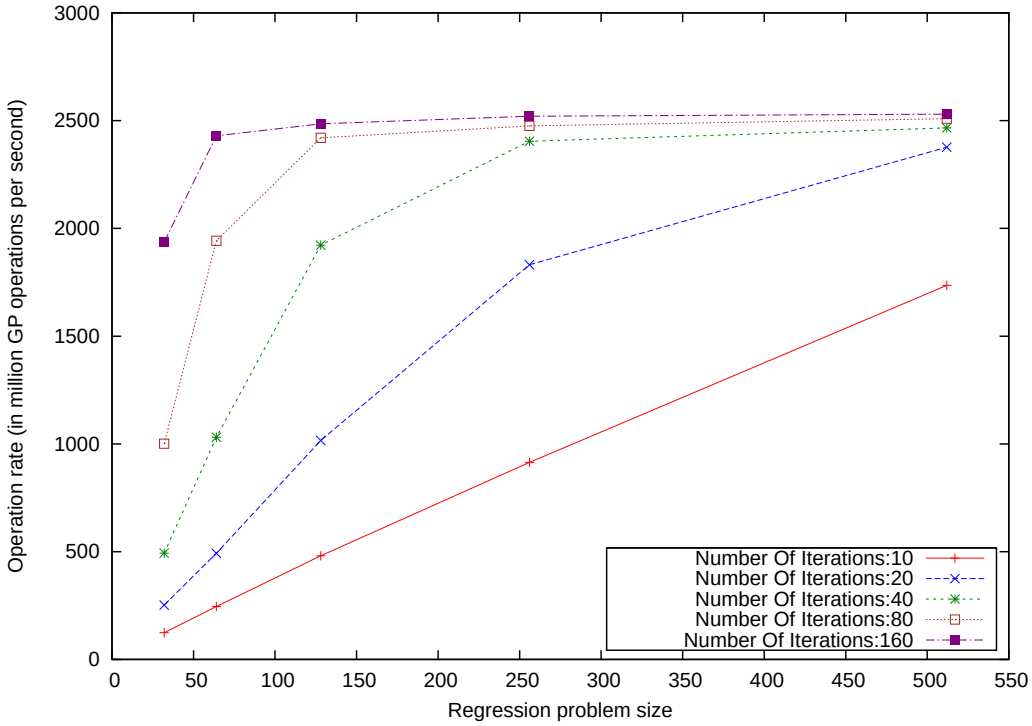


Figure 33: Operation rate (in million GP operations per second) for the multiple GPU and multiple CPU core implementation over varying number of testcases and number of iterations.

5.4 Step Three: Deme Transfers

5.4.1 Description of the Step

The parallelisation described in the previous sections was achieved by splitting the population into demes, which allows different processors to work simultaneously on different demes. If the demes are not to conduct entirely separate evolutionary runs, there must be transfer of evolved material between them and this introduces a new computational cost. This section aims to show that this cost need not undermine the appeal of the strategy. The deme transfer system's time costs are assessed and then a smarter approach is designed to reduce these time costs and this approach's costs are compared.

5.4.2 Implementation

A deme transfer policy must specify when and where the transfers occur, what is transferred and how it is incorporated into the recipient deme. In more detail, a common approach is to define some topological connectivity between the demes, to define the number of generations between transfers and to define some system for copying or moving good individuals to the topological neighbours during each interaction [19]. A typical strategy might involve organising the demes into a loop (so that every deme has exactly two neighbouring demes) and then, every 10 generations, using copies of the best 10% of individuals from each deme to overwrite random individuals in each of the two neighbouring demes.

The deme transfer code was written to require two policy classes: `DemeTransferLayoutPolicy` and `DemeTransferUpdatePolicy`. A concrete `DemeTransferLayoutPolicy` class must provide definitions of methods to specify which demes will transfer to which demes in which generations. A concrete `DemeTransferUpdatePolicy` class must provide definitions of methods to specify how to retrieve the necessary information from a donor deme and to specify how to incorporate that information into a recipient deme. When the deme transfer code is configured with these two classes, it uses them to carry out the deme transfers (and to perform various checks when running in debug mode). The class architecture for this is depicted in Figure 34.

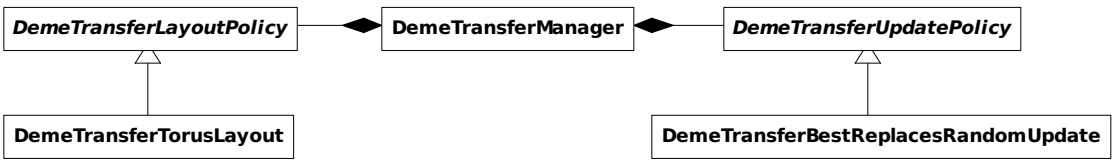


Figure 34: The `DemeTransferManager` class owns a `DemeTransferLayoutPolicy` and a `DemeTransferUpdatePolicy`. These two classes are abstract base classes with concrete instantiations such as `DemeTransferTorusLayout` and `DemeTransferBestReplacesRandomUpdate` respectively.

The normal mode of operation for the parallel architecture used in this work is to

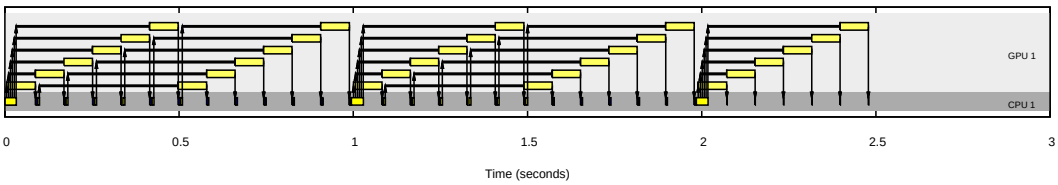
process all demes as soon as possible. Deme transfers may interrupt this pattern because a deme involved in a transfer must wait until it has received all contributions from its neighbouring demes before it can proceed to the next generation. If this is implemented on a serial architecture, there is no reason for the ordering of the tasks to make much impact on the employment of the processor so this causes no problem. However if multiple or parallel processors are being used—as is the case here—processing power may be wasted if processors sit idle whilst waiting for others to complete their work.

The most obvious method of implementing the deme transfer might be the one hinted at by the description in Section 5.2.1. That is, complete all normal processing of all demes up to the end of the generation before the deme transfer, perform the deme transfer, and then return to normal processing. This is depicted in Figure 35(a). In fact, it is possible to perform deme transfer with less idle processor time by returning each deme to normal processing as soon as possible, as shown in Figure 35(b). The deme transfer code attempts to implement this "smart" technique as far as possible with whatever `DemeTransferLayoutPolicy` and `DemeTransferUpdatePolicy` it has been configured. It is possible to switch this mechanism off so that the effects of it can be compared.

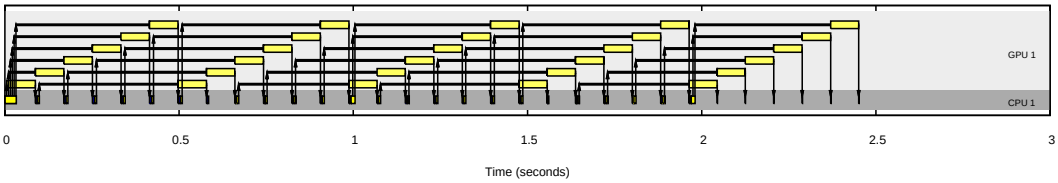
So the simple approach involves waiting for everything to be ready for the deme transfers, executing them fully and then returning to normal evolution. The smart approach aims to do as much as possible as soon as possible and to aid this, separates out donating individuals from receiving individuals. This allows demes to return to evolution, even if its neighbours have not yet taken receipt of its donations. There is another, more extreme approach which is to make deme transfers completely asynchronous. This means that demes transfer whenever they can, regardless if one deme is many generations ahead of another. This approach was rejected since it significantly changes the nature of the EC algorithm and makes reproducibility very difficult. This sort of asynchronous has been attempted for transfers between demes executing in parallel on the same GPU as discussed in Section 5.2.1.

When the GPU evaluation takes much longer than the CPU tasks, the time spent on deme transfers is usually small, relative to the total run time. Any criticism regarding the cost of using demes is more likely to be directed at the examples in which the GPU evaluation is faster. The aim is to show that this cost is small and that the smart deme transfer approach can help mitigate it, as shown in Figure 36.

All deme transfers are controlled by a `DemeTransferManager` class. To achieve the "smart" behaviour, the `DemeTransferManager` extracts any required information from a donor deme as soon as possible so that the deme is not held up from progressing to the next generation (unless it is still waiting to receive transfers from neighbouring demes). For this reason, the methods that a concrete `DemeTransferUpdatePolicy` must provide describe how to extract a `DemeUpdateRemnant` object from a donor deme and



(a) Simple approach



(b) Smart approach

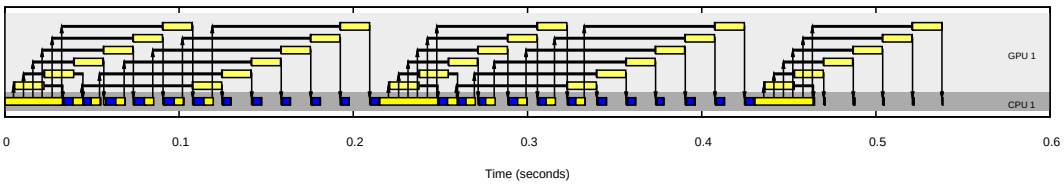
Figure 35: A smarter deme transfer mechanism can avoid the need to stop and restart all evaluation. In Subfigure 35(a), the two deme transfers (at around one second and two seconds) require all evaluation to be completely stopped and restarted. In Subfigure 35(b), the smarter deme transfer system restarts each deme's processing as soon as possible. This figure gives preliminary evidence to support the hopes that the the deme transfers are not adding much time to the run and that a smarter approach helps reduce any small amount of time that is being added. These hopes are assessed more thoroughly in Section 5.4.4. The topology used in this example is a strip (see Section 5.4.3). The problem size is 256 testcases and the number of iterations is 100.

how to incorporate a previously extracted `DemeUpdateRemnant` into a recipient deme.

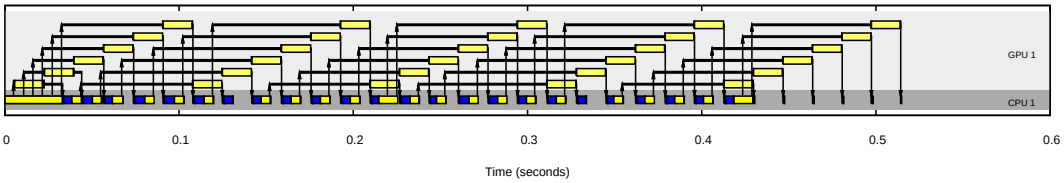
The `DemeTransferManager` class keeps track of the status of each deme and manages the interactions between them. Since the demes may be distributed over multiple CPU threads, the `DemeTransferManager` must be thread-safe. It provides a method for clients to register that a given deme is ready to donate any required transfers. It provides two methods that update demes with any appropriate transfers from other demes (if they are all ready). One of these methods updates any demes for which all transfers are ready and then returns with information about which demes, if any, were updated. Clients may use this to see if any demes are ready to receive transfers before resorting to looking for lower priority work.

If no such updates are ready and if there is no lower priority work, clients may have nothing to do until a deme is ready for updating. This happens when all the demes that a CPU thread is managing still require deme transfers from demes being managed by other threads. This is the motivation for the other `DemeTransferManager` update method, which waits until at least one of the specified demes can be updated. To do this efficiently, a standard technique in concurrent programming, the condition variable (or monitor), is used. This allows the thread to sleep, waiting for the opportunity to update one of the demes. Whenever a thread registers that a deme is ready to donate, it uses the condition variable to wake any sleeping threads so that they may check if this gives them the donation they require to update one of their demes.

All this provides a mechanism to allow each deme to be processed as soon as pos-



(a) Simple approach



(b) Smart approach

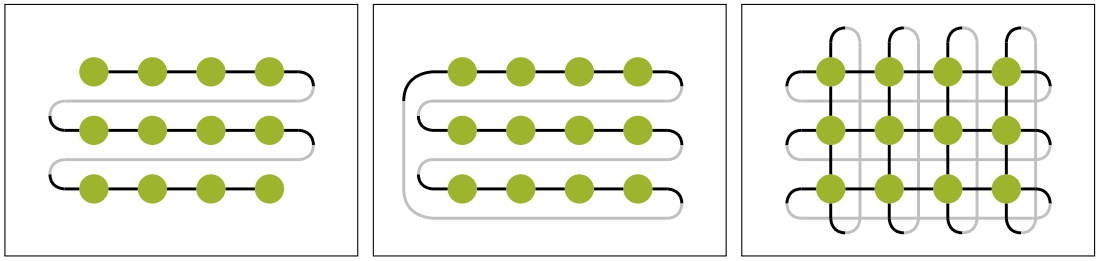
Figure 36: The smarter deme transfer mechanism makes more of a difference when the GPU is relatively faster compared to the CPU tasks. Again, the topology used in this example is a strip (see Section 5.4.3). Again, the problem size is 256 testcases but here the number of iterations is reduced to 20.

sible given the appropriate deme transfers. However this may not be sufficient if it is used in a framework that processes all of a CPU's demes in a strict order, such as: submit all demes in order, gather and immediately submit all demes in order for each generation, gather all demes in order. Under many deme transfer topologies, this arrangement would lead to the code waiting for one deme whilst another is ready for processing. Instead, the strict ordering of deme processing should be broken and each deme should be processed as independently as possible. This requires a little more work to keep track of the status of each deme. For this reason, each CPU thread manages its demes with a `FlowStageManager` object, which records each deme's progress. Further, it keeps track of the order of demes where appropriate so that it can be maintained where there is no advantage to breaking it. This is implemented with queues: when a deme is processed through from one flow stage to another, say by being submitted to the GPU, it joins the back of the queue for the next flow stage. This mechanism makes it easy to ensure that each CPU thread never sits idle if there is work to be done whilst also making it easy to prioritise tasks. For instance, since it is also important to ensure the GPU does not sit idle, submitting to the GPU is treated as a high priority task for the CPU.

5.4.3 Topologies

The topology defines which demes are neighbours of each other and so defines the pattern of deme transfers. This in turn affects the time required to carry out the deme transfers; for example, the transfers can be eradicated completely by using a topology that keeps all demes separate. For this reason the topology is worth considering. As mentioned in Section 2.1.10, various different topologies have been proposed in the

literature and their effects on the evolutionary algorithm compared. This work is not concerned with these effects of the topologies but with their effects on the delays caused by the deme transfers. The three topologies investigated in this study are the strip, the loop and the torus as illustrated in Figure 37.



(a) Strip (b) Loop (c) Torus

Figure 37: Examples of the strip, loop and torus topologies for twelve demes. The grey and black sections of the connections have no significance, other than to help clarify the image for the torus topology.

All connections in these three topologies are bidirectional, meaning that there are deme transfers from deme A to deme B in some generation if and only if there are transfers from deme B back to deme A in the same generation. This is merely a design choice of the topologies in question; the code does not enforce this property. All three topologies are also connected, meaning that it is possible to get from any deme to any other via some path of transfers.

The strip (Figure 37(a)) consists of a single line of demes in which each deme shares a connection with its two neighbours, except the first and last deme which each only share a connection with one neighbour.

The loop (Figure 37(b)) is like the strip except that the first and last demes also share a connection with each other (which can be visualised as a strip with its two ends looped around so that they meet).

The torus (Figure 37(c)) can be thought of as a rectangular grid with each deme sharing a connection with its four (horizontal and vertical) neighbours and with further connections being shared by each pair of equivalent demes in the top and bottom rows and each pair of equivalent demes in the far left and far right columns. This topological connectivity can also be visualised as a grid drawn on the surface of a torus, hence the name. To see how the two are topologically equivalent, it might be helpful to imagine drawing the grid on a square sheet of rubber and then smoothly deforming the rubber to align and glue the top and bottom edges (forming a hollow cylinder) and then to align and glue the two remaining edges (forming a torus).

To understand the different natures of these layouts it is helpful to consider some of their numerical properties as given in Table 11. This table shows that there are fewer connections between the demes in the loop than in the torus and fewer again in the strip than in the loop. This is reflected in the average minimum number of steps between

pairs of demes which is greater for the loop than for the torus and greater again for the strip. In terms of the EC algorithm, this means that we would expect a new individual of very high fitness to take more deme transfers to propagate throughout the whole population when using a strip than when using a torus.

	Strip	Loop	Torus (3x4)
Fraction of deme pairs directly connected	$\frac{11}{66}$ $(0.1\dot{6})$	$\frac{12}{66}$ $(0.\dot{1}\dot{8})$	$\frac{24}{66}$ $(0.\dot{3}\dot{6})$
Average minimum steps between demes	$4\frac{1}{3}$ $(4.\dot{3})$	$3\frac{3}{11}$ $(3.\dot{2}\dot{7})$	$1\frac{9}{11}$ $(1.\dot{8}\dot{1})$
Maximum steps between demes	11	6	3

Table 11: For a 12 deme instance of each of the layouts, this table gives the fraction of pairs of demes that are directly connected, the minimum number of steps between a pair of demes, averaged over all pairs of demes and the maximum number of steps between any pair of demes.

5.4.4 Experiments and Results

It was hoped that the results from the experiments would show that the cost of deme transfers is small and that it can be made smaller through the use of smart deme transfers. Before discussing the more robust comparative data, it is worth examining some timelines which can give an indication of what happens during the runs.

Figures 35 and 36 in Section 5.4 depict relevant results in that they show the effect of the smart deme transfer system on a strip topology using real timing data. In those figures, the ordering of the deme evaluations was unchanged by the deme transfers over the strip topology, regardless of the deme transfer approach being used. This is because the strip topology has few connections and in particular, if the demes are evaluated in order, then they also become available in order after strip-based deme transfer.

The loop and torus topologies are more connected and this makes things more interesting as shown in Figures 38 and 39 respectively. Figure 38(a) shows that the simple mechanism remains very similar (to Figure 36(a)) with a loop topology. In Figure 38(b) the first deme is not ready to be submitted again after the first deme transfer until the results of the sixth (and last) deme have been retrieved from the previous generation. In contrast, the second deme is the first to be ready for evaluation, after the results of the third deme are gathered for the second generation. After the second deme transfer, the order of the demes gets changed again.

Figure 39(a) shows that the simple mechanism remains very similar again for the toroidal topology. In Subfigure 39(b) the deme ordering also gets altered by the first deme transfer, although it is then left unchanged by the second transfer. To reiterate: these behaviours are not directly coded for each topology; they arise from the `DemeTransferManager` class dynamically querying the `DemeTransferLayoutPolicy` to calculate what processing can be done.

Figure 40 illustrates the same mechanism for twelve demes being processed by two CPU threads. When using the smart deme transfer approach, the constraints between the various demes being processed by two different threads produce an intricate pattern of evaluations.

These timelines give a feel for the workings of the deme transfer system but do not provide a robust assessment of the time cost incurred by the deme transfers or the time savings offered by the smarter deme transfers. To tackle these questions, an experiment was performed. Each of the three topologies (strip, loop and torus) was assessed along with an empty topology specifying no deme transfers. Each deme topology was configured to perform any deme transfers every two generations. This is quite a high frequency and was chosen to highlight the effects of deme transfer. The experimental setup was similar to that described in Section 4.5.2 but the runs involved 384 individuals assessed on 256 testcases for 20 iterations over 180 generations. Each run was repeated 20 times.

The results are in shown in Table 12 and are depicted in Figure 41. The two values for the empty topology are very similar, which is as expected because the style of deme transfer is irrelevant for a topology with no deme transfers. The overall message of the remaining results is clear: although there are costs associated with deme transfer, they are small and can be tackled effectively by being a bit smarter about deme transfers.

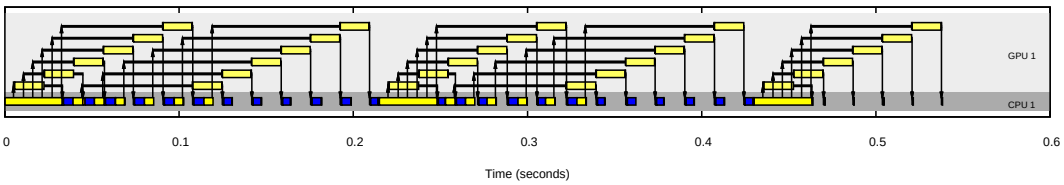
The largest percentage increase over the average of the two corresponding empty results occurred for two threads, four demes per thread and the toroidal topology: the naive deme transfers added 13.449% to the total run time but the smart deme transfers only added 1.742%. For single threaded results, the largest increase occurred with four demes per thread and the toroidal topology: the naive deme transfers added 7.571% to run time but the smart deme transfers reduced the run time by 0.101%. The slightness of all of these deme transfer costs is particularly striking in the context of the transfer frequency being set so high to accentuate the cost.

Error bars are included in Figure 41 to indicate the average plus and minus one estimated standard error, however these are so small that they can barely be seen. The largest estimated standard error is 0.037 seconds and this can just about be seen at the tip of the second column in the last group.

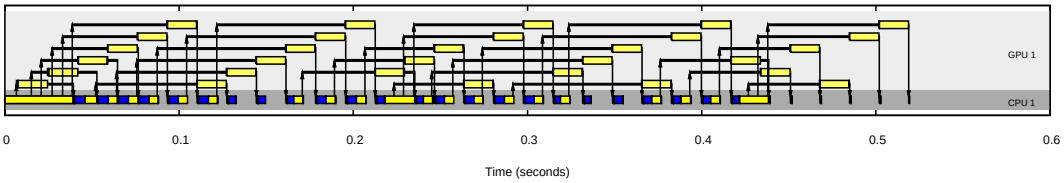
5.5 Summary and Contribution

This chapter investigated three additional steps to accelerate GPU implementations of GP. The population-parallel implementation of cyclic GP described in Chapter 4 was used to assess these steps, though they are all applicable to any GPU evaluation system (such as the data-parallel system discussed in Chapter 6).

Step one was based on the observation that the GPU is parallel in the sense of operating in parallel to the CPU, as well as in the sense of being a parallel processor. To capitalise on this, a deme-based approach was used such that the GPU can be evalu-



(a) Simple approach



(b) Smart approach

Figure 38: The effect of using smart deme transfers when using a loop topology. The problem size is 256 testcases and the number of iterations is 20.

ating one deme whilst the CPU is preparing another. The use of CUDA streams also allows the graphics card to transfer data between the two processors whilst they are both executing.

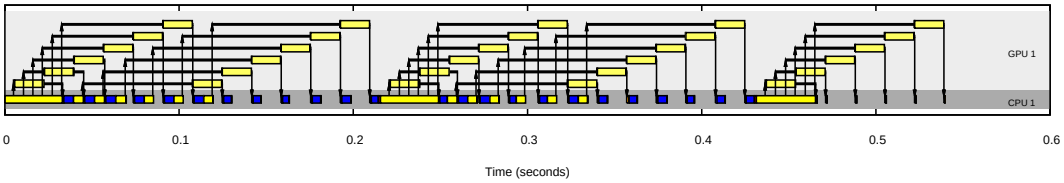
An experimental investigation was conducted into the effects of this approach. The timelines recorded for this technique revealed its workings. They showed that the data transfers accounted for a very small amount of time in these experiments and so little was gained by attempting to execute them in parallel. Instead, the timelines suggest that the savings come from allowing GPU and CPU processing to occur in parallel.

With negligible savings in parallel transfers, the factor of speed improvement would be expected to lie between one and two. The speed should not reduce unless the small management overheads outweigh the benefits, which seems unlikely. The speed should improve no more than two-fold because the time should not reduce to less than is required by the slower of the two processors (which would account for at least half the time in the serial setup). The improvements would be expected to be greatest when the two processors require the same amount of time as each other.

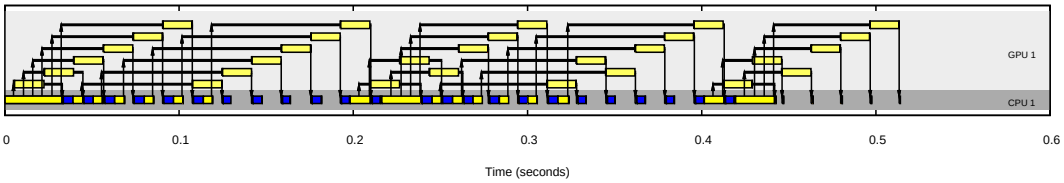
The best overall acceleration was observed with 128 testcases and 160 iterations: the total run time was reduced by 191.248 times over the CPU implementation. The best improvement over the original GPU acceleration was observed with 64 testcases and 80 iterations: the total run time was reduced by a further 1.806 times.

This work involved a number of contributions that are believed to be novel:

- The approach of using demes to allow the CPU to be preparing one deme whilst the GPU is processing another. The closest work published before a description of this work was that of Garnica et al [21], which involved the CPU completely processing one deme whilst the GPU completely processed another. As discussed in Section 5.2.1, this fails to play to the respective strengths of the two processors.



(a) Simple approach



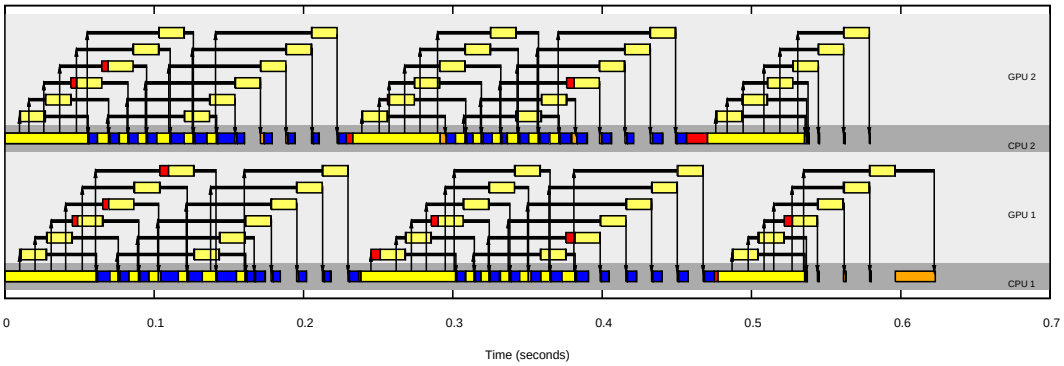
(b) Smart approach

Figure 39: The effect of using smart deme transfers when using a toroidal topology. The problem size is 256 testcases and the number of iterations is 20. This purpose of this figure is to give a qualitative feel for what is assessed more quantitavely in Figure 41.

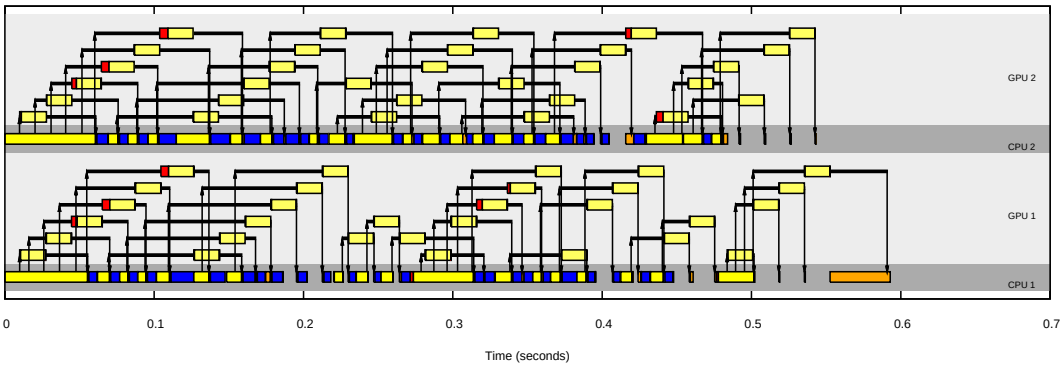
- The approach of using CUDA streams to also allow data transfers be performed whilst both processors are doing useful work.
- An explanation of key issues faced in implementing this technique and a description of successful solutions.
- The real timing data drawn from runs and depicted as timelines, showing the inner workings of the technique.
- Experimental investigation into the effect of this technique on total run time. The best result showed a further improvement of 1.806 times beyond the corresponding result in Chapter 4.

Step two built upon the work in step one by adding another GPU and using a second CPU core to drive it. This required some parts of the code to be made thread-safe. As in step one, the factor of speed improvement would be expected to lie between one and two. Again, the speed should not reduce unless the small management overheads outweigh the benefits, which seems unlikely. The speed should improve no more than two-fold because the computational resources of the processors are doubled by the technique.

The best overall acceleration was observed with 128 testcases and 160 iterations: the architecture completed the full run 375.513 times faster than the single–core CPU equivalent and evaluated 2529.779 million GP operations per second over the full run. The best improvement over the original GPU acceleration was observed with 128 testcases and 40 iterations: the total run time was reduced by a further 3.292 times. The best improvement over step one was observed with 512 testcases and 160 iterations: the total run time was reduced by a further 1.982 times.



(a) Simple approach



(b) Smart approach

Figure 40: The effect of using smart deme transfers when using a toroidal topology and twelve demes split over two CPU threads. The problem size is 256 testcases and the number of iterations is 20. This purpose of this figure is to give a qualitative feel for what is assessed more quantitavely in Figure 41.

Layout	Technique	Number of CPU threads: 1, Number of demes per CPU thread: 4	Number of CPU threads: 1, Number of demes per CPU thread: 8	Number of CPU threads: 2, Number of demes per thread: 4	Number of CPU threads: 2, Number of demes per thread: 8
Empty	Smart	9.926 s [±0.006]	19.777 s [±0.010]	10.269 s [±0.015]	20.681 s [±0.027]
	Naive	9.940 s [±0.015]	19.755 s [±0.008]	10.283 s [±0.015]	20.721 s [±0.037]
Strip	Smart	9.925 s [±0.012] -0.008 s	19.747 s [±0.021] -0.019 s	10.322 s [±0.015] +0.046 s	20.648 s [±0.026] -0.053 s
	Naive	10.661 s [±0.014] +0.728 s	20.570 s [±0.023] +0.804 s	11.587 s [±0.014] +1.311 s	22.096 s [±0.021] +1.395 s
Loop	Smart	9.913 s [±0.013] -0.020 s	19.772 s [±0.015] +0.006 s	10.424 s [±0.012] +0.148 s	20.688 s [±0.034] -0.013 s
	Naive	10.663 s [±0.005] +0.730 s	20.574 s [±0.017] +0.808 s	11.601 s [±0.011] +1.325 s	22.073 s [±0.019] +1.372 s
Torus	Smart	9.923 s [±0.010] -0.010 s	19.778 s [±0.014] +0.012 s	10.455 s [±0.015] +0.179 s	20.841 s [±0.017] +0.140 s
	Naive	10.685 s [±0.009] +0.752 s	20.615 s [±0.015] +0.849 s	11.658 s [±0.012] +1.382 s	22.199 s [±0.013] +1.498 s

Table 12: Total run duration (in seconds) over varying layouts, numbers of threads, numbers of demes per thread and smart or naive deme transfers. Each value is followed by the estimated standard error in square brackets. Each value for non-empty topologies is followed by the value's difference from the average of the two equivalent empty topology results.

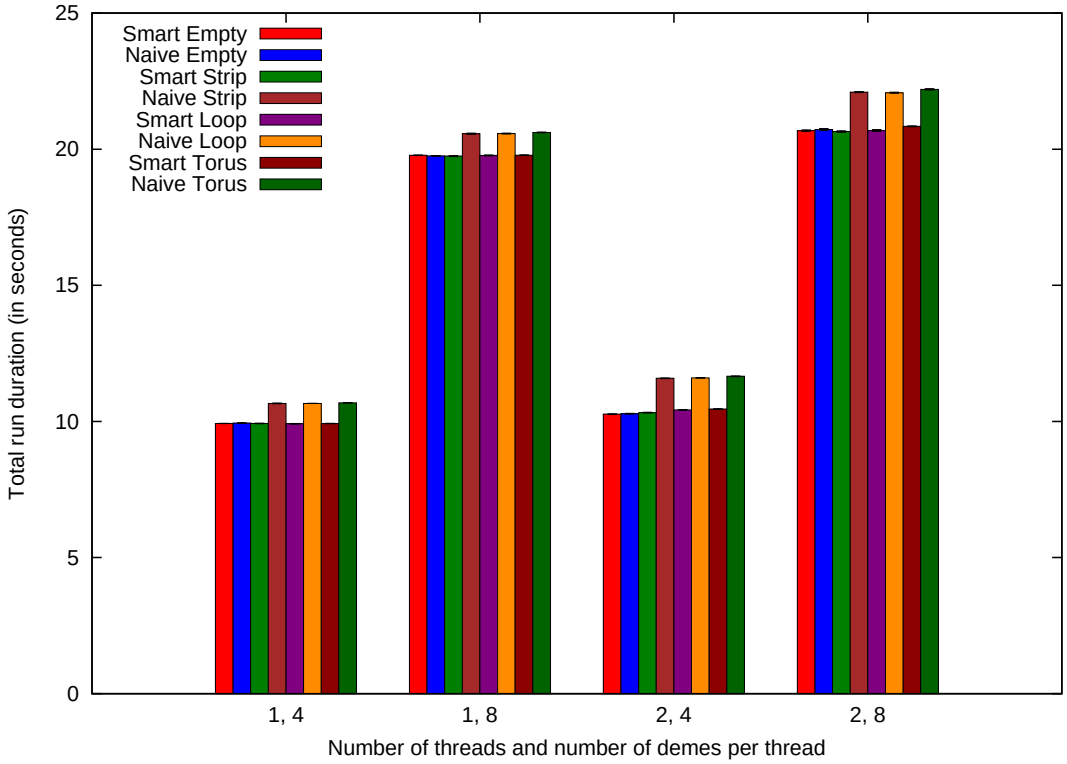


Figure 41: Total run duration (in seconds) over varying layouts and over smart or naive deme transfers. The results are grouped according to the number of threads and the number of demes per thread. Error bars indicate the average value plus and minus one estimated standard error although these are so slight that they can barely be seen.

This work involved a number of contributions that are believed to be novel:

- The use of a second GPU to improve evaluation speed and the use of a second CPU core to drive it (and to handle the increased workload).
- A combination of this technique with the one described in step one such that both CPU cores and both GPUs may all be executing simultaneously.
- Explanation of key issues that might be faced in implementing this technique and a description of successful solutions. In particular, this involved a discussion of the techniques to divide the CPU work over multiple threads in an effective and safe-manner.
- The real timing data drawn from runs and depicted as timelines, showing the inner workings of this technique when combined with step one's technique.
- Experimental investigation into the effect of this technique when used in combination with step one's technique. The best result showed an improvement of 1.982 times over the corresponding result from step one. The best combined result showed an improvement of 3.292 times over the original GPU acceleration.

Motherboards that accept three graphics cards and graphics cards with multiple GPUs are both now widely available. Limits on resources available for this work meant that it was not possible to use more than two GPUs. Nevertheless, GPU intensive resources such as GPU farms seem likely to become more widely available and the architecture being constructed in this work is well placed to take advantage of such technology.

Step three introduced the deme transfers that are necessary if the demes are not to execute entirely independent runs. This work addressed a potential criticism of the approach adopted in steps one and two: that the use of demes incurs a time cost for the deme transfers. The work aimed to show that this cost is small and that a proposed "smart' approach to deme transfers can reduce this small cost. Unlike completely asynchronous approaches, this approach does not modify the EC algorithm.

The deme-transfer was built in a flexible way such that it is configured by two policy classes: `DemeTransferLayoutPolicy` and `DemeTransferUpdatePolicy`. In simple mode or in smart mode, the `DemeTransferManager` implements these policies as efficiently as it can. Timelines for the strip, loop and torus topologies show that the `DemeTransferManager` and `FlowStageManager` manage to operate well in a wide range of circumstances. This is further supported by a timeline of a torus topology split over two threads. The two approaches (naive and smart) were experimentally compared. The largest percentage increase over the average of the two corresponding empty results occurred for two threads, four demes per thread and the torus topology: the naive deme transfers added 13.449% to the total run time but the smart deme transfers only added 1.742%. For single threaded results, the largest increase occurred with four demes per thread and the torus topology: the naive deme transfers added 7.571% to run time but the smart deme transfers reduced the run time by 0.101%. Despite a very high transfer frequency being used, the results clearly show that the cost of deme transfers is small and is reduced by the use of a smart approach to deme transfers.

This work involved a number of contributions that are believed to be novel:

- A new approach to organising deme transfers when using multiple deme transfers (such as a GPU and a CPU core or multiple CPU cores). The approach aims to reduce the impact of deme transfers as far as possible whilst preserving synchronous nature of the algorithm. Furthermore, it does this efficiently when configured with any `DemeTransferLayoutPolicy` or `DemeTransferUpdatePolicy`.

- Explanation of key issues that might be faced in implementing this technique and a description of successful solutions. In particular, this involved a discussion of the techniques to manage the tracking of different demes and the interactions between events from different threads.

- Real timing data drawn from runs and depicted as timelines, showing the inner workings of this technique. These showed the effects over multiple topologies

and over multiple threads and GPUs. The timelines displayed the reordering effect on the deme evaluations.

- Experimental investigation into the effects of this technique over varying numbers of threads, demes and over varying topologies. The results suggest that deme transfers add little to the run time of the architecture previously described and that these costs can be mitigated through the use of the smart deme transfer approach.

The asynchronous approach was rejected since it distorts the EC algorithm and makes reproducibility very difficult. The reproducibility for this approach is much better. However it should be noted that reordering the demes may alter the random numbers that are used for each deme and therefore affect the algorithm. To eradicate any theoretical possibility of this problem, it would be necessary to do something like create a separate stream of random numbers for each deme. In practice, the patterns of deme reordering appear to be robust between runs.

Combined, the three techniques described in this chapter help to squeeze even more evaluation speed out of a single, reasonably-priced machine. The techniques maintain reproducibility and only distort the EC algorithm minimally (by adopting the common EC strategy of splitting the population into demes). The techniques seek to make good use of whatever CPU cores and GPUs are available and so should be effective on new hardware configurations.

6 Data-Parallel Optimisations

6.1 Introduction

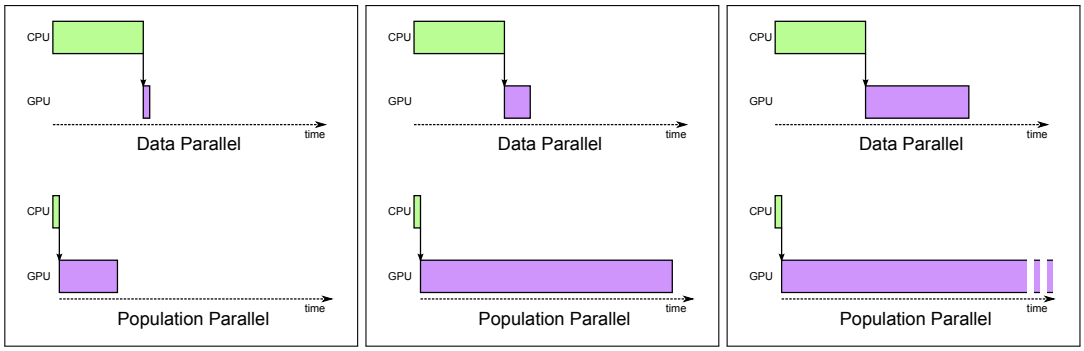
This chapter moves from a population-parallel implementation of node-based programs to a data-parallel implementation of instruction-based programs. It tackles the third objective outlined in Section 1.4: find ways to reduce compilation times of a data-parallel implementation of a form of Evolutionary Computation (EC) with linear programs so that the best evaluation speeds can be brought to bear on more moderately sized data-sets.

As described in Section 2.1.7, data-parallel methods involve dynamically writing and compiling new GPU code for each batch of individuals to be evaluated. This has several advantages: the compiler can write highly specific, heavily optimised binaries; precious on-chip Graphics Processing Unit (GPU) memory need not be used to store programs as data (as is the case when using an interpreter) and optimal GPU memory access patterns arise very naturally. As a result, the GPU evaluation speeds for data-parallel code tend to be impressive.

The problem with data-parallel techniques is that additional Central Processing Unit (CPU) time must be spent compiling the GPU codes for every batch of programs to be evaluated. In practice, this compilation overhead can be considerable (as will be demonstrated in this chapter). If the data-set is vast then this price is worth paying because the fast GPU evaluation over very many testcases makes up for the CPU compilation time, which is independent of the number of testcases. For more moderate numbers of testcases, the compilation time can mean that population-parallel methods are faster. Even if the data-set is large enough to make data-parallel faster, it may still be too small to exploit data-parallel fully. The GPU's power is being wasted because it spends much of its time sat idle waiting for the CPU to compile its code. Since data-parallel evaluation can be so remarkably fast, it requires a remarkably large data-set to reach the best speeds. This is illustrated in Figure 42.

For the best results, the data-set should be large enough so that the time spent on evaluation is much longer than the time spent on compilation. If much less data is used, then the compilation time will tend to dominate total run time. Since this chapter describes work to reduce this compilation time, it is worth asking how much data it would take to just get the evaluation and compilation times to be equal. The larger this value, the greater range of problem sizes we would expect to benefit greatly from these techniques to reduce compilation time. For the sake of a rough calculation, it is helpful to use some values from Section 6.3.4. Individuals of 300 instructions were evaluated for 20 iterations. The compilation time was 0.473 seconds per individual and the evaluation rate was 155837.159 Mgpop/s. The values were derived using a single GPU and single CPU core.

Let the population size (i.e. number of individuals) be labelled p and the data-set



(a) Few testcases (b) Many testcases (c) Very many testcases

Figure 42: It takes a lot of testcases to make data-parallel faster than population-parallel and even more to realise data-parallel's power fully. In Subfigure 42(a), the data-parallel's evaluation speed cannot compensate for its compilation time because population-parallel is finished before data-parallel's compilation is complete. In Subfigure 42(b), an increase in testcases means data-parallel is now faster than population-parallel. It is only after even more testcases are added in Subfigure 42(c) that data-parallel begins to exhibit its full power. Even here, the system speed is only around half that being achieved on the GPU and more testcases will be required to do better.

size (i.e. number of testcases) be labelled x. For the sake of approximate calculations, assume that the compilation times and evaluation times are linear over the number of individuals (an assumption that can be justified as roughly correct based on the results in Section 6.3.4 and in Section 6.4.7). The total number of instructions to be evaluated can be calculated as the number of testcases multiplied by the number of individuals multiplied by the number of instructions per individual multiplied by the number of iterations per evaluation. The evaluation time can be calculated by dividing this by the evaluation rate. The compilation time can be calculated as the compilation time per individual multiplied by the number of individuals. Setting these equal to each other using the figures in our example gives:

$$\begin{aligned} \frac{300 \times 20 \times px}{1.55837159 \times 10^{11}} &= 0.473p \\ \Rightarrow (3.850 \times 10^{-8})x &= 0.473 \\ \Rightarrow x &= 1.229 \times 10^{7} \end{aligned}$$

Hence, obtaining an evaluation time as long as the evaluation time would take over 12 million testcases. Using 1000-instruction individuals as in Section 6.5, the evaluation speed is 148366.170 Mgpop/s and the compilation time is 3.370 seconds per individual. Using these values, it would take over 25 million testcases for the evaluation time to equal the compilation time.

If this number of testcases had been very small, then there would be little value in this chapter's methods because it would be easy to provide enough testcases to ensure that the compilation time would be substantially shorter than the evaluation time.

As it stands, this result indicates the method should be of considerable value for any problems that have fewer than around $12 - -25$ million testcases, a huge amount of data. Indeed, depending on whether compilation is performed in parallel with the evaluation and whether there are other considerable CPU tasks, this chapter's techniques may still be of considerable value for problems of much greater size than this.

Even larger data-sets would be required if fewer iterations were used, or if a more powerful GPU were used (and the graphics card being used is a fairly low-level model from a range several generations old). GPU manufacturers claim that the rates of improvements of their chips have been outstripping those of CPUs [15].

Note that the work described in Section 5.2 is of some considerable help here. That section built on the observation that the GPU offers two sorts of parallelism: within the GPU and between the GPU and the CPU. Jobs can be submitted to the GPU asynchronously so that execution returns to the CPU code once the GPU starts evaluating. That work was in the context of population-parallel evaluation but can be applied without modification in this context of data-parallel evaluation. If the GPU evaluation times are of a similar magnitude to the CPU compilation times, then performing them in parallel will improve the rate of work by up to two times. Similarly, the work in Section 5.3 is applicable here and allows multiple threads to evaluate code whilst multiple GPUs evaluate the binaries. Indeed, the machine used for this research contained two GPUs and a four core CPU.

These techniques somewhat alleviate the need for huge data-sets to get the most from data-parallel but they only make much difference when the data-set is already large enough that the evaluation and compilation times are similar. A middle ground is required on which systems may use data-parallel's evaluation speeds to tackle problems with normal amounts of data but without having to pay such a high overhead in compilation time.

The correct solution to the problem must be to reduce compilation times. It would almost certainly be unwise to attempt to delve under the covers of the compiler itself since compilers are typically highly complex and heavily optimised. The alternative is to reduce the compiler's workload. This chapter investigates two ways to attempt this. The first method sends the code in a lower-level language whilst the second method aims to reduce duplication in the code sent to the compiler.

Section 6.3 describes the attempt to send the code to the compiler in a lower level language. The Compute Unified Device Architecture (CUDA) C compiler that is used to convert CUDA C into a GPU-ready binary achieves this in two steps: from CUDA C to Parallel Thread EXecution (PTX) (an assembly-like language) and from PTX to the binary. Passing PTX code to the compiler instead of CUDA C saves the compiler the work involved in the first step. It also allows more control over the low level code that is compiled into the binary used for evaluation.

Section 6.4 describes the attempt to reduce duplication in the code. This exploits

code similarities within groups of individuals sent for evaluation. The core algorithm of EC often involves evaluating the effect of fairly minor modifications to the most successful individuals seen so far, particularly for the form used in this chapter (see Section 6.2). This means that the code to be compiled involves a lot of duplication. The technique described in Section 6.4 aims to scan for and draw out this duplication in order to remove some of the work for the compiler.

Section 6.5 describes a brief investigation into how quickly 1000-instruction individuals can be compiled using a combination of the two previous techniques. Before all this, Section 6.2 gives a brief overview of the form of EC that will be used throughout the chapter.

6.2 A Brief Synopsis of TMBL

Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble") has been proposed as a baby sister to Genetic Programming (GP) [50] and is akin to linear GP. The key feature of TMBL is its focus on long term fitness growth above all else. As described in Section 3.2, it is built on the following hypothesis: *long term fitness growth is dependent on the ease with which mutations can affect an individual's behaviour without (necessarily) ruining its existing functionality*. Such changes are known as tweaks.

An analogy helps motivate this hypothesis. Imagine that you are given around a hundred toy blocks with patterns on their surfaces so that lining them up in one particular way makes their patterns fit together. Imagine you are asked to solve the puzzle but only using trial and error: no pre-planning, no writing, just considering random changes and performing them if they improve things.

Given this challenge, you would almost certainly take the puzzle, lay it out flat and solve it without much difficulty. Imagine you are then given an equivalent set of blocks but this time you must build the blocks vertically in a tower. This would be much harder. In fact, with around a hundred blocks, you might find it almost impossible. However much progress is made, at some point you have to grab some block near the bottom and ruin the prior achievements.

The argument is that the same principles hold for a GP tree flipped upside-down: at some point, changes must be made to a node near the root of the tree and that ruins all the nodes above. The lower blocks in the puzzle support the blocks above them physically; the lower nodes in the inverted GP tree support the nodes above functionally.

What went wrong when the tower became vertical? It became difficult to make changes to parts where progress had been made without damaging what had already been achieved. This view motivates the design of a representation for TMBL that is like a form of linear GP. Figure 43 shows code implementing a short excerpt of a TMBL individual.

Some of the features that make TMBL's representation distinct from standard linear GP also happen to make it more suitable for the PTX approach described in Section 6.3.

```
slot12 = slot15;
if (testcase0 >= 0) {
        slot4 += testcase0;
}
if (slot13 >= 0) {
        slot13 *= slot8;
}
if (slot7 >= 0) {
        slot3 -= testcase0;
}
slot6 = ((testcase0 == 0.0f) ? 0.0f : slot6/testcase0);
slot16 = slot14;
if (testcase0 >= 0) {
        slot17 = -95.3412549093695f;
}
slot13 = ((slot16 == 0.0f) ? 0.0f : slot13/slot16);
slot3 = ((slot4 == 0.0f) ? 0.0f : slot3/slot4);
```

Figure 43: An illustration of code one might see representing a TMBL individual. This is presented for those readers who are interested; other readers need not scrutinise the details of the code.

For GPU work, we want the execution of different threads to diverge as little as possible. Linear-style branching is bad for this because it allows threads to take completely different paths depending on data but TMBL achieves its conditionality with local *if* conditions that can be implemented in PTX without any non-uniform branches. Stacks are harder to implement than registers with PTX because the fastest type of memory is not indexable but TMBL uses registers. The techniques used here should also suit other forms of GP such as tree-based GP. Forms that involve conditional jumps, such as some varieties of linear GP may experience slower evaluation speeds.

TMBL is also well suited to the alignment work described in Section 6.4. TMBL has high computational requirements and will typically be applied to complex problems with moderately large data sets. With current technology, these factors make it important to find the quickest possible evaluation methods. TMBL is focused on long term fitness growth through a slow process of tweaks of previous successful solutions. The consequence of this is that its populations typically contain individuals that are highly similar to each other.

6.3 Technique 1: Compiling from the Lower-Level Language PTX

6.3.1 Data-Parallel with CUDA C Code

Before describing the novel PTX work, it is worth outlining the standard data-parallel techniques to which it will be compared.

CUDA requires applications to provide a function to be executed on the GPU,

known as a *kernel*. The standard approach to a CUDA data-parallel system is to write, compile and execute one or more CUDA C kernels for each batch of individuals to be evaluated. A CUDA C file contains source code in standard C with a few additional keywords and constructs. New releases of CUDA are permitting more C++ constructs in this code however for simplicity, it will still be referred to as CUDA C here.

Once the system has written out a CUDA C kernel file, there are three steps to prepare it for execution as shown in Figure 44. First, the CUDA C must be compiled to PTX, which is a lower-level language, similar to machine code. Second, the PTX code must be compiled to a "cubin" file. In earlier versions of CUDA, the cubin file was implemented as a text file but it is now a binary. Third, this binary file must be loaded onto the GPU. The first and second steps are done using the `nvcc` compiler; the third step is done using the CUDA function `cuModuleLoadData()`.

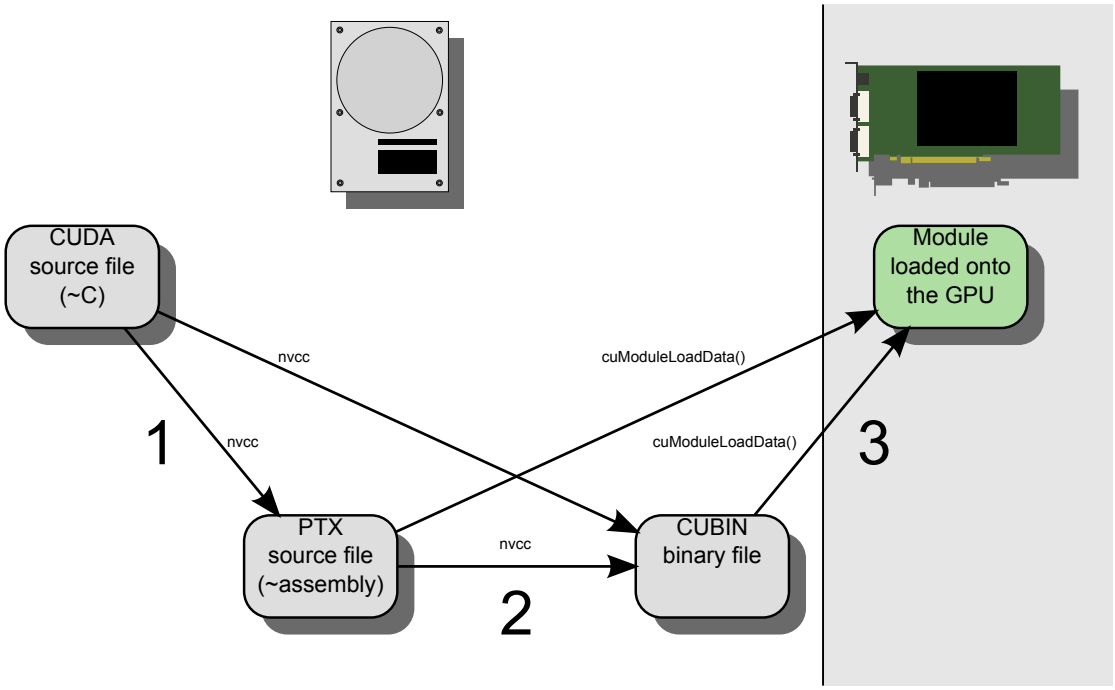


Figure 44: The steps required to compile and load source code into a callable GPU module. The three grey rectangles on the left represent files, as indicated by the hard drive icon; the green rectangle on the right represents a GPU module, as indicated by the graphics card icon. These steps will be referred to throughout the section. The nVidia compiler nvcc can be used to compile CUDA C-style source code into PTX source code (step 1), to compile PTX source code into a binary cubin file (step 2) or to perform both steps together. CUDA driver functions such as `cuModuleLoadData()` load an executable GPU module from a cubin file (step 3) or from a PTX file (by internally performing step 2 first). Existing techniques generate CUDA source files and then apply steps 1, 2 and 3; this research investigates generating PTX source code directly, hence reducing compilation time by skipping step 1.

As indicated in Figure 44, `nvcc` permits steps 1 and 2 to be done together and `cuModuleLoadData()` permits steps 2 and 3 to be done together.

Using CUDA version 3.2, the `cuModuleLoadData()` function (and its sister functions

`cuModuleLoad()` and `cuModuleLoadDataEx()`) cannot be accessed through the high-level CUDA Runtime Application Programming Interface (API) but must be accessed through the lower-level CUDA driver API, which increases the technical difficulty.

One of the advantages of the data-parallel approach is that it naturally leads to excellent memory access patterns in which consecutive threads always access consecutive memory locations. Such memory access patterns are favourable to the fastest possible access to off-chip global memory. This is worth highlighting here because the GPU code used in this section's experiments does not read testcases but calculates them dynamically instead. This may make the evaluation speeds higher than they would otherwise be. The excellent access patterns give reason to hope this effect is small because the testcase reads that are being omitted would be as fast as possible. Furthermore, there is no reason to think that this effect favours the novel techniques over the standard techniques.

6.3.2 Data-Parallel with PTX Code

The data-parallel approach achieves very high evaluation speeds but suffers a high compilation overhead, which must be paid every time a new batch is to be evaluated. Step 3 from Figure 44 is relatively quick, as will be seen. The problem lies with steps 1 and 2. The aim of this research is to circumvent step 1 by writing the source code directly in PTX. This should reduce the compilation time and may even increase the evaluation speed. This requires leaving the comfortable familiarity of C and entering the lower-level world of PTX.

So what is PTX? According to nVidia, PTX is a "low-level *parallel thread execution* virtual machine and instruction set architecture (ISA)", which "provides a stable programming model and instruction set for general purpose parallel programming." It is a low-level, assembly-like language to which CUDA C gets compiled and which, in turn, gets compiled to GPU-ready binary. Unlike assembly, it does not correspond directly to its resulting machine code binary and although it is "designed to be efficient on nVidia GPUs" it could be implemented on other parallel platforms. Importantly, the goal of PTX stated first in the nVidia documentation is to "provide a stable ISA that spans multiple GPU generations" so it should be forward compatible. As PTX is one of the CUDA resources, its tools and documentation are proprietary but freely available. It is well documented: the CUDA toolkit v3.2 contains a 199-page PTX manual, the source of this paragraph's quotations.

PTX is considerably more low-level than CUDA C, so maintaining extensive PTX code would be difficult. Fortunately this data-parallel approach only needs the PTX code to describe a skeleton and the limited instruction set of the individuals being evolved. This can be achieved with a small code base and using a simple subset of the language.

PTX's basic syntax rules will be familiar to programmers of many modern lan-

guages: whitespace may be used freely and is ignored (except in separating tokens); semi-colons separate lines; lines beginning with a # character are pre-processor directives and commenting rules are as for C/C++ (/* and */ mark comment blocks and // marks rest-of-line comments).

Table 13 provides a translation from some common C tasks to their PTX equivalents. These building blocks provide most of the tools that are needed to construct complete TMBL kernels. The float literals in the CUDA C use a trailing `f` to request floats explicitly, which stops the compiler grumbling about demoting doubles. Float literals in PTX must be specified in native hexadecimal so a trailing comment is used to provide a decimal equivalent for readability. It is worth taking a look at some of the basics of the language.

Description	CUDA C code	PTX code
Set to constant	`slot0 = -1.64101672f;`	`mov.f32 %slot0, 0fBFD20CD6; // -1.64101672`
Add	`slot4 += slot3;`	`add.f32 %slot4, %slot4, %slot3;`
Subtract	`slot1 -= testcase0;`	`sub.f32 %slot1, %slot1, %testcase0;`
Multiply	`slot0 *= slot3;`	`mul.f32 %slot0, %slot0, %slot3;`

Safe divide	`slot2 = (`
	`  (slot3 == 0.0f) ?`
	`  0.0f :  slot2/slot3`
	`)`
		`div.full.f32 %slot2, %slot2, %slot3;`
		`setp.eq.f32 %divPred, %slot3, 0f00000000;`
		`selp.f32 %slot2, 0f00000000, %slot2, %divPred;`

Test subtract	`if (slot2 > 0) {`
	`  slot0 -= testcase1;`
	`}`
		`sub.f32 %ifTemp, %slot0, %testcase1;`
		`slct.f32.f32 %slot0, %ifTemp, %slot0, %slot2;`

Test safe divide	`if (slot0 > 0) {`
	`  slot3 = (`
	`  (slot2 == 0.0f) ?`
	`  0.0f :  slot3/slot2`
	`  );`
	`}`
		`div.full.f32 %ifTemp, %slot3, %slot2;`
		`setp.eq.f32 %divPred, %slot2, 0f00000000;`
		`selp.f32 %ifTemp, 0f00000000, %ifTemp, %divPred;`
		`slct.f32.f32 %slot3, %ifTemp, %slot3, %slot0;`

Loop	`unsigned int iter=0;`
	`while(iter<noIters) {`
	`  ...`
	`  ...`
	`  ++iter;`
	`}`
		`mov.u32 %iterCtr, 0;`
		`$startOfLoop:`
		`...`
		`add.u32 %iterCtr, %iterCtr, 1;`
		`setp.ne.u32 %loopPred, %noOfIters, %iterCtr;`
		`@%loopPred bra.uni $startOfLoop;`

Program choice	`if (progId==0) {`
	`  ..`
	`}`
	`else if (progId==1) {`
	`  ..`
	`}`
		`mov.u32 %progComp, 0;`
		`setp.eq.u32 %progPred, %progId, %progComp;`
		`@%progPred bra.uni $prog0;`
		`mov.u32 %progComp, 1;`
		`setp.eq.u32 %progPred, %progId, %progComp;`
		`@%progPred bra.uni $prog1;`
		`$prog0:`
		`  ..`
		`  bra.uni $endCode;`
		`$prog1:`
		`  ..`
		`  bra.uni $endCode;`
		`$endCode:`

Table 13: A comparison of the CUDA C and PTX code used to perform various tasks. Adjacent blocks of code perform equivalent tasks but adjacent lines within them may not.

A declaration of a 32-bit unsigned integer (`.u32`) called `%foo` is written:

```
.reg .u32 %foo;
```

For convenience, a sequence of 5 numbered `%bar` registers may by declared with:

```
.reg .u32 %bar<5>;
```

The PTX code generated for the experiments described in Section 6.3.3 used 32-bit floating point numbers (`.f32`) for the evaluation's native type, 32-bit unsigned integers (`.u32`) for some of the administration and a few Boolean predicates (`.pred`) for condition testing.

A typical instruction comprises three parts: the action to perform, qualified by the register type; the destination register and the source registers. For example, the code to set `%bar1` to the result of a 32-bit floating point addition of `%bar2` and `%bar3` is:

```
add.f32 %bar1, %bar2, %bar3;
```

Conditional execution is achieved in two steps: one instruction sets a predicate register according to some test and then a second instruction conditionally executes if that register is set to true. For example, to branch to (i.e. `goto`) `$codeLocationBaz` if `(%bar3==%bar1)` the code might be:

```
setp.eq.u32 %predicateVar, %bar3, %bar1;
@%progBranchPred bra $codeLocationBaz;
```

The GPU architecture is designed to execute the same instruction in parallel on multiple data. Although CUDA permits divergence of neighbouring threads, the documentation emphasises the considerable time penalty this entails. Hence, to maximise speed, good CUDA code should minimise any such divergence.

This raises the question of whether directly-coded PTX is faster or slower than the intermediate PTX generated from CUDA C source by the `nvcc` compiler. On one hand, the PTX that `nvcc` generates from CUDA C will have all the execution speed advantages that nVidia's compiler programmers could muster. On the other hand, directly-coding PTX might allow greater control than is possible through compiling CUDA C. For instance, where the compiler cannot identify that divergence is impossible, it may generate PTX with avoidable divergences.

PTX offers its programmers two divergence-minimising tools:

- explicit conditional instructions (such as `selp` and `slct` in Table 13), which are executed by all threads but which conditionally perform some limited action depending on a predicate and
- a qualified branch instruction `bra.uni`, which indicates that a branch is guaranteed to be non-divergent.

By using the former, it is possible to code many tasks with no conditional execution and hence no possibility of divergence. By using the latter, it is possible to guarantee to the compiler that many of the remaining branches will be uniform. What is the effect of this? The PTX manual `ptx_isa_2.2.pdf` has the following to say:

> A CTA ["Cooperative Thread Array"] with divergent threads may have lower performance than a CTA with uniformly executing threads, so it is important to have divergent threads re-converge as soon as possible. All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the `.uni` suffix. For divergent control flow, the optimizing code generator automatically determines points of re-convergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.

This leaves unclear precisely how much speed improvement (if any) might be available by avoiding non-uniform branches. In turn, this leaves open the question of whether avoiding non-uniform branches produces faster code (and indeed of whether writing PTX directly can produce faster kernels at all).

Nevertheless, care was taken to try to minimise the number of non-uniform branches in case there was a potential speed benefit. To achieve this, the design assumes that there are enough testcases (with appropriate padding) such that CUDA thread blocks (or at least warps) only evaluate data for one individual. If the number of testcases is so low that this is a problem, population-parallel approaches would likely be more appropriate anyway. If this assumption is violated, the code will attempt to diverge at a branch labelled as uniform. The consequences of this are unknown.

So did this effort actually manage to reduce non-uniform branches in the directly-generated PTX? To highlight the difference, the same population of 128 individuals, each with 200 TMBL instructions, was output as a PTX file and as a CUDA C file, which was then compiled to a PTX file using `nvcc`. The directly-generated PTX contained 384 branch instructions, all of which were uniform. The PTX compiled from CUDA C contained 5205 branch instructions, 3092 (around 59%) of which were non-uniform.

It is worth noting that, although PTX looks like assembly, compiling it to a cubin file is not a matter of direct translation. This can be seen by requesting verbose output from `nvcc` with the `--ptxas-options=-v` option. This shows that the compiler often uses far fewer registers than are directly implied by the PTX source.

6.3.3 Experimental Assessment

The tactic of using PTX introduces more complexity and so can only be justified if it either reduces compilation times or improves evaluation speeds (or both). This section

describes experiments to test this. Since the purpose of the work was to implement the same algorithm using faster techniques, the experiments examined speed, not results. Tests verified that the method generated results equal to those generated by standard methods.

The initial experiments appeared to give wildly varying results. On further investigation, the reason for this was found to be `nvcc`'s optimisation capabilities. When compiling from CUDA C to PTX, `nvcc` spotted dead instructions that could never affect the output and optimised them away. This resulted in the compilations and evaluations from CUDA C code occasionally being extremely fast.

TMBL evolves individuals with very few dead instructions so the experiments were seeded with a TMBL individual evolved during a long run. This means that this issue does not affect these experiments. However it is worth underlining this point: using PTX appears to sacrifice the optimisation of inactive code. This might be more of a problem in the context of GP's infamous introns.

One of the experiments varied the number of instructions per individual. To generate individuals with fewer instructions, instructions were removed from the start of the seed individual. This may change the individual so that more of the instructions may be optimised away. Hence the evaluation rates stated for CUDA C source for low numbers of instructions may be overestimates.

The system configuration is the same as described in Section 6.4.6 and the parameters are the same except a default of eight individuals per kernel was used here. Again, the mutation rate was set such that 95% of individuals have at least one mutation. Each experiment averaged results over five repetitions and the estimated standard error bars are often too thin to be visible suggesting that the relatively small number of repetitions has been adequate to give good estimates of the means.

Some of the estimated standard error bars are slightly thicker in some areas than in others. These differences may just reflect the differences in variance that one would expect to see between such small samples; they may be due to other factors such as other processes on the test computer interfering with some runs. The differences do not seem
pronounced enough to warrant an analysis of how likely it would be to see such differences by random chance.

6.3.4 Results of Experiments

Whilst discussing the results, it would be useful to work with a couple of assumptions: that performing the compiling and loading steps in pairs ($1 + 2$ or $2 + 3$) does not massively affect the overall duration and that the load time is small enough to disregard. The validity of these assumptions is checked in Figures 45 and 46. In both cases, the values are plotted over varying number of individuals per kernel although this is not too important in either case. Figure 45 and Table 14 show that performing the steps in

pairs does not massively affect the duration. Pairing steps 2 and 3 appears to increase the duration slightly but this effect is minor and will henceforth be disregarded. Figure 46 and Table 15 show that the load times are very short (compared to compilation times from other experiments) and so will be disregarded for the rest of the analysis.

Figures 47 and 48 consider the effect of varying the number of individuals per kernel on evaluation speed and compilation time. Figure 47 and Table 16 show two striking results: that the evaluation speeds are remarkably high (compared to population-parallel evaluation speeds) and that they are even higher from PTX source code. The results are fairly steady across varying numbers of individuals per kernel.

It is very positive that the PTX can achieve higher evaluation speeds but the main aim was to reduce compilation time. Figure 48 and Table 17 show that this has been achieved effectively too. Complete compilation time per individual to generate a cubin is much lower for PTX than for CUDA C and this effect intensifies as the number of individuals per kernel increases. Interestingly the compilation time for CUDA C from PTX to cubin suggests that the reduction in compilation time is only partly explained by avoiding step 1. Presumably the rest is due to the directly-generated PTX being simpler than the PTX that `nvcc` generates from equivalent CUDA C.

Figures 49 and 50 show the effect of varying the total number of programs on evaluation speed and compilation time. As would be hoped, neither is hugely affected. However compilation times per individual for CUDA C source do vary slightly, with the time to cubin higher for particularly small or large populations. Tables 18 and 19 show the corresponding values.

Figures 51 and 52 show the effect of varying the number of instructions per TMBL individual. Figure 51 and Table 20 show that for individuals with 90 or more instructions, the evaluation speeds are fairly steady and the evaluation speeds from PTX are consistently higher than those from CUDA C. At 300 instructions, the evaluation speed is 155837.159 million operations per second from CUDA C and 191724.434 million operations per second from PTX, an increase of 23.029%. Decreasing to 60 and then 30 instructions, both evaluations speeds get higher and at 30 instructions, CUDA C is faster than PTX.

It is worth commenting that the higher evaluation speeds for low numbers of instructions may be an artifact of the way individuals are constructed. It may be that these individuals with few instructions had a high fraction that could be optimised away so the rate of evaluation appears higher than it actually was.

Figure 52 and Table 21 show that the compilation time to cubin per individual is much lower for PTX than for CUDA C for all numbers of instructions. For both PTX and CUDA C, the compilation times per individual appear to be increasing more than linearly with respect to the number of instructions. This might be because the order of the compiler's algorithm is worse than linear; it might be because of demand on limited resources such as memory. At 300 instructions, the total compilation time per

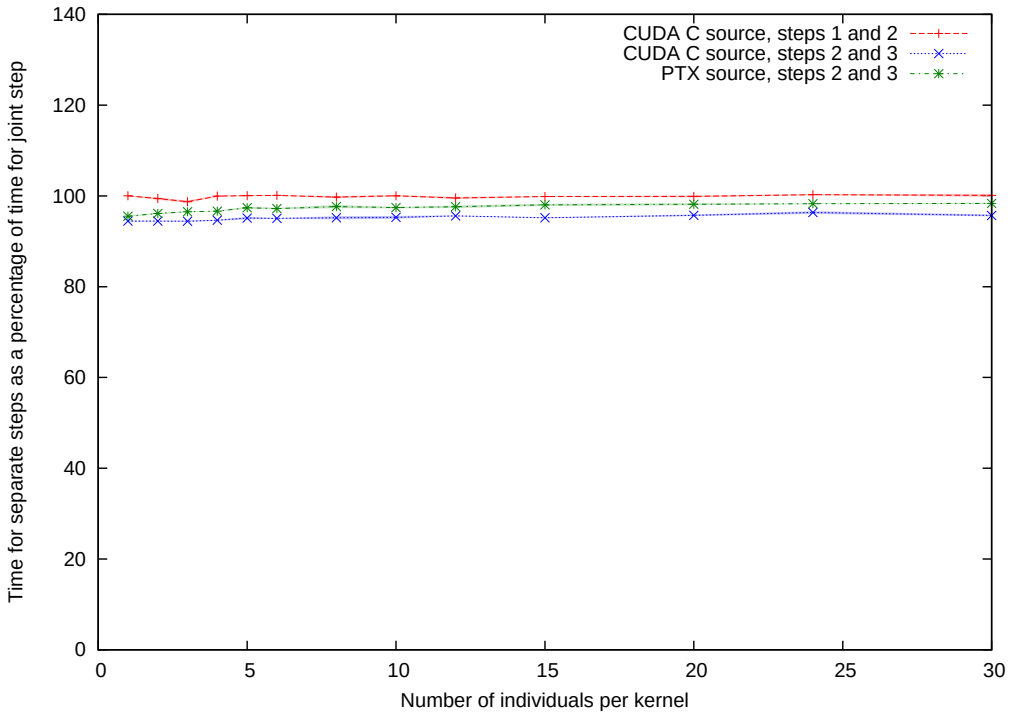


Figure 45: The linearity of combining pairs of steps over varying numbers of individuals per kernel for CUDA C and PTX source. This purpose of this is for checking the validity of assumptions as discussed in the text.

Programs per kernel	Source type	
	CUDA C	PTX
1	100.026 % [±0.093] 94.455 % [±0.126]	N/A 95.934 % [±0.115]
2	100.267 % [±0.163] 94.879 % [±0.550]	N/A 96.201 % [±0.081]
3	100.363 % [±0.334] 94.434 % [±0.189]	N/A 96.606 % [±0.147]
4	99.878 % [±0.080] 94.633 % [±0.100]	N/A 96.640 % [±0.261]
5	100.119 % [±0.498] 95.480 % [±0.849]	N/A 96.901 % [±0.088]
6	99.931 % [±0.138] 95.178 % [±0.155]	N/A 97.198 % [±0.112]
8	100.204 % [±0.105] 95.446 % [±0.307]	N/A 97.412 % [±0.159]
10	99.746 % [±0.356] 95.632 % [±0.095]	N/A 97.308 % [±0.093]
12	100.284 % [±0.121] 96.154 % [±0.223]	N/A 97.857 % [±0.205]
15	99.806 % [±0.142] 95.464 % [±0.083]	N/A 98.063 % [±0.314]
20	100.231 % [±0.319] 96.028 % [±0.484]	N/A 98.099 % [±0.142]
24	99.813 % [±0.063] 95.658 % [±0.119]	N/A 98.285 % [±0.168]
30	100.020 % [±0.135] 95.296 % [±0.322]	N/A 98.311 % [±0.183]

Table 14: The linearity of combining pairs of steps over varying numbers of individuals per kernel for CUDA C and PTX source

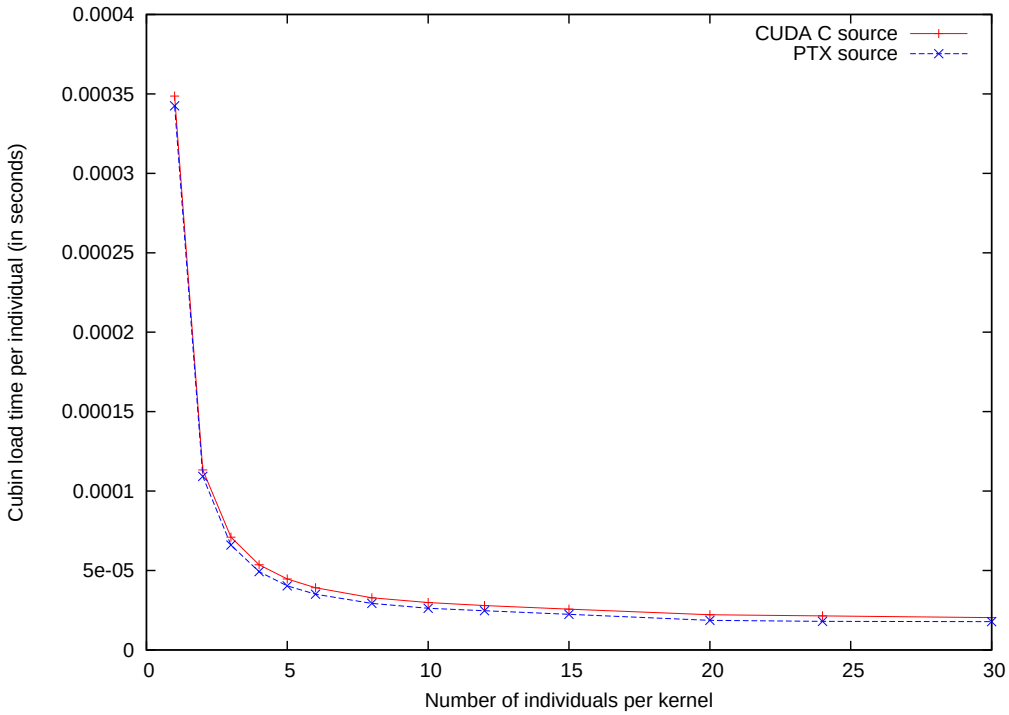


Figure 46: The cubin load time per individual over varying numbers of individuals per kernel for CUDA C and PTX source. This purpose of this is for checking the validity of assumptions as discussed in the text.

Programs per kernel	Source type	
	CUDA C	PTX
1	0.00034863 s [±0.00000010]	0.00034462 s [±0.00000203]
2	0.00011336 s [±0.00000041]	0.00010839 s [±0.00000020]
3	0.00007137 s [±0.00000006]	0.00006656 s [±0.00000024]
4	0.00005386 s [±0.00000018]	0.00004903 s [±0.00000006]
5	0.00004449 s [±0.00000005]	0.00004027 s [±0.00000007]
6	0.00003906 s [±0.00000007]	0.00003493 s [±0.00000002]
8	0.00003356 s [±0.00000008]	0.00002913 s [±0.00000004]
10	0.00002979 s [±0.00000005]	0.00002604 s [±0.00000001]
12	0.00002747 s [±0.00000021]	0.00002408 s [±0.00000004]
15	0.00002538 s [±0.00000005]	0.00002205 s [±0.00000003]
20	0.00002183 s [±0.00000008]	0.00001850 s [±0.00000019]
24	0.00002160 s [±0.00000013]	0.00001777 s [±0.00000007]
30	0.00002061 s [±0.00000005]	0.00001739 s [±0.00000007]

Table 15: The cubin load time per individual over varying numbers of individuals per kernel for CUDA C and PTX

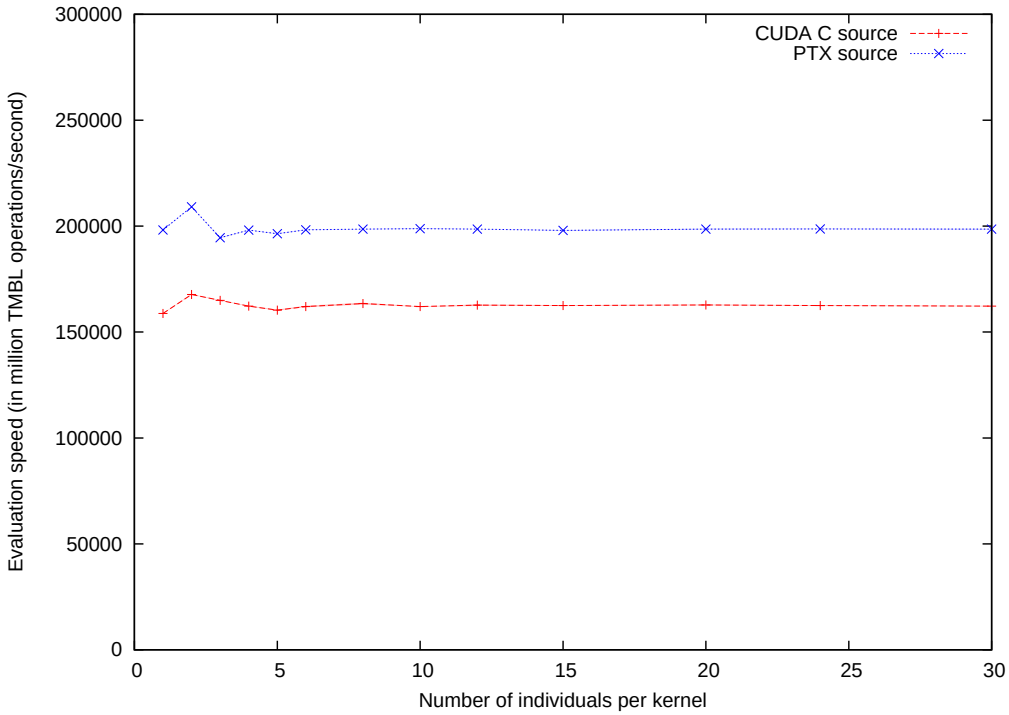


Figure 47: The effect on evaluation speed of varying the number of individuals per kernel for CUDA C and PTX source

Programs per kernel	Source type		Change
	CUDA C	PTX	
1	158805.123 Mo/s [±134.257]	197741.473 Mo/s [±87.397]	+24.518%
2	167831.260 Mo/s [±242.884]	209315.151 Mo/s [±173.477]	+24.718%
3	165032.511 Mo/s [±149.071]	194343.676 Mo/s [±143.120]	+17.761%
4	161646.680 Mo/s [±74.796]	198211.357 Mo/s [±83.347]	+22.620%
5	160425.626 Mo/s [±306.146]	196488.093 Mo/s [±43.781]	+22.479%
6	161986.958 Mo/s [±327.900]	198429.966 Mo/s [±57.069]	+22.497%
8	162159.380 Mo/s [±202.829]	198788.216 Mo/s [±98.733]	+22.588%
10	162449.723 Mo/s [±220.745]	198560.674 Mo/s [±132.443]	+22.229%
12	162621.976 Mo/s [±267.583]	198554.999 Mo/s [±110.307]	+22.096%
15	162186.285 Mo/s [±324.353]	198265.132 Mo/s [±182.975]	+22.245%
20	163202.853 Mo/s [±488.099]	198737.958 Mo/s [±180.982]	+21.774%
24	162935.422 Mo/s [±344.845]	198783.705 Mo/s [±103.458]	+22.002%
30	163270.273 Mo/s [±391.122]	199012.729 Mo/s [±135.569]	+21.892%

Table 16: The effect on evaluation speed of varying the number of individuals per kernel for CUDA C and PTX source

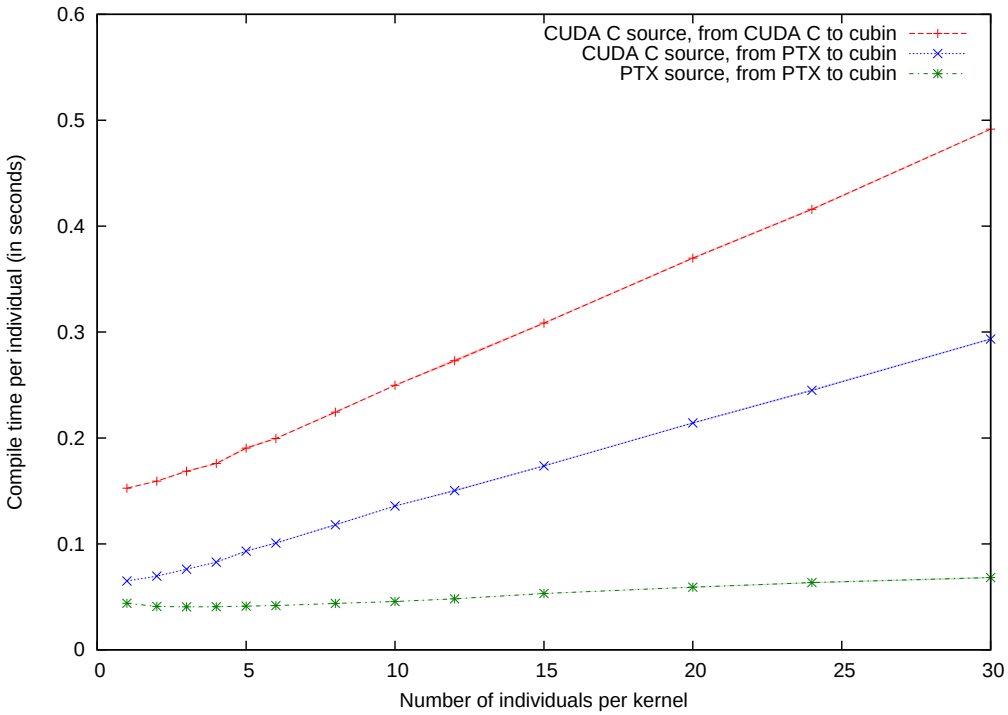


Figure 48: The effect on compilation time per individual of varying the number of individuals per kernel for CUDA C and PTX source

Programs per kernel	Source type		Change
	CUDA C	PTX	
1	0.15263 s [±0.00013] 0.06493 s [±0.00007]	N/A 0.04399 s [±0.00003]	−71.179%
2	0.15621 s [±0.00016] 0.06996 s [±0.00038]	N/A 0.04078 s [±0.00005]	−73.894%
3	0.16375 s [±0.00018] 0.07588 s [±0.00004]	N/A 0.04021 s [±0.00005]	−75.444%
4	0.17355 s [±0.00014] 0.08355 s [±0.00010]	N/A 0.04033 s [±0.00016]	−76.762%
5	0.18402 s [±0.00038] 0.09192 s [±0.00090]	N/A 0.04059 s [±0.00013]	−77.943%
6	0.19416 s [±0.00031] 0.09965 s [±0.00024]	N/A 0.04118 s [±0.00014]	−78.791%
8	0.21911 s [±0.00054] 0.11857 s [±0.00035]	N/A 0.04296 s [±0.00006]	−80.393%
10	0.24087 s [±0.00101] 0.13318 s [±0.00066]	N/A 0.04458 s [±0.00020]	−81.492%
12	0.26194 s [±0.00041] 0.14886 s [±0.00059]	N/A 0.04775 s [±0.00020]	−81.771%
15	0.29817 s [±0.00082] 0.17198 s [±0.00063]	N/A 0.05146 s [±0.00040]	−82.741%
20	0.35215 s [±0.00116] 0.20961 s [±0.00151]	N/A 0.05776 s [±0.00010]	−83.598%
24	0.39883 s [±0.00146] 0.23947 s [±0.00124]	N/A 0.06332 s [±0.00046]	−84.124%
30	0.46791 s [±0.00136] 0.28612 s [±0.00138]	N/A 0.06909 s [±0.00139]	−85.234%

Table 17: The effect on compilation time per individual of varying the number of individuals per kernel for CUDA C and PTX source (where the first value in each cell is from CUDA C to cubin and the second value is from PTX to cubin)

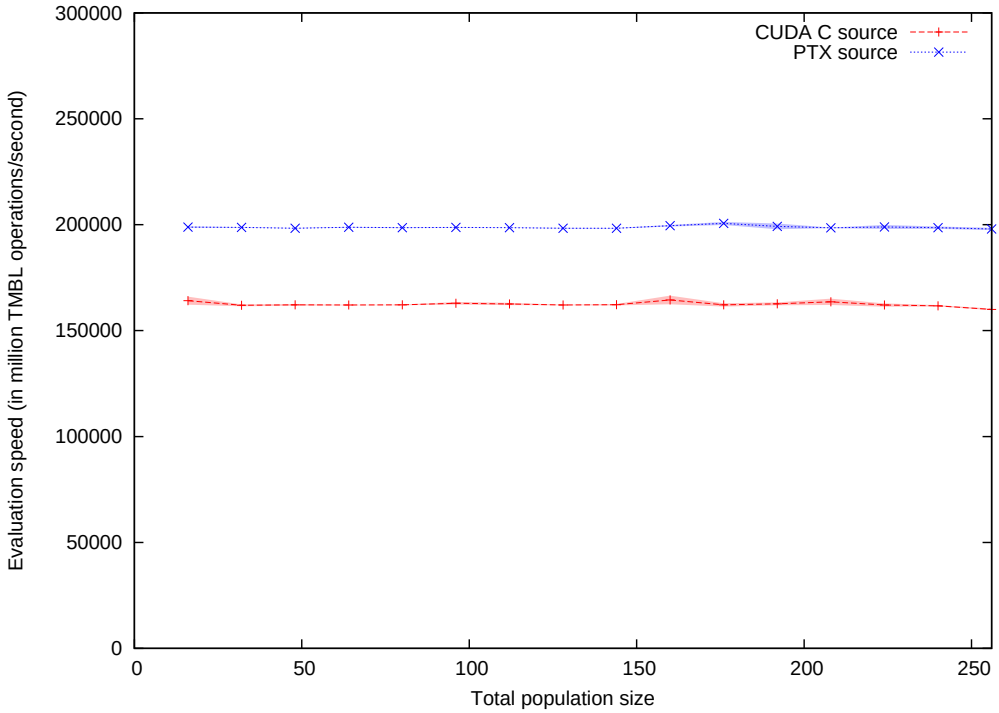


Figure 49: The effect on evaluation speed of varying population sizes for CUDA C and PTX source

Population	Source type		Change
	CUDA C	PTX	
16	164139.446 Mo/s [±1923.278]	198895.037 Mo/s [±337.057]	+21.174%
32	161943.142 Mo/s [±430.873]	198733.168 Mo/s [±138.417]	+22.718%
48	162170.397 Mo/s [±386.229]	198345.937 Mo/s [±115.541]	+22.307%
64	162087.015 Mo/s [±116.017]	198834.684 Mo/s [±177.841]	+22.672%
80	162152.265 Mo/s [±156.723]	198585.415 Mo/s [±178.455]	+22.468%
96	162918.939 Mo/s [±588.722]	198714.732 Mo/s [±129.684]	+21.972%
112	162551.626 Mo/s [±539.116]	198619.793 Mo/s [±66.806]	+22.189%
128	162103.340 Mo/s [±126.193]	198358.097 Mo/s [±162.960]	+22.365%
144	162219.577 Mo/s [±342.545]	198343.349 Mo/s [±131.611]	+22.268%
160	164478.474 Mo/s [±2060.750]	199518.908 Mo/s [±364.757]	+21.304%
176	162151.190 Mo/s [±801.189]	200620.720 Mo/s [±709.098]	+23.724%
192	162672.407 Mo/s [±677.593]	199202.037 Mo/s [±1331.087]	+22.456%
208	163576.934 Mo/s [±1493.960]	198527.188 Mo/s [±133.804]	+21.366%
224	162082.569 Mo/s [±757.031]	198914.113 Mo/s [±951.853]	+22.724%
240	161720.934 Mo/s [±299.484]	198611.198 Mo/s [±556.692]	+22.811%
256	160033.266 Mo/s [±389.726]	198008.989 Mo/s [±549.420]	+23.730%

Table 18: The effect on evaluation speed of varying population sizes for CUDA C and PTX source

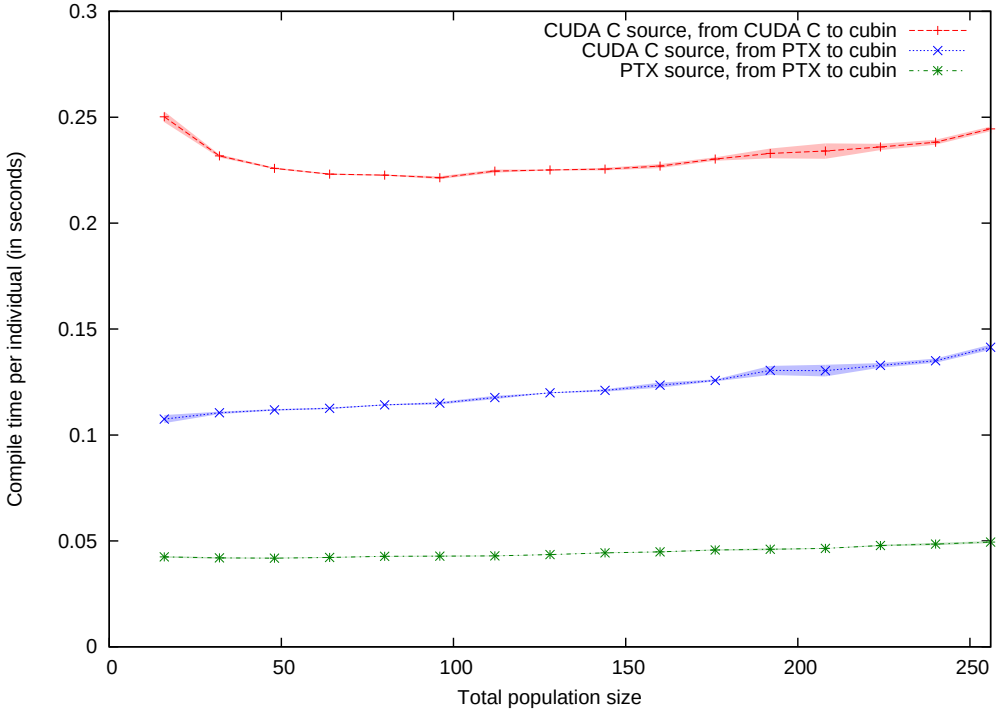


Figure 50: The effect on compilation time per individual of varying population sizes for CUDA C and PTX source

Population	Source type		Change
	CUDA C	PTX	
16	0.25020 s [±0.00250] 0.10751 s [±0.00192]	N/A 0.04243 s [±0.00038]	−83.042%
32	0.23174 s [±0.00071] 0.11047 s [±0.00059]	N/A 0.04197 s [±0.00030]	−81.889%
48	0.22588 s [±0.00037] 0.11183 s [±0.00035]	N/A 0.04188 s [±0.00018]	−81.459%
64	0.22312 s [±0.00032] 0.11260 s [±0.00033]	N/A 0.04219 s [±0.00019]	−81.091%
80	0.22271 s [±0.00027] 0.11422 s [±0.00018]	N/A 0.04272 s [±0.00016]	−80.818%
96	0.22148 s [±0.00064] 0.11500 s [±0.00058]	N/A 0.04283 s [±0.00022]	−80.662%
112	0.22457 s [±0.00075] 0.11769 s [±0.00077]	N/A 0.04290 s [±0.00015]	−80.897%
128	0.22512 s [±0.00021] 0.11989 s [±0.00012]	N/A 0.04353 s [±0.00008]	−80.664%
144	0.22555 s [±0.00061] 0.12106 s [±0.00057]	N/A 0.04436 s [±0.00014]	−80.333%
160	0.22694 s [±0.00095] 0.12352 s [±0.00119]	N/A 0.04480 s [±0.00018]	−80.259%
176	0.23029 s [±0.00080] 0.12577 s [±0.00045]	N/A 0.04570 s [±0.00028]	−80.155%
192	0.23290 s [±0.00230] 0.13047 s [±0.00212]	N/A 0.04603 s [±0.00045]	−80.236%
208	0.23406 s [±0.00364] 0.13039 s [±0.00277]	N/A 0.04644 s [±0.00008]	−80.159%
224	0.23596 s [±0.00148] 0.13289 s [±0.00111]	N/A 0.04781 s [±0.00036]	−79.738%
240	0.23820 s [±0.00117] 0.13508 s [±0.00098]	N/A 0.04851 s [±0.00053]	−79.635%
256	0.24453 s [±0.00110] 0.14141 s [±0.00121]	N/A 0.04947 s [±0.00049]	−79.769%

Table 19: The effect on compilation time per individual of varying population sizes for CUDA C and PTX source (where the first value in each cell is from CUDA C to cubin and the second value is from PTX to cubin)

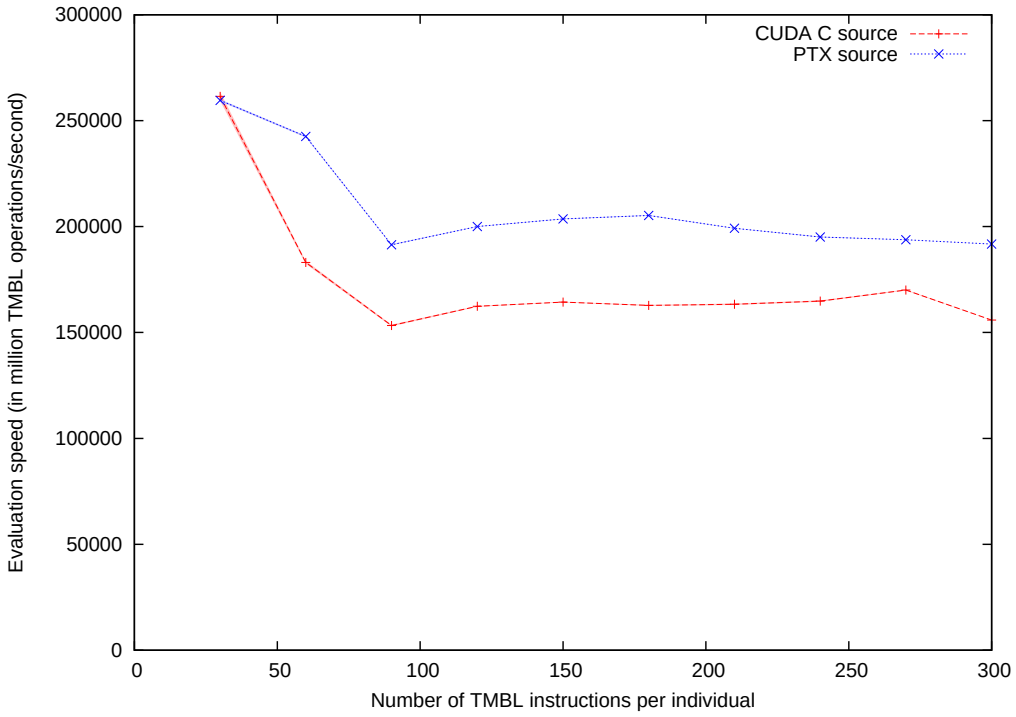


Figure 51: The effect on evaluation speed of varying numbers of TMBL instructions per individual for CUDA C and PTX source

Instructions	Source type		Change
	CUDA C	PTX	
30	261474.862 Mo/s [±2124.695]	259582.278 Mo/s [±498.381]	−0.724%
60	183085.021 Mo/s [±728.449]	242544.446 Mo/s [±372.822]	+32.476%
90	153284.933 Mo/s [±403.871]	191423.677 Mo/s [±250.859]	+24.881%
120	162355.469 Mo/s [±244.062]	200016.133 Mo/s [±140.292]	+23.196%
150	164318.236 Mo/s [±138.657]	203611.813 Mo/s [±115.007]	+23.913%
180	162802.268 Mo/s [±232.489]	205253.255 Mo/s [±162.186]	+26.075%
210	163329.531 Mo/s [±316.118]	199176.476 Mo/s [±111.705]	+21.948%
240	164816.979 Mo/s [±316.891]	195103.068 Mo/s [±154.135]	+18.376%
270	170075.087 Mo/s [±203.362]	193755.862 Mo/s [±42.270]	+13.924%
300	155837.159 Mo/s [±41.382]	191724.434 Mo/s [±38.273]	+23.029%

Table 20: The effect on evaluation speed of varying numbers of TMBL instructions per individual for CUDA C and PTX source

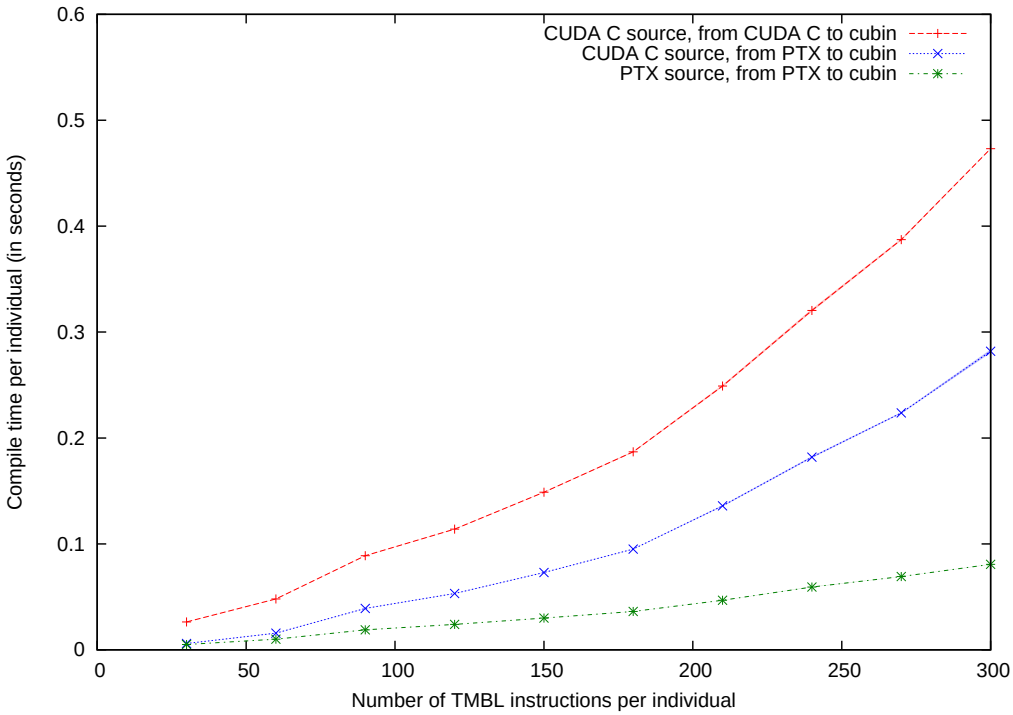


Figure 52: The effect on compilation time per individual of varying numbers of TMBL instructions per individual for CUDA C and PTX source

Instructions	Source type		Change
	CUDA C	PTX	
30	0.02636 s [±0.00006] 0.00597 s [±0.00005]	N/A 0.00505 s [±0.00000]	−80.842%
60	0.04797 s [±0.00013] 0.01571 s [±0.00009]	N/A 0.01010 s [±0.00005]	−78.945%
90	0.08881 s [±0.00014] 0.03910 s [±0.00014]	N/A 0.01890 s [±0.00004]	−78.719%
120	0.11400 s [±0.00016] 0.05314 s [±0.00012]	N/A 0.02401 s [±0.00005]	−78.939%
150	0.14878 s [±0.00028] 0.07295 s [±0.00016]	N/A 0.03004 s [±0.00010]	−79.809%
180	0.18691 s [±0.00049] 0.09506 s [±0.00010]	N/A 0.03618 s [±0.00003]	−80.643%
210	0.24913 s [±0.00096] 0.13590 s [±0.00063]	N/A 0.04674 s [±0.00004]	−81.239%
240	0.32045 s [±0.00129] 0.18191 s [±0.00076]	N/A 0.05931 s [±0.00018]	−81.492%
270	0.38727 s [±0.00089] 0.22367 s [±0.00061]	N/A 0.06917 s [±0.00009]	−82.139%
300	0.47322 s [±0.00023] 0.28188 s [±0.00161]	N/A 0.08074 s [±0.00017]	−82.938%

Table 21: The effect on compilation time per individual of varying numbers of TMBL instructions per individual for CUDA C and PTX source (where the first value in each cell is from CUDA C to cubin and the second value is from PTX to cubin)

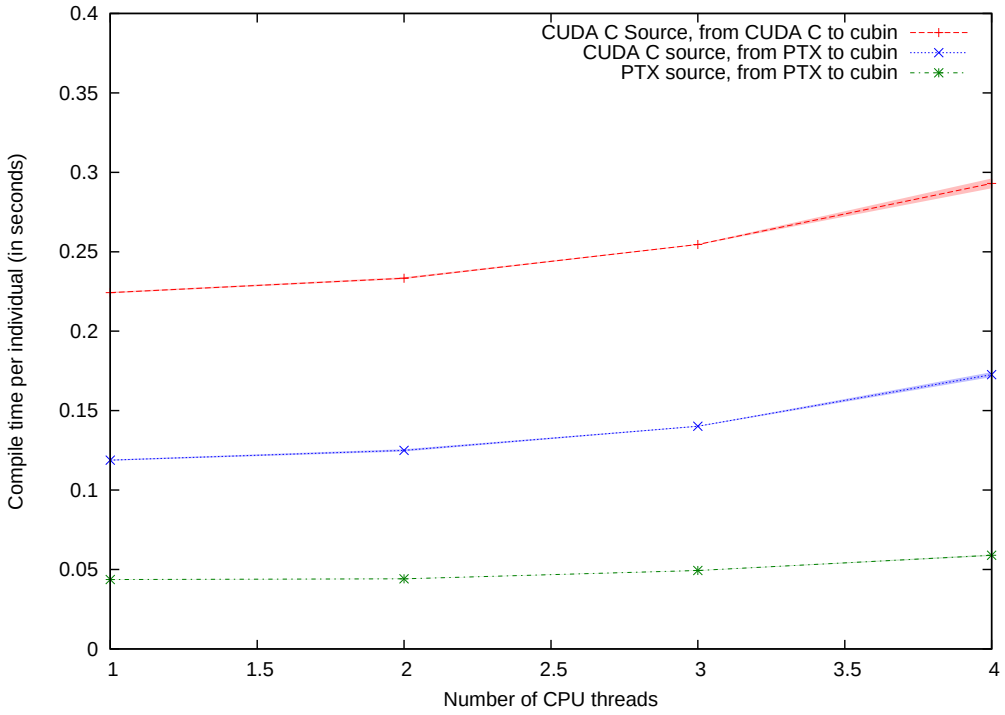


Figure 53: Using multiple threads for compilation

Number of CPU threads	Source type		Change
	CUDA C	PTX	
1	0.22428 s [±0.00042] 0.11881 s [±0.00037]	N/A 0.04365 s [±0.00016]	−80.538%
2	0.23336 s [±0.00060] 0.12498 s [±0.00072]	N/A 0.04412 s [±0.00011]	−81.094%
3	0.25455 s [±0.00027] 0.14014 s [±0.00024]	N/A 0.04936 s [±0.00015]	−80.609%
4	0.29302 s [±0.00310] 0.17267 s [±0.00153]	N/A 0.05895 s [±0.00044]	−79.882%

Table 22: Using multiple threads for compilation

individual is 0.473 seconds from CUDA C and 0.081 seconds from PTX, a decrease of 82.938%.

Figure 53 and Table 22 show the effects of compiling with multiple threads on a 4 core machine. In all cases, compilation time per individual increases when using multiple threads. This effect is not strong enough to prevent parallel compilation being worthwhile but it is somewhat disappointing.

6.3.5 Comments on Compiling from the Lower-Level Language PTX

Hopefully the findings described here will help researchers to understand these factors better and so permit them to gauge the suitability of PTX more accurately. A number of questions remain open for future investigation:

- Is it possible to improve the directly-coded PTX by manual comparing it to the PTX that `nvcc` generates from CUDA C?
- Is it possible to increase the number of instructions with a linear increase in compilation time?
- Is it possible to increase the number of parallel compilations with a smaller time penalty?
- Do the uniform branches contribute to the improved execution speed?

Beyond this, there are two obvious possibilities to pursue: using PTX to write a population-parallel interpreter and directly manipulating a data-parallel cubin binary. Despite the success of the work described, these remain daunting prospects.

With work, a PTX interpreter might be possible and it might be expected to deliver a small improvement in evaluation speed. However this does not seem worth the huge increase in the difficulty of developing and maintaining the code.

Attempts to manipulate cubin binaries directly may be unwise because nVidia recommend storing CUDA C or PTX source files rather than cubin binaries to defend against any radical changes they may make to the cubin format. Nevertheless, if it were possible to manipulate the cubin files directly rather than having to compile afresh each time, this might slash the CPU time spent preparing each individual for GPU evaluation. This aspiration motivated a brief investigation using the Python program `decuda`, which attempts to disassemble cubin files back to PTX code. This quickly revealed a complex relationship between PTX and the cubin file to which it compiles. The investigation was promptly curtailed.

6.4 Technique 2: Reducing Repeated Code through Alignment

To exploit a GPU, one must write and compile a kernel for it. The data-parallel approach involves writing such kernels each time a batch of individuals is to be evaluated. These kernels are then used for evaluation on the GPU. Several individuals may

be grouped together into one kernel. The problem of this approach is the time spent compiling.

The technique discussed in this section aimed to reduce compilation time by targeting the redundancy of compiling blocks of similar code by exploiting their similarities. Rather than sending highly redundant code like that shown on the left of Table 23 to the compiler, an aligning algorithm is first used to identify the similarities and pull them together to form code like that shown on the right of Table 23.

Unaligned	Aligned

`if (prog == 0) {`
`  ...`
`  slot1 += slot3;`
`  slot2 = 3.1096370;`
`  ...`
`}`
`else if (prog == 1) {`
`  ...`
`  slot1 += slot3;`
`  slot2 = 3.1096370;`
`  ...`
`}`
`else if (prog == 2) {`
`  ...`
`  slot1 += slot3;`
`  slot2 *= slot1;`
`  slot2 = 3.1096370;`
`  ...`
`}`
	`...`
	`slot1 += slot3;`
	`if (prog == 2) {`
	`  slot2 *= slot1;`
	`}`
	`slot2 = 3.1096370;`
	`...`

Table 23: Reducing work for the compiler through alignment. On the left, each program's code is in a separate block. Since the compiler does not know these blocks are highly similar, it repeats work by compiling each separately. On the right, the similarities have been identified first so the common instructions are pulled together and need only be compiled once.

The biggest danger with this approach is that the execution speed is reduced because the execution path through the kernel for each individual is more convoluted. Can the faster compilation outweigh the slower evaluation? If so, under what circumstances? The investigation sought to tackle these questions.

How should the individuals be aligned? It might be possible to identify these similarities by keeping track of the mutation and crossover operations performed on each individual. However that would be a rather complicated and brittle approach which would need extending with each new genetic operator. A more robust approach is to align individuals when they are to be evaluated.

6.4.1 Alignment

How can lists of instructions be aligned? There is a standard approach to aligning lists of items but it was not used in this context. To explain why, it is important to describe the standard approach first.

The principles underlying alignment algorithms are not dependent on the type of thing being aligned, so this section talks about aligning lists made up of instructions, letters, items, amino acids and coloured shapes. The simplicity of letters makes them suitable for outlining the principles.

Consider two lists of letters, `DVSGGWIVHGVRGS` and `SGGWVHGRKGSA`. An alignment of these two lists involves laying them out such that some items from one list might tally with some items from the other. Hence an alignment is a list of alignment positions, each containing the next entry from one or more of the lists. For example, one possible alignment is as follows:

```
DVSGGWIVHGVRG.....S
  |  |  |||||
..S..G..GWVHGRKGSA
```

This is a poor quality alignment for two reasons. First, relatively few pairs of letters have been aligned. Second, many of the aligned pairs do not contain matching letters. This is permitted in other contexts but for this application, instructions may only be aligned with each other if they are identical. Under these criteria, the following alignment is an improvement:

```
DVSGGWIVHGVR.GS
  |||| ||| | ||
..SGGW.VHG.RKGSA
```

These alignments involve a single pair of lists. Later, it will be necessary to create multiple alignments with more than two lists. That algorithm will be described after the single alignment algorithm, upon which it is built.

How many possible ways are there to align a list containing m items with a list containing n items? That depends on whether different alignments that tally the same pairs of items should only be counted once. If not, the number of ways of performing the alignment may be calculated iteratively. If either of the lists is empty, there is only one possible way since there are no choices that need to be made. Otherwise, there are three recipes for aligning the lists:

- take the first item off the *first* list, add it to the end of the alignment and then align the remaining items in any possible way,
- take the first item off the *second* list, add it to the end of the alignment and then align the remaining items in any possible way or

- take the first item off *both* lists, tally them at the end of the alignment and then align the remaining items in any possible way.

The total number of ways is the sum of these possibilities and so can be calculated with the following iterative formula: $F(m,n) = F(m-1,n-1) + F(m,n-1) + F(m-1,n)$, where $F(0,i) = 1$ and $F(i,0) = 1$ for any i. This formula generates the result that there are around $2.054 * 10^{75}$ possible ways to align two lists of 100 items. Fortunately things are much better than this calculation suggests since the problem exhibits optimal substructure, i.e. optimal solutions can be calculated from optimal solutions to subproblems. This makes the problem amenable to standard dynamic programming techniques.

6.4.2 The Needleman and Wunsch Algorithm

One of the core techniques in the field of bioinformatics involves aligning amino acid sequences so it has developed the area considerably. The basic algorithm used for this is the Needleman-Wunsch (NW) algorithm [64]. This is the standard alignment algorithm, against which others may be contrasted.

A scoring system is used to guide the algorithm and the standard scheme awards an alignment one point for tallying each pair of identical items. The NW algorithm is an application of dynamic programming to the process of alignment. The principle of dynamic programming is to find the optimal solution to each subproblem and then reuse these results. This can be applied to alignment by using the following subproblem: align the two query sequences but with one or more items taken off the front of either or both. The optimal solution will contain the optimal solution to one of these subproblems. This observation can be applied iteratively to build up the optimal solution in simple steps.

For two sequences of lengths m and n, this process can be represented elegantly in a matrix of dimensions $m \times n$. Using conventional matrix indexing, the sequences both start in the top-left and finish in the bottom-right. Each possible alignment can be represented as a path through the matrix between these corners. Figures 54(a) and 54(b) can be viewed as such matrices.

6.4.3 The Need for a New Alignment Algorithm

What is the computational complexity of the NW algorithm with respect to the sizes of the sequences? For each cell, the original NW algorithm scans for the maximum values in the strips running down and right from the below-right cell. To examine the $(i-1)+(j-1)-1$ such cells for every position (i,j) (where $i > 1$ and $j > 1$) in an $m \times n$ matrix the number of examinations is:

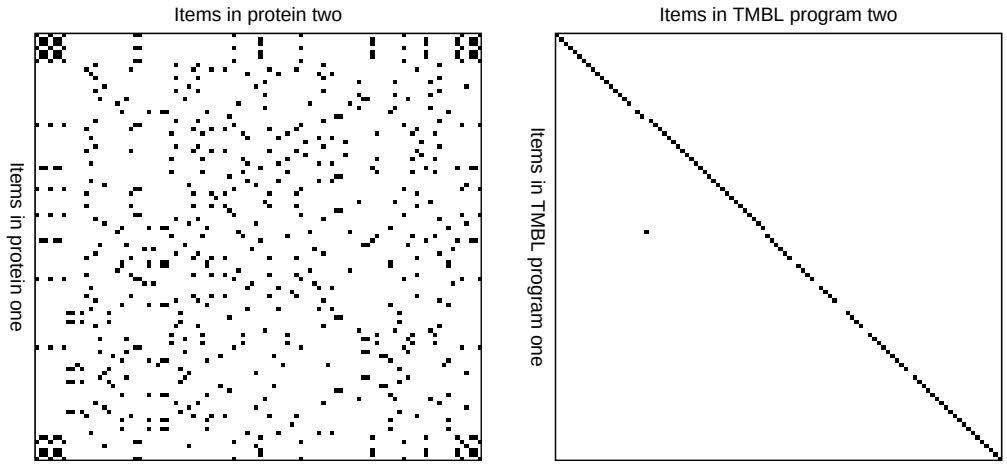
$$\sum_{i=2}^{m} \sum_{j=2}^{n} (i+j-3)$$

$$= \frac{1}{2}\left(m^2n + mn^2 - m^2 - n^2 - 4mn + 3m + 3n - 2\right)$$

This means the algorithm takes $O(n^3)$ time to align two sequences of length n and $O(n^2)$ time to align a sequence of length n against a non-trivial, fixed-length sequence. Sankoff [78] later showed how to refine the algorithm to reuse more information and hence reduce the two running times from $O(n^3)$ and $O(n^2)$ to $O(n^2)$ and $O(n)$ respectively. This is now widely referred to as the NW algorithm.

This means that NW is slower than would ideally be required here. Conversely, consideration of NW's requirements will show that they are much more stringent than those required here. NW alignments are used to provide a consistent measure of the level of sequence similarity between proteins and to identify stretches of sequences that are most similar. NW often faces very difficult problems and is relied upon to perform as good an alignment as possible.

In contrast, this problem only requires an alignment algorithm to face simple problems and to do a fairly good job quickly. Figure 54 helps to highlight the difference between the sorts of problems that might be faced.



(a) Aligning two protein sequences of length 100 (b) Aligning two TMBL programs of length 100

Figure 54: Aligning TMBL programs is very different to aligning protein sequences so NW may not be appropriate. A black pixel indicates a match between the two corresponding items such that they could be aligned. Subfigure 54(a) shows the problem of aligning two sequences of length 100 from the proteins 2yuv and 2yuz. Subfigure 54(b) shows the problem of aligning two TMBL programs with 100 instructions. The two programs share the same single parent.

Figure 54(a) shows the problem of aligning two protein sequences from the files 2yuv and 2yuz in the Protein Data Bank (PDB) [9]. These sequences score highly (35% sequence identity over 97% overlap using a gap penalty of 3), which provides good evidence of relatedness and suggests that this is a relatively easy alignment problem. A faint line can be discerned running from top left to bottom right, a likely rough path for the optimal alignment. However the signal for this path is weak and is difficult to

discern due to the substantial noise of chance matches. Since each item is one of only 20 amino acids, we would expect such "false positive" matches in 5% of cases.

Contrast this with Figure 54(b) which shows the problem of aligning two 100 instruction TMBL programs. Here, there are very few items that match by chance since there are very many possible instructions. Furthermore the signal is very strong because they have received few mutations since copying their shared parent's genome.

Note that there is another important difference: in this application two instructions may only be aligned together if they match whereas bioinformatics sequence alignments may include many aligned pairs that do not.

All of this motivates the design of a new, rough alignment algorithm.

6.4.4 A Rough Alignment Algorithm

The proposed algorithm's core idea is to just keep looking for the next matching pair. The algorithm is only ever interested in the next best step: it does not look far around for better, less direct routes and it never turns back from its current path. This narrowly focused approach is in contrast to the global approach of NW. For problems such as the one shown in Figure 54(a), the false positives could easily lead this algorithm astray through poor alignment routes. However for problems such as the one shown in Figure 54(b), the algorithm should rarely wander off track and should quickly find its way back to the main path if it does.

In more detail, the algorithm starts from just outside the top-left of the alignment matrix. It repeatedly looks for the next matching position, and moves there. When the algorithm can find no more matching pairs to the bottom-right of its current position, it stops. Non-matching items are never aligned and so can be ignored by the algorithm.

How does the algorithm choose the next matching position based on its current location? In short, it sweeps out and selects the first match it finds. The sweep is summarised in Figure 55 and depicted in Figure 56.

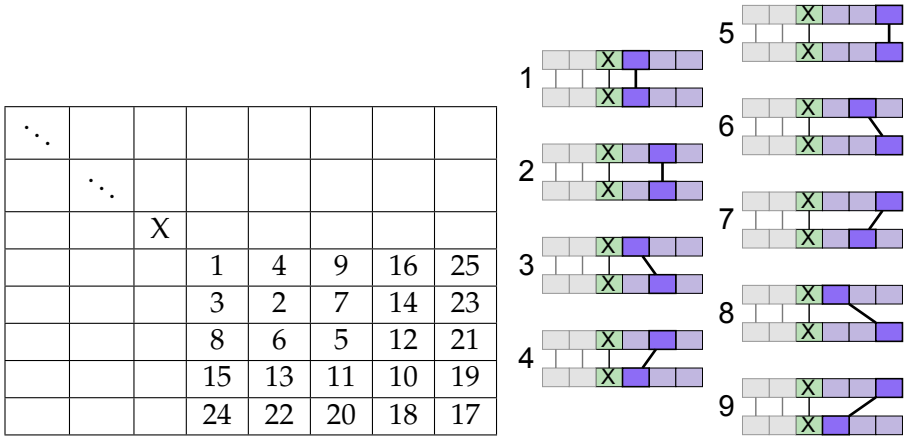
The worst case for the algorithm is aligning two sequences with no matching items. This would require the described algorithm to search the entire matrix for the first match before giving up. Hence the algorithm aligns two sequences of length n in amortised $O(n^2)$ time, like the NW algorithm. However the best case is aligning two identical sequences and the described algorithm would perform this in linear time whereas the NW algorithm would still take $O(n^2)$. Furthermore, this speed should degrade gracefully so that adding a few, small mutations should add little time to the alignment.

6.4.5 A Rough Multiple Alignment Algorithm

This mechanism provides the means to align pairs of lists as illustrated in Figure 57(a) but this is only part of the problem of forming a multiple alignment. How should the

```
Stage One (neighbouring pair alignments)
 * Align each neighbouring list pair as follows
 * Start at the top-left corner of the matrix
 * Work diagonally down-right through matches
 * If the next down-right item is not a match, search progressively
further down and right from the previous match
 * In searching, prefer:
    1st:  smaller maximum increments
    2nd:  more diagonal jumps
    3rd:  smaller first-list increments
 * Continue from the next match found
 * Repeat until there are no more matches to the bottom right of the
previous match
 * Glue the resulting alignments together
```

Figure 55: A summary of the first stage of the alignment algorithm.

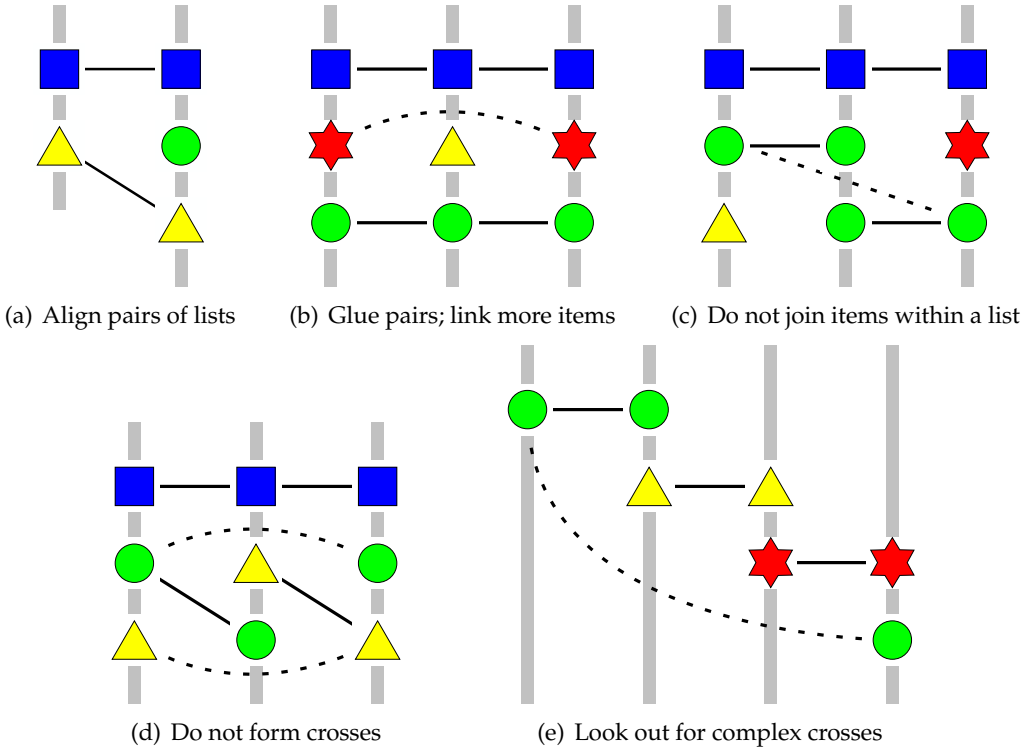


(a) First 17 search positions (b) First nine as diagrams

Figure 56: The order in which the proposed alignment algorithm sweeps to find the next match. The X denotes the current location and the numbers indicate the sequence of the sweep.

code deal with aligning more than two lists? A globally optimum NW-based algorithm can align k sequences of length n using a k-dimensional matrix. That algorithm runs in $O(n^k)$ time (with respect to both n and k) which is unacceptable for all but the smallest of cases. The compromise often adopted in bioinformatics methods is to perform all-versus-all pairwise comparisons (typically alignments) and then use the results iteratively to identify the next most similar list and add it into a core (by performing further alignments). This has much better running time since it requires $\frac{k(k-1)}{2}$ alignments and so — when used in conjunction with NW— runs in $O(n^2k^2)$ time (with respect to n and k).

Even so, this problem demands a quicker, rougher approach. Rather than aligning all $\frac{k(k-1)}{2}$ pairs, the new algorithm aligns each of the $k-1$ pairs of neighbouring lists



(a) Align pairs of lists (b) Glue pairs; link more items (c) Do not join items within a list

(d) Do not form crosses (e) Look out for complex crosses

Figure 57: Grey strips represent lists of items, coloured shapes represent items being aligned, solid black lines represent alignment links and dashed lines represent possible new links. 57(a) First, items are aligned within pairs of neighbouring lists. 57(b) Then these alignments are glued together and extra connections (e.g. the dashed line) may be formed. These links must not be formed if they connect items within a list or form crosses. 57(c) Joining items within a list (e.g. by forming the dashed line) breaks the alignment since it means one of those duplicates will be absent from the resulting code. 57(d) Forming a cross (e.g. by forming either of the dashed lines) breaks the alignment since then one joined group (e.g. joined circles) comes both before and after another (e.g. joined triangles). 57(e) Identifying some crosses (such as the one created by joining the third circle to the other two) may involve tracing back through lists that do not include any of the items to be joined (e.g. the list containing one triangle and one star).

and then glues the resulting alignments together. As shown in Figure 57(b), this may leave some easy connections missed out. For instance, if one individual with one mutated item is placed in the middle of many otherwise identical lists, the mutated item will break the connection between identical items on either side.

Figure 58 shows an example of individuals being aligned after stage one and after stage two. Notice situations such as the one at the top of Figure 58(a): the first mutated instruction has broken the connection between its neighbours. In Figure 58(b), these two groups have been connected.

The proposed algorithm's second stage takes the aligned-and-glued neighbour-pair-lists from the first stage and scans for these extra connections. This avoids spending time on another alignment as is usual in bioinformatics multiple alignments. Since these connections are not formed in an alignment process, they are vulnerable to two

(a) Stage one

(b) Stage two

Figure 58: Aligning 20 individuals, each with 80 instructions. Columns within these plots represent individuals, black marks represent individuals' instructions and rows of black marks represent aligned instructions. After stage one there are 183 positions; after stage two, more connections have been identified and there are only 137 positions.

new dangers, which are worth highlighting.

The first danger is connecting items such that two identical items within the same list are placed in the same group of equivalents. Figure 57(c) shows an example in which the proposed dashed link would result in the two green circles in the middle list being placed in the same group. Of course, this problem only occurs when a list contains two identical items. This is rare with this TMBL representation but must still be guarded against.

The second danger is forming a cross between two groups of equivalents so that each group contains some items before the other group and some items after it. Figure 57(d) shows an example in which either of the proposed dashed links would create a group that has items before another group and items after it. This would make the alignment invalid and so must be avoided. Figure 57(e) shows a more complicated

cross example in which the proposed dashed link would create a group that has items before the triangles and the stars and an item after them. This example illustrates that some crosses may only be identified by tracing through relationships and through lists that do not include any of the items to be joined.

The scan for these extra connections proceeds by searching through the individuals' instructions. For each, a list of candidates instructions for connection is drawn up. To be a candidate for connection, a pair must be matching, not yet connected and must either both start or both precede their respective lists or either both immediately precede or both immediately follow two connected items.

Such items are checked for conflicts and if there would not be any, the connection is added. When any connection has been made between two items, the algorithm follows back up the pairs that immediately precede them in case they can be sequentially connected like two sides of a zip.

The algorithm checks for the two types of conflict described earlier: overlaps and crosses. The checks for crosses are the most involved so these are only initiated when all other tests have been passed. The cross-checking subroutine checks for crosses from above the first item to below the second item and is called with both orderings of the item so that it checks for crosses in either direction.

The cross-checker scans up the equivalences from the first item and searches for any routes that lead to something below the second item. It is important to ensure that any possible cross will be found but it is also important that no more time is spent checking for crosses than is necessary. To this end, the code works on the assumption that there are no pre-existing crosses (or other conflicts) in the alignment. This allows the code to terminate searches whenever it hits a "stop": an instruction beyond which a cross cannot exist if there are no pre-existing crosses. Instructions are stops if they are equivalent to an item preceding an equivalent of the second item or if they precede another stop. A simplified version of this is summarised in Figure 59.

```
Stage Two (build extra connections)
 * Consider unconnected pairs where both items:
    - immediately precede two connected items,
    - immediately follow two connected items,
    - start their respective lists or
    - end their respective lists
 * Reject the pair if its items mismatch
 * Reject if connecting it would connect items within a list
 * Reject if connecting it would form a cross
 * If the pair has not been rejected, connect it
 * Repeat until no more connections can be made
```

Figure 59: A summary of the second stage of the alignment algorithm.

This algorithm was tested quite extensively to ensure that all potential dangers such

as crosses were identified and that the resulting aligned code is functionally equivalent to the individual programs before alignment.

6.4.6 Experimental Assessment

The CUDA platform was used for the experiments. The technique is not dependent on the platform and might be applied in other GP compilation scenarios such as compiling code for execution on a multi-core CPU. The C++ alignment code was written as a template so it can align numbers, strings or TMBL instructions.

The experiments were concerned with the effects of the alignment on compilation time and evaluation speed. The results from aligned code were verified against results from non-aligned code to identical behaviour. Otherwise, the results are not of interest here.

The individuals in the experiments were generated as children of a single seed parent using a low mutation rate. Two points should be noted here. First, this creates groups of similar individuals which will favour the alignment technique. The aim was to provide a realistic environment. However in other systems, such as tree-based GP systems, the diversity may be much greater and this may make alignment technique worthless.

Second, the seed individual is evolved and so is likely to use most of its instructions (as has been found with TMBL) so the compiler will not optimise away many of the instructions. This is in contrast to situations in which many of the instructions do not affect the output, as might be expected if the seed individual were randomly initialised. Some of the experiments varied the number of instructions and this was achieved by removing varying numbers of instructions from the start of the seed individual. This may change the individual so that more of the instructions may be optimised away. Hence the recorded evaluation rates for low numbers of instructions may be an overestimate.

Operating system	Ubuntu Linux 10.10
Linux Kernel	2.6.35-28-generic-pae
GPU Device	nVidia GeForce GTX 260 [Core 216] (216 cores, core clock speed: 590MHz shader clock speed: 1296MHz)
CUDA toolkit	v3.2
Device driver	260.19.44
`nvcc`	v0.2.1221

Table 24: Details of the system

The system configuration is provided in Table 24 and the default parameters are provided in Table 25. The last three entries refer to how the evaluation speeds were assessed: this involved timing eight consecutive launches of kernels, each executing

Number of CPU threads	1
Individuals per kernel	4
Number of individuals	120
Number of instructions	200
Number of evaluation repeats	8
Number of testcases per evaluation	65536
Number of iterations per evaluation	50

Table 25: Default parameters for the runs

all individuals for 50 iterations over 65536 testcases. For a standard population of 120 individuals, each with 200 TMBL instructions, this means executing 0.6291456×10^{12} TMBL instructions.

The mutation rate was set such that 95% of individuals have at least one mutation. For individuals with 200 instructions, this translates to a rate of 1.487% by instruction.

Each of the results in the experiments is averaged over five runs. Each line in the graphs in Section 6.4.7 has a background bar that indicates the mean value plus and minus one estimated standard error. In many cases, these bars are so thin that they cannot be seen. This suggests that the relatively small number of repetitions has been adequate to give good estimates of the means.

6.4.7 Results of Experiments

Figure 60 and Table 26 show the time per individual to align and generate the CUDA C source. This indicates that the time spent on these tasks is very small and that the alignment actually reduces this time. This is presumably because it reduces the amount of code that must be output. This might also suggest that the code-outputting code would benefit from some optimisation. Encouragingly, the graph seems to suggest that the time required to load or to align and load increases linearly with the number of TMBL instructions.

Figure 61 and Table 27 show the time taken per individual to compile from CUDA C to a cubin binary file, ready to be loaded onto the GPU. These durations are much longer than those in Figure 60. The reduction in compilation time achieved by alignment is more pronounced as the number of TMBL instructions increases. At 300 instructions, the compilation time per individual is 0.347 seconds without alignment and 0.072 seconds with, a reduction of 79.238%.

The figure appears to suggest that the growth in compilation time without alignment is faster than linear. With alignment, the increase looks as though it may be linear, although this is not clear from the graph.

Figure 62 and Table 28 demonstrate the reduction in evaluation speed caused by alignment. The slowdown reduces as the number of TMBL instructions increases and the difference is relatively small from 60 instructions upwards. At 300 instructions

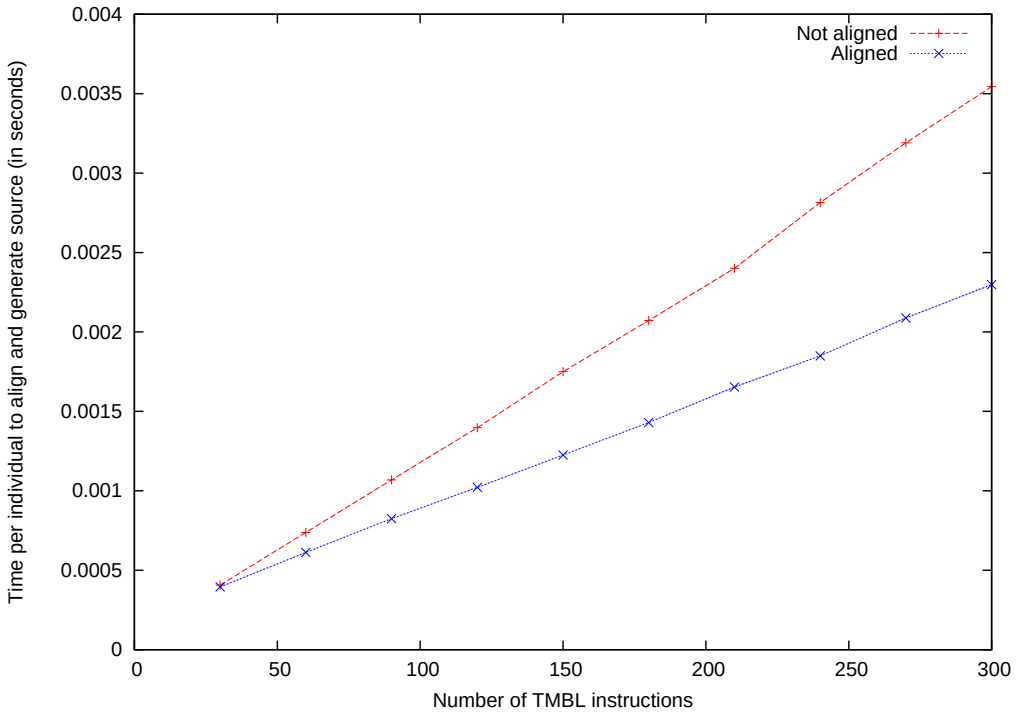


Figure 60: The time per individual to align and generate source over varying numbers of TMBL instructions with and without alignment

Instructions	Align kernel code		Change
	False	True	
30	0.0004066 s [±0.0000002]	0.0003951 s [±0.0000039]	−2.828%
60	0.0007382 s [±0.0000011]	0.0006122 s [±0.0000038]	−17.069%
90	0.0010685 s [±0.0000031]	0.0008247 s [±0.0000025]	−22.817%
120	0.0013976 s [±0.0000008]	0.0010216 s [±0.0000044]	−26.903%
150	0.0017499 s [±0.0000004]	0.0012253 s [±0.0000028]	−29.979%
180	0.0020720 s [±0.0000011]	0.0014301 s [±0.0000031]	−30.980%
210	0.0024013 s [±0.0000036]	0.0016529 s [±0.0000053]	−31.166%
240	0.0028148 s [±0.0000022]	0.0018497 s [±0.0000025]	−34.287%
270	0.0031906 s [±0.0000007]	0.0020878 s [±0.0000037]	−34.564%
300	0.0035439 s [±0.0000003]	0.0022977 s [±0.0000047]	−35.165%

Table 26: The time per individual to align and generate source over varying numbers of TMBL instructions with and without alignment

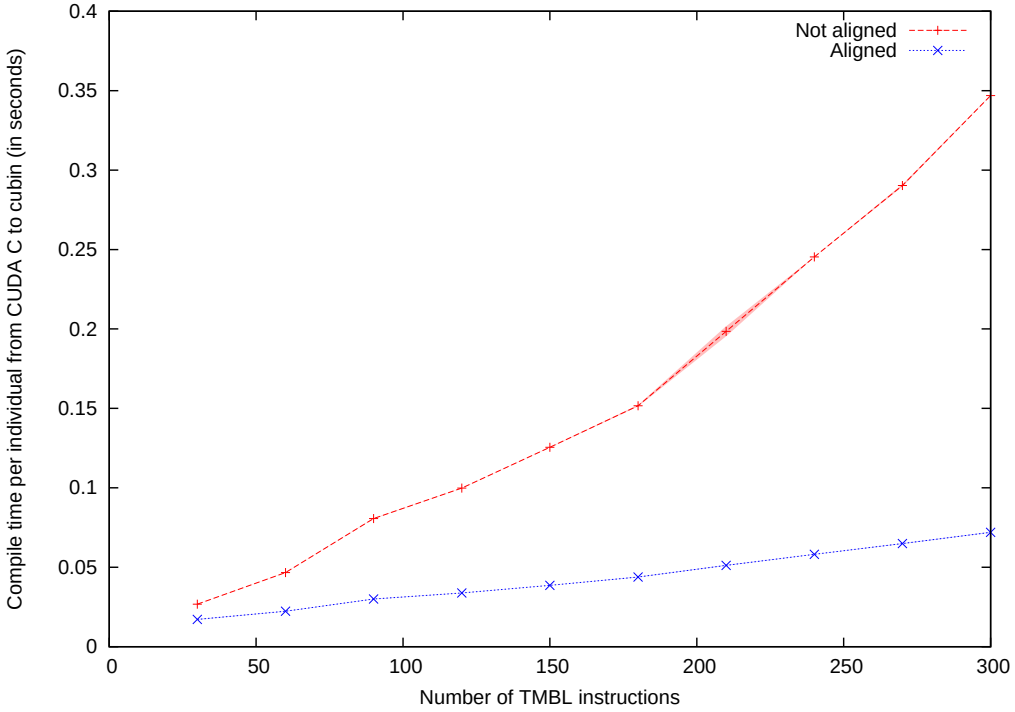


Figure 61: The time per individual to compile from CUDA C to cubin over varying numbers of TMBL instructions with and without alignment

Instructions	Align kernel code		Change
	False	True	
30	0.02675 s [±0.00004]	0.01717 s [±0.00018]	−35.813%
60	0.04670 s [±0.00008]	0.02229 s [±0.00012]	−52.270%
90	0.08067 s [±0.00019]	0.03002 s [±0.00014]	−62.787%
120	0.09988 s [±0.00008]	0.03386 s [±0.00009]	−66.099%
150	0.12550 s [±0.00014]	0.03865 s [±0.00008]	−69.203%
180	0.15167 s [±0.00043]	0.04385 s [±0.00015]	−71.089%
210	0.19851 s [±0.00320]	0.05120 s [±0.00021]	−74.208%
240	0.24535 s [±0.00013]	0.05815 s [±0.00013]	−76.299%
270	0.29033 s [±0.00071]	0.06489 s [±0.00026]	−77.650%
300	0.34694 s [±0.00014]	0.07203 s [±0.00021]	−79.238%

Table 27: The time per individual to compile from CUDA C to cubin over varying numbers of TMBL instructions with and without alignment

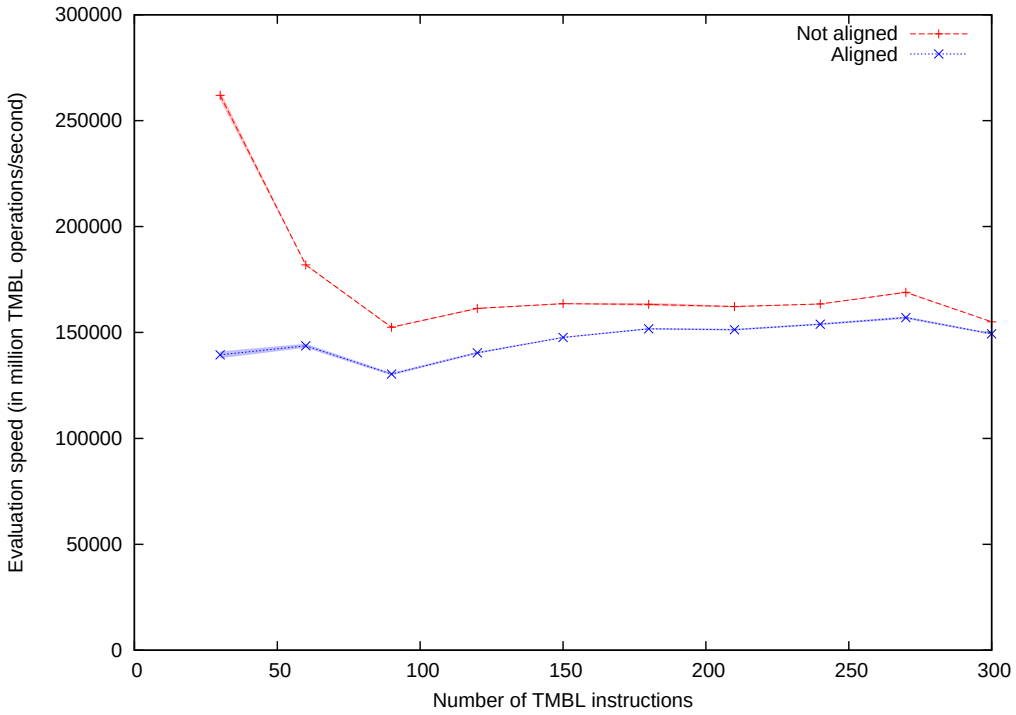


Figure 62: The evaluation speed over varying numbers of TMBL instructions with and without alignment

Instructions	Align kernel code		Change
	False	True	
30	261941.308 Mo/s [±2101.291]	139470.198 Mo/s [±1614.673]	−46.755%
60	181895.790 Mo/s [±297.537]	143734.897 Mo/s [±972.525]	−20.980%
90	152445.107 Mo/s [±253.607]	130362.974 Mo/s [±615.323]	−14.485%
120	161342.209 Mo/s [±273.564]	140385.536 Mo/s [±485.387]	−12.989%
150	163612.386 Mo/s [±232.703]	147657.551 Mo/s [±263.846]	−9.752%
180	163287.743 Mo/s [±714.749]	151757.682 Mo/s [±367.412]	−7.061%
210	162223.776 Mo/s [±327.786]	151324.779 Mo/s [±421.162]	−6.718%
240	163444.459 Mo/s [±103.636]	153908.795 Mo/s [±434.609]	−5.834%
270	168980.278 Mo/s [±269.374]	156988.850 Mo/s [±589.476]	−7.096%
300	155052.633 Mo/s [±24.321]	149383.227 Mo/s [±410.043]	−3.656%

Table 28: The evaluation speed over varying numbers of TMBL instructions with and without alignment

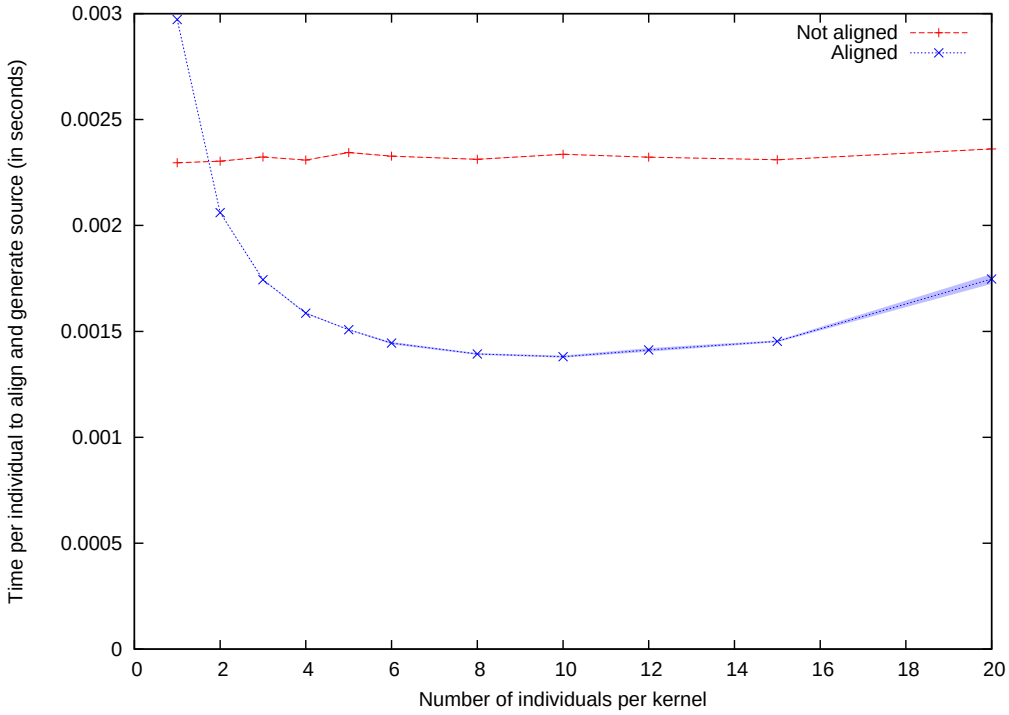


Figure 63: The time per individual to align and generate source over varying numbers of individuals per kernel with and without alignment

Programs per kernel	Align kernel code		Change
	False	True	
1	0.0022957 s [±0.0000007]	0.0029721 s [±0.0000027]	+29.464%
2	0.0023031 s [±0.0000020]	0.0020603 s [±0.0000014]	−10.542%
3	0.0023230 s [±0.0000010]	0.0017437 s [±0.0000024]	−24.938%
4	0.0023086 s [±0.0000007]	0.0015857 s [±0.0000035]	−31.313%
5	0.0023444 s [±0.0000034]	0.0015080 s [±0.0000018]	−35.677%
6	0.0023270 s [±0.0000010]	0.0014446 s [±0.0000059]	−37.920%
8	0.0023120 s [±0.0000020]	0.0013933 s [±0.0000039]	−39.736%
10	0.0023358 s [±0.0000014]	0.0013805 s [±0.0000049]	−40.898%
12	0.0023222 s [±0.0000003]	0.0014123 s [±0.0000083]	−39.183%
15	0.0023104 s [±0.0000010]	0.0014529 s [±0.0000044]	−37.115%
20	0.0023614 s [±0.0000018]	0.0017474 s [±0.0000230]	−26.002%

Table 29: The time per individual to align and generate source over varying numbers of individuals per kernel with and without alignment

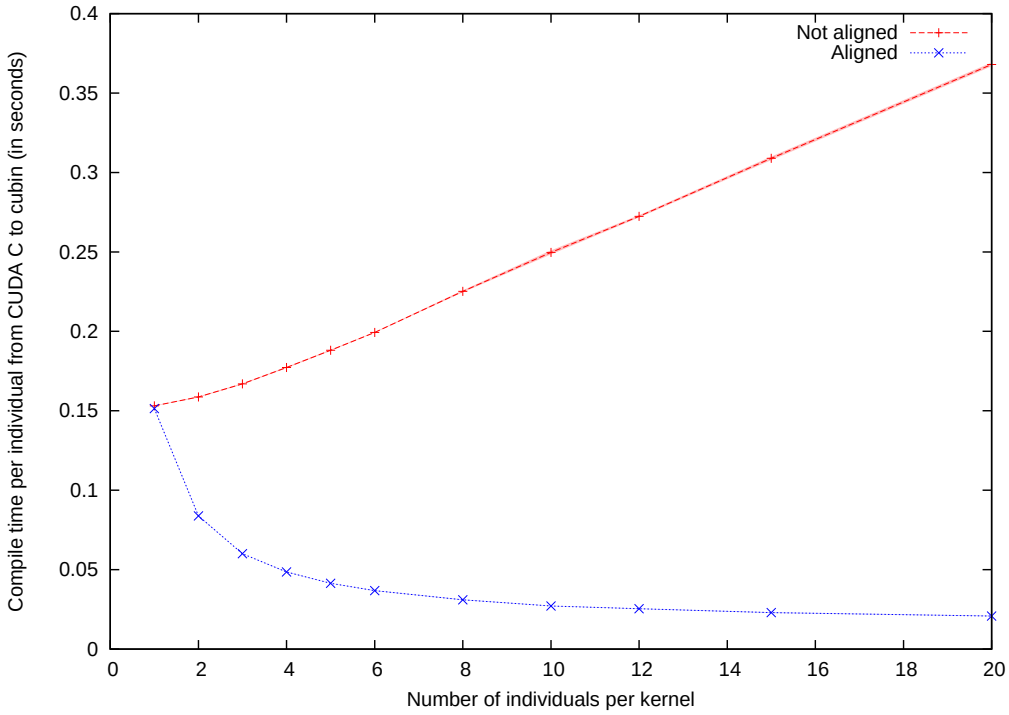


Figure 64: The time per individual to compile from CUDA C to cubin over varying numbers of individuals per kernel with and without alignment

Programs per kernel	Align kernel code		Change
	False	True	
1	0.15309 s [±0.00020]	0.15139 s [±0.00017]	−1.110%
2	0.15867 s [±0.00034]	0.08380 s [±0.00019]	−47.186%
3	0.16691 s [±0.00036]	0.06006 s [±0.00004]	−64.017%
4	0.17721 s [±0.00041]	0.04856 s [±0.00019]	−72.597%
5	0.18809 s [±0.00038]	0.04145 s [±0.00010]	−77.963%
6	0.19934 s [±0.00022]	0.03679 s [±0.00019]	−81.544%
8	0.22518 s [±0.00063]	0.03105 s [±0.00020]	−86.211%
10	0.24971 s [±0.00129]	0.02714 s [±0.00013]	−89.131%
12	0.27238 s [±0.00074]	0.02540 s [±0.00018]	−90.675%
15	0.30891 s [±0.00114]	0.02298 s [±0.00024]	−92.561%
20	0.36801 s [±0.00119]	0.02081 s [±0.00022]	−94.345%

Table 30: The time per individual to compile from CUDA C to cubin over varying numbers of individuals per kernel with and without alignment

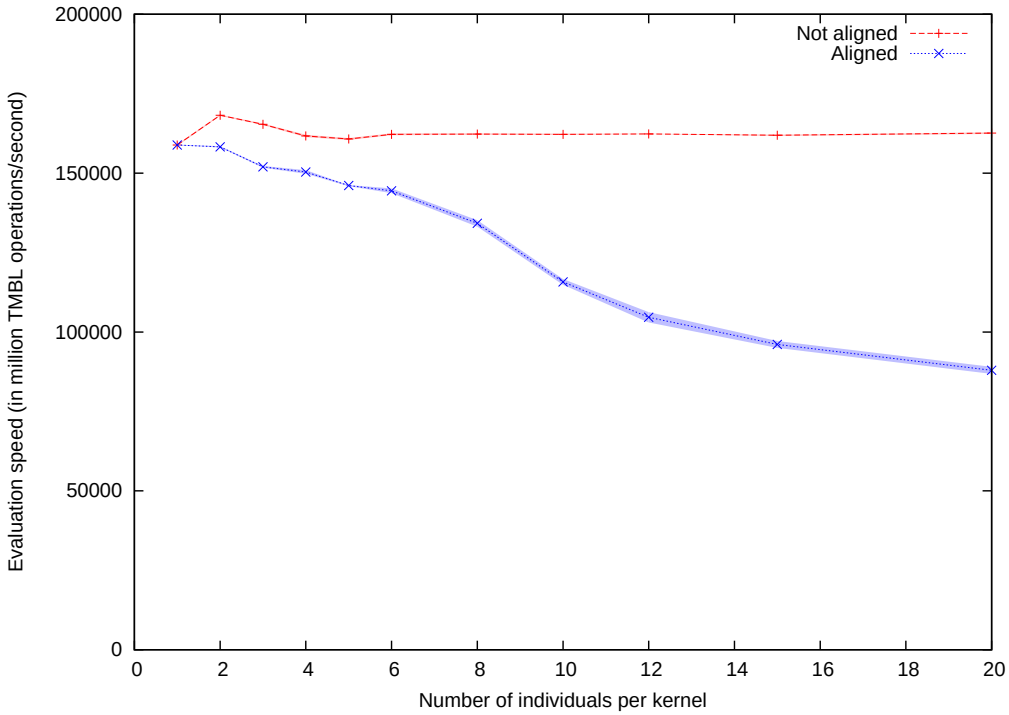


Figure 65: The evaluation speed over varying numbers of individuals per kernel with and without alignment

Programs per kernel	Align kernel code		Change
	False	True	
1	158850.182 Mo/s [±249.203]	158868.234 Mo/s [±90.088]	+0.011%
2	168211.238 Mo/s [±163.977]	158268.944 Mo/s [±207.540]	−5.911%
3	165373.722 Mo/s [±249.830]	151986.458 Mo/s [±263.702]	−8.095%
4	161710.994 Mo/s [±368.757]	150369.875 Mo/s [±557.921]	−7.013%
5	160741.163 Mo/s [±297.259]	146090.138 Mo/s [±159.134]	−9.115%
6	162218.132 Mo/s [±287.876]	144433.089 Mo/s [±749.007]	−10.964%
8	162280.312 Mo/s [±236.319]	134187.200 Mo/s [±1031.027]	−17.311%
10	162218.674 Mo/s [±208.434]	115744.987 Mo/s [±902.907]	−28.649%
12	162338.854 Mo/s [±67.455]	104626.361 Mo/s [±1618.974]	−35.551%
15	161923.293 Mo/s [±200.493]	96105.564 Mo/s [±1079.823]	−40.647%
20	162587.334 Mo/s [±252.569]	87934.720 Mo/s [±1100.069]	−45.915%

Table 31: The evaluation speed over varying numbers of individuals per kernel with and without alignment

the evaluation speed is 155052.633 million operations per second without alignment and 149383.227 million operations per second with, a decrease of only 3.656%. This may reflect the number of mutations being relatively high in the individuals with few instructions, which would mean frequent differences between individuals and hence a greater proportion of code to deal with these differences.

As with Figure 51, the higher evaluation speeds for low numbers of instructions may be an artifact of the way individuals are constructed and may be unrepresentative. This hypothesis is consistent with the fact that this does not happen for the aligned results because it is plausible that the alignment prevents code being optimised away.

Figures 63, 64 and 65 show these same properties over varying numbers of individuals in each kernel. Figure 63 and Table 29 show that the alignment time remains small across these values.

The only important point from this graph is that the alignment and generation times are all short, relative to the times involved in other tasks. Nevertheless, it is notable that the values do vary for the aligned individuals whereas, perhaps unsurprisingly, these times appear fairly constant for the unaligned individuals. Ideally the alignment and generation of a single individual should not take any more time than generation alone as is the case in this graph but since these times are all so short, this is not important.

Figure 64 and Table 30 show that the reduction in compilation time from alignment gets much larger as the number of individuals per kernel increases.

The compilation time per individual increases with the number of individuals per kernel when alignment is not used, meaning that the total compilation time increases worse than linearly. The aligned compilation time falls as the number of individuals increases. The ideal would be that the total compilation time stays completely constant so that the compilation time per individual is proportional to the inverse of the number of individuals, although this would never be possible in practice as more individuals will require more code to handle their increased number of differences.

Figure 65 and Table 31 show that increasing the number of individuals per kernel also increases the reduction in evaluation speed.

Whereas the evaluation speeds attained without alignment stay fairly consistent, the speeds achieved using alignment fall as the number of individuals to be required and hence the complexity of the resulting code increases.

Note that the lines meet at one individual per kernel in Figures 64 and 65 because at this point, there is no alignment work to be done so the input to the compiler remains the same.

6.4.8 Comments on Reducing Repeated Code through Alignment

The program conditions in the aligned source code only used `if` statements and only tested the individual's index with equality tests. Future work could tackle the evalua-

tion speed reduction by adding `else-if` and `else` statements and `greater-than` and `less-than` tests.

The technique was applied to a TMBL representation but it could equally be applied to other forms of GP. There are two issues that need to be considered in judging the applicability: the representation of the individuals and the nature of the population.

The representation is important in that it must allow similarities to be exploited to reduce duplication in the compiler's workload. In practice this should be possible for most forms, for example GP trees can be flattened into linear lists of instructions and these can then be aligned. This issue might not be the problem it initially appears.

More important is the nature of the population. The work here exploits the fact that in most TMBL generations, most individuals are mostly similar to each other. This suggests that the technique may be better suited to forms in which populations tend to contain many highly similar individuals. Where this is not true the technique is likely to be of little use. On the other hand, it doesn't appear to introduce a significant penalty so an investigation may be worthwhile.

6.5 Combining Both Techniques on 1000-Instruction Individuals

One of the most important aims of TMBL is to stimulate long term evolution by allowing mutations to make contributions in fresh areas of an individual without ruining what has already been achieved. To permit this, long evolutionary runs require fairly large individuals. Compiling populations of 1000-instruction individuals was previously found to be prohibitively slow.

An investigation was conducted to determine whether this problem could be solved by combining the two techniques described in Sections 6.3 and 6.4. Attempting this required additional code to be written to allow the alignments to be built into PTX kernels. In particular, choosing between sections of code in CUDA C requires if-statements, whereas PTX is inherently a branching language. To achieve this, the code uses two classes: `AlignmentIfElseHelper` and `AlignmentBranchHelper`. Each takes an alignment and decides how to arrange it into specific tests, blocks or branches in its respective type of language. This information is then used with the specifics of the actual language when generating the code.

The combination of techniques was assessed on 1000-instruction individuals using the parameters described Section 6.4.6. Each test was repeated 10 times. The 1000-instruction individuals were created by gluing together four repeats of the 300-instruction seed individual and then removing the first 200 instructions. The reason for this was to attempt to produce an individual with little redundant code. This is the sort of individual that is expected to be involved in TMBL work.

Table 32 shows the evaluation speeds. CUDA C speeds were 148366.170 Mgpop/s for non-aligned and 149616.796 Mgpop/s for aligned. PTX speeds were considerably

Align kernel code	Source type		Change
	CUDA C	PTX	
False	148366.170 Mo/s [±37.533]	187814.670 Mo/s [±23.757]	+26.589%
True	149616.796 Mo/s [±82.185]	192711.933 Mo/s [±132.676]	+28.804%
Change	+0.843%	+2.607%	+29.889%

Table 32: The effect on evaluation speed (in million GP operations per second) of varying whether kernel code is aligned and the language in which it is written. The bottom right value represents the change from unaligned CUDA C to aligned PTX.

higher at 187814.670 Mgpop/s for non-aligned and 192711.933 Mgpop/s for aligned. This represents an overall improvement of 29.889% from the use of both techniques.

Align kernel code	Source type		Change
	CUDA C	PTX	
False	3.37047 s [±0.00354] 1.92723 s [±0.00568]	N/A 0.44870 s [±0.00181]	−86.687%
True	0.28923 s [±0.00051] 0.15146 s [±0.00063]	N/A 0.05849 s [±0.00019]	−79.777%
Change	−91.419%	−86.965%	−98.265%

Table 33: The compilation time per thousand-instruction individual for aligned and unaligned code and for CUDA C and PTX source (where the first value in each cell is from CUDA C to cubin and the second value is from PTX to cubin). The bottom right value represents the change from unaligned CUDA C to aligned PTX.

Table 33 shows the compilation times that were observed. The complete compilation time for unaligned CUDA C was 3.370 seconds. For aligned CUDA C, it was 11.653 times faster at 0.289 seconds and for non-aligned PTX, it was 7.512 times faster at 0.449 seconds. For aligned PTX, the complete compilation time was a remarkable 57.625 times faster than for unaligned CUDA C at 0.0585 seconds.

6.6 Summary and Contribution

This chapter described two strategies to reduce the compilation times associated with data-parallel evaluation and an investigation of using these to tackle 1000-instruction TMBL individuals.

Section 6.3 described an investigation into using the low-level language PTX for data-parallel GPU evaluation, rather than the more standard CUDA C. Investigation revealed that PTX is a forward compatible, well-documented language, usable enough for small code bases such as data-parallel kernels. An implementation demonstrated two advantages to the approach, illustrated here with values derived from individuals of 300 instructions:

- considerably shorter compilation times (5.861×);
- higher resulting evaluation speeds (+23.029%).

This satisfies the aim of making data-parallel evaluation speeds accessible for moderately sized data-sets. Yet there is a price to be paid for these benefits:

- PTX is more complicated to develop and less readable than CUDA C.
- The PTX documentation is probably not as extensive as the CUDA C documentation and there is probably less PTX expertise available through CUDA forums.
- Compilation from PTX does not appear to optimise away dead code. This should not be a problem for TMBL because it is thought to use most of its code. For other forms of EC that do not, the problem can be avoided by performing intron removal before evaluation.

This work involved a number of contributions that are believed to be novel:

- The first use of PTX for data-parallel evaluation of EC individuals.
- A guide to the issues involved in attempting this.
- The design and writing of code to write out TMBL individuals in PTX code.
- An investigation into whether performing consecutive pairs of compilation and loading steps uses a similar amount of total time. This was found to be the case.
- An investigation into whether the time required to load binaries is very small compared to the time required to compile them. This was also found to be the case.
- An analysis into the effects of using PTX rather than CUDA C. The experiments recorded the various compile and load times and the evaluation speeds over varying numbers of instructions per individual and individuals per kernel, total population size and number of CPU threads.

In Section 6.4, code was written to identify and unite similarities in TMBL kernels. The aim was to reduce the compilation time whilst keeping the evaluation speed comparably high. Implementing this involved finding a way to align individuals against each other. The standard NW algorithm was examined but was found to be a slower, more thorough algorithm than was required so a new algorithm was proposed that is rough but fast. This was extended with another rough but fast algorithm for forming multiple alignments.

This work involved a number of contributions that are believed to be novel:

- The idea of speeding up the dynamic compilation of EC individuals though identifying similarities and drawing them out to reduce duplication.
- The design, coding and testing of a novel, fast algorithm to perform a multiple alignment between similar individuals in order to identify their similarities.

- The incorporation of this into an algorithm for exporting TMBL individuals in aligned CUDA C to produce code that compiles faster and evaluates comparably fast.

- An analysis of the effects of this technique. The experiments recorded the time required to align and generate the aligned CUDA C code, the compilation time, and the evaluation speed over varying numbers of instructions per individual and individuals per kernel (which defined the number of individuals to be aligned in a group).

This work was successful. Experiments showed that, at 300 instructions, the method reduced compilation time 4.817 times whilst only reducing evaluation speed by 3.656%. This satisfies the aim of making data-parallel evaluation speeds accessible for moderately sized data-sets. The amount of time spent aligning the individuals and generating the source code was relatively small and was even quicker than when no alignment was being used. Increasing the number of instructions made the technique reduce the compilation time more and reduce the evaluation speed less. Increasing the number of programs per kernel made the technique reduce the compilation time more but also reduce the evaluation speed more.

These results suggests the following guidelines: use as many instructions as can be benefited from and then tune the number of individuals per kernel to load both the GPU and CPU fully.

Section 6.5 described an attempt to combine the two techniques (alignment and PTX) to evaluate 1000-instruction TMBL individuals. Combined, the techniques reduced compilation times by an impressive 57.625 times. This work involved a number of contributions that are believed to be novel:

- The design and writing of code to write out aligned PTX code for TMBL individuals.

- An investigation into the effects combining the alignment and PTX technique.

7 Further CPU Optimisations

7.1 Introduction

This chapter tackles the final objective outlined in Section 1.4: identify the worst and most avoidable bottlenecks within the Central Processing Unit (CPU) code and tackle them. This is important because inefficient CPU code needlessly slows Genetic Programming (GP) runs and excellent Graphics Processing Unit (GPU) evaluation speeds can be ruined by CPU bottlenecks keeping the total run-time up. Thus effort was invested in optimising CPU code in other stages. This was guided by the evidence obtained from profiling the code. Two components of this optimisation work were interesting enough to merit further investigation and discussion.

The first part of the work was motivated by evidence from profiling, which indicated that considerable time was being spent generating random numbers for the tournament selection. It would be possible to make some improvements by switching the code to use a faster random number generator of lower quality. However, this sacrifice in random number quality would be a big price to pay; a study into the effect of random number quality on Genetic Algorithm (GA) performance recommended that "[...] in accordance with common practice in other fields, it is preferable to use the best PRNG [pseudo random number generator] available to avoid muddling the interpretation of the results" [13]. Hence it is better to keep higher quality random numbers and to improve the tournament selection code to require fewer of them instead. This work considers the mechanism of without-replacement tournament selection, and observes that each selection requires many random numbers but wastes many of them. If the probability distribution is understood, each selection could be made with just one random number. A mathematical analysis is performed on the probability distribution. As discussed in Section 2.3, this has previously been done for with-replacement tournament selection but not for without-replacement. The resulting formulae are used to investigate selection pressure in both cases and to construct an optimised without-replacement tournament selection.

An order analysis shows that the optimised algorithm uses $O(N)$ random numbers rather than $O(Nm)$, a significant improvement. Further analysis shows the optimised algorithm is an $O(N \log(N))$ algorithm (needed to sort the population) compared to the original which is an $O(Nm)$ algorithm. Hence the new algorithm's ability to perform better will depend on whether m is increased with respect to N at a rate asymptotically worse or better than $O(\log(N))$.

This is experimentally shown to perform without-replacement faster than the standard algorithm for most configurations in populations up to 1000. However this algorithm is only likely to be of much use for very fast implementations of Evolutionary Computation (EC) (such as in this work) since the slowest population tournament selection in the experiments is 0.056042 seconds (around $1/18^{\text{th}}$ of a second).

The discrete probability distributions for with-replacement tournament selection and without-replacement tournament selection are both extended to continuous functions. Mathematical analysis is then used to show that these continuous extensions are well behaved. These mathematical tools inspire the proposal of a new *many-from-few* measure of selection pressure strength. This many-from-few measure is used to investigate how selection pressure relates to tournament size and population size. Finally, graphs of selection pressure contours are produced, which show how tournament size must vary with respect to population size to keep selection pressure strength constant.

The second part of the work was motivated by evidence from profiling, which indicated that considerable time was spent copying individuals whilst constructing new generations. A naive strategy was being used, that took a complete temporary copy of the population and then used this as a source for generating the new generation. A first improvement replaces this with a system that updates the original population in-place. This is found to be effective but unsatisfactory since it is unable to handle crossover. Introducing crossover into the problem makes it considerably more complex. A way to represent these problems is proposed, which is effective at depicting an abundance of information. This representation is used to guide the design of a heuristic to update the population in-place with as few copies as possible. The proposed heuristic is assessed on data sets generated using a range of selection pressures. It is compared against the number of copies required for the naive strategy and against a lower bound on the minimum possible number of copies. It is found to be highly effective for reducing the number of copies toward the lower bound, particularly for population sizes more than about 100.

7.2 Optimising the Tournament Selection

For EC to succeed, it must exert selection pressure, i.e. fitter individuals must propagate to the next generation with higher probability. The method of selecting individuals for propagation is called the selection scheme. Selection schemes that select one individual may be invoked repeatedly to populate a new generation. When crossover is used, a selection scheme may be used to select two parents. The informal term "selection pressure" describes the extent to which the selection scheme makes the fittest individuals dominate the next generation. In Section 7.2.7, a new *many-from-few* measure is proposed to capture this notion.

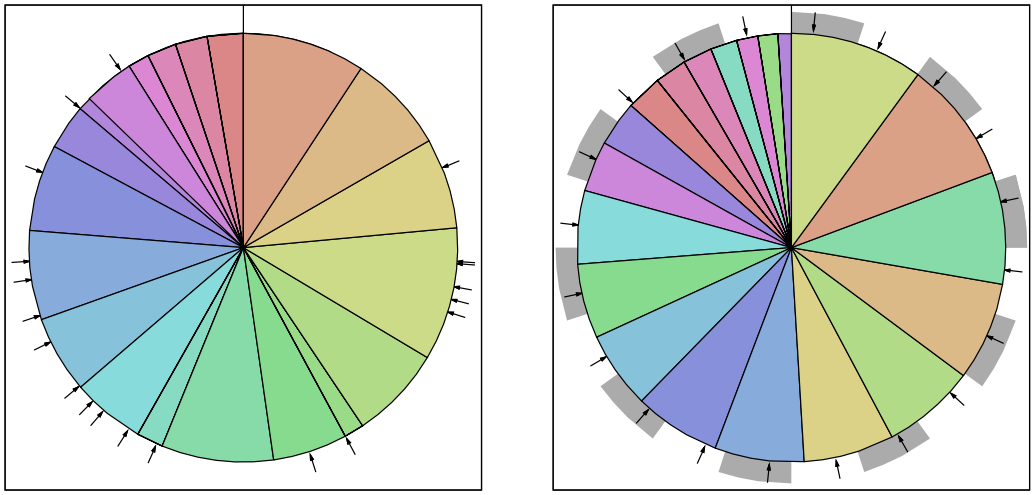
This section describes an analysis of the tournament selection scheme, with the aim of investigating whether tournament selection could be performed more efficiently. As will be seen, the analysis also helps answer the two questions researchers might most want to ask of tournament selection:

- What selection pressure is applied by a given tournament size, and what measure can give an intuitive feel for this?

- How can tournament size be varied to keep selection pressure constant between different runs with varying population sizes?

7.2.1 Common Selection Schemes

Roulette Wheel Selection (RWS) is a common selection scheme, which randomly selects individuals with probabilities in proportion to their fitnesses. Hence an individual's selection probability is its fitness divided by the total population fitness. This is known as RWS because the process of selecting the individual is akin to marking sectors of a roulette wheel in proportion to the individuals' fitnesses and then rolling a ball in the roulette wheel to make a selection.



(a) Roulette Wheel Selection (RWS)

(b) Stochastic Universal Sampling (SUS)

Figure 66: Selection schemes applied to 20 individuals. The coloured sectors represent individuals in the previous generation and their subtended angles are in proportion to their fitnesses. Each colour has no particular significance except to identify the same sector between the two subfigures. Each arrow points to an individual the scheme has selected for propagation. In both examples, some individuals are selected more than once. In Subfigure 66(a), RWS randomly chooses a point on the "roulette wheel" for each selection. In Subfigure 66(b), SUS sorts the individuals by descending fitness, randomly chooses a point within an initial sector of average fitness and then copies it round the wheel at intervals of average fitness. The grey and white patches indicate sectors of average fitness.

Figure 66(a) depicts this scheme using a population of 20 individuals that will be reused to illustrate later schemes. Running clockwise from the 12 o'clock point, the fitnesses of the individuals are: 92, 74, 68, 100, 70, 15, 56, 84, 20, 55, 58, 67, 65, 34, 10, 38, 16, 22, 24 and 27.

Baker proposed an alternative called Stochastic Universal Sampling (SUS) [5]. The roulette wheel analogy can also help to illustrate SUS. Again, the wheel's sectors denote each individual and are sized in proportion to their fitnesses, but this time the individ-

uals are sorted in descending order of fitness, say clockwise from some start point on the wheel. The average of the sectors' angles, A°, is calculated. Heading clockwise from the start point, some mark is made at a randomly chosen point between 0°and A°. Continuing clockwise from this mark, successive marks are made at regular intervals of A°. Each mark denotes an individual in the new population and the sector in which a mark falls denotes the parent of that individual. Figure 66(b) depicts this scheme using the same population of 20 individuals as before. Note that the selections are less random and less bunched than those in Figure 66(a).

Although SUS bears similarities to RWS, it differs in a few important ways. The SUS scheme selects an entire population in one pass using one random number whereas the RWS scheme only selects one individual for one pass using one random number, thus RWS must be repeated with many different random numbers to select an entire population. Also, SUS achieves a certain spread so it ensures individuals are picked from the full range of descending fitness sorted individuals whereas the RWS might only pick from the top end of that ranking. That said, if one individual's fitness dwarfs the fitnesses of all the others, both schemes are quite likely to fill the selection with copies of that one individual alone. Conversely, if all individuals have highly similar fitnesses, both selection schemes will tend to select each individual with highly similar probabilities (although RWS would do this stochastically whereas SUS would deterministically select precisely one instance of each individual).

Some authors use the term Fitness Proportionate Selection (FPS) synonymously with RWS [6]. Some others use FPS to denote a category that can be implemented using any scheme such as RWS (or presumably SUS) in which selection probabilities are assigned in proportion to fitness [59]. The distinction is unimportant here.

These approaches suffer the disadvantage of being sensitive to the measure of fitness used. For instance, adding a constant to the fitnesses evens out the selection probabilities of the individuals and so reduces the selection pressure. For this reason, these schemes' selection pressure usually weakens during an evolutionary run: at first, the fitness values are low so a moderately improved fitness confers a very significant reproductive advantage; later on, the fitness values are high so the same moderate fitness improvement has less effect.

Tournament selection offers an alternative. It involves selecting some subset of the population to compete in a tournament that is won by the competitor with the highest fitness. Figure 67 shows a tournament selection applied to the population of 20 individuals previously used in Figure 66.

The selection pressure may be varied by adjusting the tournament size: large tournaments will frequently contain one of the population's fittest members and so will exert strong selection pressure; small tournaments will frequently contain none of the population's fittest members and so will exert weak selection pressure. At one extreme, a tournament of the same size as the population will always select the fittest individual;

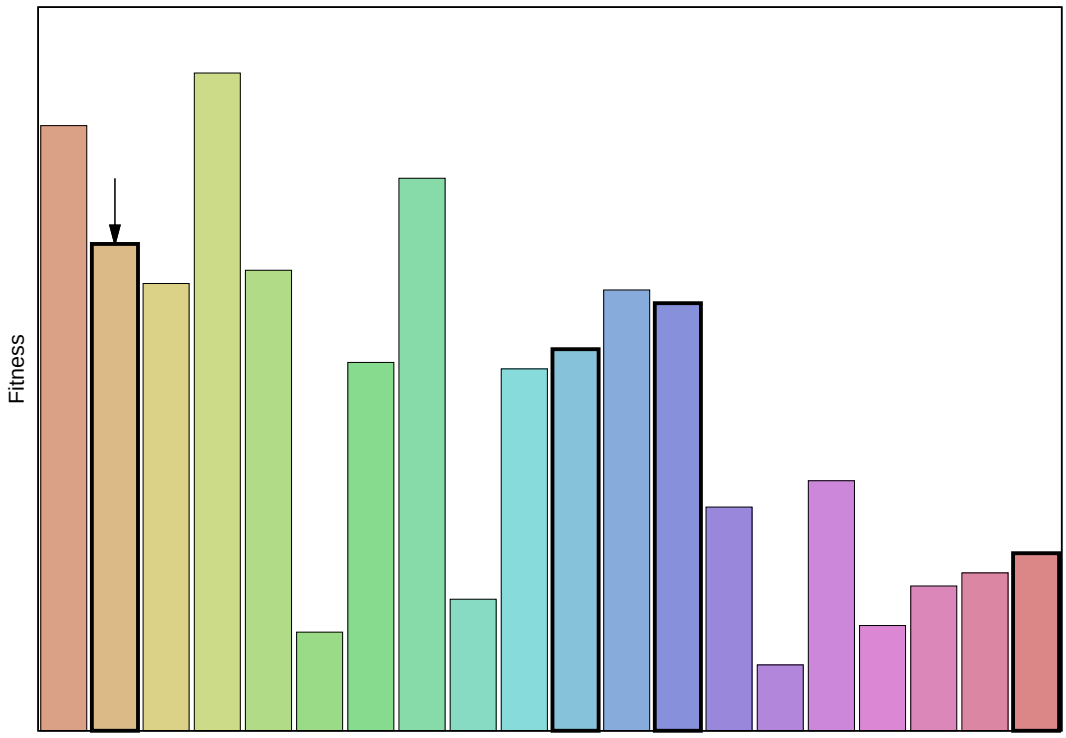


Figure 67: A single tournament selection applied to 20 individuals. Each bar represents an individual and its height is proportional to the fitness. Thick borders highlight the four individuals entered into the tournament and an arrow highlights the selected individual. It wins the tournament because it has the highest fitness of the four competitors.

at the other extreme, a tournament of size one will select randomly and uniformly.

The individuals that make up these tournaments are selected without replacement but with-replacement tournament selection is also possible. In that case, each addition to the tournament is chosen from the whole population, including those individuals that have already been added. For with-replacement tournament selection, the tournament can be arbitrarily large whereas for without-replacement, the tournament can only be as large as the population (since there can be no bigger subset).

Since tournament selection does not explicitly use fitness, it does not depend on the fitness measure. This protects it from the problem of selection pressure weakening throughout the run. Conversely, some might argue that this property exposes it to the problem of excessive sensitivity to small fitness differences.

Tournament selection has further advantages. Since it does not explicitly use fitness, it can be used for problems where no explicit fitness measure is available, provided there is some means of assessing which, if either, of two solutions is fitter. The ranking may include individuals in joint place. Such schemes are compatible with various optimisations which reduce the amount of evaluation performed. See Section 2.1.1 for more information.

For these reasons and because tournament selection is currently the most commonly used in GP [73], it was used throughout this work. Whilst performing the work in Chapter 5, profiling indicated that the tournament selection was consuming considerable CPU time. In particular, time was being spent on generating random numbers to pick the individuals for the tournaments. For a population of 1000 split into four demes of 250 with a tournament size fraction of 0.3, this entails picking 75 individuals for each tournament and hence 75000 individuals for every generation. When using a reasonable quality random number generator, this was using a non-trivial amount of time. This will be discussed further in Section 7.2.3.

Generating all these random numbers wastes resources because all that matters is the winner of the tournament. To make this clearer, consider a population of individuals sorted in descending order of fitness (such as in Figure 68). The trick of analysing selection probabilities by fitness-sorting the population has previously been deployed for with-replacement tournament selection [3] [106]. To perform a tournament selection, a random number generator repeatedly selects individuals to enter a tournament. Once this is complete, the selection is the member of the tournament that comes first in the sorted population and all other individuals are ignored. The probability of an individual winning the tournament is the probability that it will be the first of the sorted population in a randomly selected tournament.

This is shown in Figure 68. It isn't actually necessary to pick four random members of the tournament to make the selection; all that's required is the probability that each individual would win under this method.

These probabilities are independent of the fitnesses that produced the ranking so the probability distribution can be pre-calculated for any chosen set of parameters. Such distributions have been derived for with-replacement tournament selection, as discussed in Section 2.3, and this will be extended to a without-replacement tournament selection probability distribution in Section 7.2.2. This can then be used to perform a selection for any other sorted list using only one random number. The same principle may be applied to any ranking-based selection scheme. This approach also confers the benefit of code reusability since much of the work (the ordering, the random selection within a distribution and the randomising amongst joint-place individuals) is common to any ranking-based selection scheme. Adding a specific scheme is as simple as plugging in a new distribution.

This approach introduces a couple of issues worth mentioning. First, it requires all of the individuals to be ranked in descending order of fitness. As was mentioned earlier, ranking-based selection schemes offer the advantage of permitting problems with no specific measure of fitness, as long as they provide a method of determining which, if either, of any pair of individuals is fitter. However, if this method's ordering relation is not transitive (i.e. if it might produce results of the form A is fitter than B which is fitter than C which is fitter than A), then it may be necessary to use something

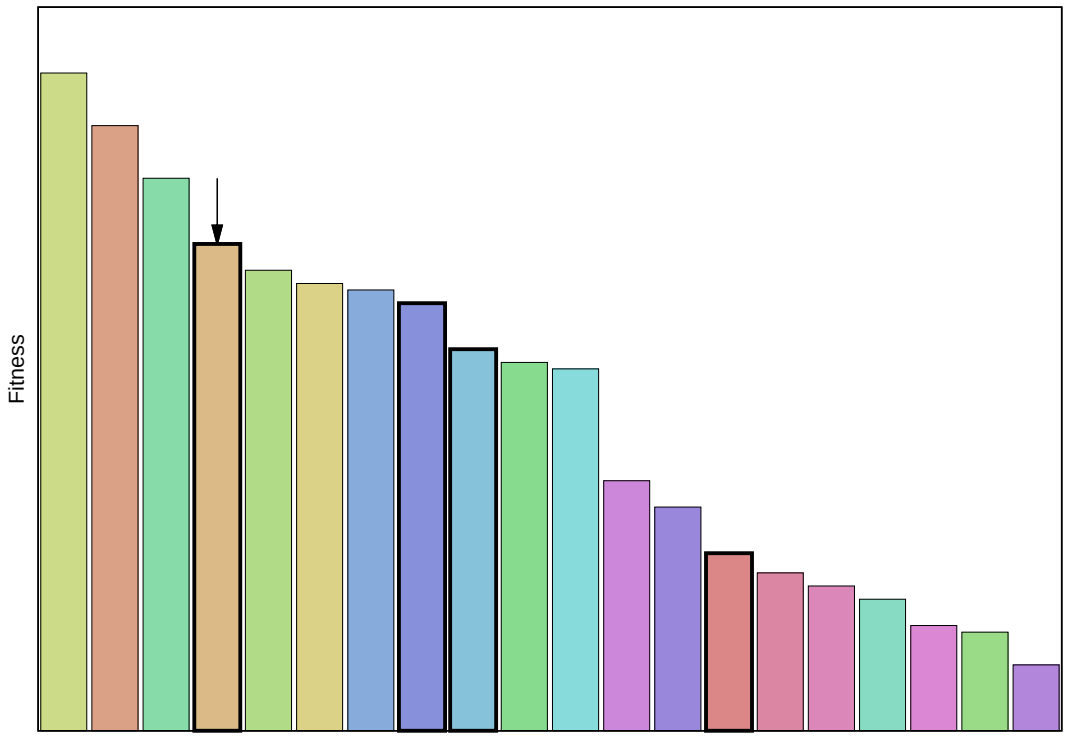


Figure 68: The same tournament selection as depicted in Figure 67. By pre-sorting the individuals, the winner is always the individual that comes first. The probability of an individual winning the tournament is the probability that it will be the first in a randomly selected tournament. These probabilities are independent of the fitnesses that produced the ordering so they can be pre-calculated for a given tournament size and population size.

like a round-robin (i.e. all-play-all) championship to generate the overall ranking of individuals. This requires an unappealing $O(n^2)$ comparisons and each may be computationally expensive.

Second, the story is slightly more complicated because, as noted earlier, a given ranking may involve multiple individuals in joint places, such as two individuals in joint fourth place as shown in Figure 69. If a tournament contains multiple joint winners, the original algorithm randomly chose a winner among them, to avoid potential drawbacks of any particular deterministic choice. The optimised algorithm can emulate this behaviour by performing an additional step after the first stage of selection. In this step, the algorithm collects any individuals holding joint place with the selected individual (including the individual itself) and then randomly chooses among them. This replicates the original algorithm's behaviour because it ensures that equally fit individuals are equally likely to be selected and that the group's total probability of selection is correct.

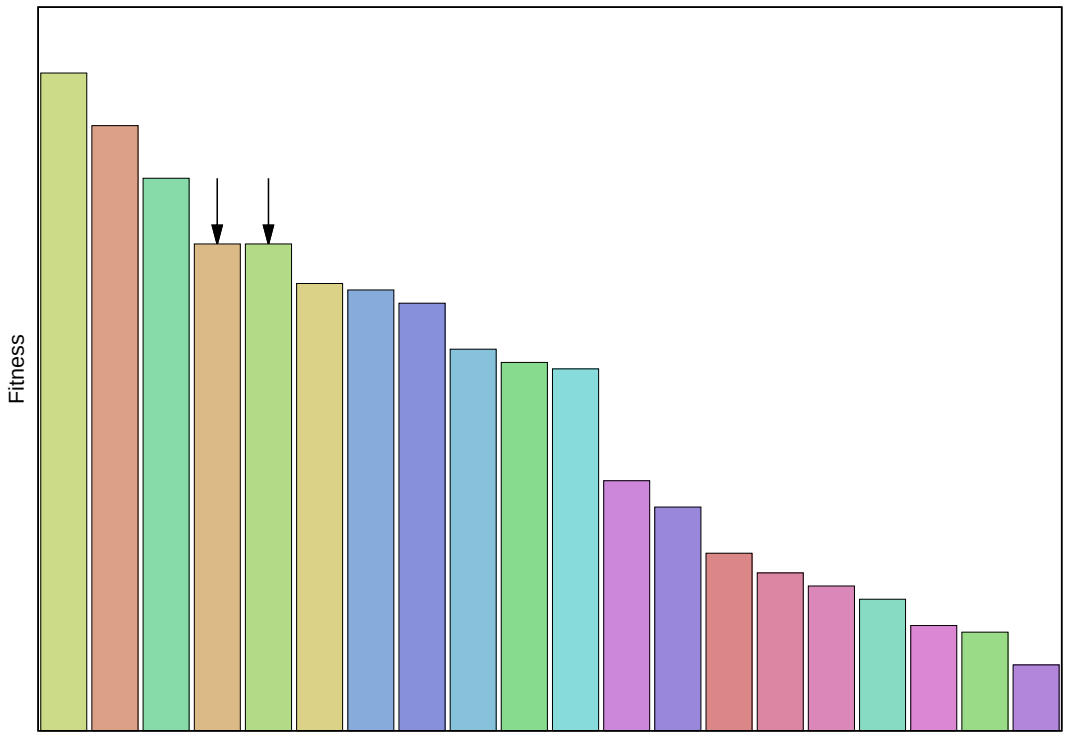


Figure 69: Figure 68 raised the possibility of pre-calculating selection probabilities, independent of the actual fitness values. This leaves the problem of individuals of equal fitness. In this figure, the fitness of the fifth fittest individual has been raised so that it equals that of the fourth fittest. These two individuals (highlighted with arrows) should have equal chance of selection, yet the pre-calculated probabilities are more likely to select the fourth-ranked individual than the fifth-ranked. Hence, if either is selected, there is an additional random choice between them to fix this problem.

7.2.2 Tournament Selection Mathematics

Implementing this technique requires the calculation of the associated probability distribution. For with-replacement tournament selection, this is fairly simple and has previously been achieved in the literature [3] [106]. The calculation for without-replacement selection is more involved because the probabilities vary with successive additions to a tournament.

Consider a population of N individuals that has been sorted by fitness such that the first individual is the fittest and the N^{th} is the least fit. Since joint places are processed in a later step as described above, for now each individual may be assumed to be strictly fitter than the next. A tournament selection with tournament size m (such that $1 \leq m \leq N$) will pick one individual. What is the probability $P(E_i)$ of the event E_i that the selected individual is the i^{th} individual in the population? What is the cumulative probability $P(C_i)$ of the event C_i that the selected individual is the i^{th} individual or any earlier (fitter) individual?

For with-replacement tournament selection, the probability that any given individ-

ual will be added to a tournament is the same for each addition. The probability that a given addition to the tournament is the i^{th} individual or earlier is $\frac{i}{N}$ so the probability that it is after the i^{th} is $1 - \frac{i}{N}$. The additions are independent so the probability that the whole tournament of size m is filled with individuals after the i^{th} is $\left(1 - \frac{i}{N}\right)^m$. This observation helps to calculate both $P(C_i)$ and $P(E_i)$.

The cumulative selection probability $P(C_i)$ is the probability that the whole tournament is not filled with individuals after the i^{th} so $P(C_i) = 1 - \left(1 - \frac{i}{N}\right)^m$. The selection probability $P(E_i)$ is the probability that the whole tournament is filled with individuals after the $(i-1)^{\text{th}}$, minus the probability that the whole tournament is filled with individuals after the i^{th} so $P(E_i) = \left(1 - \frac{i-1}{N}\right)^m - \left(1 - \frac{i}{N}\right)^m$. This may be rearranged to match other forms that have been used in the literature: $N^{-m}((N-i+1)^m - (N-i)^m)$ [3] and $\frac{(N-i+1)^m - (N-i)^m}{N^m}$ [106].

To begin the analysis for without-replacement tournament selection, observe that the tournament selection is equivalent to randomly selecting a subset $\mathcal{T}$ of size m from the set $\{1, 2, \ldots, N\}$ and the event E_i occurs if and only if i is the smallest member of $\mathcal{T}$. This observation can be combined with conditional probability as follows.

$$\begin{aligned}
P(E_i) &= P\left[(i \in \mathcal{T}) \cap (\nexists j \in \mathcal{T} | j < i)\right] \\
&= P\left[(i \in \mathcal{T}) \cap (i-1 \notin \mathcal{T}) \cap (i-2 \notin \mathcal{T}) \cap \ldots \cap (1 \notin \mathcal{T})\right] \\
&= P\left[i \in \mathcal{T}\right] . P\left[i-1 \notin \mathcal{T} | i \in \mathcal{T}\right] \ldots P\left[1 \notin \mathcal{T} | (i \in \mathcal{T}) \cap (i-1 \notin \mathcal{T}) \cap \ldots \cap (2 \notin \mathcal{T})\right] \\
&= \frac{m}{N} \left(\frac{N-m}{N-1}\right) \left(\frac{N-m-1}{N-2}\right) \cdots \left(\frac{N-m-i+2}{N-i+1}\right) \\
&= \frac{m}{N} \prod_{k=2}^{i} \frac{N-m-k+2}{N-k+1}
\end{aligned}$$

To understand the step from entries like $P\left[i \in T\right]$ to entries like $\frac{m}{N}$, it may help to visualise taking N balls numbered 1 to N and randomly choosing m balls to put into a jar labelled $\mathcal{T}$ and putting the rest in a jar labelled $!\mathcal{T}$. The probability that any one ball ends up in the $\mathcal{T}$ jar is $\frac{m}{N}$. Similarly, for entries like $P\left[i-1 \notin \mathcal{T} | i \in \mathcal{T}\right]$, we can imagine that the i^{th} ball has already been placed in jar $\mathcal{T}$ before we pick the other $m-1$ balls from $N-1$ to put in jar $\mathcal{T}$ and the $N-m$ balls from $N-1$ to put in jar $!\mathcal{T}$. The probability that any one ball now ends up in jar $!\mathcal{T}$ is $\frac{N-m}{N-1}$.

If $i \geq N - m + 2$, then the final result is 0, otherwise it is equal to:

$$\frac{m}{N} \frac{(N-m)!}{(N-m-i+1)!} \frac{(N-i)!}{(N-1)!}$$

A similar analysis may be used to evaluate the cumulative probability $P(C_i)$ of the event C_i that the selected individual is the i^{th} individual or any earlier individual. In this case it is easier to manipulate the probability of the queried event not happening.

$$
\begin{aligned}
P(C_i) &= P\left[(1 \in \mathcal{T}) \cup (2 \in \mathcal{T}) \cup \ldots \cup (i \in \mathcal{T})\right] \\
&= 1 - P\left[(1 \notin \mathcal{T}) \cap (2 \notin \mathcal{T}) \cap \ldots \cap (i \notin \mathcal{T})\right] \\
&= 1 - P\left[1 \notin \mathcal{T}\right].P\left[2 \notin \mathcal{T} | 1 \notin \mathcal{T}\right] \ldots P\left[i \notin \mathcal{T} | (1 \notin \mathcal{T}) \cap (2 \notin \mathcal{T}) \cap \ldots \cap (i-1 \notin \mathcal{T})\right] \\
&= 1 - \left(\frac{N-m}{N}\right)\left(\frac{N-m-1}{N-1}\right) \cdots \left(\frac{N-m-i+1}{N-i+1}\right) \\
&= 1 - \prod_{k=1}^{i} \frac{N-m-k+1}{N-k+1}
\end{aligned}
$$

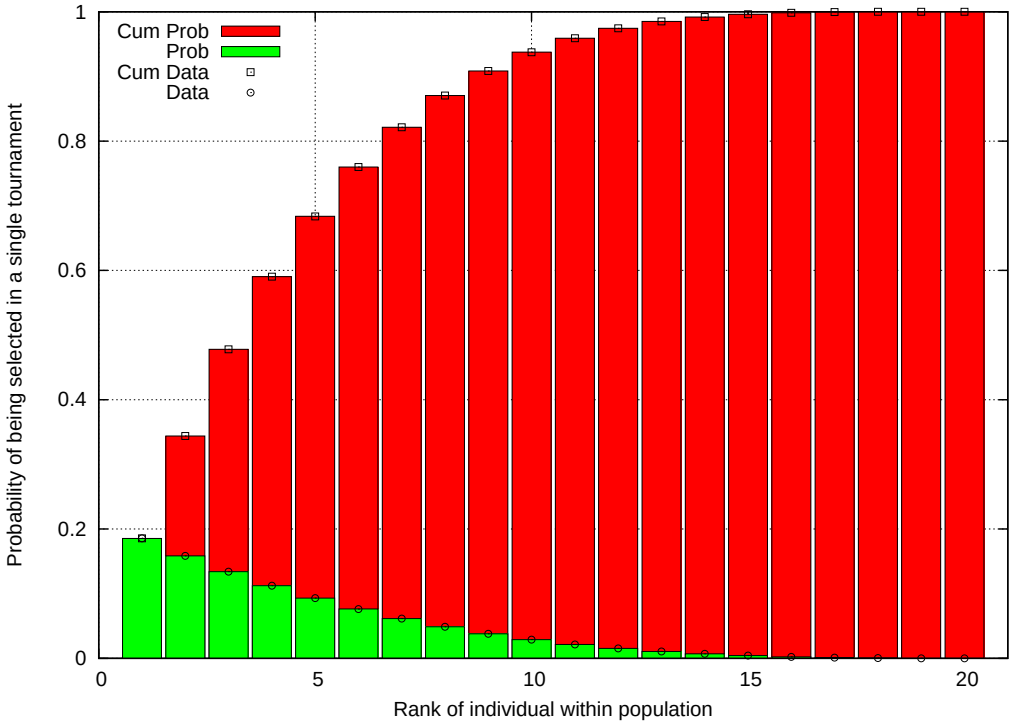
If $i \geq N - m + 1$, then the final result is 1, otherwise it is equal to:

$$1 - \frac{(N-m)!}{(N-m-i)!}\frac{(N-i)!}{(N)!}$$

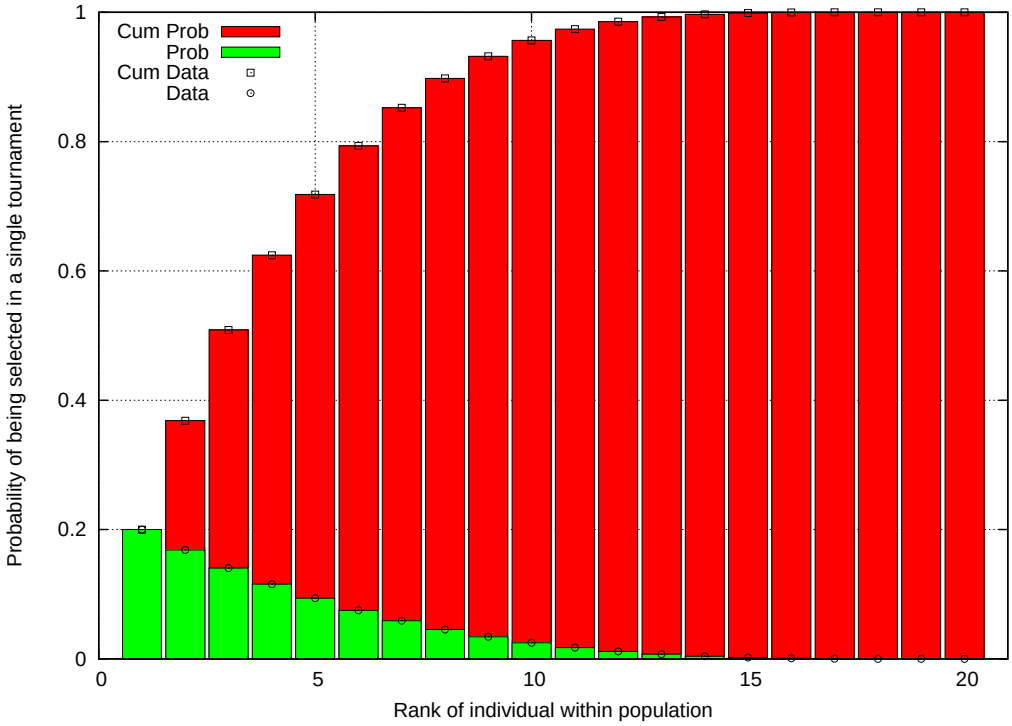
In practice, it is better to compute these values using the appropriate list of fractions rather than using the formulae with four factorials because the values for those factorials will be huge and so will be more prone to rounding errors.

Figure 70 shows what these formulae look like when plotted for a tournament size four and population size of 20. There is one graph for with-replacement and another for without-replacement. In each, the fittest individual is on the left and the least fit is on the right. Note that the expected values of "reproduction rate" (described in Section 2.3.1) are simply the selection probabilities on this graph multiplied by N. The results satisfy the following expected properties.

- The selection probabilities should be higher for fitter individuals.
- The cumulative probabilities should sum to one.
- Each probability should be the difference between the equivalent cumulative probability and the one before.
- For without-replacement, the fittest item's probability should be $\frac{m}{N}$ (in this case $\frac{4}{20}$), the chance of appearing in a tournament, because it wins any tournament in which it appears.
- For without-replacement, the $m - 1$ least fit individuals should have probability 0 because their tournaments always contain fitter members.
- For without-replacement, all but the $m - 1$ least fit individuals should have non-zero (but possibly very small) probabilities.
- For with-replacement, all individuals should have non-zero probabilities because a tournament could be populated with nothing but copies of any of the individuals.



(a) With-replacement tournament selection probabilities



(b) Without-replacement tournament selection probabilities

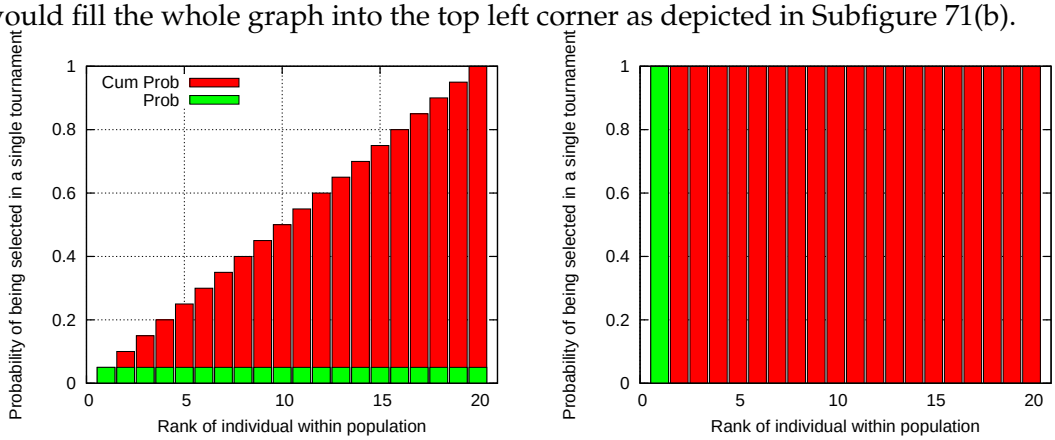
Figure 70: The probabilities of each individual in a population of 20 being selected by one tournament of size four. The individuals are ranked in descending order of fitness, the green bars indicate the probabilities and the red bars indicate the cumulative probabilities. The cumulative (red) bars stand behind the non-cumulative (green) bars, rather than being stacked atop them. The first cumulative probability bar is the same height as the bar it is behind. Fitness would be averaged out within any groups of equal fitness. For comparison, each graph has the real data from 100,000,000 tournament selections plotted on top.

- The probabilities should be well fitted by real data.

As expected, without-replacement assigns a higher selection probability to the fittest individuals than does with-replacement because it doesn't waste any of its random numbers on entering individuals into the same tournament multiple times and so is more likely to include one of the fitter individuals in each tournament. The difference is slight for this particular configuration although it can be much larger for other configurations as will be seen in Section 7.2.8.

Note that when performing with-replacement tournament selection with tournament size m from a population of size N, there are easily calculable probabilities for the number of unique individuals (from 1 to m) that the tournament will contain. For a fixed number of unique individuals, the selection probabilities will be the same as the without-replacement using that fixed-number as the tournament size. Hence the with-replacement distribution can be viewed as a weighted sum of without-replacement distributions based on the number of unique selections.

If there were no selection pressure, each green bar would be of equal height and the tops of the red bars would form a straight line between bottom-left and top-right as depicted in Subfigure 71(a). If there were maximal selection pressure, the first green bar would have height 1, all other green bars would have no height and the red bars would fill the whole graph into the top left corner as depicted in Subfigure 71(b).



(a) No selection pressure

(b) Maximal selection pressure

Figure 71: An illustration of how the graphs of Figure 70 would appear were there: no selection pressure (Subfigure 71(a)) or maximal selection pressure (Subfigure 71(b)).

Indeed, the shape of the cumulative probabilities is a fingerprint of the scheme's selection pressure, precisely describing how much the scheme favours each of the ranks. Perhaps the most interesting feature is the overall strength of the selection pressure. The closer the red bars get to the top left corner, the stronger the selection pressure; the closer to the straight line joining bottom-left to top-right, the weaker. If they were to go below that line, towards the bottom right corner, that would indicate negative selection pressure (favouring less fit individuals). None of the measures in Section 2.3.1 quite captures this notion of selection pressure strength: the degree to which the fittest

individuals are favoured. This will be revisited in Section 7.2.7, which proposes a new *many-from-few* measure of selection pressure strength.

7.2.3 Fast Tournament Selection

The motivation for this work is to reduce the time wasted in GP runs. With this in mind, can the new analysis be used to make tournament selection more efficient? A new algorithm is devised to attempt this. In each generation, the new algorithm begins by sorting the population in order of descending fitness. Then for each selection, the new algorithm selects one random number from a uniform distribution between 0 and 1 to represent the cumulative probability. It uses a pre-built database of cumulative probabilities to translate this number to the index of the selected individual as depicted in Figure 72. The database of cumulative probabilities for a given tournament configuration only need be built once per run.

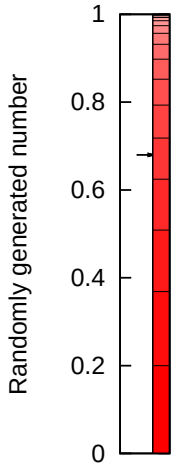


Figure 72: Cumulative probabilities may be used to map efficiently from a randomly generated number in the interval $[0, 1]$ to an individual in the population. Here, the random number 0.68 (illustrated by the arrow) picks the fifth fittest individual. The smaller cumulative probabilities are shown in front of the larger ones. This example's cumulative probabilities are those for without-replacement tournament selection using a tournament of four in a population of 20 (as in Figure 70(b)).

Does this reduce the computationally expensive random number requirements? The original algorithm required m random numbers for each of the N selections to build a new generation, hence it required $O(Nm)$ random numbers per generation. Note that the tournament size m is not discarded as a constant in this analysis because it may be increased with respect to N. The optimised algorithm requires only 1 random number per selection (or sometimes two when there are multiple individuals of equal fitness) so it requires only $O(N)$ random numbers per generation. This is a clear improvement.

Has the overall computational complexity similarly improved? The computational complexity of the original algorithm is $O(Nm)$, whereas for the new algorithm it is

$O(N\log(N))$. The new algorithm requires this for sorting the individuals and for making the selections. The code written for this research optimises the sort by performing it on the individuals' indices rather than the individuals themselves but this is still an $O(N\log(N))$ algorithm. The code uses a binary search to translate between the random cumulative probability and the corresponding individual index so this also takes $O(N\log(N))$ steps per generation. This could be improved to $O(N)$ time using a hash function but that would not circumvent the $O(N\log(N))$ time required for the sort.

So whether the new algorithm reduces the overall computational complexity with respect to N depends on how one increases m as N increases. If the increase in m is asymptotically slower than $\log(N)$, the new algorithm will have worse computational complexity; if the increase is asymptotically faster than $\log(N)$, it will be better. Even if m is not increased at all, the new algorithm will only get worse at the rate of $\log(N)$.

The random number generator used throughout this work was the `mt19937` generator from the Boost C++ Library (`www.boost.org`). The Boost documentation describes `mt19937` as having 44% approximate speed compared to their fastest generator, cycle length $2^{19937}-1$, approximate memory requirements of `625*sizeof(uint32_t)` and good uniform distribution in up to 623 dimensions. It was chosen based on the following Boost documentation advice: "If the names of the generators don't ring any bell and you have no idea which generator to use, it is reasonable to employ `mt19937` for a start: It is fast and has acceptable quality."

Figures 73, 74, 75 and 76 and Tables 34, 35, 36 and 37 show the time required for each of the algorithms to fill a population with tournament selections over various population and tournament sizes. Each value is averaged over 20 runs and each plotted line on the graphs has a bar behind it to represent the average plus and minus one standard error. Since the standard errors are so small, these can hardly be seen. The four figures are all shown on the same scale for ease of comparison. Since each plotted line represents a constant fraction of the population size, we would expect them to have underlying $O(N^2)$ behaviour for the standard algorithm and $O(n\log(N))$ for the new algorithm.

Figure 73 and Table 34 show the values for without-replacement tournament selection. The surprising pattern was that the tournament selection was slowest for the smallest tournaments. On investigation, the standard C++ function `random_sample()` emerged as the cause of this behaviour, perhaps because its implementation is optimised to minimise something other than random number generations.

It should take few random numbers to generate a small without-replacement random sample from a set. Similarly, generating a large random subset should take few random numbers because it can be constructed as the complement of a small randomly generated subset. Using this approach, a new random sampling subroutine was coded to replace `random_sample()`. Figure 74 and Table 35 show the results. Here the slowest tournament sizes are those that account for 50% of the population.

Figure 75 and Table 36 show the values for the new algorithm. These results show the proposed approach's increase in speed. For a population of 1000, the new algorithm is 8.714 times faster for 10% tournament selection and 12.442 times faster for 90% tournament selection. For tournament sizes in between, the speed improvement is even greater and at 50% tournament selection, it is 43.856 times faster.

Figure 76 and Table 37 show the values for with-replacement tournament selection using the standard algorithm. For a population of 1000, it is 7.461 times slower than the new without-replacement algorithm for 10% tournament selection and this increases to 72.972 times slower for 90% tournament selection. These results show that with-replacement tournament selection is faced with similar problems to those of without-replacement. Furthermore, as argued in Section 2.3.2, with-replacement is more wasteful with random number generations in striving for a certain selection pressure. It is not even susceptible to the subset complement trick deployed above.

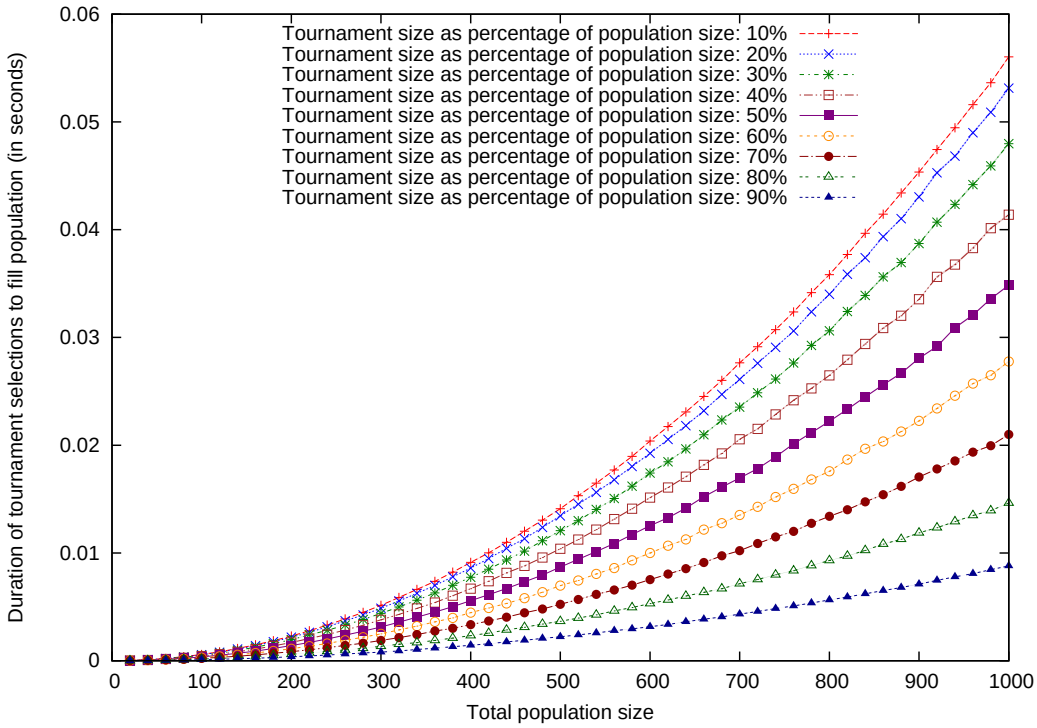


Figure 73: Time to fill a population from without-replacement tournament selections using the standard algorithm and `random_sample()`. The smaller tournaments take longer than the smaller ones due to the implementation of `random_sample()`.

It might be suggested that viewing the data this way unfairly benefits the new algorithm because many researchers might choose to increase the tournament size slower than linearly with respect to population size. For what absolute values of population and tournament size, is the new algorithm better? Figure 77 shows the answer for population sizes up to 1000. The colour of each point indicates how many of 20 comparisons showed the new algorithm to be faster (light green) or slower (dark red) than

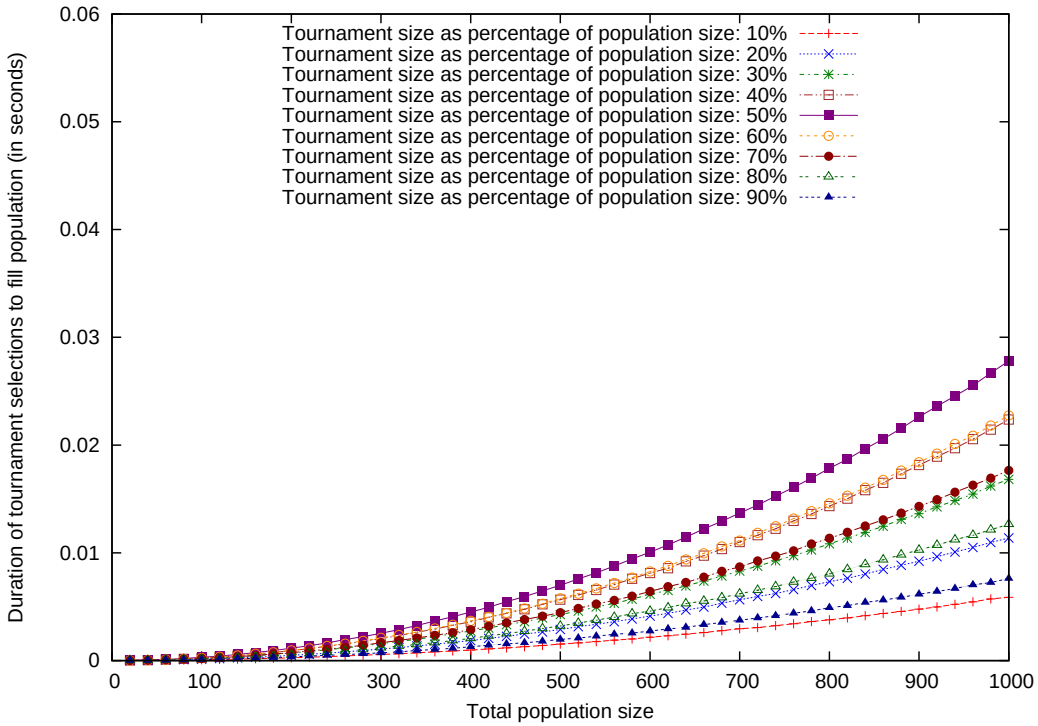


Figure 74: Time to fill a population from without-replacement tournament selections using the standard algorithm and a new sampling subroutine. Now the smallest tournaments are performed more quickly and the tournaments using 50% of the population are slowest.

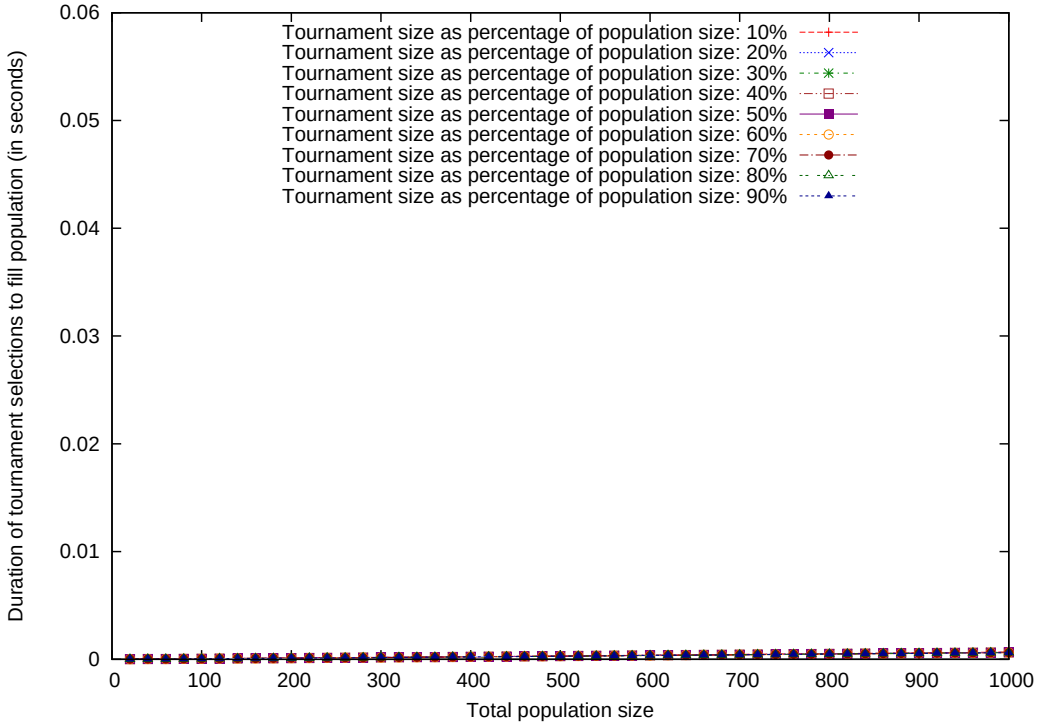


Figure 75: Time to fill a population from without-replacement tournament selections using the new algorithm. Most times are much shorter than those in Figures 73 and 74.

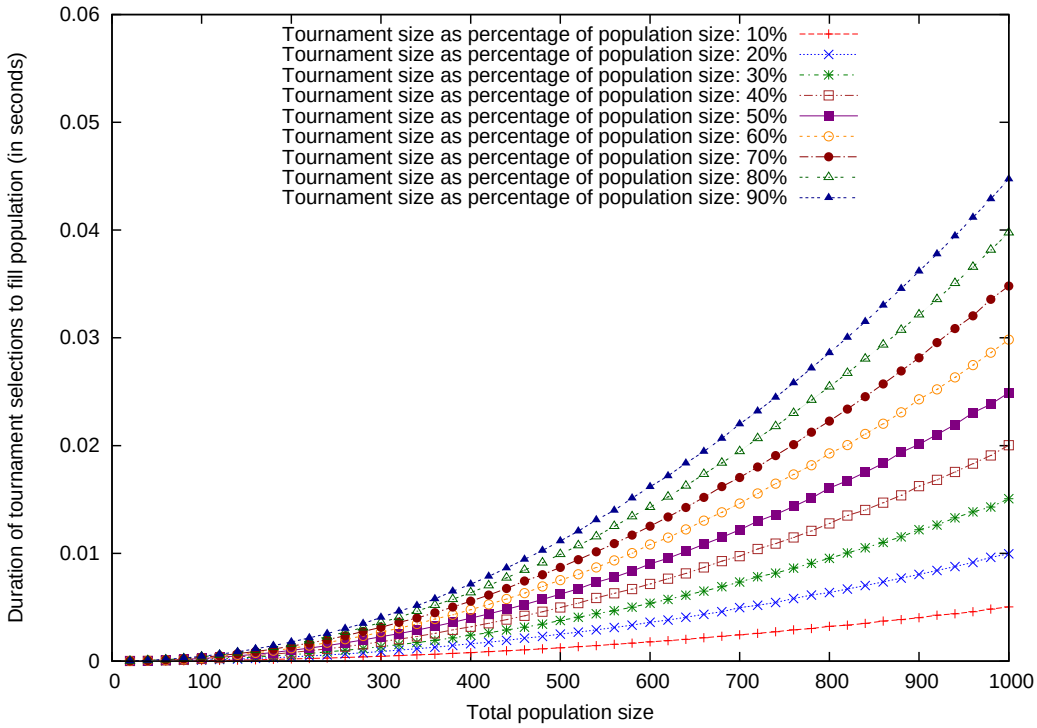


Figure 76: Time to fill a population from with-replacement tournament selections. Note that the times are similar to those in Figure 73 but here the larger tournaments account for the bigger times. Also note that for with-replacement tournament selection, the tournament size may need to be larger than the population size to achieve the desired selection pressure (see Section 7.2.8).

Population size	Tournament size as percentage of population size								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
20	0.000027 s	0.000025 s	0.000023 s	0.000020 s	0.000017 s	0.000014 s	0.000012 s	0.000010 s	0.000007 s
40	0.000096 s	0.000092 s	0.000083 s	0.000071 s	0.000060 s	0.000048 s	0.000037 s	0.000029 s	0.000018 s
60	0.000213 s	0.000202 s	0.000181 s	0.000157 s	0.000131 s	0.000104 s	0.000080 s	0.000061 s	0.000041 s
80	0.000374 s	0.000356 s	0.000319 s	0.000276 s	0.000229 s	0.000183 s	0.000138 s	0.000104 s	0.000063 s
100	0.000580 s	0.000551 s	0.000496 s	0.000428 s	0.000356 s	0.000283 s	0.000215 s	0.000153 s	0.000100 s
120	0.000833 s	0.000790 s	0.000710 s	0.000613 s	0.000510 s	0.000405 s	0.000307 s	0.000223 s	0.000140 s
140	0.001133 s	0.001086 s	0.000968 s	0.000832 s	0.000695 s	0.000552 s	0.000417 s	0.000302 s	0.000189 s
160	0.001477 s	0.001399 s	0.001260 s	0.001084 s	0.000900 s	0.000716 s	0.000542 s	0.000389 s	0.000237 s
180	0.001865 s	0.001769 s	0.001591 s	0.001373 s	0.001145 s	0.000907 s	0.000698 s	0.000482 s	0.000304 s
200	0.002298 s	0.002189 s	0.001959 s	0.001695 s	0.001407 s	0.001118 s	0.000839 s	0.000597 s	0.000369 s
220	0.002778 s	0.002636 s	0.002401 s	0.002067 s	0.001698 s	0.001354 s	0.001019 s	0.000726 s	0.000445 s
240	0.003309 s	0.003152 s	0.002821 s	0.002432 s	0.002024 s	0.001603 s	0.001212 s	0.000863 s	0.000528 s
260	0.003879 s	0.003710 s	0.003306 s	0.002848 s	0.002369 s	0.001899 s	0.001418 s	0.001001 s	0.000620 s
280	0.004495 s	0.004252 s	0.003829 s	0.003299 s	0.002738 s	0.002179 s	0.001632 s	0.001159 s	0.000721 s
300	0.005140 s	0.004871 s	0.004391 s	0.003799 s	0.003134 s	0.002501 s	0.001884 s	0.001347 s	0.000807 s
320	0.005871 s	0.005551 s	0.004986 s	0.004306 s	0.003582 s	0.002856 s	0.002144 s	0.001531 s	0.000904 s
340	0.006618 s	0.006259 s	0.005618 s	0.004860 s	0.004037 s	0.003220 s	0.002431 s	0.001685 s	0.001043 s
360	0.007370 s	0.006990 s	0.006290 s	0.005400 s	0.004517 s	0.003607 s	0.002752 s	0.001889 s	0.001164 s
380	0.008220 s	0.007789 s	0.007001 s	0.006033 s	0.005023 s	0.003974 s	0.003018 s	0.002099 s	0.001296 s
400	0.009125 s	0.008597 s	0.007739 s	0.006691 s	0.005573 s	0.004468 s	0.003345 s	0.002336 s	0.001437 s
420	0.010013 s	0.009477 s	0.008499 s	0.007375 s	0.006128 s	0.004897 s	0.003696 s	0.002558 s	0.001574 s
440	0.010988 s	0.010426 s	0.009348 s	0.008153 s	0.006707 s	0.005319 s	0.004039 s	0.002859 s	0.001749 s
460	0.012015 s	0.011326 s	0.010170 s	0.008806 s	0.007332 s	0.005801 s	0.004453 s	0.003110 s	0.001898 s
480	0.013051 s	0.012366 s	0.011137 s	0.009581 s	0.007951 s	0.006362 s	0.004809 s	0.003405 s	0.002072 s
500	0.014114 s	0.013439 s	0.012075 s	0.010395 s	0.008693 s	0.006971 s	0.005231 s	0.003667 s	0.002224 s
520	0.015317 s	0.014532 s	0.013000 s	0.011241 s	0.009414 s	0.007453 s	0.005685 s	0.003964 s	0.002400 s
540	0.016488 s	0.015626 s	0.014038 s	0.012165 s	0.010130 s	0.008055 s	0.006155 s	0.004266 s	0.002580 s
560	0.017709 s	0.016795 s	0.015052 s	0.013145 s	0.010881 s	0.008604 s	0.006562 s	0.004621 s	0.002788 s
580	0.018964 s	0.018032 s	0.016202 s	0.014105 s	0.011712 s	0.009324 s	0.007007 s	0.004945 s	0.002959 s
600	0.020384 s	0.019247 s	0.017418 s	0.015145 s	0.012533 s	0.010001 s	0.007529 s	0.005307 s	0.003161 s
620	0.021728 s	0.020532 s	0.018465 s	0.016071 s	0.013265 s	0.010675 s	0.008046 s	0.005649 s	0.003360 s
640	0.023090 s	0.021819 s	0.019654 s	0.017075 s	0.014163 s	0.011257 s	0.008546 s	0.006006 s	0.003616 s
660	0.024524 s	0.023201 s	0.020974 s	0.018179 s	0.015220 s	0.012171 s	0.009112 s	0.006343 s	0.003858 s
680	0.026016 s	0.024714 s	0.022346 s	0.019240 s	0.016109 s	0.012759 s	0.009736 s	0.006683 s	0.004076 s
700	0.027646 s	0.026109 s	0.023536 s	0.020552 s	0.016953 s	0.013532 s	0.010213 s	0.007147 s	0.004328 s
720	0.029143 s	0.027603 s	0.024863 s	0.021511 s	0.017829 s	0.014287 s	0.010874 s	0.007536 s	0.004576 s
740	0.030724 s	0.029084 s	0.026135 s	0.022851 s	0.018924 s	0.015193 s	0.011492 s	0.007973 s	0.004843 s
760	0.032373 s	0.030609 s	0.027631 s	0.024174 s	0.020133 s	0.015957 s	0.012009 s	0.008361 s	0.005084 s
780	0.034157 s	0.032383 s	0.029257 s	0.025277 s	0.021167 s	0.016826 s	0.012751 s	0.008838 s	0.005360 s
800	0.035831 s	0.034014 s	0.030623 s	0.026481 s	0.022215 s	0.017579 s	0.013401 s	0.009303 s	0.005638 s
820	0.037700 s	0.035872 s	0.032413 s	0.027933 s	0.023334 s	0.018671 s	0.014010 s	0.009720 s	0.005915 s
840	0.039666 s	0.037400 s	0.033904 s	0.029398 s	0.024487 s	0.019663 s	0.014752 s	0.010235 s	0.006214 s
860	0.041443 s	0.039351 s	0.035617 s	0.030873 s	0.025612 s	0.020342 s	0.015408 s	0.010825 s	0.006517 s
880	0.043415 s	0.041017 s	0.036956 s	0.032016 s	0.026679 s	0.021273 s	0.016179 s	0.011278 s	0.006793 s
900	0.045347 s	0.043045 s	0.038710 s	0.033542 s	0.028063 s	0.022255 s	0.017057 s	0.011870 s	0.007113 s
920	0.047445 s	0.045279 s	0.040702 s	0.035618 s	0.029161 s	0.023419 s	0.017796 s	0.012351 s	0.007457 s
940	0.049462 s	0.046823 s	0.042346 s	0.036760 s	0.030904 s	0.024593 s	0.018538 s	0.012911 s	0.007786 s
960	0.051609 s	0.049000 s	0.044188 s	0.038296 s	0.032078 s	0.025715 s	0.019364 s	0.013483 s	0.008090 s
980	0.053643 s	0.050899 s	0.045918 s	0.040145 s	0.033586 s	0.026489 s	0.019958 s	0.013933 s	0.008452 s
1000	0.056042 s	0.053136 s	0.047977 s	0.041386 s	0.034849 s	0.027765 s	0.020997 s	0.014642 s	0.008799 s

Table 34: Time to fill a population from without-replacement tournament selections using the standard algorithm and using `random_sample()`. These values are depicted in Figure 73.

Population size	Tournament size as percentage of population size								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
20	0.000008 s	0.000011 s	0.000013 s	0.000015 s	0.000018 s	0.000016 s	0.000014 s	0.000012 s	0.000009 s
40	0.000019 s	0.000028 s	0.000037 s	0.000045 s	0.000055 s	0.000046 s	0.000038 s	0.000032 s	0.000022 s
60	0.000034 s	0.000053 s	0.000073 s	0.000094 s	0.000115 s	0.000095 s	0.000077 s	0.000061 s	0.000043 s
80	0.000053 s	0.000089 s	0.000124 s	0.000160 s	0.000196 s	0.000162 s	0.000129 s	0.000102 s	0.000064 s
100	0.000077 s	0.000132 s	0.000187 s	0.000243 s	0.000317 s	0.000248 s	0.000197 s	0.000146 s	0.000100 s
120	0.000105 s	0.000185 s	0.000265 s	0.000344 s	0.000424 s	0.000350 s	0.000278 s	0.000210 s	0.000138 s
140	0.000141 s	0.000252 s	0.000356 s	0.000465 s	0.000571 s	0.000473 s	0.000373 s	0.000288 s	0.000180 s
160	0.000177 s	0.000318 s	0.000461 s	0.000603 s	0.000741 s	0.000639 s	0.000481 s	0.000359 s	0.000221 s
180	0.000218 s	0.000398 s	0.000578 s	0.000755 s	0.000932 s	0.000772 s	0.000612 s	0.000439 s	0.000282 s
200	0.000264 s	0.000487 s	0.000709 s	0.000927 s	0.001151 s	0.000944 s	0.000738 s	0.000540 s	0.000342 s
220	0.000316 s	0.000583 s	0.000851 s	0.001116 s	0.001383 s	0.001138 s	0.000895 s	0.000653 s	0.000425 s
240	0.000373 s	0.000688 s	0.001010 s	0.001324 s	0.001641 s	0.001352 s	0.001059 s	0.000776 s	0.000484 s
260	0.000430 s	0.000808 s	0.001180 s	0.001541 s	0.001950 s	0.001571 s	0.001236 s	0.000902 s	0.000553 s
280	0.000498 s	0.000931 s	0.001362 s	0.001783 s	0.002220 s	0.001832 s	0.001425 s	0.001038 s	0.000638 s
300	0.000567 s	0.001066 s	0.001568 s	0.002053 s	0.002538 s	0.002110 s	0.001649 s	0.001189 s	0.000724 s
320	0.000642 s	0.001207 s	0.001776 s	0.002333 s	0.002879 s	0.002374 s	0.001876 s	0.001362 s	0.000810 s
340	0.000722 s	0.001361 s	0.001985 s	0.002635 s	0.003260 s	0.002663 s	0.002104 s	0.001495 s	0.000934 s
360	0.000820 s	0.001522 s	0.002230 s	0.002933 s	0.003680 s	0.002989 s	0.002352 s	0.001672 s	0.001031 s
380	0.000893 s	0.001695 s	0.002497 s	0.003271 s	0.004068 s	0.003327 s	0.002593 s	0.001857 s	0.001141 s
400	0.000982 s	0.001860 s	0.002742 s	0.003610 s	0.004500 s	0.003692 s	0.002862 s	0.002056 s	0.001287 s
420	0.001085 s	0.002062 s	0.003025 s	0.004015 s	0.004946 s	0.004062 s	0.003185 s	0.002291 s	0.001430 s
440	0.001187 s	0.002246 s	0.003340 s	0.004386 s	0.005463 s	0.004425 s	0.003502 s	0.002510 s	0.001545 s
460	0.001291 s	0.002475 s	0.003602 s	0.004765 s	0.005936 s	0.004849 s	0.003810 s	0.002731 s	0.001655 s
480	0.001399 s	0.002659 s	0.003967 s	0.005218 s	0.006476 s	0.005282 s	0.004120 s	0.002980 s	0.001795 s
500	0.001534 s	0.002887 s	0.004274 s	0.005653 s	0.006983 s	0.005778 s	0.004454 s	0.003209 s	0.001961 s
520	0.001640 s	0.003143 s	0.004603 s	0.006094 s	0.007583 s	0.006211 s	0.004837 s	0.003489 s	0.002087 s
540	0.001765 s	0.003367 s	0.004978 s	0.006556 s	0.008146 s	0.006699 s	0.005244 s	0.003754 s	0.002262 s
560	0.001897 s	0.003611 s	0.005317 s	0.007051 s	0.008781 s	0.007201 s	0.005593 s	0.004029 s	0.002441 s
580	0.002009 s	0.003869 s	0.005703 s	0.007593 s	0.009447 s	0.007702 s	0.005987 s	0.004354 s	0.002553 s
600	0.002171 s	0.004133 s	0.006153 s	0.008130 s	0.010058 s	0.008281 s	0.006421 s	0.004598 s	0.002739 s
620	0.002305 s	0.004378 s	0.006517 s	0.008577 s	0.010744 s	0.008785 s	0.006865 s	0.004907 s	0.002939 s
640	0.002452 s	0.004696 s	0.006935 s	0.009211 s	0.011443 s	0.009381 s	0.007257 s	0.005287 s	0.003083 s
660	0.002611 s	0.004967 s	0.007378 s	0.009744 s	0.012220 s	0.009953 s	0.007750 s	0.005537 s	0.003341 s
680	0.002776 s	0.005331 s	0.007857 s	0.010354 s	0.012923 s	0.010599 s	0.008282 s	0.005845 s	0.003542 s
700	0.002951 s	0.005648 s	0.008322 s	0.010995 s	0.013683 s	0.011135 s	0.008698 s	0.006200 s	0.003740 s
720	0.003072 s	0.005951 s	0.008765 s	0.011646 s	0.014435 s	0.011853 s	0.009251 s	0.006543 s	0.003963 s
740	0.003249 s	0.006227 s	0.009267 s	0.012243 s	0.015256 s	0.012472 s	0.009705 s	0.006877 s	0.004167 s
760	0.003423 s	0.006586 s	0.009742 s	0.012963 s	0.016080 s	0.013179 s	0.010187 s	0.007298 s	0.004393 s
780	0.003623 s	0.006972 s	0.010264 s	0.013602 s	0.016936 s	0.013864 s	0.010827 s	0.007650 s	0.004608 s
800	0.003790 s	0.007324 s	0.010837 s	0.014333 s	0.017839 s	0.014590 s	0.011349 s	0.008061 s	0.004910 s
820	0.003965 s	0.007633 s	0.011375 s	0.015033 s	0.018691 s	0.015305 s	0.011909 s	0.008471 s	0.005104 s
840	0.004161 s	0.008031 s	0.011903 s	0.015821 s	0.019661 s	0.016043 s	0.012496 s	0.008921 s	0.005394 s
860	0.004366 s	0.008466 s	0.012464 s	0.016491 s	0.020562 s	0.016767 s	0.013058 s	0.009358 s	0.005613 s
880	0.004552 s	0.008851 s	0.013100 s	0.017331 s	0.021539 s	0.017629 s	0.013688 s	0.009821 s	0.005872 s
900	0.004778 s	0.009220 s	0.013637 s	0.018155 s	0.022611 s	0.018412 s	0.014316 s	0.010262 s	0.006152 s
920	0.004981 s	0.009651 s	0.014271 s	0.018927 s	0.023603 s	0.019209 s	0.014935 s	0.010720 s	0.006411 s
940	0.005220 s	0.010077 s	0.014858 s	0.019709 s	0.024550 s	0.020108 s	0.015626 s	0.011216 s	0.006688 s
960	0.005449 s	0.010505 s	0.015452 s	0.020545 s	0.025532 s	0.020869 s	0.016285 s	0.011645 s	0.007027 s
980	0.005726 s	0.010958 s	0.016203 s	0.021439 s	0.026685 s	0.021817 s	0.016928 s	0.012144 s	0.007248 s
1000	0.005873 s	0.011385 s	0.016820 s	0.022365 s	0.027805 s	0.022746 s	0.017648 s	0.012651 s	0.007627 s

Table 35: Time to fill a population from without-replacement tournament selections using the standard algorithm and a new random sampling subroutine. These values are depicted in Figure 74.

Population size	Tournament size as percentage of population size								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
20	0.000014 s	0.000014 s	0.000014 s	0.000013 s	0.000013 s	0.000013 s	0.000013 s	0.000013 s	0.000013 s
40	0.000024 s	0.000023 s	0.000023 s	0.000023 s	0.000022 s	0.000022 s	0.000022 s	0.000022 s	0.000022 s
60	0.000035 s	0.000034 s	0.000033 s	0.000033 s	0.000033 s	0.000032 s	0.000032 s	0.000031 s	0.000031 s
80	0.000046 s	0.000045 s	0.000044 s	0.000043 s	0.000043 s	0.000042 s	0.000043 s	0.000042 s	0.000042 s
100	0.000057 s	0.000056 s	0.000055 s	0.000054 s	0.000053 s	0.000053 s	0.000053 s	0.000052 s	0.000052 s
120	0.000068 s	0.000068 s	0.000066 s	0.000065 s	0.000064 s	0.000064 s	0.000063 s	0.000062 s	0.000062 s
140	0.000081 s	0.000080 s	0.000078 s	0.000076 s	0.000076 s	0.000076 s	0.000075 s	0.000074 s	0.000074 s
160	0.000093 s	0.000091 s	0.000089 s	0.000088 s	0.000087 s	0.000086 s	0.000087 s	0.000086 s	0.000084 s
180	0.000105 s	0.000103 s	0.000101 s	0.000099 s	0.000098 s	0.000098 s	0.000097 s	0.000096 s	0.000096 s
200	0.000117 s	0.000115 s	0.000113 s	0.000112 s	0.000110 s	0.000109 s	0.000108 s	0.000107 s	0.000107 s
220	0.000130 s	0.000127 s	0.000124 s	0.000123 s	0.000128 s	0.000120 s	0.000120 s	0.000118 s	0.000117 s
240	0.000143 s	0.000139 s	0.000136 s	0.000134 s	0.000134 s	0.000131 s	0.000137 s	0.000130 s	0.000128 s
260	0.000156 s	0.000152 s	0.000149 s	0.000147 s	0.000145 s	0.000144 s	0.000144 s	0.000142 s	0.000140 s
280	0.000174 s	0.000164 s	0.000161 s	0.000159 s	0.000158 s	0.000157 s	0.000156 s	0.000154 s	0.000154 s
300	0.000181 s	0.000176 s	0.000173 s	0.000172 s	0.000187 s	0.000168 s	0.000168 s	0.000166 s	0.000164 s
320	0.000194 s	0.000188 s	0.000186 s	0.000186 s	0.000182 s	0.000180 s	0.000179 s	0.000178 s	0.000177 s
340	0.000207 s	0.000201 s	0.000199 s	0.000197 s	0.000195 s	0.000194 s	0.000192 s	0.000190 s	0.000188 s
360	0.000220 s	0.000215 s	0.000212 s	0.000209 s	0.000208 s	0.000205 s	0.000205 s	0.000202 s	0.000199 s
380	0.000235 s	0.000229 s	0.000225 s	0.000222 s	0.000219 s	0.000218 s	0.000216 s	0.000213 s	0.000212 s
400	0.000246 s	0.000242 s	0.000238 s	0.000236 s	0.000232 s	0.000229 s	0.000229 s	0.000226 s	0.000225 s
420	0.000262 s	0.000255 s	0.000249 s	0.000247 s	0.000245 s	0.000244 s	0.000240 s	0.000237 s	0.000236 s
440	0.000276 s	0.000267 s	0.000262 s	0.000259 s	0.000256 s	0.000255 s	0.000254 s	0.000249 s	0.000249 s
460	0.000288 s	0.000281 s	0.000277 s	0.000273 s	0.000270 s	0.000285 s	0.000265 s	0.000263 s	0.000259 s
480	0.000303 s	0.000294 s	0.000289 s	0.000285 s	0.000282 s	0.000279 s	0.000277 s	0.000275 s	0.000273 s
500	0.000316 s	0.000307 s	0.000301 s	0.000297 s	0.000294 s	0.000293 s	0.000289 s	0.000286 s	0.000283 s
520	0.000330 s	0.000321 s	0.000315 s	0.000312 s	0.000309 s	0.000307 s	0.000304 s	0.000302 s	0.000301 s
540	0.000343 s	0.000335 s	0.000329 s	0.000325 s	0.000323 s	0.000321 s	0.000318 s	0.000315 s	0.000315 s
560	0.000355 s	0.000347 s	0.000342 s	0.000338 s	0.000336 s	0.000333 s	0.000331 s	0.000327 s	0.000326 s
580	0.000369 s	0.000361 s	0.000356 s	0.000351 s	0.000349 s	0.000345 s	0.000343 s	0.000342 s	0.000337 s
600	0.000384 s	0.000374 s	0.000369 s	0.000364 s	0.000361 s	0.000359 s	0.000354 s	0.000354 s	0.000369 s
620	0.000397 s	0.000387 s	0.000382 s	0.000377 s	0.000374 s	0.000373 s	0.000370 s	0.000367 s	0.000364 s
640	0.000412 s	0.000403 s	0.000396 s	0.000390 s	0.000388 s	0.000388 s	0.000381 s	0.000380 s	0.000379 s
660	0.000426 s	0.000415 s	0.000409 s	0.000405 s	0.000402 s	0.000401 s	0.000395 s	0.000394 s	0.000390 s
680	0.000441 s	0.000430 s	0.000424 s	0.000420 s	0.000415 s	0.000410 s	0.000409 s	0.000408 s	0.000406 s
700	0.000454 s	0.000444 s	0.000436 s	0.000431 s	0.000427 s	0.000424 s	0.000422 s	0.000421 s	0.000418 s
720	0.000471 s	0.000457 s	0.000450 s	0.000445 s	0.000440 s	0.000438 s	0.000433 s	0.000432 s	0.000432 s
740	0.000484 s	0.000472 s	0.000464 s	0.000458 s	0.000456 s	0.000449 s	0.000447 s	0.000444 s	0.000442 s
760	0.000497 s	0.000486 s	0.000477 s	0.000471 s	0.000467 s	0.000463 s	0.000460 s	0.000459 s	0.000455 s
780	0.000515 s	0.000505 s	0.000495 s	0.000489 s	0.000486 s	0.000481 s	0.000478 s	0.000473 s	0.000470 s
800	0.000529 s	0.000518 s	0.000507 s	0.000503 s	0.000496 s	0.000494 s	0.000491 s	0.000487 s	0.000485 s
820	0.000540 s	0.000529 s	0.000520 s	0.000517 s	0.000512 s	0.000506 s	0.000501 s	0.000500 s	0.000497 s
840	0.000556 s	0.000546 s	0.000537 s	0.000532 s	0.000526 s	0.000521 s	0.000517 s	0.000512 s	0.000511 s
860	0.000570 s	0.000560 s	0.000551 s	0.000546 s	0.000538 s	0.000534 s	0.000529 s	0.000526 s	0.000525 s
880	0.000585 s	0.000574 s	0.000564 s	0.000559 s	0.000554 s	0.000546 s	0.000541 s	0.000538 s	0.000535 s
900	0.000600 s	0.000588 s	0.000578 s	0.000572 s	0.000567 s	0.000564 s	0.000557 s	0.000552 s	0.000547 s
920	0.000614 s	0.000599 s	0.000592 s	0.000585 s	0.000579 s	0.000572 s	0.000571 s	0.000565 s	0.000563 s
940	0.000628 s	0.000618 s	0.000607 s	0.000599 s	0.000595 s	0.000588 s	0.000583 s	0.000581 s	0.000577 s
960	0.000644 s	0.000631 s	0.000623 s	0.000615 s	0.000607 s	0.000600 s	0.000595 s	0.000592 s	0.000590 s
980	0.000660 s	0.000643 s	0.000634 s	0.000629 s	0.000622 s	0.000616 s	0.000609 s	0.000604 s	0.000605 s
1000	0.000674 s	0.000660 s	0.000650 s	0.000641 s	0.000634 s	0.000627 s	0.000623 s	0.000617 s	0.000613 s

Table 36: Time to fill a population from without-replacement tournament selections using the new algorithm. These values are depicted in Figure 75.

Population size	Tournament size as percentage of population size								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
20	0.000004 s	0.000005 s	0.000007 s	0.000012 s	0.000011 s	0.000013 s	0.000015 s	0.000016 s	0.000019 s
40	0.000009 s	0.000017 s	0.000025 s	0.000033 s	0.000042 s	0.000050 s	0.000058 s	0.000062 s	0.000073 s
60	0.000019 s	0.000037 s	0.000056 s	0.000072 s	0.000089 s	0.000108 s	0.000125 s	0.000139 s	0.000157 s
80	0.000033 s	0.000065 s	0.000097 s	0.000127 s	0.000159 s	0.000190 s	0.000222 s	0.000249 s	0.000288 s
100	0.000052 s	0.000101 s	0.000149 s	0.000197 s	0.000248 s	0.000298 s	0.000347 s	0.000402 s	0.000445 s
120	0.000072 s	0.000143 s	0.000214 s	0.000285 s	0.000355 s	0.000431 s	0.000500 s	0.000568 s	0.000638 s
140	0.000098 s	0.000195 s	0.000291 s	0.000387 s	0.000487 s	0.000583 s	0.000686 s	0.000769 s	0.000869 s
160	0.000127 s	0.000253 s	0.000379 s	0.000507 s	0.000640 s	0.000760 s	0.000890 s	0.001005 s	0.001141 s
180	0.000161 s	0.000319 s	0.000480 s	0.000644 s	0.000803 s	0.000966 s	0.001110 s	0.001284 s	0.001433 s
200	0.000198 s	0.000394 s	0.000592 s	0.000798 s	0.000997 s	0.001195 s	0.001394 s	0.001587 s	0.001772 s
220	0.000240 s	0.000477 s	0.000719 s	0.000958 s	0.001204 s	0.001433 s	0.001674 s	0.001901 s	0.002155 s
240	0.000284 s	0.000565 s	0.000855 s	0.001140 s	0.001432 s	0.001705 s	0.001993 s	0.002265 s	0.002549 s
260	0.000335 s	0.000672 s	0.001009 s	0.001344 s	0.001695 s	0.002040 s	0.002350 s	0.002704 s	0.003019 s
280	0.000390 s	0.000779 s	0.001171 s	0.001559 s	0.001968 s	0.002337 s	0.002757 s	0.003104 s	0.003492 s
300	0.000445 s	0.000893 s	0.001345 s	0.001807 s	0.002265 s	0.002688 s	0.003136 s	0.003600 s	0.004060 s
320	0.000509 s	0.001014 s	0.001528 s	0.002035 s	0.002550 s	0.003056 s	0.003599 s	0.004052 s	0.004623 s
340	0.000578 s	0.001151 s	0.001724 s	0.002297 s	0.002870 s	0.003447 s	0.004001 s	0.004641 s	0.005155 s
360	0.000643 s	0.001291 s	0.001932 s	0.002576 s	0.003223 s	0.003903 s	0.004489 s	0.005201 s	0.005783 s
380	0.000714 s	0.001455 s	0.002152 s	0.002903 s	0.003589 s	0.004304 s	0.005021 s	0.005741 s	0.006487 s
400	0.000792 s	0.001614 s	0.002411 s	0.003182 s	0.003982 s	0.004775 s	0.005567 s	0.006361 s	0.007138 s
420	0.000880 s	0.001754 s	0.002629 s	0.003509 s	0.004383 s	0.005264 s	0.006133 s	0.007015 s	0.007870 s
440	0.000958 s	0.001925 s	0.002884 s	0.003847 s	0.004807 s	0.005770 s	0.006737 s	0.007735 s	0.008635 s
460	0.001049 s	0.002126 s	0.003151 s	0.004208 s	0.005256 s	0.006312 s	0.007432 s	0.008456 s	0.009448 s
480	0.001145 s	0.002291 s	0.003438 s	0.004587 s	0.005724 s	0.006928 s	0.008011 s	0.009130 s	0.010276 s
500	0.001240 s	0.002516 s	0.003768 s	0.004978 s	0.006212 s	0.007519 s	0.008699 s	0.009916 s	0.011154 s
520	0.001342 s	0.002687 s	0.004074 s	0.005382 s	0.006716 s	0.008064 s	0.009411 s	0.010726 s	0.012065 s
540	0.001441 s	0.002898 s	0.004394 s	0.005857 s	0.007308 s	0.008694 s	0.010148 s	0.011564 s	0.013104 s
560	0.001557 s	0.003116 s	0.004674 s	0.006292 s	0.007792 s	0.009348 s	0.010912 s	0.012521 s	0.013988 s
580	0.001669 s	0.003343 s	0.005011 s	0.006694 s	0.008351 s	0.010028 s	0.011691 s	0.013421 s	0.015122 s
600	0.001785 s	0.003615 s	0.005371 s	0.007169 s	0.009026 s	0.010820 s	0.012528 s	0.014282 s	0.016187 s
620	0.001901 s	0.003817 s	0.005729 s	0.007651 s	0.009546 s	0.011459 s	0.013371 s	0.015252 s	0.017193 s
640	0.002023 s	0.004067 s	0.006112 s	0.008139 s	0.010177 s	0.012223 s	0.014261 s	0.016264 s	0.018367 s
660	0.002170 s	0.004342 s	0.006499 s	0.008692 s	0.010841 s	0.013031 s	0.015192 s	0.017349 s	0.019456 s
680	0.002301 s	0.004590 s	0.006904 s	0.009285 s	0.011486 s	0.013806 s	0.016198 s	0.018399 s	0.020663 s
700	0.002439 s	0.004962 s	0.007336 s	0.009742 s	0.012185 s	0.014629 s	0.017034 s	0.019468 s	0.022006 s
720	0.002577 s	0.005180 s	0.007823 s	0.010404 s	0.012981 s	0.015565 s	0.018017 s	0.020680 s	0.023192 s
740	0.002724 s	0.005443 s	0.008164 s	0.010912 s	0.013604 s	0.016464 s	0.019067 s	0.021793 s	0.024466 s
760	0.002910 s	0.005803 s	0.008624 s	0.011480 s	0.014357 s	0.017326 s	0.020097 s	0.023012 s	0.025812 s
780	0.003026 s	0.006125 s	0.009078 s	0.012087 s	0.015126 s	0.018176 s	0.021247 s	0.024224 s	0.027191 s
800	0.003224 s	0.006364 s	0.009538 s	0.012780 s	0.016059 s	0.019273 s	0.022280 s	0.025457 s	0.028607 s
820	0.003342 s	0.006689 s	0.010043 s	0.013517 s	0.016739 s	0.020044 s	0.023390 s	0.026728 s	0.030035 s
840	0.003506 s	0.007007 s	0.010516 s	0.014026 s	0.017524 s	0.021076 s	0.024542 s	0.028056 s	0.031504 s
860	0.003719 s	0.007349 s	0.011019 s	0.014700 s	0.018359 s	0.022034 s	0.025717 s	0.029356 s	0.033013 s
880	0.003848 s	0.007702 s	0.011535 s	0.015390 s	0.019372 s	0.023091 s	0.026923 s	0.030726 s	0.034564 s
900	0.004026 s	0.008045 s	0.012193 s	0.016231 s	0.020118 s	0.024290 s	0.028148 s	0.032147 s	0.036184 s
920	0.004257 s	0.008415 s	0.012627 s	0.016828 s	0.021033 s	0.025225 s	0.029563 s	0.033578 s	0.037776 s
940	0.004390 s	0.008778 s	0.013294 s	0.017552 s	0.021952 s	0.026332 s	0.030856 s	0.035069 s	0.039449 s
960	0.004579 s	0.009163 s	0.013857 s	0.018326 s	0.023051 s	0.027466 s	0.032041 s	0.036578 s	0.041171 s
980	0.004830 s	0.009636 s	0.014299 s	0.019091 s	0.023838 s	0.028628 s	0.033573 s	0.038157 s	0.042894 s
1000	0.005029 s	0.009951 s	0.015075 s	0.020030 s	0.024907 s	0.029834 s	0.034808 s	0.039758 s	0.044732 s

Table 37: The effect on old style tournament selection duration (in seconds) of varying total population size, tournament size fraction.

the standard algorithm (with the improved random sampling subroutine). Figure 77(a) shows a detail of Figure 77(b).

The total number of points in Figure 77(b) is 31125. This is as expected since there is a point for each of the multiples of four less than each of the multiples of four up to 1000, hence $n(n-1)/2$ points where $n = 250 = 1000/4$. For 98.911% of these points, the new algorithm was faster in all 20 cases; for 0.790%, it was slower in all 20 cases; only 0.299% represent configurations for which the 20 results were not unanimous. The new algorithm spends extra time sorting the population but aims to reduce the number of random number generations. How small (or large) does the tournament size have to be before the saving in random number generations is not worth the cost of the sort? Figure 77 shows that the new algorithm is only slower when the tournament size is very small or when it is very large in a small population.

7.2.4 The Effect of Tournament Size on Selection Pressure

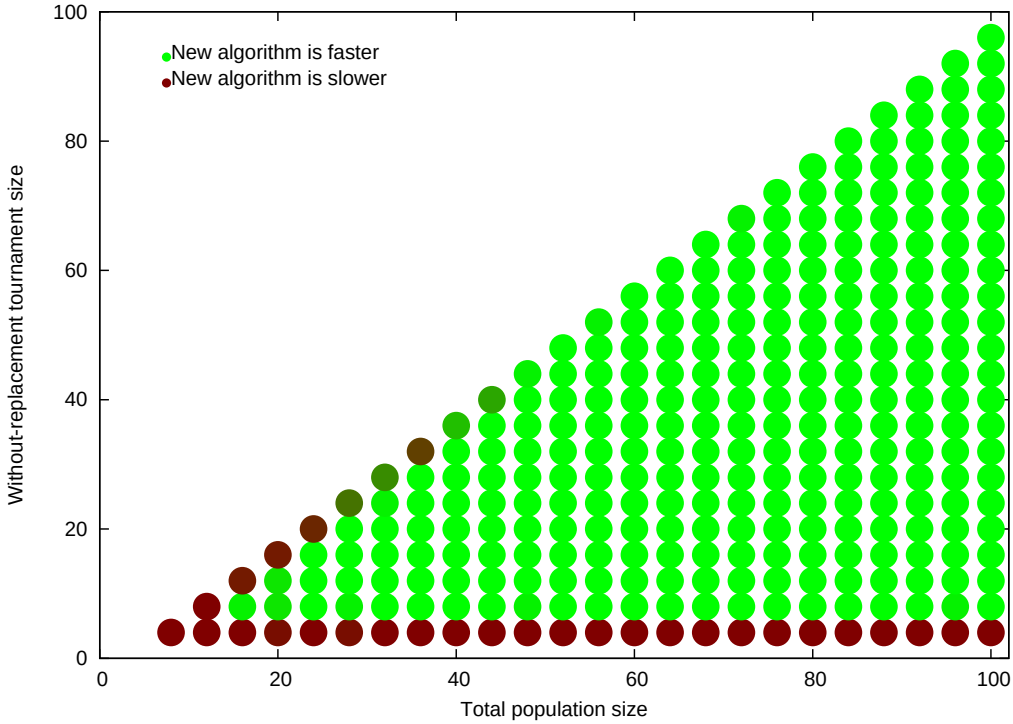
This analysis also supplied tools to help unearth deeper insights into the role of tournament size on selection pressure for without-replacement tournament selection. In particular, it can show how to adjust the tournament size to maintain equivalent selection pressure as the population size is varied. Without the benefit of the analysis, two obvious candidate tactics present themselves: maintain an identical tournament size or maintain an identical ratio between the tournament size and the population size.

Some thought suggests that neither of these can be quite right and examples of the use of each in the literature help develop this line of thought. It should be noted that neither of the papers in the following examples specifies whether the tournament selection is performed with or without replacement. The following discussion will work on the presumption that they are using without-replacement tournament selection. This highlights the need for the replacement status to be included when reporting experimental details.

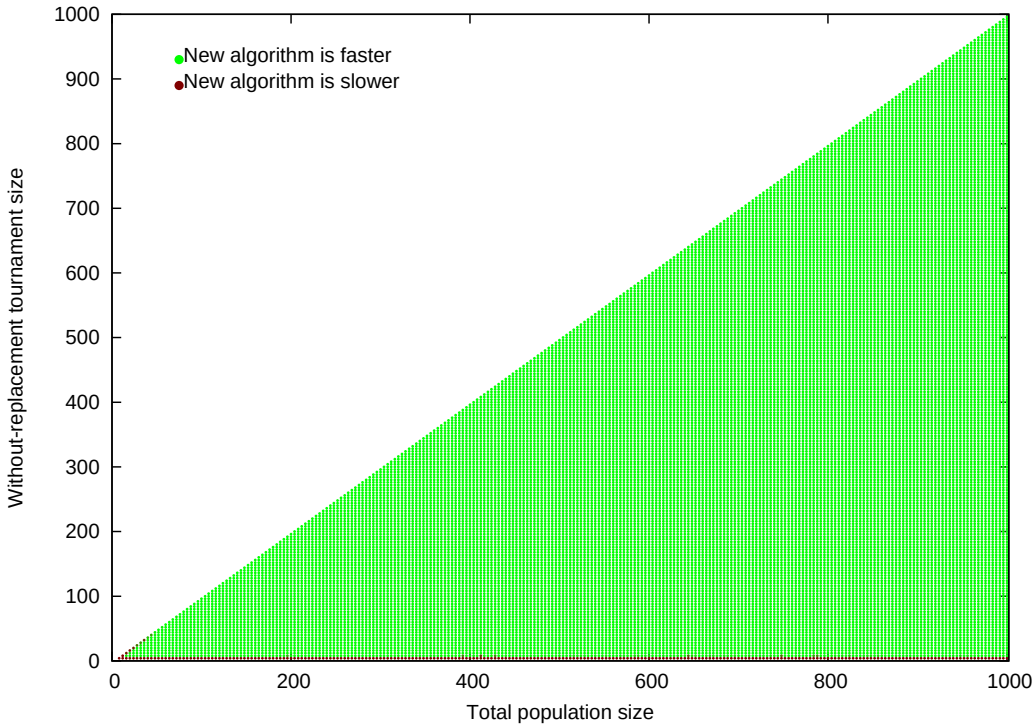
One study examining the role of demes [19] compared the results of evolutionary runs in populations of 2500 using different population structures such as one deme of 2500 or ten demes of 250. The ratio of tournament size to sub-population size was kept constant at 10%.

Using the cumulative formulae derived in Section 7.2.2 with tournament sizes of 10% shows that a tournament selection from a deme of 250 is 95.088% likely to come from the best 10.8% of individuals whereas a tournament selection from a deme of 2500 is 95.375% likely to come from the best 1.16%. This shows that 10% tournament selection provides very much stronger selection pressure in a deme of 2500 than in a deme of 250. Hence holding the tournament size a constant ratio of the population size as it varies, does not ensure constant selection pressure.

Another study compared short runs with big populations against long runs with small populations [23]. As the population size was varied over values between 50



(a) Results for population sizes up to 100



(b) Results for population sizes up to 1000

Figure 77: The without-replacement configurations for which the new algorithm is faster than the standard algorithm (with the improved random sampling subroutine). Light green points indicate configurations for which the new algorithm is faster; dark red points indicate configurations for which the algorithm is slower. Each point represents twenty trials: where the results were not unanimous, the point is shaded between the two colours according to the fraction that were faster. Points are placed at tournament sizes and population sizes divisible by four.

and 400, a constant tournament size of four individuals was used. Using the formulae derived in Section 7.2.2 shows that such a tournament selection in a population of 400 is 92.362% likely to come from the best 47.25% whereas in a population of 50, it is 92.380% likely to come from the best 46% of individuals. The difference is smaller in this particular example (than in the previous example). Nevertheless, the fact that the difference exists shows that holding the absolute tournament size constant as the population size varies, does not ensure constant selection pressure either.

Although the selection pressure varies considerably less in the latter example, it is possible to produce a larger change in selection pressure by using more extreme values: e.g. in a population of six, a tournament selection using four individuals is 93.333% likely to come from the best 33% of individuals.

So it turns out that it should indeed be possible to do better than either of these strategies. This attempt will benefit from a bit more mathematics.

7.2.5 Extending Tournament Selection Mathematics Beyond Integers

In Sections 7.2.7 and 7.2.8, it will turn out to be useful to extend the probability formulae to continuous functions over real values of i and m. This is because the continuous functions will make possible a consistent way to compare selection pressure between different tournament selection configurations.

For with-replacement, there is no work to do since the functions are already continuous with respect to i and m. For without-replacement, continuous functions can be achieved by replacing the factorial function with the Γ function, which extends it to real and complex numbers such that $\Gamma(x) = (x-1)!$ for all positive integers x.

It is important to remember that there is no claim that the results it provides between integers will be meaningful in this analysis. This is only one of an infinite number of functions which extend the factorial function, although the Bohr-Möllerup theorem states it is the only one defined for all positive real numbers that is logarithmically convex [2] and Section 7.2.6 shows that the resulting formulae are well behaved.

At this point, i and m are still assumed to be positive integers and so the equation $\Gamma(x) = (x-1)!$ can be used to replace the factorials in the formula for values $i \geq N - m + 2$:

$$\begin{aligned} P(E_i) &= \frac{m}{N} \frac{(N-m)!}{(N-m-i+1)!} \frac{(N-i)!}{(N-1)!} \\ &= \frac{m}{N} \frac{\Gamma(N+1-m)}{\Gamma(N+2-m-i)} \frac{\Gamma(N+1-i)}{\Gamma(N)} \end{aligned}$$

Now the assumption that i and m are integers may be dropped to reveal a continuous function. In practice, this form is rather hard to calculate and will be prone to large rounding errors in the huge Γ function values. What is required is a form that allows the value to be calculated as before for $\lfloor i \rfloor$ and $\lfloor m \rfloor$ and then modified according to the

differences $i - \lfloor i \rfloor$ and $m - \lfloor m \rfloor$. This can be obtained by multiplying top and bottom by various Γ functions of the floors of i and m, which produces something akin to the original formula using $\lfloor i \rfloor$ and $\lfloor m \rfloor$ multiplied by ratios of various Γ functions.

$$\begin{aligned}
&\frac{m}{N}\frac{\Gamma(N+1-m)}{\Gamma(N+2-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N)} \\
&= \frac{m}{N}\frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)}\frac{\Gamma(N+1-\lfloor m \rfloor)}{\Gamma(N+2-\lfloor m \rfloor-\lfloor i \rfloor)}\frac{\Gamma(N+2-\lfloor m \rfloor-\lfloor i \rfloor)}{\Gamma(N+2-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)}\frac{\Gamma(N+1-\lfloor i \rfloor)}{\Gamma(N)} \\
&= \frac{m}{N}\frac{(N-\lfloor m \rfloor)!}{(N-\lfloor m \rfloor-\lfloor i \rfloor+1)!}\frac{(N-\lfloor i \rfloor)!}{(N-1)!}\frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)}\frac{\Gamma(N+2-\lfloor m \rfloor-\lfloor i \rfloor)}{\Gamma(N+2-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)}
\end{aligned}$$

There is no danger of multiplying top and bottom by zero since there is no value $z \in \mathbb{C}$ such that $\Gamma(z) = 0$. The same technique can be used on the formula for cumulative probability for values $i \geq N - m + 1$:

$$\begin{aligned}
P(C_i) &= 1 - \frac{(N-m)!}{(N-m-i)!}\frac{(N-i)!}{(N)!} \\
&= 1 - \frac{\Gamma(N+1-m)}{\Gamma(N+1-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1)}
\end{aligned}$$

Again, on dropping the assumption that i and m are integers, something akin to the original formula may be retrieved using $\lfloor i \rfloor$ and $\lfloor m \rfloor$ and multiplied by ratios of various Γ functions

$$\begin{aligned}
&1 - \frac{\Gamma(N+1-m)}{\Gamma(N+1-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1)} \\
&= 1 - \frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)}\frac{\Gamma(N+1-\lfloor m \rfloor)}{\Gamma(N+1-\lfloor m \rfloor-\lfloor i \rfloor)}\frac{\Gamma(N+1-\lfloor m \rfloor-\lfloor i \rfloor)}{\Gamma(N+1-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)}\frac{\Gamma(N+1-\lfloor i \rfloor)}{\Gamma(N+1)} \\
&= 1 - \frac{(N-\lfloor m \rfloor)!}{(N-\lfloor m \rfloor-\lfloor i \rfloor)!}\frac{(N-\lfloor i \rfloor)!}{(N)!}\frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)}\frac{\Gamma(N+1-\lfloor m \rfloor-\lfloor i \rfloor)}{\Gamma(N+1-m-i)}\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)}
\end{aligned}$$

These particular forms of the continuous extensions are of particular value since the math library of the Boost C++ Library (`www.boost.org`) provides a function `tgamma_delta_ratio()` which is specifically designed for accurately computing ratios $\frac{\Gamma(a)}{\Gamma(a+\delta)}$ where δ may be small compared to a. Hence, these arrangements of the formulae can be used in an algorithm which is essentially the same as for integers but with the result multiplied and divided by some easily calculable values.

7.2.6 Showing the Continuous Extensions are Well Behaved

Before going further, it is worth demonstrating that the formulae that have been derived are well behaved (positive, continuous and monotonic with respect to i and m). The with-replacement formulae are fairly straightforward but things are not so simple

for the without-replacement formulae. First, it is worth noting that Γ is strictly positive for all real values greater than zero.

Proposition 1. *If f is a log-convex function on $\mathbb{R}$ and $y \in \mathbb{R}$ such that $y > 0$, then $\frac{f(x)}{f(x-y)}$ is monotonically increasing with respect to x wherever $f(x) > 0$ and $f(x-y) > 0$.*

Proof. Remember the log-convexity of f means that $\log f$ is convex, i.e. for $t \in [0,1]$ and x_1 and x_2 in the domain of f:

$$\begin{aligned} &\log(f(tx_1 + (1-t)x_2)) \leq t \log f(x_1) + (1-t)\log(f(x_2)) \\ &\Rightarrow f(tx_1 + (1-t)x_2) \leq f(x_1)^t f(x_2)^{1-t} \end{aligned}$$

Now the aim is to show that for $a \in \mathbb{R}$ and $b \in \mathbb{R}$ such that $a < b$, $f(a) > 0$, $f(a-y) > 0$, $f(b) > 0$ and $f(b-y) > 0$:

$$\frac{f(a)}{f(a-y)} \leq \frac{f(b)}{f(b-y)}$$

There are three possibilities:

- $b - y = a$,
- $b - y < a$ or
- $b - y > a$

To show the result for $b - y = a$, use $t = \frac{1}{2}$, $x_1 = a - y$ and $x_2 = b$ with the log-convexity of f:

$$\begin{aligned} f\left(\frac{a-y}{2} + \frac{b}{2}\right) &\leq \sqrt{f(a-y)}\sqrt{f(b)} \\ \Rightarrow f(a) &\leq \sqrt{f(a-y)}\sqrt{f(b)} \\ \Rightarrow f(a)f(a) &\leq f(a-y)f(b) \\ \Rightarrow f(a)f(b-y) &\leq f(a-y)f(b) \\ \Rightarrow \frac{f(a)}{f(a-y)} &\leq \frac{f(b)}{f(b-y)} \text{ (since } f(a-y) > 0 \text{ and } f(b-y) > 0) \end{aligned}$$

To show the results for $b - y < a$, use $t = \frac{a-b+y}{y}$, $x_1 = a - y$ and $x_2 = a$:

$$\begin{aligned} f\left(\frac{a-b+y}{y}(a-y) + \frac{b-a}{y}a\right) &\leq f(a-y)^{\frac{a-b+y}{y}} f(a)^{\frac{b-a}{y}} \\ \Rightarrow f(b-y)^y &\leq f(a-y)^{a-b+y} f(a)^{b-a} \text{ (since } y > 0) \\ \Rightarrow \frac{f(a)^{a-b}}{f(a-y)^{a-b+y}} &\leq \frac{1}{f(b-y)^y} \\ &\text{(since } f(a) > 0, f(a-y) > 0 \text{ and } f(b-y) > 0) \end{aligned}$$

Then use $t = \frac{b-a}{y}$, $x_1 = b - y$ and $x_2 = b$:

$$f\left(\frac{b-a}{y}(b-y) + \frac{a-b+y}{y}b\right) \leq f(b-y)^{\frac{b-a}{y}} f(b)^{\frac{a-b+y}{y}}$$
$$\Rightarrow f(a)^y \leq f(b-y)^{b-a} f(b)^{a-b+y} \text{ (since } y > 0)$$
$$\Rightarrow \frac{f(a)^y}{1} \leq \frac{f(b)^{a-b+y}}{f(b-y)^{a-b}}$$

Multiplying these two (positive) results together and then using $a - b + y > 0$:

$$\frac{f(a)^{a-b+y}}{f(a-y)^{a-b+y}} \leq \frac{f(b)^{a-b+y}}{f(b-y)^{a-b+y}}$$
$$\Rightarrow \frac{f(a)}{f(a-y)} \leq \frac{f(b)}{f(b-y)}$$

To show the results for $b - y > a$, use $t = \frac{b-y-a}{b-a}$, $x_1 = a - y$ and $x_2 = b - y$:

$$f\left(\frac{b-y-a}{b-a}(a-y) + \frac{y}{b-a}(b-y)\right) \leq f(a-y)^{\frac{b-y-a}{b-a}} f(b-y)^{\frac{y}{b-a}}$$
$$\Rightarrow f(a)^{b-a} \leq f(a-y)^{b-y-a} f(b-y)^y \text{ (since } b - a > 0)$$
$$\Rightarrow \frac{f(a)^{b-a}}{f(a-y)^{b-y-a}} \leq \frac{1}{f(b-y)^{-y}} \text{ (since } f(a-y) > 0)$$

Then use $t = \frac{y}{b-a}$, $x_1 = a$ and $x_2 = b$:

$$f\left(\frac{y}{b-a}a + \frac{b-y-a}{b-a}b\right) \leq f(a)^{\frac{y}{b-a}} f(b)^{\frac{b-y-a}{b-a}}$$
$$\Rightarrow f(b-y)^{b-a} \leq f(a)^y f(b)^{b-y-a} \text{ (since } b - a > 0)$$
$$\Rightarrow \frac{f(a)^{-y}}{1} \leq \frac{f(b)^{b-y-a}}{f(b-y)^{b-a}} \text{ (since } f(a) > 0 \text{ and } f(b-y) > 0)$$

Multiplying these two (positive) results together and then using $b - y - a > 0$:

$$\frac{f(a)^{b-y-a}}{f(a-y)^{b-y-a}} \leq \frac{f(b)^{b-y-a}}{f(b-y)^{b-y-a}}$$
$$\Rightarrow \frac{f(a)}{f(a-y)} \leq \frac{f(b)}{f(b-y)}$$

So the required result has been shown for all three possibilities.

□

Proof. (outline of refined version) During the *viva voce* examination of this thesis, one of the examiners, Professor Qingfu Zhang, showed how this proof could be improved by taking the log of both sides of the desired inequality and then defining $g() = \log(f())$. This gives $g(a) - g(a-y) \leq g(b) - g(b-y)$, so the aim becomes to show that this

inequality is true if $g()$ is convex. It is now substantially more intuitive than in the original formulation to see that this is true. This improved clarity also helped me to see how to avoid splitting the proof into three different cases. This can be done by using the symmetry in the definition of convexity which means that the statement of convexity using $t \in [0,1]$ is also true using $(1-t) \in [0,1]$, regardless of whether $t < (1-t)$, $t = (1-t)$ or $t > (1-t)$. Using this to continue the proof in both formulations (without worrying about the details of where the original assumptions of positivity must be deployed) :

The log-convexity of f gives us that for any $t \in [0,1]$:

$$f(tx_1 + (1-t)x_2) \leq f(x_1)^t f(x_2)^{1-t}$$

Since $t \in [0,1]$, then also $(1-t) \in [0,1]$ and so:

$$f((1-t)x_1 + tx_2) \leq f(x_1)^{1-t} f(x_2)^t$$

From the first, it may be derived that:

$$\frac{f(x_2)}{f(tx_1 + (1-t)x_2)} \geq \left(\frac{f(x_2)}{f(x_1)}\right)^t$$

...and from the second, it may be derived that:

$$\frac{f((1-t)x_1 + tx_2)}{f(x_1)} \leq \left(\frac{f(x_2)}{f(x_1)}\right)^t$$

Combining these two inequalities gives:

$$\frac{f(x_2)}{f(tx_1 + (1-t)x_2)} \geq \frac{f((1-t)x_1 + tx_2)}{f(x_1)}$$

Now the desired result can be obtained by plugging the following values into this inequality:

$$\begin{aligned} x_1 &= a - y \\ x_2 &= b \\ t &= \frac{y}{b - a + y} \end{aligned}$$

The convexity of g gives us that for any $t \in [0,1]$:

$$g(tx_1 + (1-t)x_2) \leq tg(x_1) + (1-t)g(x_2)$$

Since $t \in [0,1]$, then also $(1-t) \in [0,1]$ and so:

$$g((1-t)x_1 + tx_2) \leq (1-t)g(x_1) + tg(x_2)$$

From the first, it may be derived that:

$$g(x_2) - g(tx_1 + (1-t)x_2) \geq t(g(x_2) - g(x_1))$$

...and from the second, it may be derived that:

$$g((1-t)x_1 + tx_2) - g(x_1) \leq t(g(x_2) - g(x_1))$$

Combining these two inequalities gives:

$$\begin{aligned} g(x_2) - g(tx_1 + (1-t)x_2) \geq \\ g((1-t)x_1 + tx_2) - g(x_1) \end{aligned}$$

Now the desired result can be obtained by plugging the following values into this inequality:

$$\begin{aligned} x_1 &= a - y \\ x_2 &= b \\ t &= \frac{y}{b - a + y} \end{aligned}$$

□

Proposition 2. *If $k, c \in \mathbb{N}^0$ and $\delta, \gamma \in \mathbb{R}^+$ such that $k > \delta + \gamma$ then $\frac{\Gamma(k+c-\delta)}{\Gamma(k+c)} \frac{\Gamma(k)}{\Gamma(k-\delta-\gamma)}$ is positive and monotonically decreasing with respect to δ.*

Proof. Start by expanding the left hand side fraction using $\Gamma(z+1) = z\Gamma(z)$:

$$\begin{aligned}
&\frac{\Gamma(k+c-\delta)}{\Gamma(k+c)} \frac{\Gamma(k)}{\Gamma(k-\delta-\gamma)} \\
&= \frac{(k+c-1-\delta)}{(k+c-1)} \frac{(k+c-2-\delta)}{(k+c-2)} \cdots \frac{(k-\delta)}{k} \frac{\Gamma(k-\delta)}{\Gamma(k)} \frac{\Gamma(k)}{\Gamma(k-\delta-\gamma)} \\
&= \frac{(k+c-1-\delta)}{(k+c-1)} \frac{(k+c-2-\delta)}{(k+c-2)} \cdots \frac{(k-\delta)}{k} \frac{\Gamma(k-\delta)}{\Gamma(k-\delta-\gamma)}
\end{aligned}$$

This is positive because each of the fractions is and they are positive because each of the numerators and denominators are. The Γ values are positive because their arguments are. The product will be monotonically decreasing if each of the fractions are. This is clearly true for all but the last and can be seen for that case using Proposition 1 and the log-convexity of Γ.

□

Now return to the formulae for the selection probability and for the cumulative probability:

$$\begin{aligned}
P(E_i) &= \frac{m}{N} \frac{\Gamma(N+1-m)}{\Gamma(N+2-m-i)} \frac{\Gamma(N+1-i)}{\Gamma(N)} \\
P(C_i) &= 1 - \frac{\Gamma(N+1-m)}{\Gamma(N+1-m-i)} \frac{\Gamma(N+1-i)}{\Gamma(N+1)}
\end{aligned}$$

These are both continuous where all the arguments to the Γ functions are strictly positive because then all the numerators and denominators are strictly positive and continuous. To show that they are monotonically decreasing and increasing respectively with respect to both i and m, it will suffice to show that this is true between any integer values. To do this, it will help to use the equivalent forms derived earlier and to observe that, if varying either i or m, only two of the fractions vary:

$$\begin{aligned}
P(E_i) &= \frac{m}{N} \frac{(N-\lfloor m \rfloor)!}{(N-\lfloor m \rfloor - \lfloor i \rfloor + 1)!} \frac{(N-\lfloor i \rfloor)!}{(N-1)!} \frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)} \frac{\Gamma(N+2-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+2-m-i)} \frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)} \\
P(C_i) &= 1 - \frac{(N-\lfloor m \rfloor)!}{(N-\lfloor m \rfloor - \lfloor i \rfloor)!} \frac{(N-\lfloor i \rfloor)!}{(N)!} \frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)} \frac{\Gamma(N+1-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+1-m-i)} \frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)}
\end{aligned}$$

To demonstrate the selection probability is monotonically decreasing with respect to i, use Proposition 2 by setting $k = N + 2 - \lfloor m \rfloor - \lfloor i \rfloor$, $c = \lfloor m \rfloor - 1$, $\delta = i - \lfloor i \rfloor$ and $\gamma = m - \lfloor m \rfloor$ to show the following is monotonically decreasing with respect to i :

$$\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)} \frac{\Gamma(N+2-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+2-m-i)}$$

To demonstrate the selection probability is monotonically decreasing with respect to m, use Proposition 2 by setting $k = N + 2 - \lfloor m \rfloor - \lfloor i \rfloor$, $c = \lfloor i \rfloor - 1$, $\delta = m - \lfloor m \rfloor$ and $\gamma = i - \lfloor i \rfloor$ to show the following is monotonically decreasing with respect to m :

$$\frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)} \frac{\Gamma(N+2-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+2-m-i)}$$

To demonstrate the cumulative probability is monotonically increasing with respect to i, use Proposition 2 by setting $k = N + 1 - \lfloor m \rfloor - \lfloor i \rfloor$, $c = \lfloor m \rfloor$, $\delta = i - \lfloor i \rfloor$ and $\gamma = m - \lfloor m \rfloor$ to show the following is monotonically decreasing with respect to i :

$$\frac{\Gamma(N+1-i)}{\Gamma(N+1-\lfloor i \rfloor)} \frac{\Gamma(N+1-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+1-m-i)}$$

To demonstrate the cumulative probability is monotonically increasing with respect to m, use Proposition 2 by setting $k = N + 1 - \lfloor m \rfloor - \lfloor i \rfloor$, $c = \lfloor i \rfloor$, $\delta = m - \lfloor m \rfloor$ and $\gamma = i - \lfloor i \rfloor$ to show the following is monotonically decreasing with respect to m :

$$\frac{\Gamma(N+1-m)}{\Gamma(N+1-\lfloor m \rfloor)} \frac{\Gamma(N+1-\lfloor m \rfloor - \lfloor i \rfloor)}{\Gamma(N+1-m-i)}$$

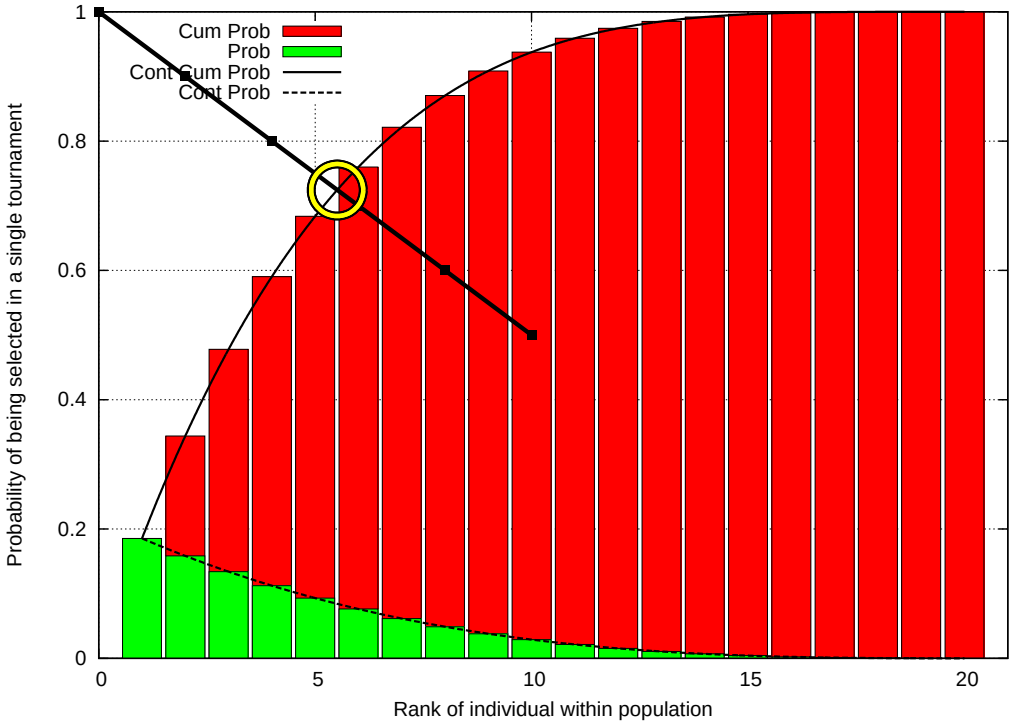
This argument has skimmed over the difficulties at the boundaries where, say, $m + i \approx N + 1$. By returning to the original formulae, it is possible to handle these points at which the selection probability is zero and the cumulative probability is one. Care was required to write the code in such as way as to calculate these values correctly.

7.2.7 A New Measure of Selection Pressure: Many-From-Few

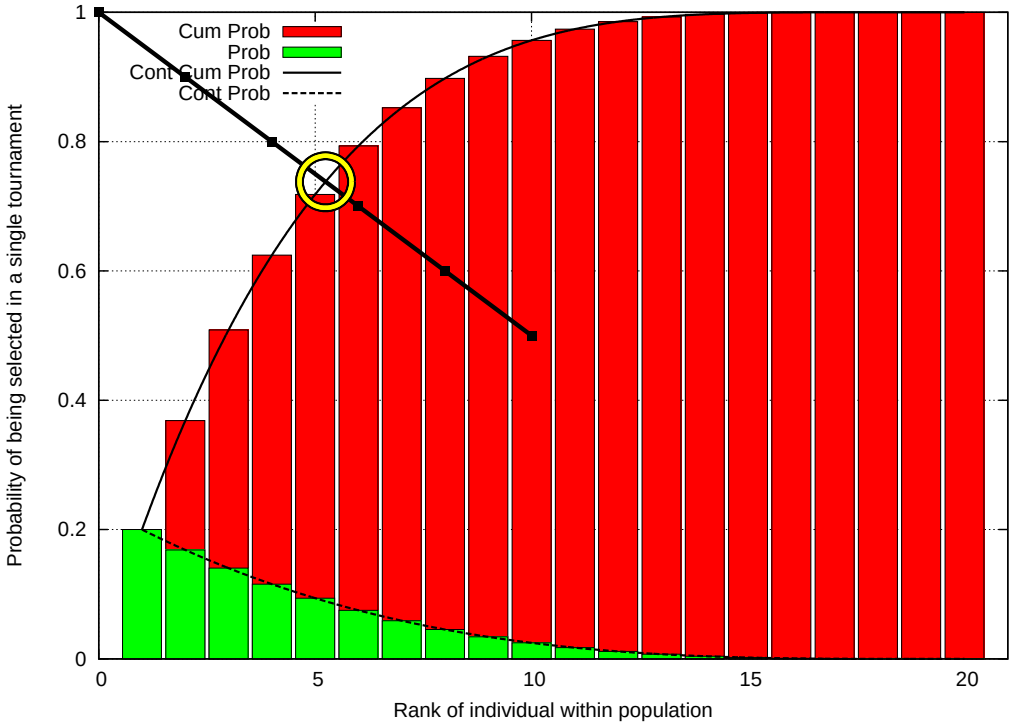
Figure 78 depicts examples of the continuous formulae, plotted in front of the discrete formulae that they extend over continuous values of i. As with the previous Figure 70, tournament size is four and the population size is 20. The figure illustrates pleasing features of the continuous extensions: they are smooth, match the discrete values at the appropriate points and are monotonically increasing or decreasing with respect to i as would be hoped.

These continuous functions return the analysis to the question of how to compare the selection pressure of different tournament selection configurations. This is a difficult problem without the benefit of the continuous functions because the two configurations might have relatively prime (i.e. coprime) population sizes. In this case, no discrete subsets will represent equal fractions of their populations and hence no direct comparisons will be possible (unless two of the cumulative probabilities happen to match).

A new measure of selection pressure is proposed called the *many-from-few* mea-



(a) With-replacement tournament selection probabilities and continuous extensions



(b) Without-replacement tournament selection probabilities and continuous extensions

Figure 78: The probabilities shown in Figure 70 may be extended to smooth continuous functions. In each case, the continuous extensions of probability and cumulative probability are shown as a dashed black curve and a solid black curve respectively. The thick black line shows all points where the fraction of the population and the probability would sum to one, marked off in divisions of 10%. The many-from-few measure identifies the point where this intersects with the cumulative probability (as highlighted by the yellow annulus).

sure. It indicates where the cumulative probability crosses the thick black line shown in Figure 78. That line represents those points at which the two percentages (i.e. x% likelihood that selection will choose from the best y%) add up to 100%. This measure recalls the structure of the Pareto Principle ("roughly 80% of the effects come from 20% of the causes").

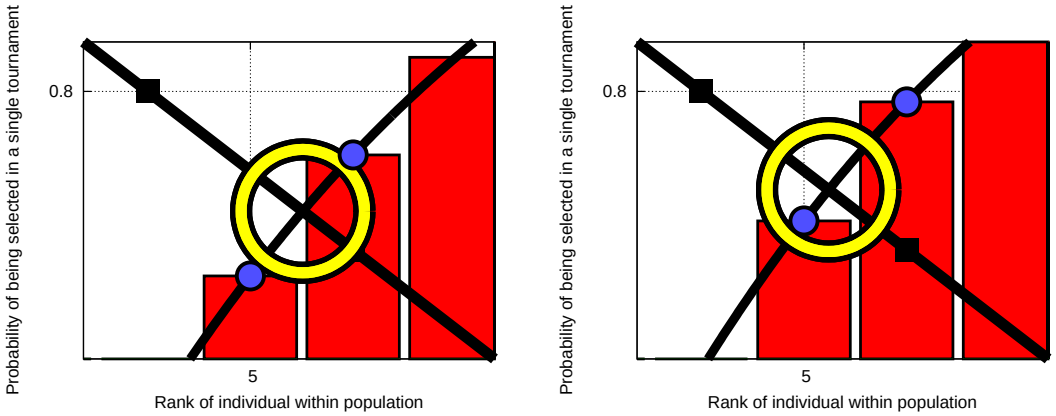
The continuous functions allow exact comparisons of the selection pressure of configurations at intermediate values. In other cases, it might be more appropriate to find the many-from-few measure by finding the discrete point nearest the line. Note that the many-from-few measure could easily be empirically recorded from a run using any selection scheme using a histogram of the selection frequencies for the various fitness ranks.

In Figure 78, marks on the thick black line indicate points every 10% from 50%/50% up to 100%/0%. "50% likely from best 50%" represents no selection pressure at all, which occurs when the tournament size is 1. For without-replacement tournament selection, the selection pressure will be at its strongest when the tournament size is equal to the population size; the selection pressure of this configuration tends to "100% likely from best 0%" as the population size tends to infinity. "95% likely from best 5%" represents selection pressure so strong that (in a population of unique fitness values) we would expect the fittest twentieth of the population to account for nineteen twentieths of the population in the next generation.

Figure 78(a) shows that a with-replacement tournament of size four in a population of 20 would be expected to select 72.449% from the best 27.551% of the individuals. Figure 78(b) shows that a without-replacement tournament of size four in a population of 20 would be expected to select 73.796% from the best 26.204% of the individuals. The difference between the two types of tournament selection happens to be small for this particular configuration but Section 7.2.8 shows that the difference can be much greater for other configurations.

Strictly speaking, the many-from-few values from the continuous function are only meaningful in terms of the information they give about the neighbouring integer values as indicated in Figure 79. Consider an example: the proposed measure for a without-replacement tournament size of four in a population of 20 (as in Figure 78(b)), is 73.796% likely from the best 26.204%. What does this mean? It means that a selection from any smaller subset of best individuals is less likely and a selection from any larger subset of best individuals is more likely. Indeed, as it turns out the nearest integer values show that any selection is 71.827% likely to come from the best 25.000% and 79.340% to come from the best 30.000%.

This measure is valuable because it provides an intuitive feel for the strength of selection pressure exerted by any given tournament selection configuration. It is only one value drawn from an fingerprint of selection pressure but since this can be done in a standardised way, it allows direct comparison between different configurations.



(a) With-replacement tournament selection (b) Without-replacement tournament selection

Figure 79: Strictly speaking, the many-from-few values (each indicated with a yellow annulus) from the continuous functions are only meaningful in terms of the information they give about the neighbouring integer values (each indicated with a blue spot). Subfigures 79(a) and 79(b) show details of Subfigures 78(a) and 78(b) respectively, with additional highlighting of the integer values neighbouring the many-from-few points.

Should we be concerned about using just one value to represent an entire selection pressure fingerprint for (say without-replacement) tournament selection? Figure 80 shows the continuous cumulative probability distributions for two very different without-replacement tournament configurations: a tournament of 10 in a population of 20 and a tournament of 14 in a population of 20000. These two setups were chosen to illustrate this point because, despite having very different configurations, they have very similar many-from-few values. With a tournament of 10 in a population of 20, we would expect 86.492% to come from the best 13.508% of the individuals. With a tournament of 14 in a population of 20000, we would expect 86.621% to come from the best 13.379% of the individuals. Figure 80 shows how similar the corresponding selection pressure fingerprints are by scaling the latter example to the range of the former. The lines are so similar it is difficult to see them both. This helps allay concerns about the dangers of representing an entire tournament selection fingerprint with a single many-from-few value.

7.2.8 Calculating Many-From-Few Values and Studying Selection Pressure

How can the many-from-few value be calculated precisely? For with-replacement tournament selection, this is fairly trivial. The labels are as before: i is the index of a specific individual and m is the size of the tournament. For simplicity, define $k = \frac{i}{m}$ and then observe that at the many-from-few point the cumulative probability is equal to $1 - k$:

$$1 - (1-k)^m = 1 - k$$
$$\Rightarrow m = \frac{\log(k)}{\log(1-k)}$$

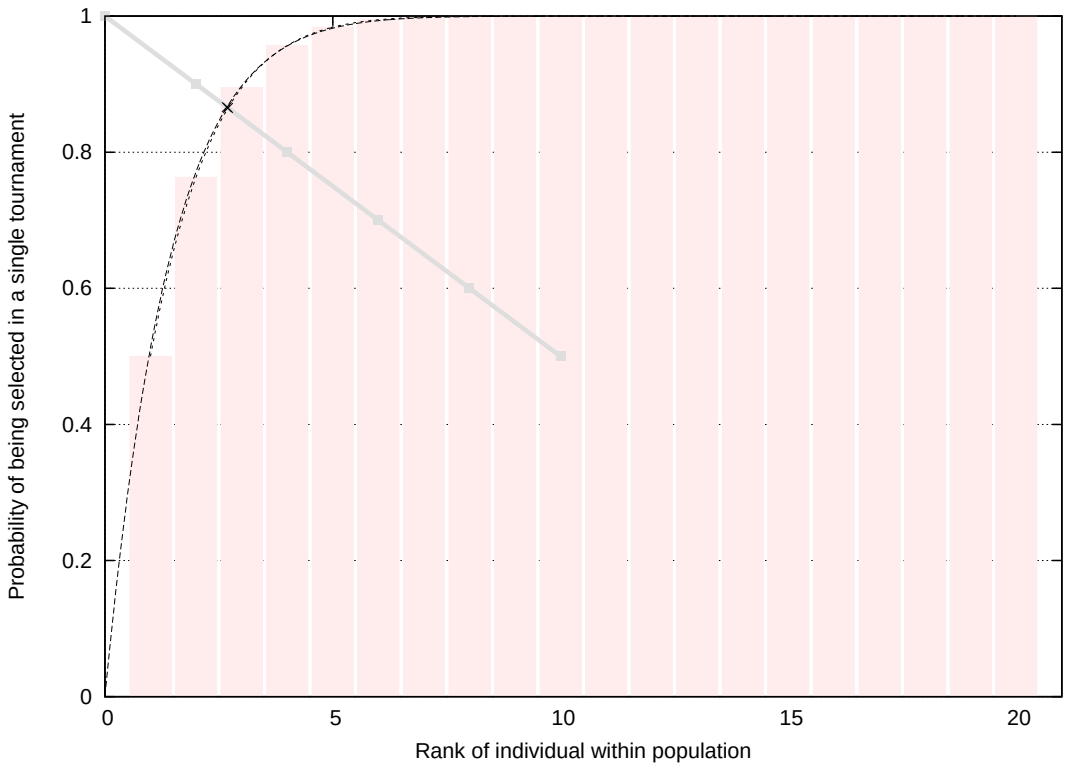


Figure 80: Very similar continuous cumulative probability distributions for two very different without-replacement tournament selection configurations. The graph shows one line representing a tournament of 10 in a population of 20 and the corresponding discrete bars behind in pale pink. On top of this, another line representing a tournament of 14 in a population of 20000 has been added, scaled down on the x-axis by a factor of 1000. The two many-from-few are indicated by two (very close) crosses. The two resulting lines are so similar that it is hard to see them both.

Hence it is easy to calculate m in terms of k. Note that the cumulative probability is expressible solely in terms of k (i's fraction through the population) and m. This illustrates that the selection pressure for with-replacement tournament selection is independent of the population size N as has previously been observed [106].

The without-replacement points of intersection are harder to find and require trial and error. To this end, a bisection algorithm is used repeatedly to subdivide the range known to contain the point until the location of the point is found to an acceptable degree of accuracy.

Examples selection pressure values generated using these methods are given in Tables 38 and 39 for with-replacement and without-replacement tournament selection respectively.

These methods can determine the many-from-few value for any given configuration. Is it also possible to determine the tournament size that exerts a given selection pressure for a given population size? This can be done using another bisection process using the continuity of the formulae over tournament size. For without-replacement tournament selection, this involves a double bisection process. In the outer bisection, a

range of tournament sizes is narrowed down to the value that gives the correct selection pressure. For each guess of the outer bisection, an inner bisection determines the selection pressure, as before.

Examples of tournament size values generated by these methods are shown in Tables 40 and 41 for with-replacement and without-replacement tournament selection respectively.

This supplies the last tools required to answer the question about the relationship between tournament size, population size and the resulting selection pressure. Figures 81 and 82 display the fruits of this labour: graphs of selection pressure contours for with-replacement and without-replacement respectively. Each line on these graphs represents a specific strength of selection pressure and its shape shows how to adjust tournament size (if at all) to maintain constant selection pressure with varying population size.

Tournament size													
5	10	15	20	25	30	35	40	45	50	60	70	80	90
75.488	83.508	87.195	89.390	90.870	91.946	92.769	93.423	93.955	94.399	95.098	95.627	96.043	96.380

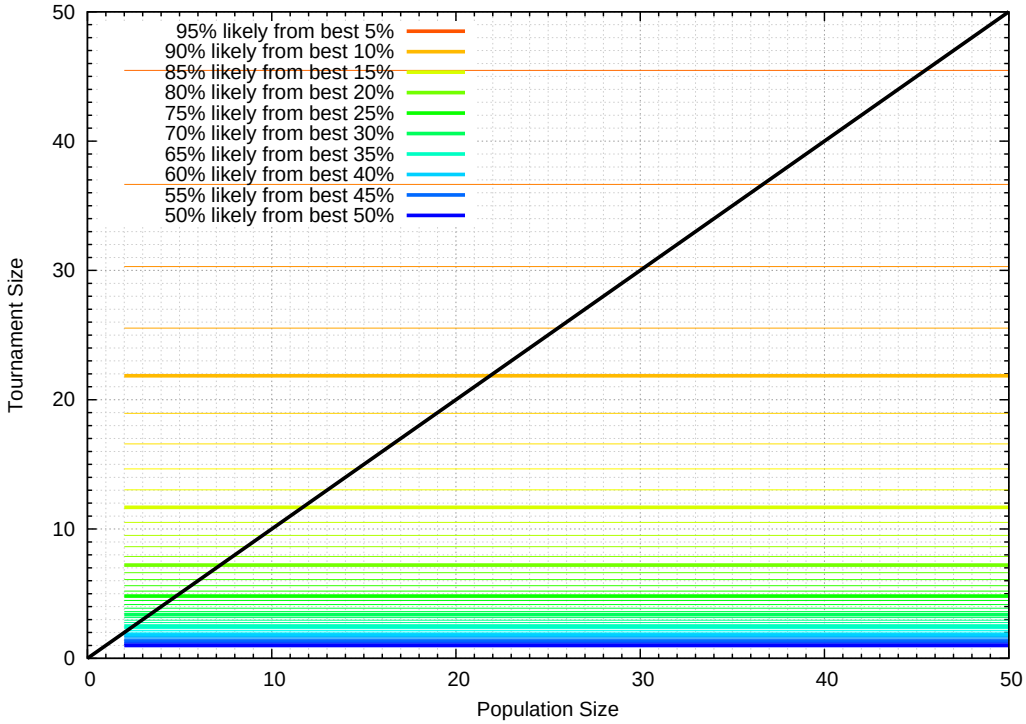
Table 38: The many-from-few selection pressure (as a percentage) exerted by with-replacement tournament selection for various tournament sizes.

The black lines indicate where tournament size equals population size. This is an upper bound for without-replacement (Figure 82) but not for with-replacement (Figure 81). These graphs clearly show how different with-replacement is from without-replacement. The contours in Figure 81 are all perfectly horizontal and are all higher than their equivalents in Figure 82 (because with-replacement requires larger tournaments to achieve the same selection pressure).

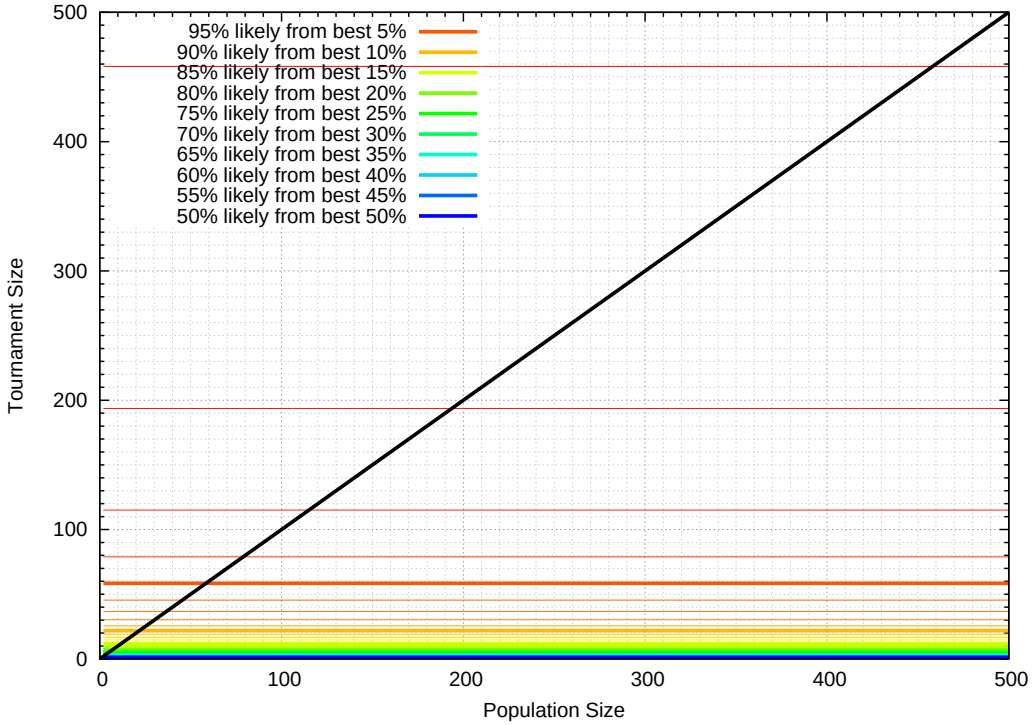
Here it is possible to see more clearly that neither the "constant" strategy nor the "constant ratio" strategy described earlier is capable of dealing with all without-replacement situations. It is worth considering some examples. As a population grows from 250 to 500, adding just one to a tournament of 25 slightly increases the selection pressure (which was "91.192% from the best 8.808%"). As a population size grows from 25 to 50, a tournament of size 24 should be increased to 40 to maintain constant selection pressure. Perhaps surprisingly, this selection pressure is only the same as from 75 out of 750, namely "96% from the best 4%".

Again, these figures are generated using a continuous extension of a discrete formula and so should strictly be interpreted in the following way. If a line indicating "x% likely from the best y%" passes through some point m for population size N then:

- for any integer tournament size less than or equal to m, and for any integer index $i <= N * y/100$, the cumulative probability for selection within the best i members of the population is less than or equal to $x/100$ and
- for any integer tournament size greater than or equal to m, and for any integer

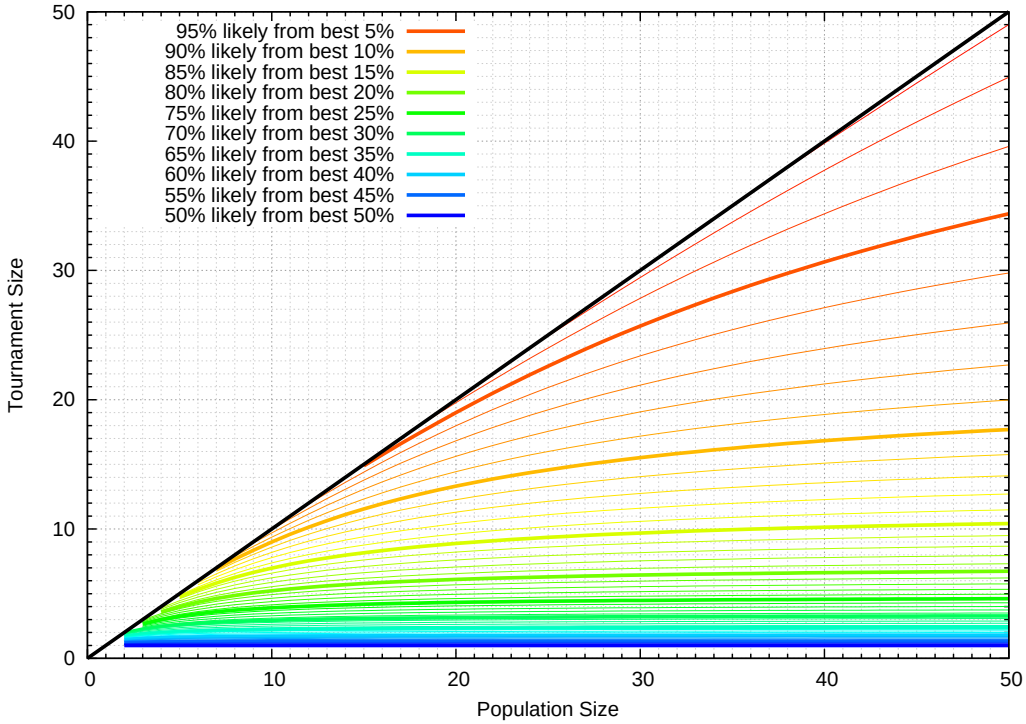


(a) Selection pressure contours for with-replacement tournaments in population sizes up to 50

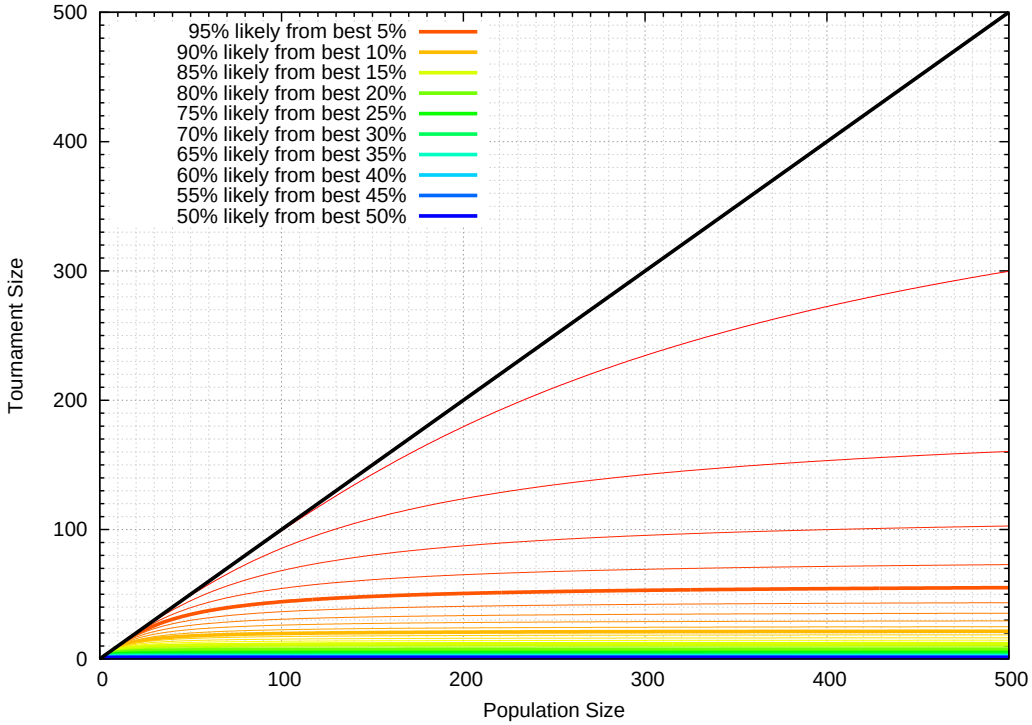


(b) Selection pressure contours for with-replacement tournaments in population sizes up to 500

Figure 81: This figure shows contours of constant selection pressure exerted by tournament selection with replacement over varying population and tournament sizes. Each thin line represents a change of 1% in both percentages from the neighbouring lines. The lines are generated using a continuous extension of a discrete formula and so, strictly speaking, should be interpreted as described in the main text. The thick black line indicates where the tournament size is equal to the population size.



(a) Selection pressure contours for without-replacement tournaments in population sizes up to 50



(b) Selection pressure contours for without-replacement tournaments in population sizes up to 500

Figure 82: This figure shows contours of constant selection pressure exerted by tournament selection without replacement over varying population and tournament sizes. Each thin line represents a change of 1% in both percentages from the neighbouring lines. The lines are generated using a continuous extension of a discrete formula and so, strictly speaking, should be interpreted as described in the main text. The thick black line indicates where the tournament size is equal to the population size.

Pop	Tournament size (as percentage of population size)												
	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	60%	70%	80%	90%
3							51.087	54.117	56.863	59.384	63.917	67.983	71.775	75.474
4					50.000	53.847	57.172	60.107	62.741	65.139	69.412	73.211	76.748	80.231
5				50.000	54.530	58.296	61.519	64.340	66.853	69.128	73.150	76.701	80.000	83.272
6				53.596	58.052	61.721	64.836	67.545	69.944	72.104	75.901	79.233	82.320	85.399
7			50.940	56.535	60.903	64.472	67.482	70.085	72.379	74.436	78.034	81.173	84.075	86.979
8			53.479	59.002	63.278	66.748	69.659	72.164	74.363	76.328	79.748	82.720	85.459	88.206
9			55.672	61.116	65.300	68.675	71.493	73.907	76.020	77.902	81.165	83.987	86.583	89.190
10		50.000	57.596	62.956	67.049	70.334	73.065	75.397	77.431	79.237	82.360	85.050	87.519	90.000
11		51.768	59.303	64.579	68.584	71.783	74.433	76.689	78.651	80.389	83.385	85.957	88.312	90.680
12		53.366	60.834	66.025	69.944	73.063	75.638	77.823	79.719	81.395	84.276	86.742	88.995	91.260
13		54.821	62.216	67.324	71.162	74.205	76.709	78.829	80.664	82.284	85.060	87.430	89.590	91.762
14		56.154	63.475	68.500	72.261	75.231	77.670	79.729	81.508	83.075	85.755	88.038	90.114	92.202
15		57.383	64.627	69.572	73.258	76.161	78.537	80.540	82.267	83.786	86.378	88.581	90.581	92.590
16		58.520	65.687	70.554	74.169	77.008	79.326	81.276	82.955	84.429	86.940	89.069	90.998	92.936
17		59.577	66.667	71.459	75.006	77.783	80.047	81.947	83.581	85.014	87.449	89.510	91.375	93.247
18		60.564	67.576	72.296	75.777	78.497	80.709	82.563	84.155	85.548	87.914	89.912	91.717	93.528
19		61.487	68.424	73.073	76.491	79.156	81.320	83.130	84.682	86.039	88.340	90.279	92.029	93.783
20	50.000	62.355	69.216	73.796	77.156	79.768	81.885	83.655	85.169	86.492	88.732	90.616	92.315	94.016
21	50.876	63.171	69.959	74.473	77.775	80.338	82.411	84.141	85.621	86.912	89.094	90.928	92.578	94.230
22	51.708	63.942	70.657	75.107	78.354	80.870	82.901	84.595	86.041	87.302	89.430	91.216	92.821	94.427
23	52.501	64.672	71.315	75.703	78.897	81.368	83.360	85.018	86.433	87.665	89.743	91.484	93.047	94.609
24	53.258	65.363	71.937	76.265	79.408	81.835	83.790	85.415	86.800	88.005	90.035	91.733	93.257	94.779
25	53.982	66.020	72.525	76.795	79.890	82.275	84.194	85.787	87.144	88.324	90.308	91.967	93.453	94.936
30	57.179	68.875	75.057	79.063	81.940	84.141	85.902	87.357	88.591	89.660	91.451	92.939	94.266	95.586
40	62.073	73.096	78.732	82.312	84.848	86.769	88.292	89.542	90.596	91.504	93.014	94.259	95.362	96.454
50	65.706	76.117	81.309	84.561	86.841	88.555	89.907	91.010	91.936	92.731	94.047	95.125	96.076	97.013
60	68.547	78.413	83.239	86.228	88.308	89.863	91.082	92.074	92.904	93.615	94.787	95.743	96.582	97.407
70	70.850	80.233	84.750	87.523	89.441	90.868	91.983	92.888	93.642	94.287	95.346	96.208	96.962	97.701
80	72.767	81.719	85.972	88.564	90.347	91.669	92.699	93.532	94.225	94.817	95.786	96.573	97.259	97.930
90	74.393	82.960	86.985	89.422	91.091	92.324	93.283	94.057	94.700	95.247	96.143	96.868	97.499	98.114
100	75.797	84.017	87.841	90.143	91.715	92.872	93.770	94.494	95.094	95.604	96.438	97.111	97.696	98.265
110	77.023	84.929	88.575	90.759	92.246	93.338	94.184	94.864	95.428	95.906	96.687	97.316	97.862	98.392
120	78.106	85.726	89.213	91.293	92.705	93.740	94.540	95.183	95.714	96.165	96.900	97.491	98.003	98.500
130	79.071	86.429	89.773	91.761	93.106	94.091	94.850	95.460	95.963	96.390	97.085	97.643	98.126	98.594
140	79.939	87.056	90.271	92.175	93.460	94.400	95.123	95.703	96.182	96.588	97.247	97.776	98.233	98.675
150	80.723	87.619	90.715	92.544	93.776	94.674	95.366	95.919	96.376	96.763	97.390	97.893	98.327	98.747
160	81.437	88.127	91.115	92.875	94.058	94.920	95.582	96.112	96.549	96.919	97.518	97.997	98.411	98.811
170	82.091	88.589	91.478	93.174	94.313	95.141	95.778	96.286	96.705	97.059	97.632	98.091	98.486	98.868
180	82.691	89.011	91.808	93.446	94.545	95.342	95.955	96.443	96.846	97.185	97.736	98.175	98.554	98.919
190	83.245	89.398	92.110	93.695	94.756	95.525	96.116	96.586	96.974	97.301	97.830	98.252	98.616	98.966
200	83.758	89.755	92.387	93.923	94.949	95.693	96.263	96.717	97.091	97.406	97.916	98.322	98.672	99.008
210	84.235	90.085	92.644	94.133	95.127	95.847	96.398	96.837	97.198	97.502	97.994	98.386	98.723	99.047
220	84.680	90.391	92.881	94.328	95.292	95.990	96.523	96.948	97.297	97.591	98.066	98.445	98.770	99.082
230	85.096	90.676	93.101	94.508	95.444	96.121	96.639	97.051	97.389	97.673	98.133	98.499	98.813	99.115
240	85.487	90.943	93.307	94.676	95.586	96.244	96.746	97.146	97.474	97.750	98.195	98.550	98.854	99.146
250	85.853	91.192	93.499	94.832	95.718	96.358	96.846	97.234	97.553	97.821	98.253	98.597	98.891	99.174
300	87.404	92.236	94.298	95.483	96.266	96.830	97.259	97.599	97.878	98.112	98.490	98.789	99.044	99.289
400	89.569	93.666	95.382	96.359	97.001	97.461	97.810	98.085	98.310	98.499	98.802	99.042	99.246	99.440
500	91.029	94.609	96.091	96.929	97.477	97.868	98.163	98.397	98.587	98.746	99.001	99.202	99.373	99.536
600	92.089	95.285	96.595	97.332	97.812	98.154	98.412	98.615	98.780	98.919	99.140	99.314	99.462	99.602
700	92.900	95.796	96.974	97.633	98.062	98.367	98.596	98.777	98.924	99.047	99.243	99.396	99.527	99.650
800	93.542	96.197	97.270	97.869	98.257	98.532	98.740	98.903	99.035	99.146	99.322	99.460	99.577	99.688
900	94.066	96.521	97.508	98.058	98.413	98.665	98.855	99.003	99.124	99.225	99.385	99.511	99.617	99.718
1000	94.503	96.790	97.705	98.213	98.542	98.774	98.949	99.086	99.197	99.289	99.437	99.552	99.650	99.742
1100	94.872	97.016	97.871	98.344	98.650	98.866	99.028	99.155	99.258	99.344	99.480	99.587	99.677	99.762
1200	95.190	97.210	98.012	98.455	98.741	98.943	99.095	99.213	99.309	99.389	99.517	99.616	99.700	99.779
1300	95.467	97.378	98.134	98.551	98.820	99.010	99.152	99.264	99.354	99.429	99.548	99.641	99.720	99.794
1400	95.710	97.524	98.241	98.635	98.889	99.068	99.203	99.308	99.392	99.463	99.575	99.663	99.737	99.807
1500	95.926	97.654	98.335	98.709	98.950	99.120	99.247	99.346	99.426	99.493	99.599	99.682	99.752	99.818
1600	96.119	97.770	98.418	98.775	99.004	99.165	99.286	99.380	99.457	99.520	99.621	99.699	99.765	99.828
1700	96.293	97.874	98.493	98.834	99.052	99.206	99.321	99.411	99.483	99.544	99.640	99.714	99.777	99.836
1800	96.450	97.967	98.561	98.887	99.096	99.243	99.352	99.438	99.508	99.565	99.657	99.728	99.788	99.844
1900	96.593	98.052	98.622	98.935	99.135	99.276	99.381	99.463	99.530	99.585	99.672	99.740	99.798	99.851
2000	96.723	98.130	98.678	98.978	99.171	99.306	99.407	99.486	99.549	99.602	99.686	99.751	99.806	99.858
2100	96.843	98.201	98.729	99.018	99.204	99.333	99.431	99.506	99.568	99.618	99.699	99.762	99.814	99.864
2200	96.954	98.266	98.776	99.055	99.234	99.359	99.452	99.525	99.584	99.633	99.711	99.771	99.822	99.869
2300	97.056	98.327	98.820	99.089	99.261	99.382	99.472	99.543	99.600	99.647	99.721	99.779	99.828	99.874
2400	97.151	98.383	98.860	99.120	99.287	99.404	99.491	99.559	99.614	99.659	99.731	99.787	99.834	99.879
2500	97.240	98.435	98.897	99.149	99.311	99.424	99.508	99.574	99.627	99.671	99.741	99.795	99.840	99.883
3000	97.605	98.648	99.050	99.268	99.408	99.505	99.578	99.635	99.680	99.718	99.778	99.825	99.864	99.900
4000	98.090	98.929	99.250	99.424	99.535	99.612	99.669	99.714	99.750	99.780	99.827	99.863	99.894	99.922
5000	98.401	99.108	99.377	99.522	99.614	99.679	99.726	99.764	99.793	99.818	99.857	99.887	99.913	99.936
6000	98.618	99.233	99.465	99.590	99.670	99.725	99.766	99.798	99.823	99.845	99.878	99.904	99.925	99.946
7000	98.780	99.325	99.530	99.640	99.710	99.759	99.795	99.823	99.845	99.864	99.893	99.916	99.935	99.953
8000	98.906	99.396	99.580	99.679	99.741	99.785	99.817	99.842	99.862	99.879	99.905	99.925	99.942	99.958
9000	99.006	99.452	99.620	99.710	99.766	99.806	99.835	99.858	99.876	99.891	99.914	99.933	99.948	99.962
10000	99.088	99.499	99.652	99.735	99.786	99.823	99.849	99.870	99.887	99.900	99.922	99.939	99.952	99.965
11000	99.157	99.537	99.679	99.755	99.803	99.837	99.861	99.880	99.896	99.908	99.928	99.944	99.956	99.968
12000	99.216	99.570	99.702	99.773	99.818	99.848	99.871	99.889	99.903	99.915	99.933	99.948	99.960	99.971
13000	99.266	99.598	99.722	99.788	99.830	99.859	99.880	99.897	99.910	99.921	99.938	99.951	99.962	99.973
14000	99.310	99.622	99.739	99.801	99.840	99.867	99.887	99.903	99.915	99.926	99.942	99.954	99.965	99.974
15000	99.348	99.644	99.754	99.813	99.850	99.875	99.894	99.909	99.920	99.930	99.945	99.957	99.967	99.976
16000	99.382	99.663	99.767	99.823	99.858	99.882	99.900	99.914	99.925	99.934	99.948	99.959	99.969	99.977
17000	99.413	99.680	99.779	99.832	99.865	99.888	99.905	99.918	99.929	99.937	99.951	99.962	99.970	99.978
18000	99.440	99.695	99.790	99.840	99.872	99.893	99.910	99.922	99.932	99.940	99.953	99.963	99.972	99.980
19000	99.465	99.709	99.799	99.847	99.877	99.898	99.914	99.926	99.935	99.943	99.956	99.965	99.973	99.980
21000	99.508	99.733	99.816	99.860	99.888	99.907	99.921	99.932	99.941	99.948	99.959	99.968	99.975	99.982
22000	99.527	99.743	99.823	99.865	99.892	99.911	99.924	99.935	99.943	99.950	99.961	99.969	99.976	99.983
23000	99.544	99.753	99.830	99.871	99.896	99.914	99.927	99.937	99.945	99.952	99.963	99.971	99.977	99.984
24000	99.560	99.761	99.836	99.875	99.900	99.917	99.930	99.940	99.947	99.954	99.964	99.972	99.978	99.984
25000	99.575	99.770	99.841	99.880	99.904	99.920	99.932	99.942	99.949	99.955	99.965	99.973	99.979	99.985

Table 39: The many-from-few selection pressure (as a percentage) exerted by without-replacement tournament selection for various tournament sizes and populations sizes.

Many-from-few selection pressure													
55%	60%	65%	70%	75%	80%	85%	90%	0.95%	96%	97%	98%	99%
1.336	1.794	2.437	3.376	4.819	7.213	11.673	21.854	58.404	78.852	115.123	193.639	458.211

Table 40: The tournament size required to achieve various selection pressures using with-replacement tournament selection.

Pop	Many-from-few selection pressure												
	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	0.96	0.97	0.98	0.99
3	1.247	1.539	1.877	2.258	2.662	3.000	3.000	3.000	3.000	3.000	3.000	3.000	3.000
4	1.266	1.592	1.988	2.459	3.000	3.574	4.000	4.000	4.000	4.000	4.000	4.000	4.000
5	1.279	1.628	2.063	2.602	3.254	4.000	4.750	5.000	5.000	5.000	5.000	5.000	5.000
6	1.288	1.653	2.117	2.707	3.449	4.346	5.324	6.000	6.000	6.000	6.000	6.000	6.000
7	1.294	1.671	2.158	2.788	3.602	4.630	5.826	6.935	7.000	7.000	7.000	7.000	7.000
8	1.299	1.685	2.189	2.851	3.725	4.864	6.264	7.670	8.000	8.000	8.000	8.000	8.000
9	1.303	1.697	2.214	2.902	3.826	5.061	6.646	8.359	9.000	9.000	9.000	9.000	9.000
10	1.306	1.706	2.235	2.944	3.910	5.228	6.980	9.000	10.000	10.000	10.000	10.000	10.000
11	1.309	1.713	2.252	2.980	3.981	5.371	7.275	9.594	11.000	11.000	11.000	11.000	11.000
12	1.311	1.720	2.266	3.010	4.041	5.496	7.537	10.143	12.000	12.000	12.000	12.000	12.000
13	1.313	1.725	2.278	3.035	4.094	5.605	7.770	10.651	13.000	13.000	13.000	13.000	13.000
14	1.314	1.730	2.289	3.058	4.140	5.701	7.978	11.121	14.000	14.000	14.000	14.000	14.000
15	1.316	1.734	2.299	3.077	4.180	5.786	8.166	11.557	14.890	15.000	15.000	15.000	15.000
16	1.317	1.738	2.307	3.095	4.216	5.862	8.337	11.961	15.749	16.000	16.000	16.000	16.000
17	1.318	1.741	2.314	3.110	4.248	5.931	8.491	12.336	16.590	17.000	17.000	17.000	17.000
18	1.319	1.744	2.321	3.124	4.277	5.993	8.633	12.685	17.414	17.988	18.000	18.000	18.000
19	1.320	1.746	2.326	3.136	4.303	6.049	8.762	13.011	18.217	18.885	19.000	19.000	19.000
20	1.321	1.749	2.332	3.148	4.327	6.100	8.881	13.315	19.000	19.771	20.000	20.000	20.000
21	1.321	1.751	2.337	3.158	4.349	6.147	8.991	13.600	19.762	20.644	21.000	21.000	21.000
22	1.322	1.753	2.341	3.167	4.369	6.191	9.092	13.867	20.503	21.505	22.000	22.000	22.000
23	1.323	1.754	2.345	3.176	4.387	6.230	9.186	14.118	21.223	22.351	23.000	23.000	23.000
24	1.323	1.756	2.349	3.184	4.404	6.267	9.274	14.354	21.922	23.183	24.000	24.000	24.000
25	1.324	1.757	2.352	3.191	4.419	6.302	9.356	14.577	22.601	24.000	24.938	25.000	25.000
30	1.326	1.763	2.366	3.221	4.482	6.441	9.693	15.520	25.700	27.846	29.450	30.000	30.000
40	1.328	1.771	2.383	3.258	4.563	6.622	10.140	16.830	30.654	34.378	37.744	39.832	40.000
50	1.330	1.775	2.394	3.281	4.612	6.735	10.423	17.694	34.377	39.588	44.936	49.000	50.000
60	1.331	1.778	2.401	3.297	4.646	6.811	10.618	18.306	37.251	43.779	51.093	57.593	60.000
70	1.331	1.781	2.406	3.308	4.670	6.866	10.760	18.761	39.526	47.200	56.363	65.522	70.000
80	1.332	1.782	2.410	3.316	4.688	6.908	10.869	19.113	41.367	50.033	60.893	72.779	79.884
90	1.332	1.783	2.413	3.323	4.703	6.941	10.954	19.393	42.885	52.414	64.813	79.395	89.522
100	1.333	1.785	2.415	3.328	4.714	6.968	11.023	19.622	44.158	54.438	68.229	85.420	99.000
110	1.333	1.785	2.417	3.332	4.723	6.989	11.080	19.811	45.239	56.179	71.226	90.909	108.274
120	1.333	1.786	2.419	3.336	4.731	7.008	11.128	19.971	46.169	57.692	73.875	95.918	117.310
130	1.333	1.787	2.420	3.339	4.738	7.023	11.169	20.108	46.977	59.018	76.230	100.497	126.084
140	1.333	1.787	2.421	3.341	4.744	7.036	11.204	20.226	47.686	60.189	78.336	104.693	134.581
150	1.334	1.788	2.422	3.344	4.749	7.048	11.234	20.329	48.312	61.230	80.230	108.548	142.792
160	1.334	1.788	2.423	3.346	4.753	7.058	11.261	20.420	48.869	62.163	81.942	112.098	150.712
170	1.334	1.788	2.424	3.347	4.757	7.067	11.285	20.501	49.368	63.002	83.495	115.376	158.343
180	1.334	1.789	2.425	3.349	4.760	7.075	11.306	20.573	49.818	63.761	84.912	118.411	165.688
190	1.334	1.789	2.426	3.350	4.763	7.082	11.325	20.638	50.225	64.451	86.208	121.226	172.752
200	1.334	1.789	2.426	3.352	4.766	7.089	11.342	20.697	50.595	65.081	87.399	123.845	179.544
210	1.334	1.789	2.427	3.353	4.768	7.094	11.357	20.750	50.934	65.659	88.497	126.286	186.072
220	1.334	1.790	2.427	3.354	4.771	7.100	11.371	20.798	51.244	66.190	89.512	128.566	192.346
230	1.334	1.790	2.428	3.355	4.773	7.105	11.384	20.843	51.530	66.680	90.452	130.700	198.375
240	1.334	1.790	2.428	3.356	4.775	7.109	11.396	20.884	51.793	67.134	91.327	132.701	204.170
250	1.334	1.790	2.428	3.356	4.776	7.113	11.407	20.921	52.038	67.556	92.142	134.581	209.741
300	1.335	1.791	2.430	3.360	4.783	7.130	11.451	21.073	53.031	69.278	95.506	142.495	234.577
400	1.335	1.791	2.432	3.364	4.792	7.150	11.506	21.265	54.309	71.516	99.954	153.345	272.479
500	1.335	1.792	2.433	3.366	4.798	7.163	11.539	21.381	55.096	72.907	102.760	160.415	299.738
600	1.335	1.792	2.433	3.368	4.801	7.171	11.561	21.459	55.630	73.854	104.691	165.381	320.161
700	1.335	1.792	2.434	3.369	4.804	7.177	11.577	21.515	56.015	74.542	106.101	169.059	335.987
800	1.335	1.793	2.434	3.370	4.806	7.181	11.589	21.557	56.306	75.063	107.175	171.891	348.591
900	1.335	1.793	2.435	3.370	4.807	7.185	11.598	21.590	56.534	75.471	108.021	174.138	358.858
1000	1.335	1.793	2.435	3.371	4.808	7.188	11.606	21.616	56.717	75.800	108.704	175.965	367.378
1100	1.335	1.793	2.435	3.371	4.809	7.190	11.612	21.637	56.868	76.071	109.268	177.480	374.559
1200	1.335	1.793	2.435	3.372	4.810	7.192	11.617	21.655	56.994	76.298	109.740	178.755	380.693
1300	1.335	1.793	2.435	3.372	4.811	7.193	11.621	21.671	57.101	76.490	110.142	179.844	385.991
1400	1.335	1.793	2.435	3.372	4.811	7.195	11.625	21.684	57.193	76.656	110.488	180.785	390.614
1500	1.335	1.793	2.436	3.372	4.812	7.196	11.628	21.695	57.272	76.800	110.789	181.605	394.682
1600	1.335	1.793	2.436	3.373	4.812	7.197	11.631	21.705	57.342	76.926	111.053	182.327	398.290
1700	1.335	1.793	2.436	3.373	4.813	7.198	11.634	21.714	57.404	77.037	111.287	182.968	401.510
1800	1.335	1.793	2.436	3.373	4.813	7.199	11.636	21.721	57.459	77.137	111.496	183.540	404.403
1900	1.335	1.793	2.436	3.373	4.813	7.199	11.638	21.728	57.508	77.226	111.683	184.053	407.015
2000	1.336	1.793	2.436	3.373	4.814	7.200	11.639	21.735	57.552	77.306	111.852	184.517	409.386
2100	1.336	1.793	2.436	3.373	4.814	7.201	11.641	21.740	57.593	77.378	112.005	184.938	411.547
2200	1.336	1.793	2.436	3.373	4.814	7.201	11.643	21.745	57.629	77.445	112.144	185.322	413.525
2300	1.336	1.793	2.436	3.373	4.814	7.202	11.644	21.750	57.663	77.505	112.271	185.674	415.343
2400	1.336	1.793	2.436	3.374	4.814	7.202	11.645	21.755	57.693	77.561	112.388	185.997	417.018
2500	1.336	1.793	2.436	3.374	4.815	7.203	11.646	21.759	57.721	77.612	112.496	186.295	418.568
3000	1.336	1.793	2.436	3.374	4.815	7.204	11.651	21.774	57.834	77.816	112.928	187.492	424.845
4000	1.336	1.794	2.436	3.374	4.816	7.206	11.656	21.794	57.976	78.073	113.472	189.004	432.872
5000	1.336	1.794	2.437	3.375	4.817	7.208	11.660	21.806	58.061	78.228	113.799	189.919	437.786
6000	1.336	1.794	2.437	3.375	4.817	7.208	11.662	21.814	58.118	78.332	114.019	190.532	441.104
7000	1.336	1.794	2.437	3.375	4.817	7.209	11.664	21.820	58.159	78.406	114.175	190.972	443.495
8000	1.336	1.794	2.437	3.375	4.818	7.209	11.665	21.824	58.190	78.461	114.293	191.302	445.300
9000	1.336	1.794	2.437	3.375	4.818	7.210	11.666	21.828	58.213	78.504	114.385	191.560	446.710
10000	1.336	1.794	2.437	3.375	4.818	7.210	11.666	21.830	58.232	78.539	114.459	191.767	447.842
11000	1.336	1.794	2.437	3.375	4.818	7.210	11.667	21.833	58.248	78.567	114.519	191.936	448.772
12000	1.336	1.794	2.437	3.375	4.818	7.210	11.668	21.834	58.261	78.591	114.569	192.077	449.548
13000	1.336	1.794	2.437	3.375	4.818	7.211	11.668	21.836	58.272	78.611	114.612	192.196	450.207
14000	1.336	1.794	2.437	3.375	4.818	7.211	11.668	21.837	58.281	78.628	114.648	192.299	450.772
15000	1.336	1.794	2.437	3.375	4.818	7.211	11.669	21.838	58.289	78.643	114.680	192.388	451.263
16000	1.336	1.794	2.437	3.375	4.818	7.211	11.669	21.839	58.297	78.656	114.707	192.466	451.693
17000	1.336	1.794	2.437	3.375	4.818	7.211	11.669	21.840	58.303	78.668	114.732	192.534	452.073
18000	1.336	1.794	2.437	3.375	4.818	7.211	11.669	21.841	58.309	78.678	114.753	192.596	452.411
19000	1.336	1.794	2.437	3.375	4.818	7.211	11.670	21.842	58.314	78.687	114.773	192.650	452.714
21000	1.336	1.794	2.437	3.375	4.818	7.211	11.670	21.843	58.322	78.703	114.806	192.744	453.234
22000	1.336	1.794	2.437	3.375	4.818	7.211	11.670	21.843	58.326	78.709	114.820	192.785	453.458
23000	1.336	1.794	2.437	3.375	4.818	7.211	11.670	21.844	58.329	78.715	114.834	192.822	453.664
24000	1.336	1.794	2.437	3.375	4.818	7.212	11.670	21.844	58.332	78.721	114.846	192.856	453.852
25000	1.336	1.794	2.437	3.375	4.818	7.212	11.671	21.845	58.335	78.726	114.857	192.887	454.025

Table 41: The tournament size required to achieve various selection pressures for various population sizes using without-replacement tournament selection.

index $i >= N * y/100$, the cumulative probability for selection within the best i members of the population is greater than or equal to $x/100$.

Figures 81 and 82 are tremendously useful reference resources for EC practitioners that use tournament selection. Section 7.2.4 proposed the hypothesis that neither keeping the tournament size constant nor keeping the tournament size ratio constant would guarantee constant selection pressure. This graph shows that hypothesis to be true and gives a good deal more information on these relationships.

7.3 Enhancing the Individual Copying Strategy

7.3.1 The Problem of Updating the Individuals

Profiling also revealed that one of the CPU's most computationally expensive tasks was building the new generation. This task comes after the individuals have been evaluated, the fitnesses have been calculated, the selections have been made and all that remains is to construct the new generation. Each new creation consists of a copy of a selected individual, a possible crossover drawing material from another individual and a possible mutation. The order in which the new creations appear does not matter.

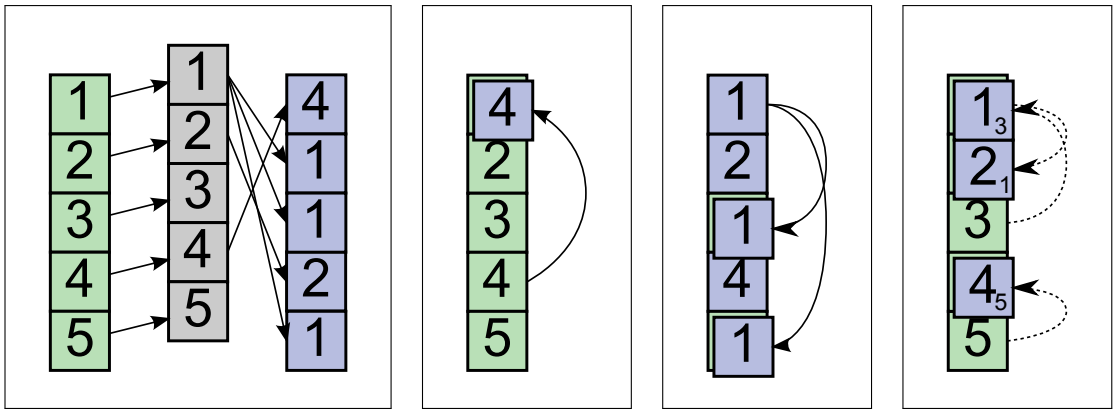
The natural approach is to take a complete copy of the population, and then build the new individuals in-place in the original array using the copy of the population for sources as shown in Figure 83(a). This is simple to implement and was adopted in the original code. However profiling revealed that this approach was time consuming and that the copying of individuals accounted for most of this. This is perhaps unsurprising since much of this work uses no crossover and the mutation operators often make a small change. The relevant copy function was optimised but the problem remained.

If optimisation could not reduce the time per copy much further, perhaps it could tackle the number of copies. For a population of N individuals, the naive technique requires $2N$ copies: one for each of the N individuals in the copy of the population and another for each of the N new individuals constructed in-place.

Is it possible to use fewer copies, perhaps by avoiding the copy to a temporary population? First, is it possible if crossover is not used? If no temporary population is available then the array of individuals must be updated in-place. The difficulty with this is that unless care is taken, individuals that are still needed, get overwritten. For example say a population of five is to be updated based on the selections 4, 1, 1, 2 and 1. If this order is used as in Figure 83(b), the first step overwrites the first individual with the fourth and so ruins the attempt because the first individual is now unavailable as a source for later steps. At the time each new individual is created, its source individual must still be intact.

It is possible to do better by using the freedom to reorder the steps and targets? Figure 83(c) shows that the previous problem may be solved by copying the first individual to the third slot and mutating, copying the first individual to the fifth slot and mutating and then mutating the remaining individuals. The second version of the code implemented a simple algorithm: "for each selection of an individual after the first one, perform one copy onto an individual that has not been selected; mutate all".

This algorithm tackles any possible list of selections. How many copies does it require? Observe that there must be at least one copy for each of the old individuals that are not selected at all since other individuals must be copied into each of those individuals' slots. This improved algorithm performs exactly that minimum possible number of copies. If each old individual is selected once, it performs no copies. Unfortunately,



(a) Attempt using a temporary copy of the population (the blue squares should be on top of the green squares)

(b) Naive attempt at building in-place

(c) Good attempt at building in-place

(d) Failed attempt to build in-place with crossover

Figure 83: Four attempts at building a new population, the last of which involves crossover. Green squares represent individuals in the previous generation, blue squares represent new individuals that have been mutated and grey squares represent temporary copies. A square's number indicates the location in the previous generation's population of the individual (or its primary parent). A subscript number indicates a secondary parent for crossover. The first three subfigures illustrate attempts to build a population based on the selections 4, 1, 1, 2 and 1 (in any order). Subfigure 83(a) shows an attempt which performs five copies of the originals to a temporary store and then uses them as sources to build the new individuals in the original locations with another five copies. Subfigure 83(b) attempts to reduce copies by constructing in-place. This goes awry because the first step overwrites individual 1 with 4 but 1 is still required for future steps. Subfigure 83(c) reorders and succeeds using only two copies. Subfigure 83(d) attempts to handle crossover and build the recipes $4 \to 5$, $1 \to 3$, $1 \to 5$, $2 \to 1$ and $1 \to 3$. Recipes $4 \to 5$ and $2 \to 1$ are built without problem but then any third step fails, e.g. the $1 \to 3$ here destroys individual 1 for remaining steps. Note that $1 \to 5$ cannot be built in slot 5 because the primary parent (1) must be copied on top of the destination (5) before the crossover, which then requires the secondary parent (5).

the algorithm has one major problem: it does not deal with crossover.

Note that although many of the runs in this research did not use crossover, it was deemed important that this optimisation should not prevent it because preventing crossover would be a severe restriction to GP. For this reason, the crossover was included as part of the problem. Unfortunately, crossover makes the problem more difficult because it means new individuals now specify two parents that must still be intact at the time of creation. Each new creation is now specified by a recipe of the form: "build an individual from the old i^{th} individual, with a possible drawing of crossover material from the old j^{th} individual, and then mutate". Such recipes will henceforth be written $i \to j$.

Temporary copies may be unavoidable once crossover is included: given the recipes $1 \to 2$, $2 \to 3$ and $3 \to 1$, each individual appears in two recipes so building any of the recipes in any of the array slots will ruin the source for at least one future recipe. For

this reason, a scratch area may be used.

Since the crossover used in this work is typically computationally cheaper than a copy and is often not used at all, it is assumed that the crossover must work with a slot containing (a copy of) the first parent rather than the second. Figure 83(d) shows an attempt with crossover going awry for a version of the previously solved problem that has been extended to use the recipes $4 \to 5$, $1 \to 3$, $1 \to 5$, $2 \to 1$ and $1 \to 3$. In addition to a slot being unusable if it appears in any remaining recipes, a slot is unusable for building a recipe in which it appears as the second part because the first stage of the recipe will overwrite the individual that is required as a source for the second part.

The problem's complexity now obstructs clear thinking. For instance, consider a list of recipes that is tackled in Section 7.3.3 below: $2 \to 27$, $3 \to 12$, $3 \to 16$, $3 \to 17$, $3 \to 23$, $3 \to 24$, $3 \to 26$, $4 \to 3$, $4 \to 22$, $4 \to 25$, $6 \to 15$, $11 \to 4$, $11 \to 12$, $11 \to 15$, $11 \to 16$, $11 \to 26$, $12 \to 15$, $12 \to 28$, $13 \to 8$, $14 \to 4$, $14 \to 13$, $14 \to 22$, $16 \to 3$, $16 \to 21$, $16 \to 25$, $16 \to 27$, $21 \to 20$, $24 \to 22$, $25 \to 4$ and $30 \to 6$. How can one think about how to find a good solution for these recipes?

7.3.2 Picturing the Problem

To think clearly about this problem, it was important to find a good way to depict it. Figure 84(a) shows an example of the chosen representation. Orange nodes represent individuals in the previous generation; arrows represent recipes for new individuals such that an arrow from node a to node b represents the recipe $a \to b$. Table 42 shows how the constraints of the problem may now be translated to rules of a game on the diagram.

Constraints of the problem	Rules of the game
The task is to form a list of steps that build a list of recipes. Each step builds a new individual in one of the array's slots, after which the recipe is done and the slot is unavailable for use.	The task is to form a list of moves that eradicate all nodes and arrows. Each move involves dragging an arrow onto a node, after which the arrow and the node disappear.
An individual in the array may not be used as a destination if it is still mentioned in any future recipes.	An arrow may not be dragged onto a node if that node has any remaining arrows touching it (other than the base of the dragged arrow).
Copying an individual i to a new slot j costs one copy whereas building an individual in the same slot as its first parent requires no copies.	Dragging an arrow onto a new node costs one point whereas sucking an arrow into its own base node is free.
When desperate, it may be necessary to spend one copy to make a temporary copy of one of the old individuals that can be used as the source of future copies.	When desperate, it may be necessary to spend one point to make a specific node immediately removable. An arrow can then be dragged onto it, even if it still has other arrows attached.
The aim is to use as few copies as possible.	The aim is to use as few points as possible.

Table 42: Translating between constraints of the problem and equivalent rules of a game to be played on diagrams like that in Figure 84(a).

Figures 84(b)-84(f) show one possible way to play this game for the problem de-

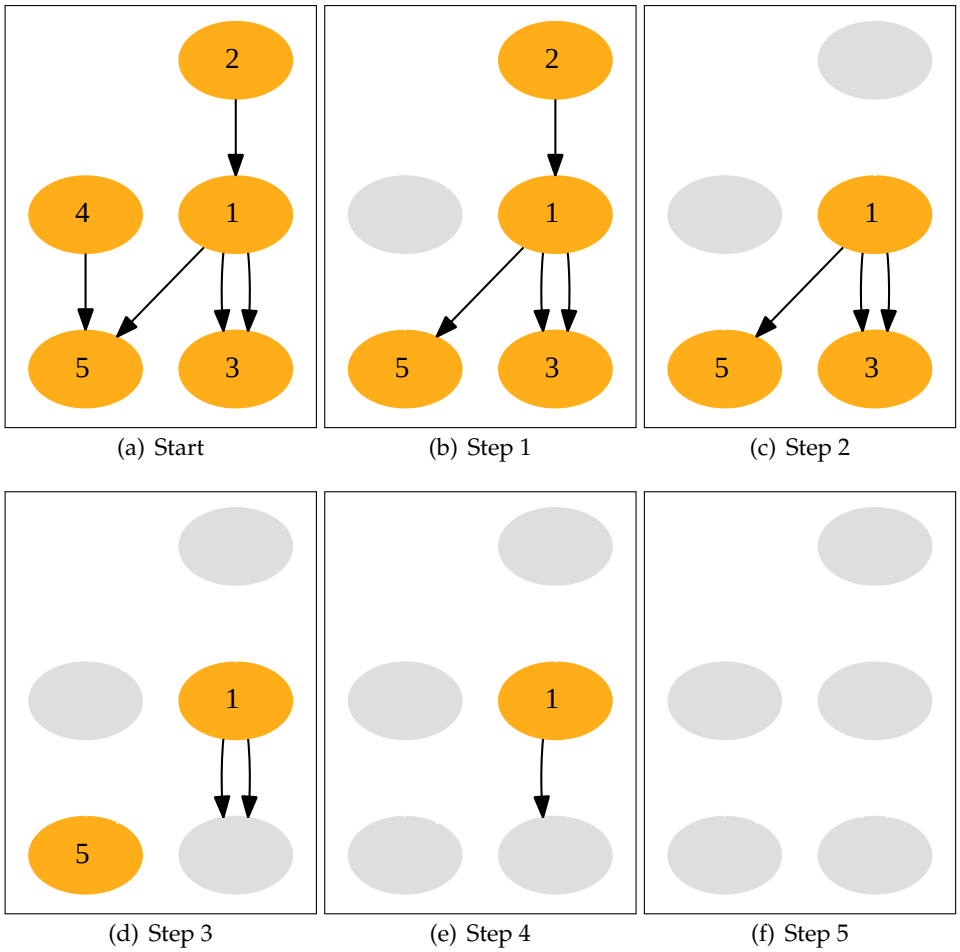


Figure 84: Orange nodes represent individuals in the previous generation; arrows represent recipes for new individuals such that an arrow from node a to node b represents the recipe $a \to b$. 84(a) The recipes for the problem are: $4 \to 5$, $1 \to 3$, $1 \to 5$, $2 \to 1$, $1 \to 3$. 84(b) Step 1: the $4 \to 5$ arrow is sucked into the 4. 84(c) Step 2: the $2 \to 1$ arrow is sucked into the 2. 84(d) Step 3: the 3 is copied to a temporary and the $1 \to 5$ arrow is dragged onto the 3. 84(e) Step 4: one $1 \to 3$ arrow is dragged onto the 5. 84(f) Step 5: the other $1 \to 3$ arrow is sucked into the 1.

scribed earlier and attempted in Figure 83(d). The first step sucks the $4 \to 5$ arrow into node 4, which costs no points. The second step sucks the $2 \to 1$ arrow into node 2, which costs no points. At this juncture, no more normal moves can be played so one point is spent making node 3 immediately removable and then another is spent dragging the $1 \to 5$ arrow onto node 3. The fourth move drags one of the $1 \to 3$ arrows onto node 5 at a cost of one point. The fifth move sucks the other $1 \to 3$ arrow into node 1. Three points are spent in total.

This representation is even more useful because it allows a solution to be shown on the same diagram as the problem. Figure 85 shows this for the same problem and solution depicted in Figures 84(a)-84(f). The number of copies required for a particular solution can be read off the figure as the number of solid arrows plus the number of brown nodes, in this case three.

Although not essential, the code was written such that the crossover source is always different from the copy source so the diagrams never involve arrows from a node to itself.

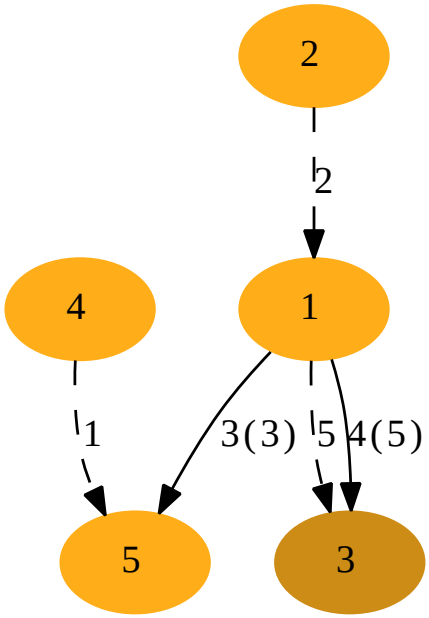


Figure 85: The problem and solution from Figure 84 compressed into one diagram. The details of the nodes and arrows are as for Figure 84 and the labels on the arrows describe the solution. The arrows are labelled with the order in which they are to be processed (i.e. by the number of the step in which they should be processed). If an arrow's label has a second number in brackets, the arrow is to be dragged onto the node with that number, otherwise it its to be sucked into the node at its base. Arrows that are to be sucked into their base are dashed; arrows that are to be dragged to other nodes are solid. Nodes that will need to be copied to a temporary are coloured brown rather than orange. In this case, there are two solid arrows and one brown node indicating three copies.

7.3.3 A Heuristic to Tackle the Problem

The aim was now to find a heuristic that quickly produces strategies that are usually fairly good. It is much easier to reason about this when using the diagrams as a guide.

Since sucking an arrow is free whereas dragging an arrow costs a point, the heuristic should aim to suck as many arrows as possible. The diagrams suggest that sucking an arrow into its base is an "easy" move that can free up other moves and can do little harm. This suggests sucking moves should always be done before trickier moves and that they should be done iteratively because sucking one arrow into its base node can free up another node to suck its arrow. Making a node immediately writable should only be used as a last resort. This leads to the following rules:

- Iteratively suck as many arrows into their base nodes as possible.

- When no more arrows can be sucked into their base nodes, drag arrows onto each of the nodes that are currently free or immediately writable:
  - Prefer dragging arrows that are not due to be sucked into their base nodes when fewer arrows point to them.
  - If an arrow can be sucked into its base node now, always do that rather than drag it elsewhere.
- Keep alternating between the previous two steps.
- When nothing else can be done (no self-writes, no free nodes, no immediately writable nodes) identify the node with the most arrows pointing to it and make it immediately writable.
- Keep following the above steps until the task is complete.

This leaves one difficult decision: how to prioritise which arrows to drag onto free and immediately writable nodes. This question was tackled with a combination of intuition and informal empirical investigation of different possibilities. The chosen tactic aims to free up nodes as fast as possible by targeting arrows pointing to nodes that are closest to being freed. More precisely the strategy selects the arrow that has the fewest arrows pointing to the node to which it points. If there is a tie, it selects the arrow that has the fewest arrows leaving the node to which it points. If there is further tie, it just selects one of the joint winners arbitrarily (but deterministically and hence reproducibly).

Figure 86 shows a solution generated by this heuristic for a problem of 30 recipes. The solution begins with steps $1-3$ sucking arrows into nodes. Note that step 3 sucks $6 \to 15$ into its base but this is only possible after $30 \to 6$ is sucked into its base in step 1.

Steps $4-15$ drag arrows onto free nodes, avoiding dragging any arrows that could be sucked into their base nodes. These steps prioritise arrows pointing to nodes that can be freed up with the least work. For instance, steps $4-7$ each free up a node by themselves. Steps 9, 12, 13 and 14 use nodes that were only freed up by earlier moves in the same group. For instance step 9 moves arrow $14 \to 13$ to node 17 which was only freed up in step 4.

By step 29, there are two remaining nodes, 16 and 3, and two remaining arrows, $16 \to 3$ and $3 \to 16$. This cannot be solved without making one node immediately writable. Node 3 is made immediately writable, step 29 sucks $3 \to 16$ into node 3 and step 30 sucks $16 \to 3$ into node 16.

In total, the solution uses 18 copies. 17 of the nodes have no arrows leaving them so this provides a lower bound for the best possible solution. Hence in this solution, making node 3 immediately writable was the only "wasted" copy. Moreover, 17 copies

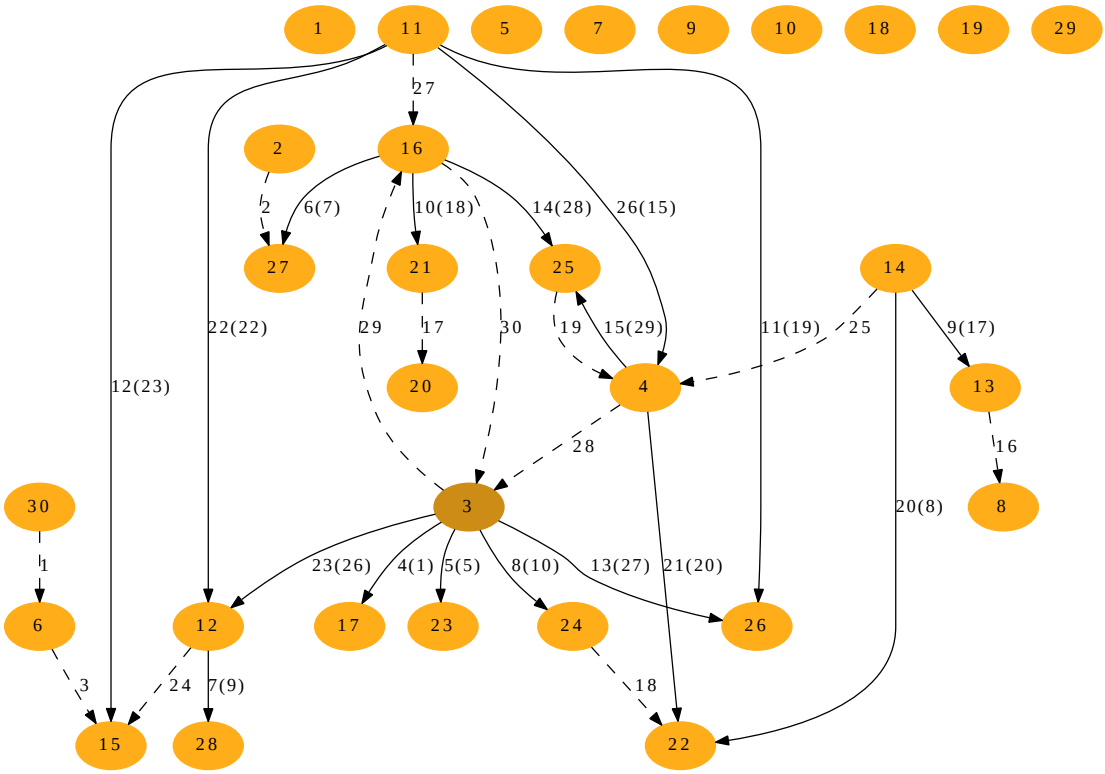


Figure 86: A solution to a more realistically sized problem than in Figure 85. A population of 30 is being updated with the recipes: $2 \to 27$, $3 \to 12$, $3 \to 16$, $3 \to 17$, $3 \to 23$, $3 \to 24$, $3 \to 26$, $4 \to 3$, $4 \to 22$, $4 \to 25$, $6 \to 15$, $11 \to 4$, $11 \to 12$, $11 \to 15$, $11 \to 16$, $11 \to 26$, $12 \to 15$, $12 \to 28$, $13 \to 8$, $14 \to 4$, $14 \to 13$, $14 \to 22$, $16 \to 3$, $16 \to 21$, $16 \to 25$, $16 \to 27$, $21 \to 20$, $24 \to 22$, $25 \to 4$ and $30 \to 6$.

is only a lower bound; perhaps it is impossible with fewer than 18 copies. Either way, 18 copies is a big improvement over the 60 required for the naive population-copying approach.

7.3.4 Assessment of the Heuristic

The aim of this work was to reduce the number of copies required to produce new generations. Profiling revealed that the mutation and crossover were not computationally expensive relative to the copy operation. The time required to perform these copies should be proportional to the number of copies although the average time per copy may vary significantly depending on the particulars of the copy operator. For these reasons, the heuristic was assessed based on the number of copy operations rather than the time taken. Of course, it is quite possible that a mutation operator could be written that was so slow, it would render these optimisations irrelevant.

Table 43 shows the results of 500 runs over varying population sizes. The data for the tests were generated using tournament selection because it is used throughout this work and is most widely used in GP [73]. The effectiveness of the algorithm may depend on the distribution of selections: few individuals selected many times each, or

Pop	Selection pressure							
	55%	60%	65%	70%	75%	80%	85%	90%	95%
10	3.494 [±0.046] 20.000 4.798 [±0.049]	4.336 [±0.044] 20.000 5.464 [±0.046]	4.276 [±0.044] 20.000 5.396 [±0.047]	5.280 [±0.037] 20.000 6.320 [±0.037]	5.962 [±0.036] 20.000 6.964 [±0.036]	6.594 [±0.033] 20.000 7.594 [±0.033]	7.532 [±0.027] 20.000 8.534 [±0.027]	8.362 [±0.021] 20.000 9.362 [±0.021]	9.000 [±0.000] 20.000 [±0.000] 10.000 [±0.000]
20	7.212 [±0.063] 40.000 8.774 [±0.068]	8.640 [±0.064] 40.000 9.892 [±0.066]	8.690 [±0.061] 40.000 9.938 [±0.064]	10.192 [±0.057] 40.000 11.290 [±0.058]	11.546 [±0.053] 40.000 12.604 [±0.053]	13.404 [±0.048] 40.000 14.416 [±0.048]	15.148 [±0.041] 40.000 16.150 [±0.041]	16.644 [±0.031] 40.000 17.646 [±0.031]	18.326 [±0.021] 40.000 [±0.000] 19.326 [±0.021]
30	10.930 [±0.076] 60.000 12.580 [±0.081]	12.946 [±0.074] 60.000 14.304 [±0.077]	13.006 [±0.075] 60.000 14.376 [±0.078]	15.168 [±0.076] 60.000 16.348 [±0.077]	17.130 [±0.064] 60.000 18.214 [±0.064]	19.730 [±0.061] 60.000 20.764 [±0.061]	22.840 [±0.048] 60.000 23.848 [±0.048]	25.394 [±0.038] 60.000 26.396 [±0.038]	27.646 [±0.023] 60.000 [±0.000] 28.646 [±0.023]
40	14.566 [±0.091] 80.000 16.322 [±0.095]	17.110 [±0.091] 80.000 18.546 [±0.093]	17.380 [±0.087] 80.000 18.808 [±0.089]	20.338 [±0.086] 80.000 21.560 [±0.087]	24.580 [±0.072] 80.000 25.710 [±0.073]	27.568 [±0.067] 80.000 28.612 [±0.067]	30.150 [±0.060] 80.000 31.162 [±0.059]	33.708 [±0.046] 80.000 34.712 [±0.046]	36.926 [±0.030] 80.000 [±0.000] 37.926 [±0.030]
50	18.222 [±0.102] 100.000 20.090 [±0.108]	21.690 [±0.099] 100.000 23.142 [±0.100]	21.516 [±0.100] 100.000 23.024 [±0.104]	25.506 [±0.099] 100.000 26.806 [±0.099]	30.490 [±0.083] 100.000 31.612 [±0.084]	34.102 [±0.073] 100.000 35.152 [±0.073]	37.558 [±0.066] 100.000 38.572 [±0.067]	42.136 [±0.053] 100.000 43.142 [±0.053]	46.066 [±0.033] 100.000 [±0.000] 47.066 [±0.033]
60	21.824 [±0.106] 120.000 23.784 [±0.114]	25.854 [±0.114] 120.000 27.440 [±0.116]	25.920 [±0.100] 120.000 27.512 [±0.105]	30.584 [±0.098] 120.000 31.902 [±0.100]	36.662 [±0.093] 120.000 37.784 [±0.093]	40.904 [±0.082] 120.000 41.966 [±0.083]	45.818 [±0.072] 120.000 46.822 [±0.072]	50.284 [±0.054] 120.000 51.286 [±0.054]	55.354 [±0.034] 120.000 [±0.000] 56.354 [±0.034]
70	25.476 [±0.117] 140.000 27.512 [±0.121]	30.478 [±0.109] 140.000 32.010 [±0.113]	30.276 [±0.118] 140.000 31.856 [±0.120]	35.436 [±0.108] 140.000 36.786 [±0.112]	42.948 [±0.105] 140.000 44.144 [±0.104]	47.570 [±0.089] 140.000 48.684 [±0.090]	53.210 [±0.078] 140.000 54.220 [±0.079]	58.992 [±0.064] 140.000 59.994 [±0.064]	64.688 [±0.039] 140.000 [±0.000] 65.688 [±0.039]
80	29.350 [±0.122] 160.000 31.442 [±0.125]	34.588 [±0.122] 160.000 36.240 [±0.127]	34.534 [±0.128] 160.000 36.186 [±0.133]	40.410 [±0.123] 160.000 41.842 [±0.123]	48.978 [±0.103] 160.000 50.206 [±0.103]	54.274 [±0.101] 160.000 55.366 [±0.103]	60.846 [±0.082] 160.000 61.862 [±0.082]	67.118 [±0.067] 160.000 68.120 [±0.067]	73.696 [±0.043] 160.000 [±0.000] 74.698 [±0.043]
90	32.900 [±0.129] 180.000 35.042 [±0.134]	38.882 [±0.135] 180.000 40.660 [±0.140]	38.750 [±0.132] 180.000 40.426 [±0.135]	45.496 [±0.131] 180.000 46.936 [±0.130]	54.860 [±0.113] 180.000 56.088 [±0.114]	60.926 [±0.107] 180.000 62.028 [±0.107]	68.192 [±0.085] 180.000 69.220 [±0.085]	75.560 [±0.068] 180.000 76.560 [±0.068]	83.044 [±0.044] 180.000 [±0.000] 84.044 [±0.044]
100	36.644 [±0.145] 200.000 38.778 [±0.152]	43.016 [±0.139] 200.000 44.714 [±0.143]	43.382 [±0.145] 200.000 45.078 [±0.147]	50.450 [±0.140] 200.000 51.922 [±0.142]	60.870 [±0.123] 200.000 62.104 [±0.124]	67.660 [±0.113] 200.000 68.790 [±0.112]	75.784 [±0.090] 200.000 76.828 [±0.091]	84.356 [±0.073] 200.000 85.360 [±0.073]	92.234 [±0.045] 200.000 [±0.000] 93.234 [±0.045]
110	40.288 [±0.139] 220.000 42.542 [±0.144]	47.492 [±0.147] 220.000 49.306 [±0.152]	47.454 [±0.154] 220.000 49.210 [±0.155]	55.678 [±0.140] 220.000 57.126 [±0.141]	67.028 [±0.131] 220.000 68.268 [±0.132]	74.290 [±0.114] 220.000 75.430 [±0.115]	83.186 [±0.097] 220.000 84.240 [±0.096]	92.486 [±0.070] 220.000 93.486 [±0.070]	101.388 [±0.048] 220.000 [±0.000] 102.388 [±0.048]
120	43.940 [±0.155] 240.000 46.180 [±0.160]	52.146 [±0.151] 240.000 53.976 [±0.152]	52.040 [±0.158] 240.000 53.922 [±0.161]	60.602 [±0.145] 240.000 62.148 [±0.148]	72.964 [±0.136] 240.000 74.260 [±0.134]	80.922 [±0.114] 240.000 82.068 [±0.115]	90.934 [±0.098] 240.000 91.966 [±0.099]	100.998 [±0.081] 240.000 102.004 [±0.080]	110.588 [±0.052] 240.000 [±0.000] 111.588 [±0.052]
130	47.354 [±0.164] 260.000 49.646 [±0.169]	55.988 [±0.160] 260.000 57.798 [±0.160]	56.068 [±0.154] 260.000 57.884 [±0.160]	65.714 [±0.157] 260.000 67.276 [±0.158]	78.986 [±0.127] 260.000 80.324 [±0.128]	87.926 [±0.121] 260.000 89.136 [±0.122]	98.134 [±0.101] 260.000 99.178 [±0.102]	109.192 [±0.079] 260.000 110.198 [±0.079]	119.836 [±0.052] 260.000 [±0.000] 120.836 [±0.052]
140	51.360 [±0.157] 280.000 53.702 [±0.161]	60.452 [±0.170] 280.000 62.282 [±0.172]	60.598 [±0.169] 280.000 62.474 [±0.172]	70.620 [±0.167] 280.000 72.182 [±0.168]	85.324 [±0.144] 280.000 86.662 [±0.144]	94.530 [±0.127] 280.000 95.764 [±0.127]	105.786 [±0.113] 280.000 106.836 [±0.114]	117.562 [±0.084] 280.000 118.572 [±0.084]	129.252 [±0.054] 280.000 [±0.000] 130.252 [±0.054]
150	55.156 [±0.172] 300.000 57.588 [±0.175]	64.868 [±0.169] 300.000 66.772 [±0.173]	65.020 [±0.170] 300.000 66.940 [±0.174]	75.822 [±0.164] 300.000 77.370 [±0.164]	91.128 [±0.148] 300.000 92.448 [±0.148]	101.142 [±0.136] 300.000 102.336 [±0.137]	112.884 [±0.113] 300.000 113.944 [±0.113]	125.644 [±0.089] 300.000 126.650 [±0.089]	138.284 [±0.059] 300.000 [±0.000] 139.284 [±0.059]
160	58.646 [±0.173] 320.000 61.050 [±0.178]	69.004 [±0.182] 320.000 70.980 [±0.186]	69.296 [±0.173] 320.000 71.192 [±0.182]	80.750 [±0.165] 320.000 82.312 [±0.167]	97.338 [±0.155] 320.000 98.726 [±0.157]	107.814 [±0.140] 320.000 109.070 [±0.142]	120.818 [±0.117] 320.000 121.884 [±0.116]	134.010 [±0.091] 320.000 135.014 [±0.091]	147.522 [±0.056] 320.000 [±0.000] 148.522 [±0.056]
170	62.826 [±0.180] 340.000 65.256 [±0.185]	73.664 [±0.184] 340.000 75.592 [±0.185]	73.566 [±0.192] 340.000 75.530 [±0.193]	85.892 [±0.167] 340.000 87.526 [±0.167]	103.462 [±0.159] 340.000 104.856 [±0.160]	114.834 [±0.141] 340.000 116.040 [±0.142]	128.128 [±0.124] 340.000 129.210 [±0.124]	143.214 [±0.096] 340.000 144.216 [±0.096]	156.760 [±0.061] 340.000 [±0.000] 157.764 [±0.061]
180	66.056 [±0.194] 360.000 68.510 [±0.199]	77.706 [±0.187] 360.000 79.744 [±0.188]	77.356 [±0.184] 360.000 79.298 [±0.188]	90.942 [±0.180] 360.000 92.590 [±0.181]	109.364 [±0.158] 360.000 110.778 [±0.160]	121.508 [±0.144] 360.000 122.796 [±0.145]	135.676 [±0.128] 360.000 136.762 [±0.128]	151.608 [±0.094] 360.000 152.616 [±0.094]	166.118 [±0.058] 360.000 [±0.000] 167.118 [±0.058]
190	69.766 [±0.188] 380.000 72.310 [±0.192]	82.464 [±0.190] 380.000 84.472 [±0.193]	81.932 [±0.188] 380.000 83.918 [±0.191]	96.072 [±0.188] 380.000 97.734 [±0.189]	115.576 [±0.165] 380.000 116.988 [±0.165]	128.022 [±0.148] 380.000 129.274 [±0.148]	143.166 [±0.129] 380.000 144.242 [±0.129]	160.134 [±0.103] 380.000 161.134 [±0.103]	175.224 [±0.064] 380.000 [±0.000] 176.224 [±0.064]
200	73.558 [±0.202] 400.000 76.100 [±0.206]	86.684 [±0.189] 400.000 88.712 [±0.191]	86.508 [±0.200] 400.000 88.488 [±0.203]	101.314 [±0.187] 400.000 103.086 [±0.188]	121.612 [±0.165] 400.000 123.034 [±0.167]	134.756 [±0.155] 400.000 136.034 [±0.156]	150.452 [±0.128] 400.000 151.538 [±0.129]	168.500 [±0.105] 400.000 169.502 [±0.105]	184.606 [±0.064] 400.000 [±0.000] 185.606 [±0.064]
210	77.082 [±0.198] 420.000 79.750 [±0.205]	90.610 [±0.199] 420.000 92.656 [±0.199]	90.946 [±0.212] 420.000 92.998 [±0.210]	106.226 [±0.186] 420.000 107.974 [±0.191]	127.688 [±0.183] 420.000 129.096 [±0.184]	141.534 [±0.156] 420.000 142.816 [±0.156]	157.916 [±0.145] 420.000 159.030 [±0.145]	176.936 [±0.107] 420.000 177.940 [±0.107]	193.776 [±0.069] 420.000 [±0.000] 194.776 [±0.069]
220	80.486 [±0.204] 440.000 83.144 [±0.207]	95.252 [±0.201] 440.000 97.340 [±0.204]	95.330 [±0.197] 440.000 97.408 [±0.202]	110.940 [±0.199] 440.000 112.722 [±0.200]	133.912 [±0.178] 440.000 135.432 [±0.180]	148.196 [±0.150] 440.000 149.528 [±0.152]	165.386 [±0.143] 440.000 166.504 [±0.143]	185.012 [±0.108] 440.000 186.026 [±0.108]	202.996 [±0.072] 440.000 [±0.000] 203.998 [±0.072]
230	84.272 [±0.217] 460.000 87.026 [±0.223]	99.486 [±0.210] 460.000 101.562 [±0.210]	99.218 [±0.206] 460.000 101.280 [±0.206]	116.176 [±0.198] 460.000 117.920 [±0.198]	139.564 [±0.183] 460.000 141.040 [±0.184]	154.488 [±0.177] 460.000 155.802 [±0.176]	172.798 [±0.133] 460.000 173.906 [±0.133]	193.450 [±0.109] 460.000 194.450 [±0.109]	212.230 [±0.073] 460.000 [±0.000] 213.230 [±0.073]
240	88.144 [±0.224] 480.000 90.810 [±0.227]	103.514 [±0.222] 480.000 105.598 [±0.220]	103.740 [±0.211] 480.000 105.782 [±0.215]	121.116 [±0.207] 480.000 122.918 [±0.209]	145.878 [±0.190] 480.000 147.368 [±0.195]	161.874 [±0.166] 480.000 163.170 [±0.165]	180.804 [±0.144] 480.000 181.920 [±0.144]	201.932 [±0.109] 480.000 202.936 [±0.109]	221.516 [±0.068] 480.000 [±0.000] 222.516 [±0.068]
250	92.034 [±0.224] 500.000 94.786 [±0.228]	108.034 [±0.232] 500.000 110.100 [±0.233]	107.858 [±0.222] 500.000 109.968 [±0.226]	126.178 [±0.220] 500.000 128.032 [±0.221]	151.716 [±0.194] 500.000 153.238 [±0.196]	168.202 [±0.181] 500.000 169.506 [±0.183]	187.898 [±0.142] 500.000 189.042 [±0.143]	210.168 [±0.109] 500.000 211.176 [±0.109]	230.684 [±0.076] 500.000 [±0.000] 231.684 [±0.076]
260	95.468 [±0.221] 520.000 98.176 [±0.230]	111.616 [±0.228] 520.000 113.714 [±0.230]	112.436 [±0.224] 520.000 114.564 [±0.227]	131.608 [±0.213] 520.000 133.334 [±0.214]	158.066 [±0.204] 520.000 159.598 [±0.204]	175.286 [±0.177] 520.000 176.628 [±0.177]	195.666 [±0.153] 520.000 196.820 [±0.154]	218.382 [±0.121] 520.000 219.388 [±0.121]	239.826 [±0.074] 520.000 [±0.000] 240.828 [±0.074]
270	99.322 [±0.225] 540.000 102.010 [±0.229]	116.364 [±0.238] 540.000 118.550 [±0.240]	116.598 [±0.216] 540.000 118.696 [±0.216]	136.470 [±0.223] 540.000 138.412 [±0.226]	163.854 [±0.201] 540.000 165.318 [±0.204]	181.704 [±0.181] 540.000 183.044 [±0.181]	202.812 [±0.148] 540.000 203.944 [±0.149]	226.998 [±0.124] 540.000 228.000 [±0.124]	248.928 [±0.075] 540.000 [±0.000] 249.928 [±0.075]
280	103.022 [±0.235] 560.000 105.836 [±0.239]	120.858 [±0.238] 560.000 122.992 [±0.238]	120.770 [±0.239] 560.000 123.000 [±0.241]	141.120 [±0.231] 560.000 142.938 [±0.234]	170.002 [±0.212] 560.000 171.540 [±0.213]	188.264 [±0.193] 560.000 189.612 [±0.195]	210.468 [±0.158] 560.000 211.648 [±0.161]	235.256 [±0.116] 560.000 236.270 [±0.116]	258.510 [±0.080] 560.000 [±0.000] 259.510 [±0.080]
290	106.470 [±0.244] 580.000 109.324 [±0.245]	125.246 [±0.257] 580.000 127.436 [±0.258]	125.304 [±0.249] 580.000 127.452 [±0.249]	146.426 [±0.236] 580.000 148.254 [±0.236]	176.210 [±0.210] 580.000 177.730 [±0.213]	195.384 [±0.193] 580.000 196.748 [±0.195]	217.646 [±0.168] 580.000 218.782 [±0.169]	243.916 [±0.119] 580.000 244.930 [±0.119]	267.662 [±0.080] 580.000 [±0.000] 268.662 [±0.080]
300	109.794 [±0.238] 600.000 112.558 [±0.247]	130.004 [±0.243] 600.000 132.206 [±0.246]	129.674 [±0.240] 600.000 131.964 [±0.243]	151.840 [±0.231] 600.000 153.682 [±0.233]	181.820 [±0.210] 600.000 183.442 [±0.213]	201.716 [±0.194] 600.000 203.122 [±0.193]	225.312 [±0.168] 600.000 226.460 [±0.169]	251.956 [±0.124] 600.000 252.970 [±0.124]	276.830 [±0.079] 600.000 [±0.000] 277.830 [±0.079]

Table 43: Lower bound, copies for the naive algorithm and copies for the new algorithm over varying population sizes and selection pressures. Square brackets contain standard errors.

many individuals selected few times each. Hence the without-replacement tournament sizes were varied to produce a range of selection pressures. The many-from-few measure described in Section 7.2.7 was used to prescribe the selection pressures and hence tournament sizes.

Note that in most cases, it will not be possible to achieve the exact stated selection pressure for the stated population size due the discrete nature of the tournament sizes. For this reason, slight oddities are to be expected, particularly for lower population sizes and selection pressures. This may explain the similarity between the values for selection pressures of 60% and 65% in Figures 88 and 88(a).

It would be helpful to know how the resulting number of copies compares to the best number of copies that could possibly be achieved. For this reason each result is compared against both the naive algorithm's $2N$ copies and a lower bound. As described earlier, a lower bound can be obtained for a given problem from the number of slots that are not mentioned as the first part of any recipes. Again, it is worth stressing that the lowest possible number of copies may still be greater than this lower bound so a result that doesn't reach this lower bound may still be doing as well as it possibly can. For example, the lower bound for the problem with recipes $1 \rightarrow 2$, $2 \rightarrow 3$ and $3 \rightarrow 1$ is zero yet the problem cannot be solved without using copies.

Figure 87 shows two subsets of these results: those generated with a high selection pressure of 95% (Figure 87(a)) and those generated with a low selection pressure of 55% (Figure 87(b)). The resulting number of copies appears to increase as a linear factor of the population size and is lower in the case of low selection pressure. The lower bounds indicate that this is largely because the higher selection pressure scenarios simply need more copies.

Figure 88 shows the data to highlight the percentage reduction that the algorithm's number of copies represents compared to the naive algorithm's $2N$ copies. Figure 88(a) shows the percentage of the reduction from $2N$ to zero. This illustrates the same pattern that the reductions are more impressive for data from lower selection pressures. For populations of 10, the reductions vary from 50.000% for 95% selection pressure, up to 76.010% for 55% selection pressure. These results improve as the population size increases and for all populations of 100 or larger (up to 300), the reductions vary from 53.383% for 95% selection pressure up to 80.611% for 55% selection pressure. Figure 88(b) shows the percentage of the reduction from $2N$ to the lower bound. This shows that the reduction is often very close to the best possible, particularly for population sizes above 100 and that this is not affected much by the selection pressure used to generate the data. The worst reductions are for population size 10 and selection pressure 95%: 90.909% reduction to the lower bound. For population sizes of 100 or above (up to 300), all reductions toward the lower bound are at least 98.694%.

Figure 89 shows the average absolute number of copies above the lower bound from the same data. All average values are below three in the population range shown.

The absolute number of copies appears to be growing slowly with regard to population size and is larger for lower selection pressures.

7.4 Summary and Contribution

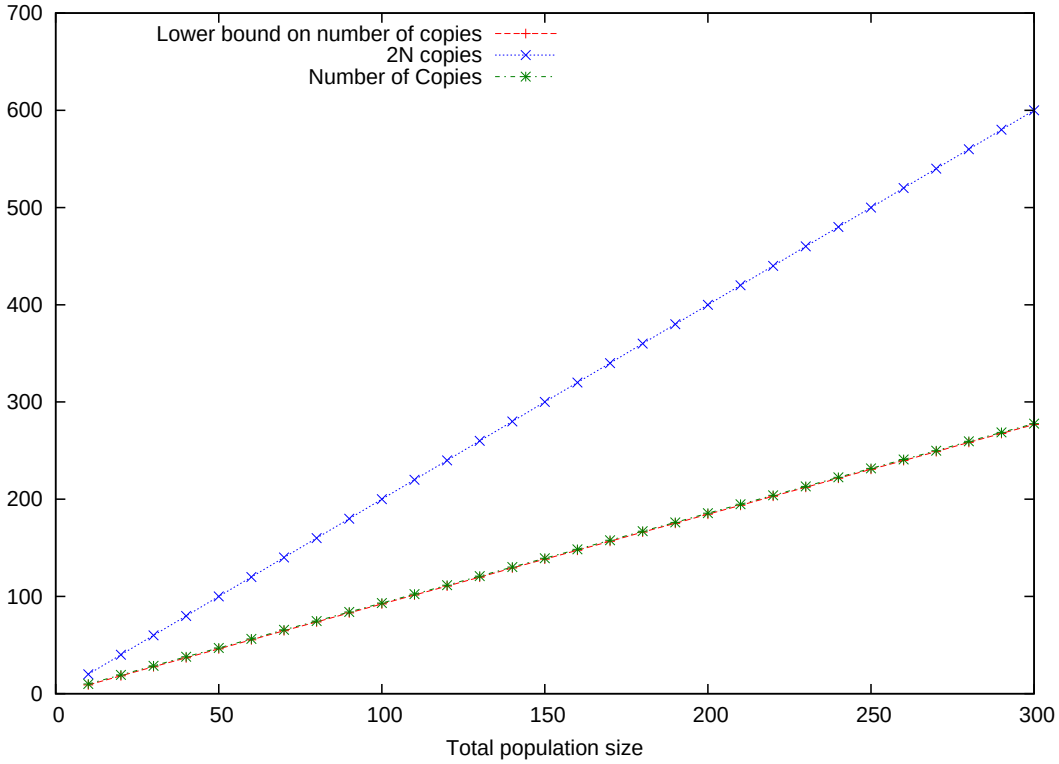
Chapters 4 and 6 created effective GPU evaluators to reduce the time spent on evaluation. Chapter 5 made these more useful by showing how the GPU work could be interleaved with CPU work and how multiple GPUs and CPU cores could be recruited. This still leaves the danger that, depending on the setup, time wasted on CPU tasks leaves the GPU sat idle for longer than necessary. Hence this chapter described two optimisations to the CPU code of a GP run.

The benefit of these two optimisations will vary according to the details of the setup to which they are applied. They will have the greatest effect for those setups in which the time taken by the optimised tasks is greatest. This will tend to occur for those setups that do not have sufficient testcases to ensure the GPU evaluation time is the slowest part of the run. The acceleration ratio offered by these optimisations is unlikely to be comparably high to the acceleration achieved by the initial move to the GPU but is of value because it applies on top of the GPU acceleration. These optimisations help to ensure that the full computational power of the GPU can be brought to bear on a wider range of setups.

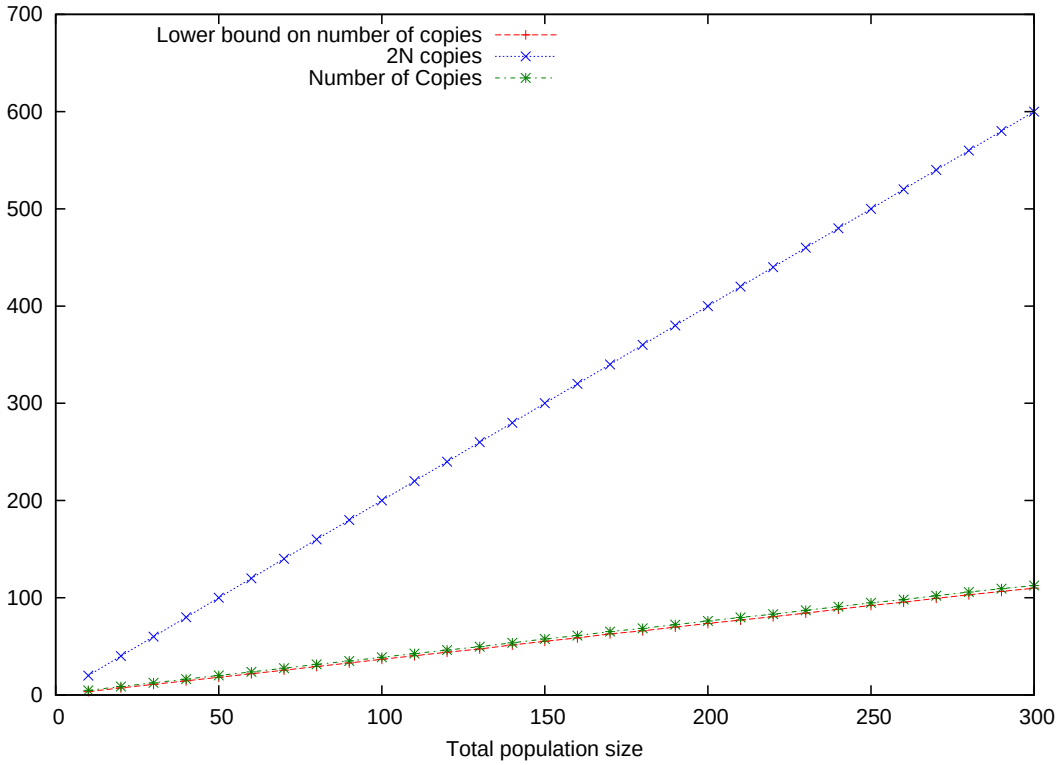
The first optimisation involved optimising without-replacement tournament selection in an attempt to reduce the number of random numbers it required. Through mathematical analysis, the algorithm was optimised. The new algorithm was shown to be faster than the standard algorithm for most configurations up to populations of 1000, even after the standard algorithm's random sampling subroutine was improved. However this algorithm is only likely to be of much use for very fast implementations of EC (such as in this work) since the slowest population tournament selection in the experiments was 0.056042 seconds (around $1/18^{\text{th}}$ of a second). The mathematical analysis also generated insight into the relationship between a tournament selection configuration and the strength of its selection pressure.

This work involved a number of contributions that are believed to be novel:

- The mathematical analysis of without-replacement tournament selection which is considerably more challenging than the with-replacement analysis that has previously been performed in the literature.
- The optimised tournament selection algorithm it generated.
- The investigation into the affect of this work on the run time of the algorithm.
- The continuous extensions of the probability formulae.
- The mathematical demonstration that these extensions are well behaved.

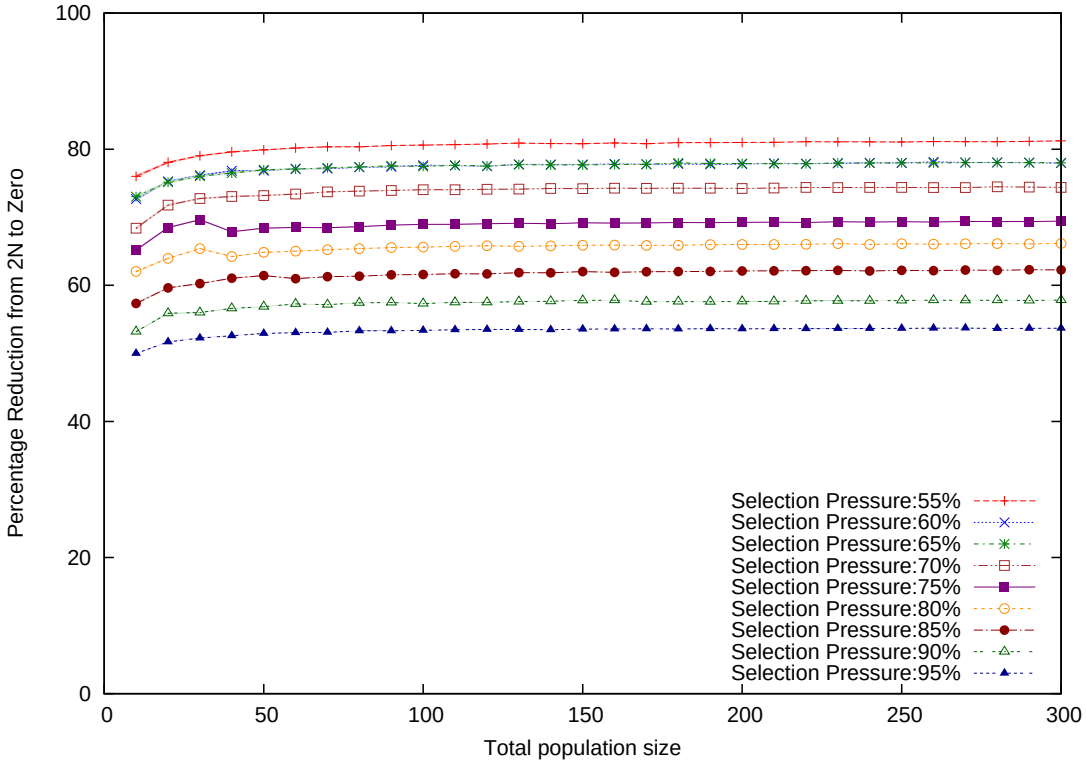


(a) Average performance over data generated with high selection pressure

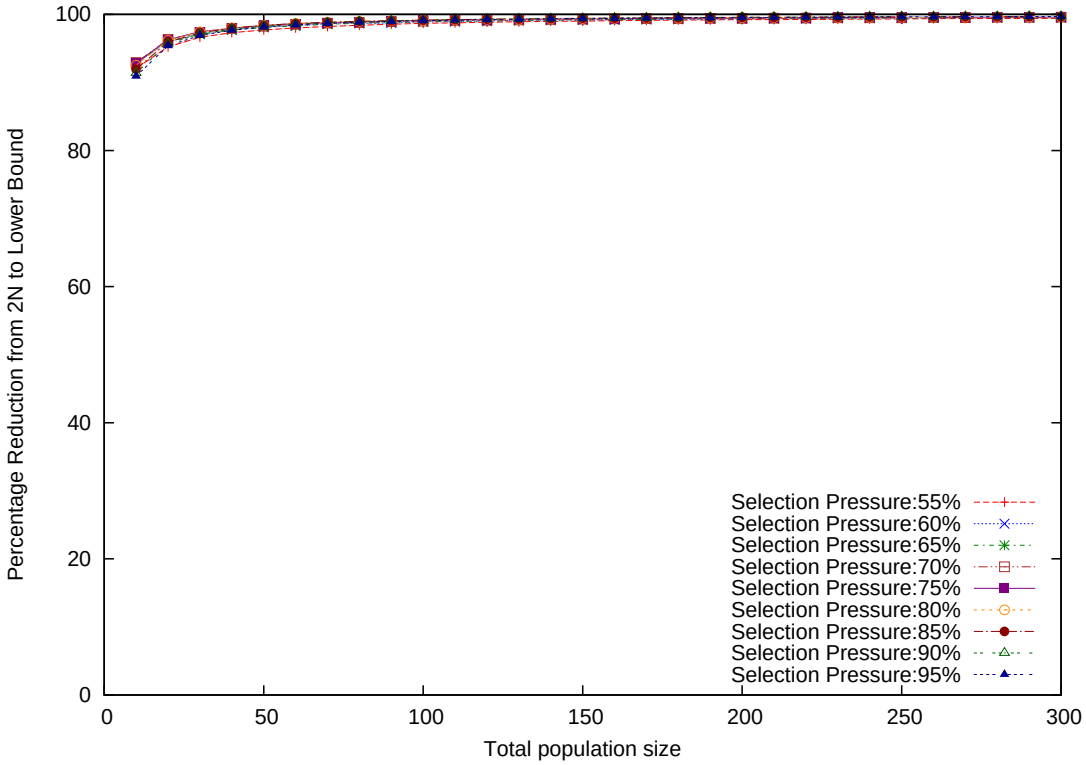


(b) Average performance over data generated with low selection pressure

Figure 87: The average number of copies required by the algorithm compared to the $2N$ copies required for the naive algorithm and the lower bound. It is not possible to do better than the lower bound and the best possible result may be larger than the lower bound. Each line has a bar behind it to represent the average plus and minus one standard error. Since the standard errors are so small, these can hardly be seen.



(a) Average percentage reduction from $2N$ copies to zero copies



(b) Average percentage reduction from $2N$ copies to the lower bound number of copies

Figure 88: The results represent considerable reductions in the numbers of copies and get close to the lower bounds. The average percentage reduction towards zero and towards the lower bound from the $2N$ copies required for the naive algorithm. Each line has a bar behind it to represent the average plus and minus one standard error. Since the standard errors are so small, these can hardly be seen.

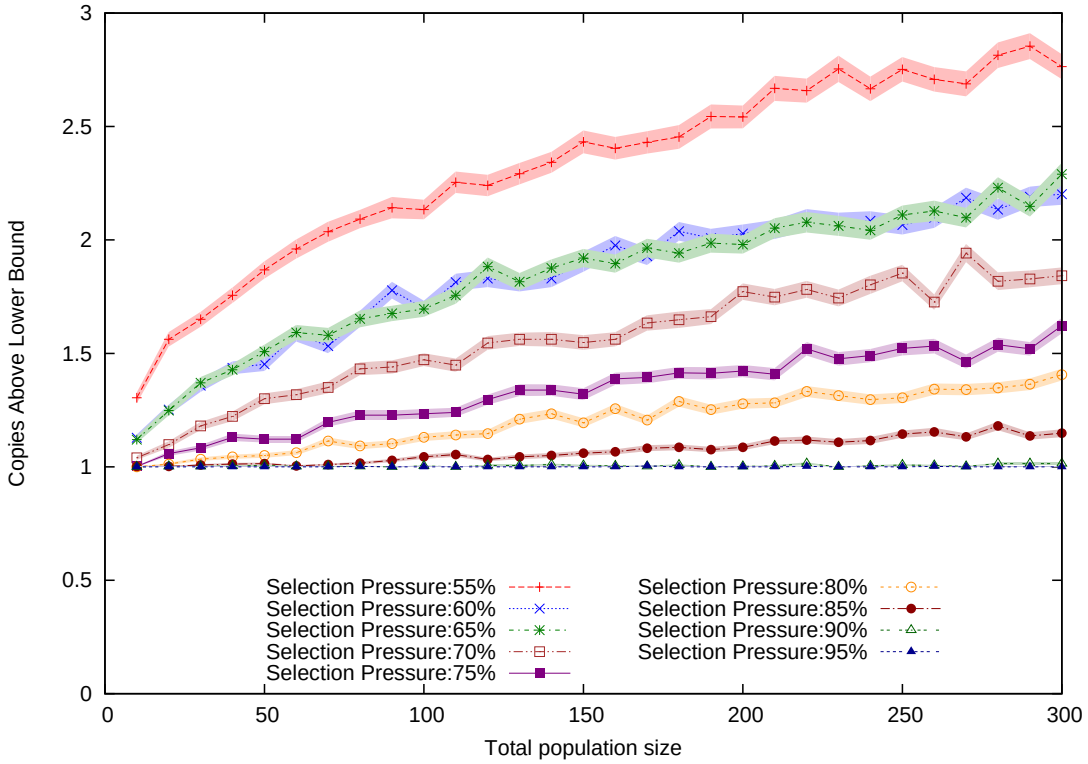


Figure 89: The average absolute number of copies above the lower bound. Each line has a bar behind it to represent the average plus and minus one standard error.

- The new *many-from-few* measure, an intuitive measure of selection pressure.
- A method to determine this measure for a given tournament configuration.
- A method to determine the tournament size that would exert a given value of this selection pressure measure (for a given population size in the case of without-replacement tournament selection).
- Maps of the contours of constant selection pressure over varying population size and tournament size.

This work suggests two key recommendations for all researchers publishing work involving tournament selection.

1. When reporting experiments using tournament selection, one should always state whether the tournament selection is being used with-replacement or without because these schemes are quite different.
2. One should also state (at least a rough indication of) the many-from-few selection pressure measure when reporting use of tournament selection. This value conveys a direct sense of what is happening in the selection scheme and makes it much easier to compare between different configurations. The tables in this

work should provide appropriate values for most practical configurations and the formulae can be used to calculate the details in other cases.

The second optimisation involved optimising the algorithm to construct a new population based on the previous population and the selections. The aim was to reduce the number of copies of individuals required. A useful representation was found to depict the problem and this was used to guide the design of a heuristic. This was found to be highly effective at reducing the number of copies particularly when viewed as a percentage from $2N$ copies to the lower bound. The minimum reduction in the number of copies was 50.000% for 95% selection pressure in a population of 10 and this rose to a reduction of 80.611% for 55% selection pressure in a population of 100. A lower bound was used to give some indication of how far the heuristic went towards achieving the best possible reduction. The worst reductions to the lower bound were for population size 10 and selection pressure 95%: 90.909% reduction to the lower bound. For population sizes of 100 or above (up to 300), all reductions toward the lower bound were at least 98.694%.

This work involved a number of contributions that are believed to be novel:

- The analysis of the problem faced in reducing copies in EC.
- The new representation to depict the problem thus making it much easier to contemplate.
- The new proposed heuristic to tackle real instances of the problem.
- The investigation into the effectiveness of this heuristic at reducing copies.

The benefit of these two optimisations will vary considerably according to the details of the run. For setups where the selection and copying account for a very small part of the run time or are executed in paralled with other tasks such as GPU evaluation that takes longer, there will be little benefit to be had. This will tend to occur for CPU evaluation, data-parallel GPU evaluation or for data parallel population parallel runs in which the number of testcases is not so large as to ensure the GPU computation.

8 Conclusions and Future Work

8.1 Conclusions

This thesis described work to accelerate Genetic Programming (GP) and Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble") as far as possible on one machine, primarily through the use of the Graphics Processing Unit (GPU). As discussed in Chapter 1, the motivation for this work was to provide the tools to research how to stimulate long-term fitness growth when evolving behaviours. The work in this thesis contributes by taking a necessary and important first step toward this crucial aim of stimulating long term fitness growth. More must be done in the future to build on this first step.

More generally, the work contributes to the broader fields of GP and Evolutionary Computation (EC) by reducing run times. The tools also have wide applicability in any situation where there is reason to perform these forms of EC more quickly. Accelerating these forms of EC is of such importance to the research community that it has developed into an independent research topic in its own right, as can be seen from the volume of literature on the topic as discussed in Chapter 2. Accelerating EC improves our ability to research, develop and apply it.

As described in Chapter 2, there are two main approaches to accelerating GP with the GPU: population-parallel and data-parallel. The work described in this thesis made contributions to both these techniques (as covered in Chapters 4 and 6 respectively). Other chapters looked at how these techniques can be complemented with tactics to speed up the whole run on a single machine: overlapping GPU and Central Processing Unit (CPU) work, recruiting a further GPU and CPU core and performing intra-population transfers efficiently (Chapter 5) and reducing wasted CPU time (Chapter 7).

The main work began in Chapter 4, which described work to develop a population-parallel evaluator. Since the research was aimed at providing tools to allow the investigation of long-term fitness growth, it was deemed important that this evaluator should permit a wide range of representations. Cyclic GP was chosen because it is a powerful representation that encompasses many of the more common forms of GP. Cyclic GP imposes extra memory requirements and so providing this flexibility of representation required substantial work to accommodate the requirements within an efficient Compute Unified Device Architecture (CUDA) system. Chapter 4 described the design of the system in detail and then described experiments to assess the resulting performance. On a realistic configuration, the architecture was found to run 175.703 times faster than the single–core CPU equivalent when measured over the whole run.

Providing a powerful GPU evaluator only realises part of the potential of a single machine as it leaves either the CPU or the GPU idle for most of the run and ignores any extra GPU and CPU resources available. Chapter 5 described three steps to develop

a GPU evaluator into a faster overall system. Step one showed how to better utilise the evaluator in the context of the whole run by using demes to allow the GPU and the CPU to work simultaneously on different tasks. The described approach uses the strengths of both types of processor and uses them both simultaneously. It also avoids introducing unwanted asynchronous components that would corrupt the standard algorithm and make reproducibility very difficult. This technique was found to reduce the total run time by up to a further 1.806 times. Step two showed what is involved in recruiting further GPUs and further CPU cores. This is likely to be of much interest because the current trend appears to be toward greater numbers of CPU cores and GPUs in individual machines. This technique was found to reduce total run time by up to a further 1.982 times over the first step and by up to a further 3.292 times in combination with the first step. Step three showed that the cost of the transfers required to use demes was small, even compared to the run time of a highly accelerated architecture and hence that deme transfers need not deter researchers from using demes in the ways described in the previous steps. Furthermore, for those who are concerned about the impact of these deme transfers, the work demonstrated how these costs can be reduced by separating out donating from receiving to minimise unnecessary delays. The combination of the first two techniques were found to reduce the total run time by up to a further 3.292 times. The worst increase seen in run time caused by the deme-transfers was only 13.449% and with smart transfers, this increase was only 1.742%.

Chapter 6 switched to the other major category of approaches to using the GPU to accelerate GP: data-parallel. Whereas the population-parallel evaluator described in Chapter 4 had covered most node-based representations of GP (through cyclic GP), this chapter's data-parallel evaluator covered the other side of GP representations: linear. In particular, it implemented TMBL, a representation similar to linear GP proposed as part of the investigation into long term fitness growth (and discussed further in Section 3.2). The chapter contributed by showing two possible ways to tackle the biggest problem of data-parallel GPU approaches, namely the time required for compilation. It also identifies issues involved with these techniques and their possible solutions. The two techniques were encoding individuals in a lower-level GPU language and aligning individuals to reduce duplication of code. The first technique was seen to reduce compilation times by up to 5.861 times and to increase evaluation speeds by up to 23.029%. The second technique was seen to reduce compilation time by up to 4.817 times whilst only reducing evaluation speed by 3.656%. Combined, the techniques were seen to reduce the compilation times by up to 57.625 times on large, 1000-instruction TMBL individuals.

Chapter 7 described work to optimise two of the processes that were wasting the most CPU time: tournament selection and individual copying. The work on optimising tournament selection showed how without-replacement tournament selection may be implemented in quite a simple way so as not to require as many random number

generations. This technique was seen to be faster than the standard technique for most tournament configurations in populations of up to 1000. Furthermore, the mathematical analysis used to perform this is a contribution in itself and was used to make a further contribution by underpinning a new measure of selection pressure. This measure was used to analyse selection pressure in a wide range of configurations of both with-replacement and without-replacement tournament selection. Two figures provided a strikingly clear illustration of the patterns of selection pressure for these two schemes. The second target for optimisation was the copying of individuals to construct new generations. This was achieved by proposing an appropriate heuristic to reduce the number of copies required. To devise such a heuristic, a suitable way to represent example problems was proposed. At worst the technique was seen to deliver a 50% reduction in the number of copies and was shown to deliver at worst a 90.909% reduction towards a lower bound on the minimum possible number of copies.

The techniques described in this thesis combine to form a system that is highly effective at achieving the aim set out in Section 1.4, namely:

Aim *Accelerate program evolution (GP) as much as possible on a single, reasonably-priced machine with as little distortion of the algorithm as possible. This should be done flexibly to allow for a wide range of forms and problems.*

It seems doubtful that, for example, TMBL can be performed at speeds much greater than those achieved by this system on equivalent technology because the system already keeps the GPU busy for most of the run performing TMBL floating point operations about as fast as could be hoped for. One exception is that it may be possible to reduce the CPU time spent on CUDA compilation quite considerably by writing CUDA binaries directly. This might allow the GPU to be kept busy for smaller data-sets. The complexity and future compatibility issues arising from this potential technique were deemed to outweigh any future benefit as described in Section 6.3.5.

The work has achieved these high speeds at the price of increased complexity. The thesis has given an indication of the complexity involved in the various steps and the resulting speed increases. To decide which, if any, of these techniques to adopt, each individual researcher must consider whether they offer sufficient benefits to warrant the greater complexity. The complexity has been curbed by designing and writing quality code that attempts to keep components independent.

8.2 Future Work

The work described in this thesis opens up several possible lines of further research. One avenue would be to take the techniques used to construct the population-parallel evaluator in Chapter 4 and use them for a population-parallel evaluator of linear forms such as linear GP or TMBL. If the quantity of memory required for each individual were small enough, it would be possible to achieve this using only one thread per evaluation

of a single program–testcase combination. To provide more memory per individual, it would be necessary to attempt to divide evaluation over multiple threads. Dividing the evaluation of a linear sequence of instructions over multiple parallel processors is more difficult than doing the same for a set of cyclic nodes. This is because the order of execution of the instructions is important. Nevertheless, it would be possible to perform a CPU analysis of each individual before submission to the GPU to assess how the list of instructions might be teased apart into several shorter lists of instructions. This is difficult since the analysis must ensure that if the lists are executed in parallel (with each step synchronised) the original behaviour is preserved and furthermore, that there are no potential read/write conflicts that might deliver differing results over multiple executions. In fact, this technique was successfully coded but it could not be investigated sufficiently thoroughly within the scope of this thesis. Informal assessment suggested that the technique was successful but that on rare occasions it had to be abandoned for individuals that could not be teased apart as required. On these occasions, small thread blocks had to be used to evaluate those individuals at much slower speed.

In general, it would be more difficult to produce efficient GPU evaluators for forms of linear GP that include conditional branches since these are likely to induce divergences between neighbouring threads, which is very costly on a GPU architecture.

If there is sufficient cause, it might be sensible to extend the existing population-parallel evaluator to automatically identify batches of individuals that are cycle-free and switch to a simpler non-cyclic evaluator.

The work on recruiting further GPUs and CPU cores in Section 5.3 could easily be extended by adding further processors. It is currently possible to have many CPU cores on a single machines through increased numbers of cores per processor and increased numbers of processors per motherboard. Similarly, it is possible to have more GPUs on a single machine. For instance, the nVidia GTX 590 (in the most recent generation of graphics cards at the time of writing) contains two GPUs and at least two such cards may be hosted in one machine. The architecture described in Section 5.3 should be ready to usefully incorporate this additional processing power.

Another possibility would be to investigate applying the techniques that were used to reduce data-parallel compilation times in Chapter 6 to node-based forms (such as tree-based GP). The first technique of writing individuals in Parallel Thread EXecution (PTX) should be just as applicable to node-based forms as it was to TMBL. This could open up tree-based GP with moderate data-sets to the remarkable evaluation speeds seen in the data-parallel work. It is perhaps less clear how effective the second technique of code alignment would be when applied to tree-based GP. Certainly, it should be possible to apply the technique since the code to evaluate the tree could be linearised and then aligned. However, it is not clear whether small mutations in trees will necessarily correlate well with small changes in the linearised code and this may limit the ability to reduce redundancy. It is also less clear that there is as much similarity within

a tree-based GP population are as there is within a TMBL population. If the level of similarity is much lower, this may also reduce the effectiveness of the alignment technique.

As mentioned in Section 6.6, it is theoretically possible to cut compilation time completely by directly manipulating cubin binary files. As also discussed in Section 6.6, an initial investigation suggested that this would be an extraordinarily complicated (and potentially brittle) approach.

This thesis described research to create two evaluators (one population-parallel and one data-parallel), make them as fast as possible and then work to keep the GPU (s) as busy as possible on those evaluators throughout the run. The aim of this was to squeeze as much GP/TMBL performance out of a machine as possible and this was rather successful. In particular, the total system is capable of performing TMBL runs on fairly moderate numbers of testcases such that the GPU spends most of the run performing TMBL floating point operations about as fast as could be hoped for. There are potential areas for future exploration as discussed above but the success of this acceleration work means that their expected rewards would be relatively small.

Instead, the work described in this thesis really opens up a different line of research, the one that provided the motivation at the start of the thesis: exploring how we can be more effective at harnessing the creative power of evolution by natural selection. It is hoped that this will provide us with a powerful tool that will contribute to progress on a wide range of problems.

References

[1] D. Andre and J. R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. *Information Sciences*, 106(3-4):201–218, 1998.

[2] E. Artin. *The gamma function*. Holt, Rinehart and Winston (New York), 1964.

[3] T. Bäck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *In Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 57–62. IEEE Press, 1994.

[4] T. Bäck. Generalized convergence models for tournament- and (μ, λ)-selection. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 2–8, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[5] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, Jan. 1998.

[7] N. H. Barton, D. E. Briggs, J. A. Eisen, D. B. Goldstein, and N. H. Patel. *Evolution*. Cold Spring Harbor Laboratory Press, 2007.

[8] F. H. Bennett III, J. R. Koza, J. Shipman, and O. Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[9] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Res*, 28:235–242, 2000.

[10] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 9–16, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[11] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evol. Comput.*, 4:361–394, December 1996.

[12] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[13] E. Cantú-Paz. On random numbers and the performance of genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 311–318, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[14] P. T. Cattani and C. G. Johnson. ME-CGP: Multi expression Cartesian genetic programming. In *Proceedings of the 2010 IEEE World Congress on Computational Intelligence*, July 2010.

[15] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 7-11 July 2007. ACM Press.

[16] J. Clegg, J. A. Walker, and J. F. Miller. A new crossover technique for cartesian genetic programming. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1580–1587, London, 7-11 July 2007. ACM Press.

[17] M. Ebner, M. Reinhardt, and J. Albert. Evolution of vertex and pixel shaders. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 261–270, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[18] S. E. Eklund. Time series forecasting using massively parallel genetic programming. In *Proceedings of Parallel and Distributed Processing International Symposium*, pages 143–147, 22-26 Apr. 2003.

[19] F. Fernandez, M. Tomassini, and L. Vanneschi. Studying the influence of communication topology and migration on distributed genetic programming. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 51–63, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[20] K.-L. Fok, T.-T. Wong, and M.-L. Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, Mar.-Apr. 2007.

[21] O. Garnica, J. L. Risco-Martin, J. Hidalgo, and J. Lanchares. Speeding-up resolution of deceptive problems on a parallel gpu-cpu architecture. In *Parallel Architectures and Bioinspired Algorithms (at PACT)*, 2008.

[22] C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 9-14 Oct. 1994. Springer-Verlag.

[23] C. Gathercole and P. Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 111–118, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[24] C. Gathercole and P. Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[25] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. San Francisco, CA: Morgan Kaufmann, 1991.

[26] R. Groß, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[27] S. Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[28] S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[29] S. Harding and W. Banzhaf. Genetic programming on GPUs for image processing. *International Journal of High Performance Systems Architecture*, 1(4):231–240, 2008.

[30] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. In A. Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 285–292, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[31] S. Harding, J. F. Miller, and W. Banzhaf. Developments in cartesian genetic programming: self-modifying cgp. *Genetic Programming and Evolvable Machines*, 11(3-4):397–439, 2010.

[32] S. L. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, page 2, Canada, 2007. IEEE Computer Society.

[33] S. L. Harding and W. Banzhaf. Distributed genetic programming on GPUs using CUDA. In I. Hidalgo, F. Fernandez, and J. Lanchares, editors, *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, Sept. 13 2009.

[34] S. L. Harding, J. F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 1021–1028, London, 7-11 July 2007. ACM Press.

[35] M. I. Heywood and A. N. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 44–59, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[36] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. Aug. 2008.

[37] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Comput.*, 9:3–12, January 2005.

[38] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi,

and W. B. Langdon, editors, *Genetic Programming: 4th European conference*, pages 302–312, Berlin, 2001. Springer.

[39] W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[40] H. Katagiri, K. Hirasawa, J. Hu, and J. Murata. Network structure oriented evolutionary model-genetic network programming-and its comparison with genetic programming. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 219–226, San Francisco, California, USA, 9-11 July 2001.

[41] M. M. Khan, G. M. Khan, and J. F. Miller. Evolution of neural networks using cartesian genetic programming. In *IEEE Congress on Evolutionary Computation (CEC 2010)*, Barcelona, Spain, 18-23 July 2010. IEEE Press.

[42] A. Koenig. Why are vectors efficient? *JOOP*, 11(5):71–75, 1998.

[43] F. Kühling, K. Wolff, and P. Nordin. Brute-force approach to automatic induction of machine code on CISC architectures. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 288–297, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[44] W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In A. I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, and A. S. Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 Apr. 2010. Springer.

[45] W. B. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 Mar. 2008. Springer.

[46] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, Oct. 2008. Special Issue on Distributed Bioinspired Algorithms.

[47] W. B. Langdon and P. Nordin. Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic*

Programming, Proceedings of EuroGP'2001, volume 2038 of *LNCS*, pages 313–324, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[48] J. Lässig and D. Sudholt. The benefit of migration in parallel evolutionary algorithms. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1105–1112, New York, NY, USA, 2010. ACM.

[49] T. E. Lewis and G. D. Magoulas. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1379–1386, Montreal, 8-12 July 2009. ACM.

[50] T. E. Lewis and G. D. Magoulas. Tweaking a tower of blocks leads to a TMBL: Pursuing long term fitness growth in program evolution. In *IEEE Congress on Evolutionary Computation (CEC 2010)*, pages 4465–4472, Barcelona, Spain, 18-23 July 2010. IEEE Press.

[51] J.-M. LI, X.-J. WANG, R.-S. HE, and Z.-X. CHI. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, pages 855–862, Washington, DC, USA, 2007. IEEE Computer Society.

[52] D. Lim, Y. Jin, Y.-S. Ong, and B. Sendhoff. Generalizing surrogate-assisted evolutionary computation. *Trans. Evol. Comp*, 14:329–355, June 2010.

[53] T. V. Luong, N. Melab, and E.-G. Talbi. Gpu-based island model for evolutionary algorithms. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1089–1096, New York, NY, USA, 2010. ACM.

[54] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410, New York, NY, USA, 2009. ACM.

[55] O. Maitre, P. Collet, and N. Lachiche. Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In A. I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, and A. S. Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 301–312, Istanbul, 7-9 Apr. 2010. Springer.

[56] J. Meyer-Spradow and J. Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, editors, *Eurographics 2003 Short Paper Proceedings*, pages 301–306, 2003.

[57] J. Miller. What bloat? cartesian genetic programming on boolean problems. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, San Francisco, California, USA, 9-11 July 2001.

[58] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[59] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[60] T. Motoki. Calculating the expected loss of diversity of selection schemes. *Evol. Comput.*, 10:397–422, December 2002.

[61] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evol. Comput.*, 1:25–49, March 1993.

[62] H. Mühlenbein and D. Schlierkamp-Voosen. The science of breeding and its application to the breeder genetic algorithm (bga). *Evol. Comput.*, 1:335–360, December 1993.

[63] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nvidia CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, Dec. 2009. Special issue on parallel and distributed evolutionary algorithms, part I.

[64] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[65] P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[66] P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances*

in Genetic Programming 3, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[67] nVidia. Cuda compute unified device architecture programming guide v1.1. `http://www.nvidia.com/object/cuda_develop.html`, 2007.

[68] C. Perez-Miguel, J. Miguel-Alonso, and A. Mendiburu. Evaluating the cell broadband engine as a platform to run estimation of distribution algorithms. In *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, pages 2491–2498, New York, NY, USA, 2009. ACM.

[69] R. Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, Sept. 1996.

[70] R. Poli. Sub-machine-code GP: New results and extensions. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 65–82, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[71] R. Poli and W. B. Langdon. Running genetic programming backward. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 9, pages 125–140. Springer, Ann Arbor, 12-14 May 2005.

[72] R. Poli and W. B. Langdon. Backward-chaining evolutionary algorithms. *Artificial Intelligence*, 170(11):953–982, Aug. 2006.

[73] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[74] P. Pospíchal, J. Jaroš, and J. Schwarz. Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[75] W. F. Punch. How effective are multiple populations in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 308–313, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[76] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel GP on the G80 GPU. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De

Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 98–109, Naples, 26-28 Mar. 2008. Springer.

[77] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, 2009.

[78] D. Sankoff. Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A.*, 69:4–6, 1972.

[79] S. Shirakawa, S. Ogino, and T. Nagao. Graph structured program evolution. In H. Lipson, editor, *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1686–1693. ACM, 2007.

[80] N. T. Siebel, A. Jordt, and G. Sommer. Accelerating neuro-evolution by compilation to native machine code. In *International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain, 18-23 July 2010. IEEE Press.

[81] N. Soca, J. L. Blengio, M. Pedemonte, and P. Ezzatti. Pugace, a cellular evolutionary algorithm framework on gpus. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[82] G. Squillero. MicroGP - an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, Sept. 2005. Published online: 17 August 2005.

[83] H. Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[84] H. Sutter. *More exceptional C++: 40 new engineering puzzles, programming problems, and solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[85] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[86] A. Teller. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 5 Dec. 1998.

[87] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of*

the Second Annual Conference, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[88] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[89] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2523–2530, New York, NY, USA, 2009. ACM.

[90] Turton, Openshaw, and Diplock. Some geographic applications of genetic programming on the Cray T3D supercomputer. In C. R. Jesshope and A. V. Shafarenko, editors, *UK Parallel'96*, pages 135–150, University of Surrey, 3-5 July 1996. Springer.

[91] Z. Vasicek and L. Sekanina. Hardware accelerators for cartesian genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 230–241, Naples, 26-28 Mar. 2008. Springer.

[92] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*. Accepted for future publication.

[93] J. A. Walker and J. F. Miller. Evolution and acquisition of modules in cartesian genetic programming. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 187–197, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[94] J. A. Walker and J. F. Miller. Investigating the performance of module acquisition in cartesian genetic programming. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1649–1656, Washington DC, USA, 25-29 June 2005. ACM Press.

[95] J. A. Walker, J. F. Miller, and R. Cavill. A multi-chromosome approach to standard and embedded cartesian genetic programming. In M. Keijzer, M. Cattolico, D. Arnold, V. Babovic, C. Blum, P. Bosman, M. V. Butz, C. Coello Coello,

D. Dasgupta, S. G. Ficici, J. Foster, A. Hernandez-Aguirre, G. Hornby, H. Lipson, P. McMinn, J. Moore, G. Raidl, F. Rothlauf, C. Ryan, and D. Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 903–910, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[96] J. A. Walker, K. Völk, S. L. Smith, and J. F. Miller. Parallel evolution using multi-chromosome cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 10:417–445, December 2009.

[97] D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the third international conference on Genetic algorithms*, pages 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[98] G. Wilson and W. Banzhaf. Linear genetic programming GPGPU on microsoft's xbox 360. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[99] G. Wilson and W. Banzhaf. Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms. *Genetic Programming and Evolvable Machines*, 11(2):147–184, June 2010.

[100] G. C. Wilson and W. Banzhaf. A comparison of cartesian genetic programming and linear genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 182–193, Naples, 26-28 Mar. 2008. Springer.

[101] G. C. Wilson and W. Banzhaf. Deployment of CPU and GPU-based genetic programming on heterogeneous devices. In A. I. Esparcia, Y. ping Chen, G. Ochoa, E. Ozcan, M. Schoenauer, A. Auger, H.-G. Beyer, N. Hansen, S. Finck, R. Ros, D. Whitley, G. Wilson, S. Harding, W. B. Langdon, M. L. Wong, L. D. Merkle, F. W. Moore, S. G. Ficici, W. Rand, R. Riolo, N. Kharma, W. R. Buckley, J. Miller, K. Stanley, J. Bacardit, W. Browne, J. Drugowitsch, N. Beume, M. Preuss, S. L. Smith, S. Cagnoni, J. DeLeo, A. Floares, A. Baughman, S. Gustafson, M. Keijzer, A. Kordon, C. B. Congdon, L. D. Merkle, and F. W. Moore, editors, *GECCO Workshop on Computational intelligence on consumer games and graphics hardware (CIGPU-2009)*, pages 2531–2538, Montreal, 8-12 July 2009. ACM.

[102] A. Wirawan, C. Kwoh, N. Hieu, and B. Schmidt. Cbesw: Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9(1):377, 2008.

[103] M. L. Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2515–2522, New York, NY, USA, 2009. ACM.

[104] M.-L. Wong, T.-T. Wong, and K.-L. Fok. Parallel evolutionary algorithms on graphics processing unit. In D. Corne, Z. Michalewicz, B. McKay, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, G. Raidl, K. C. Tan, and A. Zalzala, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2286–2293, Edinburgh, Scotland, UK, 2-5 Sept. 2005. IEEE Press.

[105] J. R. Woodward. Complexity and cartesian genetic programming. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 260–269, Budapest, Hungary, 10 - 12 April 2006. Springer.

[106] H. Xie, M. Zhang, and P. Andreae. Another investigation on tournament selection: modelling and visualisation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1468–1475, New York, NY, USA, 2007. ACM.

[107] Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In L. Wang, K. Chen, and Y.-S. Ong, editors, *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, volume 3612 of *Lecture Notes in Computer Science*, pages 1051–1059, Changsha, China, Aug. 27-29 2005. Springer.

[108] M. Yue. A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1, \forall l$ for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica (English Series)*, 7:321–331, 1991.