TMBL Kernels for CUDA GPUs Compile Faster Using PTX

Computational Intelligence on Consumer Games and Graphics Hardware

Tony E Lewis
Computer Science and Information Systems
Birkbeck, University of London
London, UK
tony@dcs.bbk.ac.uk

George D Magoulas
Computer Science and Information Systems
Birkbeck, University of London
London, UK
gmagoulas@dcs.bbk.ac.uk

ABSTRACT

Many of the most effective attempts to harness the power of the Graphics Processing Unit (GPU) to accelerate Genetic Programming (GP) have dynamically compiled code for individuals as they are to be evaluated. This approach executes very quickly on the GPU but is slow to compile, hence only vast data-sets fully reap its rewards.

To reduce compilation time, we generate and compile code in the lower-level language PTX. We investigate this in the context of implementing Tweaking Mutation Behaviour Learning (TMBL) on the GPU. We find that for programs of 300 instructions, using PTX reduces the compile time 5.861 times and even increases the evaluation speed by 23.029%.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Performance

Keywords

Tweaking Mutation Behaviour Learning (TMBL), Parallel Thread EXecution (PTX), Graphics Card, Graphics Processing Unit (GPU), CUDA

1. INTRODUCTION

This paper describes an investigation into coding individuals in a lower-level, assembly-like language for evaluating them on the Graphics Processing Unit (GPU). This relates to two lineages of previous research: the use of Central Processing Unit (CPU) assembly or even machine code to encode individuals and the use of GPUs to evaluate them. In both cases, researchers have been motivated by wanting to feed the computational hunger of Genetic Programming (GP) with fast fitness evaluations.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

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 Systems (CGPS) [9].

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. Later work managed to incorporate such functionality as "arithmetic operators, large indexed memory, automatic decomposition into subfunctions and subroutines (ADFs), conditional constructs i.e. if-the-else, jumps loop structures, recursion, protected functions string and list functions" [10]. That 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 [6]. 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 [4].

Squillero's motivation for evolving machine code programs in his μGP system [13] 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 [12]. 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 [2], [8] provide a link between the two research lineages. 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 [8], in the other it was the high-level C-like language of nVidia's Cg framework [2].

In recent years, researchers have started to look to the GPU for more power. Early attempts at exploiting the GPU for GP were data-parallel [1] [3] which is a GPU version of the compiling method described before. Later works employed population-parallel methods which use GPU interpreters [5] [11]. In contrast to the situation with the CPU, compiling methods on the GPU are perhaps simpler than interpreting methods.

The evaluation speeds achieved by data-parallel methods are often remarkable but are marred by the CPU time spent on compiling kernels. This problem is minimal when the data-set is vast because then each kernel is evaluated on very many test-cases and so the evaluation time is much larger than the compilation time. For more reasonably sized data-sets, the compilation time ruins the benefits of data-parallel methods.

This paper explores the possibility of moving one level deeper, from the C style code of Compute Unified Device Architecture (CUDA) to assembly style Parallel Thread EXecution (PTX) code. The aim of this is to reduce the compile time so that reasonably sized data-sets can still get the benefit of data-parallel evaluation speeds. Any gain in evaluation speed would be a side benefit.

2. TMBL

The form of Evolutionary Computation (EC) used for this paper is Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble"), which has been proposed as a baby sister to GP [7]. The key feature of TMBL is its focus on long term fitness growth above all else. 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 preplanning, 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 it's necessary 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 it. 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.

Some of the features that make TMBL's representation distinct from standard linear GP also happen to make it more suitable for this task. For GPU work, we want the execution of different threads to diverge as little as possible. Linear-style branching is bad for this 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 and 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 be experience slower evaluation speeds.

3. 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. This research followed the precedent of much of the recent work in the field by using nVidia's CUDA framework to exploit the GPU.

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 but for simplicity, it will 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 1. 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()`.

As indicated in Figure 1, `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 API but must be accessed through the lower-level CUDA driver API.

One of the advantages of the data-parallel approach is that it naturally leads to excellent memory access patterns that are favourable to the fastest possible access to off-chip global memory. This is worth highlighting because the GPU code used in work doesn't read test-cases but calculates them dynamically instead. This may make the evaluation speeds higher than they would otherwise be but the excellent access patterns give reason to hope this effect is small. Furthermore, there is no reason to think that this effect favours the novel techniques over the standard techniques.

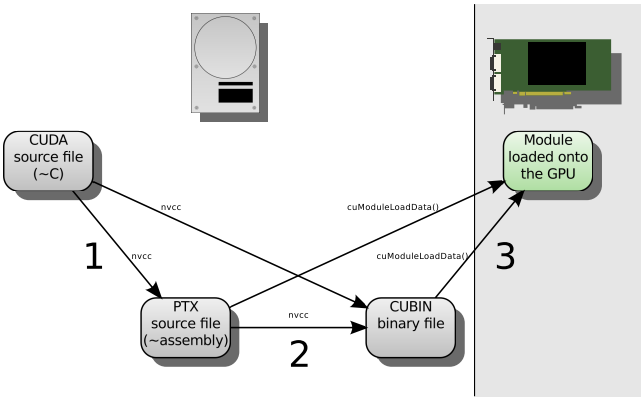


Figure 1: The steps required to compile and load source code into a callable GPU module. The three grey rectangles on the left represent files, as indicated by the hard drive icon; the green rectangle on the right represents a GPU module, as indicated by the graphics card icon. These steps will be referred to throughout the paper. 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.

4. 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 1 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 languages: whitespace may be used freely and is ignored (except in separating tokens); semi-colons separate lines; lines beginning with a `#` character are preprocessor directives and commenting rules are as for C/C++ (`/*` and `*/` mark comment blocks and `//` marks rest-of-line comments).

Table 1 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. It is worth taking a look at some of the basics of the language.

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 5 used 32-bit floating point numbers (`.f32`) for the evaluation's native type, 32-bit unsigned integers (`.u32`) for some of the admin 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 (ie `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.

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 1: 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. 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.

PTX offers its programmers two divergence-minimising tools:

- explicit conditional instructions (such as `selp` and `slct` in Table 1), 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 test-cases (with appropriate padding) such that CUDA thread blocks (or at least warps) only evaluate data for one individual. If the number of test-cases 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 passing the `--ptxas-options=-v` option to `nvcc` which causes verbose output. This shows that the compiler often uses far fewer registers than are directly implied by the PTX source.

5. EXPERIMENTS

The tactic of using PTX introduces more complexity and so can only be justified if it either reduces compile 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.

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 provided in Table 2 and the default parameters are provided in Table 3. The last three entries refer to the process used to assess evaluation speed. This involved timing eight consecutive launches of kernels, each executing the entire population for 50 iterations over 65536 test-cases. 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.

Operating system	Ubuntu Linux 10.10
Linux Kernel	2.6.35-28-generic-pae
GPU Device	nVidia GeForce GTX 260 (Core 216)
CUDA toolkit	v3.2
Device driver	260.19.44
NVCC	v0.2.1221

Table 2: The system configuration

Number of CPU threads	1
Individuals per kernel	8
Number of individuals	120
Number of instructions	200
Number of evaluation repeats	8
Number of test-cases per evaluation	65536
Number of iterations per evaluation	50

Table 3: Default parameters for the runs

Each experiment averaged results over five repetitions. The lines on the graphs in Section 6 all have a bar behind them indicating the mean plus and minus one estimated standard error. This bar is often too thin to be visible. This suggests that the relatively small number of repetitions has been adequate to give good estimates of the means.

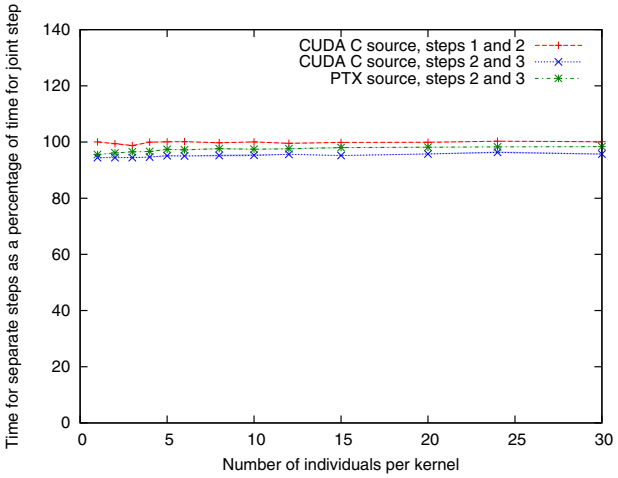
6. RESULTS

In 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 duration and that the load time is small enough to disregard. The validity of these assumptions is checked in Figure 2. In both cases, the values are plotted over varying number of individuals per kernel although this is not too important in either case. Figure 2(a) shows that performing the steps in pairs does not massively affect the duration. Pairing steps 2 and 3 appears to slightly increase the duration but this effect is minor and will henceforth be disregarded. Figure 2(b) shows that the load times are very short (compared to compile times from other experiments) and so will be disregarded for the rest of the analysis.

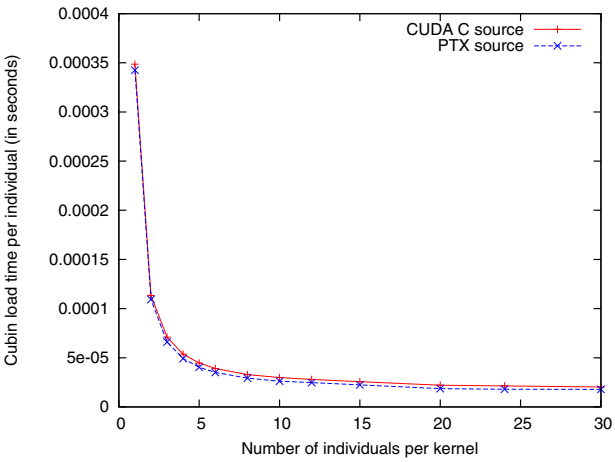
Figure 3 considers the effect of varying the number of individuals per kernel on evaluation speed and compile time. Figure 3(a) shows two striking results: that the evaluation speeds are remarkably high and that they are even higher from PTX source code. The results are fairly steady across varying numbers 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 3(b) shows that this has been achieved effectively too. Complete compile 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 compile time for CUDA C from PTX to cubin suggests that the reduction in compile 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.

Figure 4 shows the effect of varying the total number of programs on evaluation speed and compile time. As would be hoped, neither is hugely affected. However compile times per individual for CUDA C source do vary slightly, with the

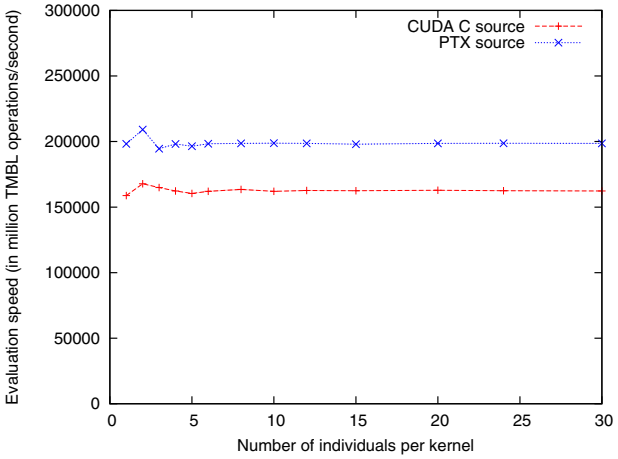


(a) Linearity of combining pairs of steps

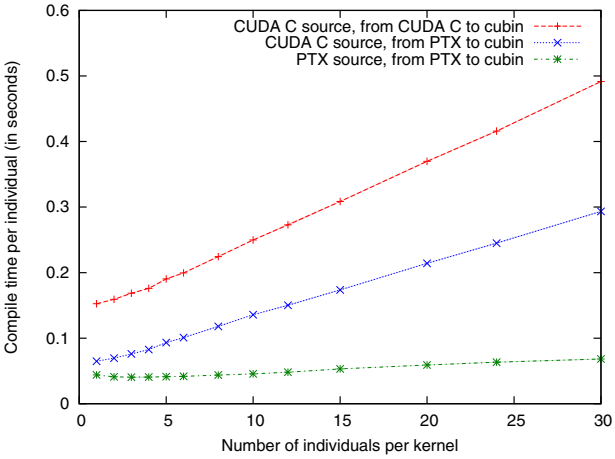


(b) Cubin load time per individual

Figure 2: The steps' linearity and the load times over varying numbers of individuals per kernel for CUDA C and PTX



(a) Evaluation speed



(b) Compile time per individual

Figure 3: The effect of varying numbers individuals per kernel for CUDA C and PTX

time to cubin higher for particularly small or large populations.

Figure 5 shows the effect of varying the number of instructions per TMBL individual. Figure 5(a) shows 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.

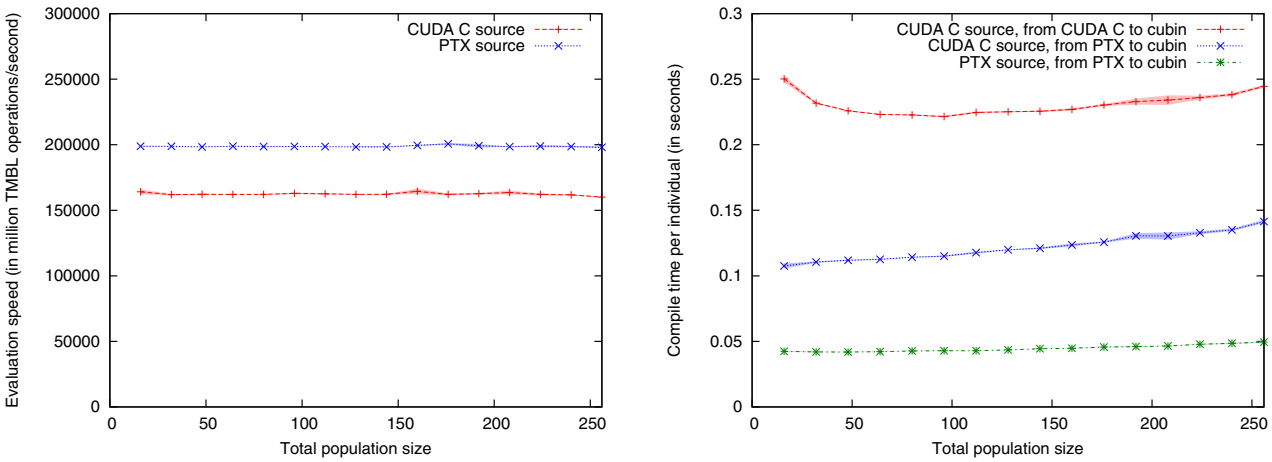
Figure 5(b) shows that the compile time to cubin per individual is much lower for PTX than for CUDA C for all numbers of instructions. For both PTX and CUDA C, the compile times per individual appear to be increasing more than linearly with respect to the number of instructions. This might be because 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 compile time per individual is 0.473 seconds from CUDA C and 0.081 seconds from PTX, a decrease of 82.938%.

Figure 6 shows the effects of compiling with multiple threads on a 4 core machine. In all cases, compile 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.

7. CONCLUSION

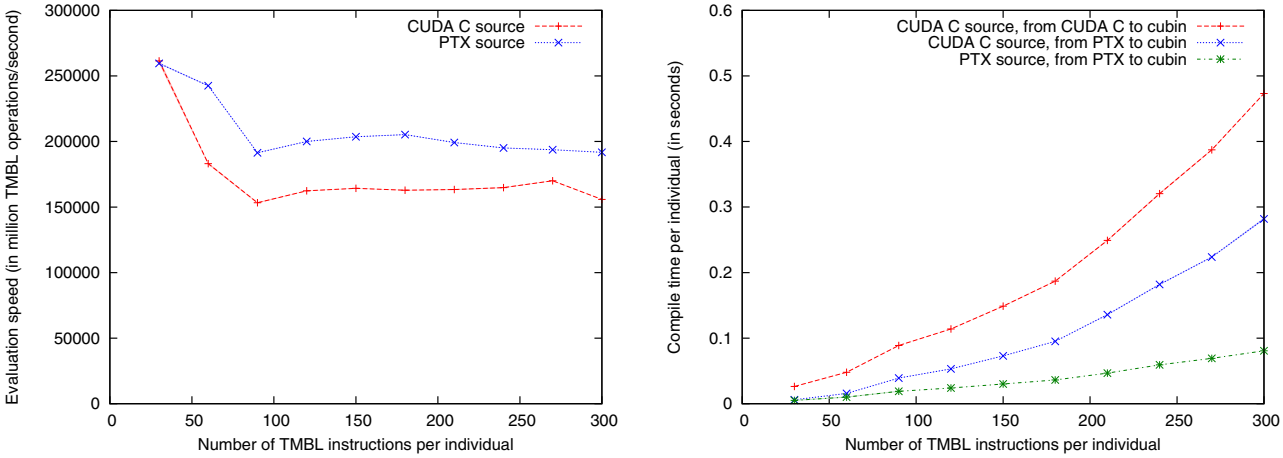
This paper described an investigation into using the low-level language PTX for data-parallel GPU evaluation, rather than the more standard CUDA C. An initial consideration of the idea revealed that PTX is a forward compatible, well-documented language, usable enough for small code bases such as data-parallel kernels. An implementation of the idea demonstrated two advantages to the approach, illustrated here with values derived from individuals of 300 instructions:

- considerably shorter (5.861×) compile times;
- higher resulting evaluation speeds (+23.029%).



(a) Evaluation speed (b) Compile time per individual

Figure 4: The effect of varying population sizes for CUDA C and PTX



(a) Evaluation speed (b) Compile time per individual

Figure 5: The effect of varying numbers of TMBL instructions per individual for CUDA C and PTX

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 problem can be avoided by performing intron removal before evaluation.

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 compile 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 in this paper, 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 directly manipulate cubin binaries 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. Nev-

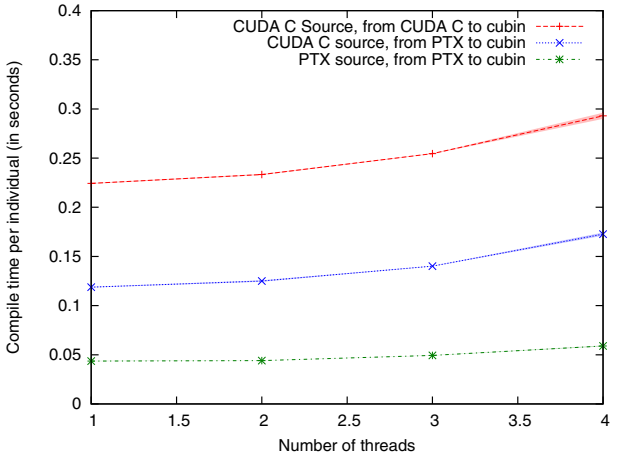


Figure 6: Using multiple threads for compilation

ertheless, 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.

8. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] G. Squillero. MicroGP - an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, Sept. 2005. Published online: 17 August 2005.