Technical Report TR-09-05, 18 June 2009, Department of Computer Science, King's College London
Revised entry to GPUs for Genetic and Evolutionary Computation Competition, GECCO 2009, 8 July.

A CUDA SIMT Interpreter for Genetic Programming

W. B. Langdon CREST centre, King's College, London, WC2R 2LS, UK

ABSTRACT

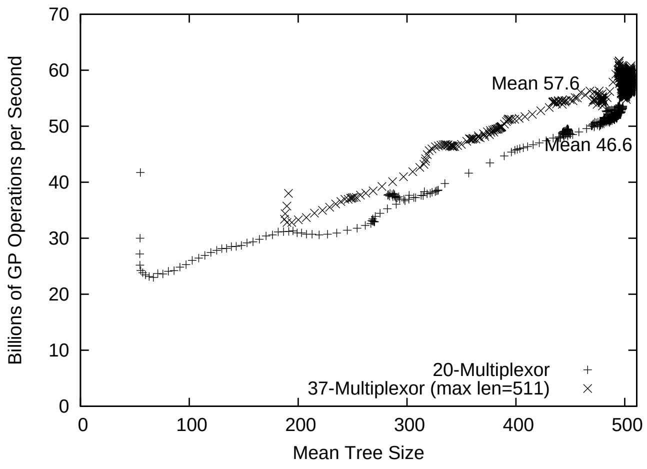
A Single Instruction Multiple Thread CUDA interpreter provides SIMD like parallel evaluation of the whole GP population of $\frac{1}{4}$ million RPN expressions on graphics cards and nVidia Tesla T10P. Using sub-machine code GP a sustain peak performance of 212 billion GP operations per second (3300 speed up) and an average of 4.5 peta GP ops per day is reported for a single card on a Boolean induction benchmark never attempted before, let alone solved.

1. INTRODUCTION

There are two main approaches to running genetic programming on highly parallel hardware such as GPUs: 1) compiling evolved programs and running multiple fitness cases in parallel, 2) interpreting multiple programs in parallel. The compiled approach suffers from the overhead of running the compiler. However recent work [1] demonstrates parallel compilation of the GP population on multiple workstations. Interpreters can run programs immediately but interpreted code is slower than optimised compiler generated machine code. GPU interpreters typically gain their speed by evaluating the whole population in parallel but are also free to run fitness cases in parallel or mixtures of the two approaches.

SIMD GPU interpreters evaluate each GP tree by treating it as a reverse polish (RPN) expression which is evaluated via a stack in single pass. I.e. without the recursive back tracking normally associated with trees. The stack required careful implementation in RapidMind but is straight forward with nVidia CUDA. For every instruction, SIMD interpreters use cond or if branches to skip through the whole instruction set and only evaluate the current instruction. (This causes threads to diverge, so we also tried a data flow approach in which the ifs were replaced by evaluating all possibilities and using array indexes to chose from them. However it was not faster.) The new approach uses the full power of CUDA to gain the best performance from G80 parallel hardware. E.g. by using shared and constant memory, where possible, in preference to local and global memory.

The approach is suitable for use with many types of GP however we demonstrate it on two Boolean benchmark problems (20-multiplexor and 37-multiplexor) where CUDA allows access to another level of parallelism. Sub-machine code GP uses parallel bit or byte level operations, to execute up to 32 (or 64) fitness cases simultaneously. Using pseudo random sampling of test cases with a population of a quarter of a million programs a single T10P Tesla is able to solve the 20-multiplexor problem in less than an hour, whereas [2] estimated it would take more than 4 years. Peak sustained performance of just over 212 billion GP operations/second was achieved when testing all $2^{37} = 137$ billion fitness cases for solutions to the 37-multiplexor. Probably compiled code would be still faster. When including all activity on the CPU as well as the GPU across the whole run the single



Tesla averaged more than 52 billion GPop/s. This is more than [1] measured for a compiled approach using a cluster of 14+ workstations each equipped with a low end GPU.

2. ALGORITHM

This is the first genetic programming implementation to exploit sub-machine code level parallelism inherent in every GPU. However it can obviously be used in any Boolean problem. Indeed many non-evolutionary algorithms with a large logic based component could benefit from this approach to exploiting bit-level parallelism. The sub-machine code approach has also been used in the continuous domain (by using 8-bit precision) and in graphics (e.g. 5×5 OCR). It is straight forward to implement in CUDA compared to other high-level GPGPU languages like RapidMind 2.

The genetic programming individuals are created and manipulated by crossover and mutation as RPN expressions in exactly the same format as is used by the GPU. I.e. the data is not converted between the host CPU and the GPU.

The host CPU accounts for only 9% of the total run time. Therefore no great effort has been spent on optimising the host side C++ code. Doubtless small efficiencies could be made to reduce the host side overhead.

Inspection of the nvcc compiler output suggests there are 100 PTX instructions in the main loop. On average about 30 will be executed per GP primitive. It takes the T10P 652 seconds to run a solution containing 1005 primitives 2^{37} times (when the interpreter achieves its peak performance). Taking into account that 32 test cases are run simultaneously and the T10P has 192 1.08 GHz cores, this means about one PTX instruction is executed per clock tic per core. Suggesting the T10P is fully loaded.

Although every effort has been made to get the best from the T10P Tesla's 192 cores, the CUDA code should run on any modern G80 GPU. The current implementation requires that the whole stack be stored in shared memory. This limits the number of CUDA threads per block. With smaller shared memory it would not be possible to have the 192 threads needed to fully use the GPU.

The interpreter architecture fits the GPU philosophy and has enabled us to solve a GP benchmark never even attempted before let alone solved.

3. SPEED

The best speed (212 10^9 GPop/s) is achieved by loading a single program into constant memory. We did experiment with loading the GP population into shared memory, at the expense of reducing the memory available for the stack etc. and hence reducing the number of threads per block. We had anticipated that reading the programs to be interpreted from on-chip shared memory would be enormously faster than reading it from off-chip global memory. However the advantage of shared v. global memory is not large. This appears to due to hardware speed up techniques like coalesced reads (and possibly caching, although the manual suggests a cache is not used) and the large number (262 144) of threads in use. Hence for the normal GP population the interpreter places each GP individual in global memory. (The large global memory makes it feasible to allow far bigger programs than are needed to solve the multiplexor problems.)

Other approaches might entail loading a small number programs (rather than the whole population) into shared memory (hence allowing more space for the stack etc.) and allocating multiple fitness evaluation threads to each. The elegant one program one thread approach means threads for short programs may finish before those for longer ones, so decreasing average performance. However when a size limit is imposed on GP trees it is common for them to grow towards this size, first increasing size variation but ultimately leading to populations of similarly sized programs (see Figure on first page).

The interpreter gives on average 52 billion GPop/s compared to about 800 million GPop/s we previously reported. Most of the 60 fold speed comes from the use of sub-machine code GP. Compared to 3GHz CPU running 32-bit sub machine code GP we get a 34 fold speed up (800 fold, 3300 peak, if it does not use sub machine code GP).

Using CUDA makes it easy to use sub-machine code GP, to expand the stack and to direct the placement of data structures to on-chip memory.

While Koza initially used a tree depth of 17 a stack depth of 16 is sufficient for most GP experiments. The interpreter has been used with multiple arity experiments. For GP primitives which take more than two inputs (e.g. if) the maximum stack depth can be more than the maximum tree depth. We can enforce a stack limit which is different from the tree depth limit. Alternatively the existing tree depth limit can be retained and the kernel configured to allow a bigger stack. However because the stack resides in shared memory, this means each block can have fewer threads.

In GPU/Tesla with larger shared memory, the number of threads can be increased above 240. This may increase performance by allowing more threads to further conceal global memory latency. In GP large populations are common so the one GP individual per GPU thread model can easily take advantage of more GPU cores. Conversely in smaller GPUs, having fewer cores will only reduce performance approximately linearly. With very small GP populations the interpreter would easily allow multiple fitness cases to be spread across multiple threads.

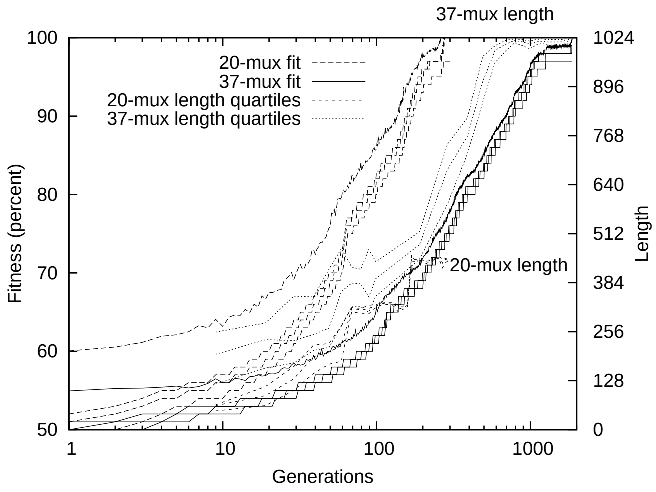


Figure 2: Evolution of fitness and size in 20-multiplexor and 37-multiplexor. 3 lengths are quartiles and median, showing bloat and small range of sizes. Fitness quartiles are given to 1%, hence their stepped apparence. The log-linear rise in fitness is reminiscent of the coupon collector suggesting major building blocks are equally difficult.

Table 1: CUDA GP for 20 and 37 multiplexor

Terminals:	20 or 37 Boolean inputs D0–D36
Functions:	AND, OR, NAND, NOR
Fitness:	Pseudo random sample of 2048 of 1 048 576 or 8192 of 137 438 953 472 fitness cases.
Tournament:	4 members run on same random sample.
Population:	262 144
Initial pop:	Ramped half-and-half 4:5 (20-mux) or 5:7
Parameters:	50% subtree crossover, 5% subtree 45% point mutation. Max depth 16, max size 511 or 1023.
Termination:	20 000 generations

4. IMPACT

The CUDA code will be made available via FTP.

The 37-mux has 137 billion fitness cases. It have never been attempted before, let alone solved.

We have used random numbers generated on the GPU to sample the test cases. Yet the CUDA SIMT interpreter gave us the power to show all the evolved solutions do generalise.

Currently Tesla are available with up to 960 cores, suggesting a further increase in performance of at least 5 fold might be possible.

Acknowledgment

The Tesla T10P early engineering sample was supplied by nVidia.

I am greatful for the assistance of Timothy Lanfear and Gernot Ziegler of nVidia and Simon Harding of Memorial.

5. REFERENCES

[1] HARDING, S. Personal communication, May 2009.

[2] YANAGIYA, M. Efficient genetic programming based on binary decision diagrams. In *IEEE CEC* 1995 234–239.