EuroGP-2010, Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum and A. Sima Uyar, *Eds.*, LNCS 6021, p146–158, Istanbul, 7-9 April, Springer.

A Many Threaded CUDA Interpreter for Genetic Programming

W. B. Langdon

CREST centre, Department of Computer Science,
King's College, London, Strand, 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 reverse polish notation (RPN) expressions on graphics cards and nVidia Tesla. Using sub-machine code tree GP a sustain peak performance of 665 billion GP operations per second (10,000 speed up) and an average of 22 peta GP ops per day is reported for a single GPU card on a Boolean induction benchmark never attempted before, let alone solved.

1 Introduction

There are two main approaches to running genetic programming [10,1,17,20] on highly parallel hardware such as GPUs: 1) compiling evolved programs and running multiple fitness cases in parallel [7,3] 2) interpreting multiple programs in parallel [15,21,16,23,18,4]. The compiled approach suffers from the overhead of running the compiler on the host computer. However Harding [8] has recently

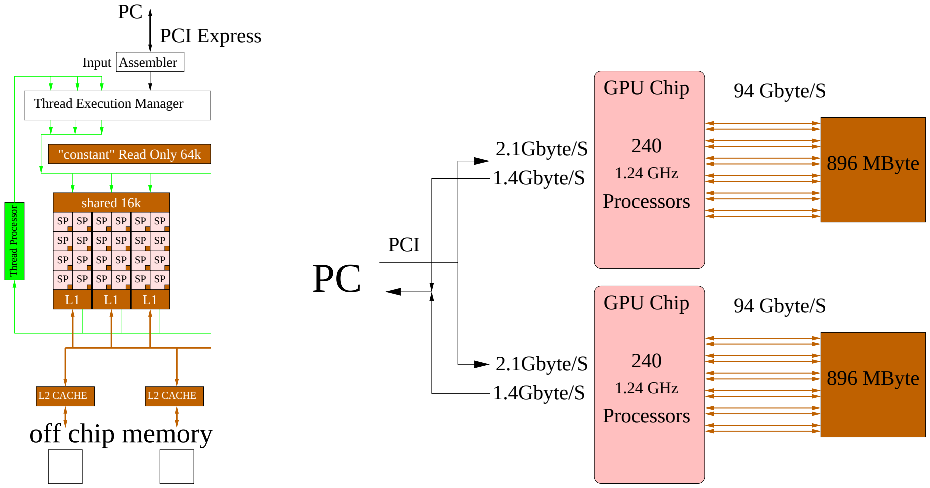

Fig. 1. Left: nVidia G80 GPU multi processor. Right: A GeForce 295 GTX contains 2×10 multi processor on two chips. Each stream processor (SP) obeys the same instruction at the same time. However each has its own registers and its own access to shared and constant memory. For efficiency multi-processors try to coalesce multiple separate access to off chip memory into a single access.

demonstrated 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, as we shall see, GPUs can also run fitness cases in parallel, or mixtures of the two approaches [11].

The essential feature of parallelism in current generation graphic processing units is that they are intended to run programs on multiple data. Graphical applications often require the rapid real time transformation of many data items. This can be performed efficiently in parallel because essentially the same transformation is applied to each datum and the data do not interact. E.g. the two dimensional appearance of a complex three dimensional scene is calculated by using one program to calculate the appearance of the many thousands of three dimensional elements independently. Separate programs are used to deal with cases where elements overlap or obscure each other.

High end GPUs typically contain a few multi-processors, each of which operate in parallel. Each multi-processor is a tightly integrated unit and in some ways resembles the earlier single instruction multiple data (SIMD) parallel computers. Both provide a limited form of parallelism which is convenient to implement in hardware. The hardware gains its speed by having many stream processing units doing the same operation at the same time on different data. See Figure 1. However unlike MasPar SIMD supercomputers of twenty years ago, GPUs are mass market consumer electronics devices for computer games and priced for the hobbyist not the corporation. Hundreds of millions of GPU have been sold rather than approximately 250 MasPar MP-2.

In GPU terminology each stream processor is running a thread. At any instant, all the threads do the same thing. But this raises a problem. What if the program contains branches? E.g. `if(data==0){} else {}`. If the contents of `data` are different in different threads, the hardware will decide either to do the `if` or the `else`. It executes all the threads whose instance of `data` puts them down the same route. The hardware stalls all the other threads. This is known as divergence.[1] At some point the hardware will stop the active threads and restart those it stalled. Eventually the whole program will be run. However divergence is a major source of inefficiency.

nVidia's CUDA has a fairly complicated memory hierarchy. However the most important distinction for performance is the small amount of memory ($\approx$1 megabyte) on the GPU chip and the very much bigger memory on the GPU card ($\approx$1 gigabyte). (Currently the GPU has no direct access to the host computer's RAM. Instead data must be explicitly copied to and from the GPU by the PC. See Figure 1) The delay in reading from off-chip memory is hundreds of times more than access to on-chip memory. This is so big that it makes sense for the hardware to pause threads which are waiting for off-chip data and start

[1] Divergence can be avoided by a data flow approach in which the ifs are replaced by evaluating all possibilities and using array indexes to chose from them. However interpreting a single GP individual across multiple test cases can be faster.

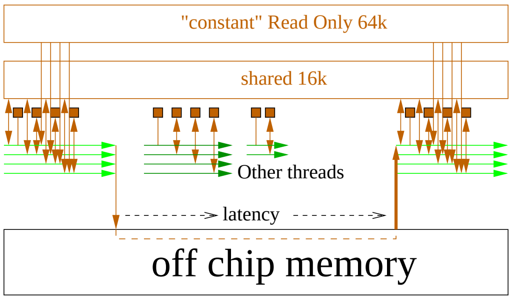

Fig. 2. Reading from onboard GPU memory causes active threads to stall possibly allowing other threads to be active. In contrast access to small areas of read only and shared memory are very much faster. The interpreter stack is placed in shared memory. To reduce bank conflicts stacks can be interleaved to use every 16^{th} memory word.

others which are ready to go. See Figure 2. The hardware can seamlessly handle many thousands of threads. (In the case of the 37-Mux we use 262 144 threads.) This all happens transparently for the CUDA programmer.

A single instruction multiple data (SIMD) interpreter for GP was originally proposed by Hugues Juille [9] for the MasPar MP-2 computer. It has recently been used for nVidia GeForce 8800 graphics hardware by ourselves [15] and Robilliard [21]. These 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 2 [15] 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.

The SIMD 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. Submachine code GP uses parallel bit or byte level operations, to execute up to 32 (or 64) fitness cases simultaneously [19]. Using pseudo random sampling of test cases with a population of a quarter of a million programs a single GPU is able to solve the 20-multiplexor problem. Peak sustained performance of just over 445 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 295 GTX averaged 254 billion GPop/s. In contrast Harding [8] measured, for a compiled approach using a cluster of 14+ workstations each equipped with a low end GPU, a best peak rate of 12.74 billion GP OP/sec for Cartesian GP on a data intensive graphics task.

Table 1. Genetic Programming Parameters for Solving 20 and 37 Multiplexors

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. New samples for each tournament and each generation.
Population:	262 144
Initial pop:	Ramped half-and-half 4:5 (20-Mux) or 5:7 (37-Mux)
Parameters:	50% subtree crossover, 5% subtree 45% point mutation. Max depth 15, max size 511 (20-Mux) or 1023 (37-Mux).
Termination:	5 000 generations

2 Genetic Programming Benchmarks

The original intentions was simply to use the 20 Boolean Multiplexor [10] as an impressive demonstration of the GPU. After all it has never been solved by a tree GP before. ([24] used a totally different representation.) The details of our GP are given in Table 1. The choice of population size was motivated by the capacity of the GPU. While the terminal and function sets are those often used for the even parity benchmark [10]. The resulting evolutions are plotted in Figure 3. Solutions are found in generation 423 (20-Mux) and 2866 (37-Mux).

3 RPN GPU Sub-machine-code Genetic Programming

This is the first genetic programming implementation to exploit sub-machine code level parallelism inherent in every GPU. Indeed it is the first time sub-machine-code GP has been used with reverse polish expressions. 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) [19]. It is straight forward to implement in CUDA compared to other high-level GPGPU languages like RapidMind 2.

4 Genetic Programming on the Host Computer

The GPU is only used for fitness evaluation. When a generation has been interpreted the fitness values of the current individuals are returned to the host. All other operations (crossover, mutation, selection, gathering statistics etc.) are performed by the Linux host computer.

The genetic programming trees are created and manipulated by crossover and mutation as Reverse Polish Notation (RPN) expressions. This is exactly the same format as is used by the GPU. I.e. the data is not converted between the host CPU and the GPU.

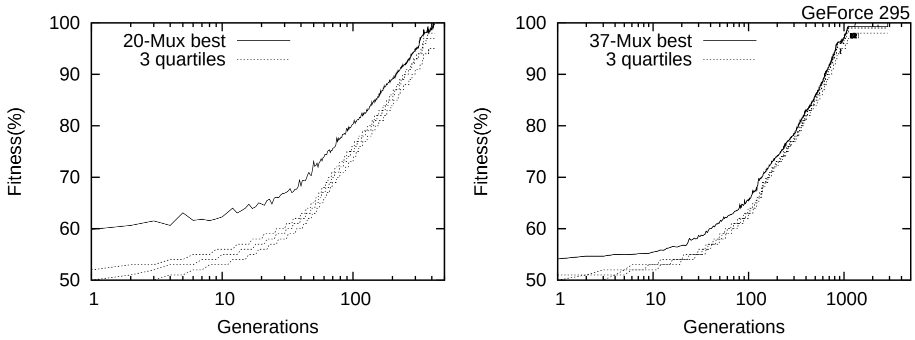

Fig. 3. Evolution of fraction of test cases passed when solving the 20-multiplexor and 37-multiplexor. Dotted lines show three quaters of the population evolves to have fitness near that of the best. (The worst in population, not shown, also starts near 50% but falls towards zero, almost mirroring the best. This may be due to tiny programs being generated by subtree crossover [13, Figure 6] which have poor fitness.) The log-linear rise in fitness over most of the evolution is reminiscent of the coupon collector suggesting major building blocks are equally difficult. This 295 run found a 37-Mux solution in gen 1325 v. 2865 for the Tesla.

It is common in efficient C++ genetic programming implementations for run time to be totally dominated by the time taken for fitness evaluation and so crossover etc. can be discounted. However for the 37-Mux, due to the speed of the interpreter on the 295 GTX these normally inexpensive operations amount to 43% of the total run time. As fitness evaluation in the 20-Mux is less computationally demanding, this rises to 73%. Since the interpreter has been our focus, no effort has been spent on optimising the host side C++ code. Doubtless some efficiencies could be made to reduce the host side overhead.

Lewis proposed [18] a nice scheme with two GPUs which uses overlapping threads on a quad core computer to ensure both GPUs and CPUs are kept busy and says overlapping execution gave almost a threefold speed increase. He says his twin 112 core super clocked and overclocked GPUs gave up to 4 billion GP operations per second for his cyclic Cartesian GP system.

5 Randomised Test Suite Sub-sampling

The final research area was to use the 20-Mux to demonstrate statistically sound sampling [12,22]. We devised CUDA code which randomly generated samples [14] and tested all members of the same tournament on them. It continued to do this until statistical tests could demonstrate one of the four candidates was better than the other three. While successful, this was eventually abandoned for three reasons. 1) As the population converged, more and more tests would be required to reliably differentiate between the best and second best candidates. Indeed it was even considered adding a statistical test to stop evaluation if it was probably that there was no difference between the best two candidates. 2) The number of random samples needed is highly variable. Since we were using a single thread per program at the time, this lead to many cases where all but the last four programs on a multi-processor had finished. Thus most of the multi-processor was idle. Yet it could not be reassigned to other tasks untill the last four had finished. (Multiple threads will be discussed in Section 7.) 3) However the most compelling reason was we realised that sophisticated eradication of 99% of chance was not needed.

If a fixed number of samples are used, some tournaments are settled by chance. This means sometimes individuals are selected to be parents who would not have won (and so would have died childless) if all test cases had been run. Nevertheless the addition of limited noise in the selection scheme did not prevent solutions from evolving. The size of the sample was set by starting with a power of two and doubling it until a solution was found. For 20-Mux only 2048 samples of 1 048 576 were enough. Whereas for the 37-Mux, 8192 were sufficient.

6 CUDA Code

A fragment showing the main interpreter loop C++ code is given in Figure 4. This CUDA kernel runs in parallel simultaneously in thousands of different threads. Figure 4. shows the main data structures used by sub-machine code GP. Reverse polish expressions are evaluated sequentially from the start to the end (indicated by OPNOP). Terminals are pushed onto the stack (which is in `__shared__` memory). In sub-machine code GP the first five inputs correspond to different 32-bit patterns (read from `__constant__ train`). The other inputs cause either 32 0s or 32 1s to be pushed onto the stack. The binary Boolean functions pop both their 32-bit arguments from the stack and push their 32-bit result back onto the stack. `runprog` leaves its answer on the top of each thread's stack.

Where there are not enough threads to permit all fitness cases to be run in parallel (i.e. `runprog` is used serially) it might be advantageous to copy `Pop` onto the chip itself. E.g. when proving the evolved solution on all 2^n fitness cases, it is copied to constant memory. (All the solutions have also been verified by extracting them and running them in a conventional computer.) Lewis reports [18] success with loading the population into shared memory. However shared memory is very limited and so we use all of it to hold the stacks rather than read-only cache copies of the programs. It appears to be more important to put the stacks close to the stream processors since they are both read and written to and used repeatedly. Indeed, given sufficient threads, the programs are only read once (so a cache is pointless). In cases where the programs are very small (so the stacks are also small) and each is run many times, it might be advantageous to use some of the on chip (i.e. constant or shared) memory to cache the population.

While Koza initially used a tree depth of 17 [10], in order to interpret 256 programs (i.e. 256 stacks) per multi-processor simultaneously, the stack size was dropped to 15. (Occupying $15 \times 256 = 3840$ of the 4032 available `int`.) Fortunately there are solutions to both benchmarks which can be evaluated with stacks of only 15 and GP is able to find them.

```
__constant__ const unsigned int train[8] =
  {0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00,0xffff0000,0,0,0};
extern __shared__ unsigned int shared_array[];
#define stack(sp) Stack[(sp)*blockDim.x+threadIdx.x]
__device__ inline void runprog(unsigned char* const Pop, const unsigned int prog,
                               const unsigned int test32, const int LEN) {
#define AND(A,B)  ((A) & (B))
#define OR(A,B)   ((A) | (B))
#define push(x) {stack(SP) = x; SP++;}
  unsigned int* Stack = shared_array;
  int SP = 0;
  for(unsigned int PC = 0;; PC++){
    const optype opcode = Pop[PC+(prog*LEN)]; //SETOPCODE;
    if(opcode==OPNOP) break;

    const int r = opcode - firstinput;
    if((r & (~7))==0) {push(train[r]);} //OP1
    else {
    const int r5 = opcode-inputd5; //ninputs <= 37bits
    if((r5 & (~31))==0) {
    if(test32 & (1<<r5)) {push(0xffffffff);}
    else                 {push(0x00000000);}
    } else {
    const unsigned int sp1 = stack(SP-1);
    const unsigned int sp2 = stack(SP-2);
    SP -= 2;
    switch(opcode) {
      case OPAND:  push( AND(sp1,sp2)); break;
      case OPOR:   push(  OR(sp1,sp2)); break;
      case OPNAND: push(~AND(sp1,sp2)); break;
      case OPNOR:  push( ~OR(sp1,sp2)); break;
}}}}}
```

Fig. 4. C++ CUDA code fragment for the sub-machine code GP SIMD reverse polish expression tree interpreter

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. Either crossover etc. can be modified to enforce a stack limit rather than the conventional tree depth limit. Alternatively the existing tree depth limit can be retained and the corresponding maximum stack depth calculated. The kernel must then be configured to allow this stack size. Typically this means each block can have fewer threads, which will tend to reduce performance.

7 Speed

Performance depends both on the number of fitness cases run in parallel by the interpreter (nparallel) and the the number of copies of the interpreter run in parallel per multiprocessor (block_size). See Figure 5. Each 20-Mux tree is evaluated 64 times on randomly selected inputs. (Remember using sub-machine code GP means each evaluation covers 32 fitness cases, making a total of $32 \times 64 = 2048$.) The interpreter allows nparallel=1, 2, 4, 8, 16, 32 or 64 threads to be used per 20-Mux individual. Since all 64 evaluation must be run, each RPN expression in each thread is evaluated by a for loop 64, 32, 16, 8, 4, 2 or 1 times.

We are limited, by shared memory, to at most block_size=256 threads per multiprocessor. This means that if we test each individual with one thread (i.e. run it sequentially in a for loop 64 times) we can test up to 256 programs in

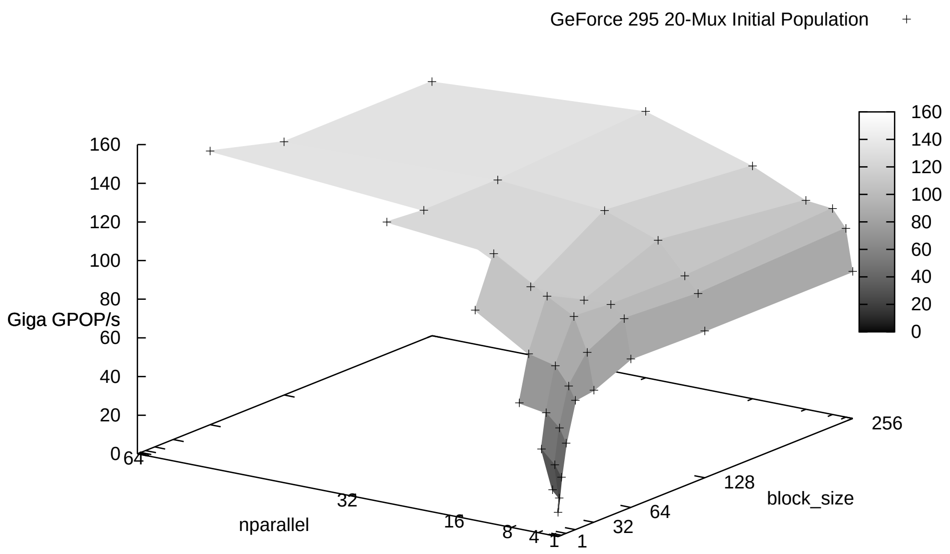

Fig. 5. Speed of interpreter v. number of fitness cases in nparallel threads and number of threads per multi-processor block. (CUDA grid size adjusted so the total number of threads is always $\frac{1}{4}$ million.)

parallel in each multi-processor. If we use two threads per program, we can simultaneously test 128 programs in each multi-processor. And so on until with maximum parallelism per 20-Mux program (i.e. nparallel=64) 1, 2 or 4 programs can be run in a single multi-processor block.

With 64, 128 or 256 threads per block, CUDA is approximately twice as fast when interpreting all 64 fitness cases in parallel compared to running them in sequence but interpreting multiple programs in the same block (see Figure 5). Evaluating the same expression in multiple threads should mean they do not diverge, so we expect better performance. However it is gratifying that the original single program-single thread SIMD approach [15] (which was designed for problems with a small number of fitness cases) gets within 50% of the speed where all the fitness cases are run in parallel. The fact that block_sizes 64, 128 and 256 give much the same performance suggests we are not getting any benefit (such as coalesce reads) by running multiple adjacent programs in the same block.

The interpreter tends to speed up in later generations as the trees get bigger (see Figure 7). Nevertheless the initial random population, i.e. Figure 5, is indicative of the general tradeoff between evaluating trees in parallel and the number of threads per multi-processor (Figure 6).

The slight difference between the two fastest configurations shown in Figure 5 remains small throughout the run. Figure 6 (right) shows maximum parallelism, whereby each program is run only once, is slightly faster (except in generation 0). Hence we use 64 threads per 20-Mux tree. This is consistent with Robilliard *et al.* [21] recommendation to run the interpreter so that each program's fitness cases are run in parallel.

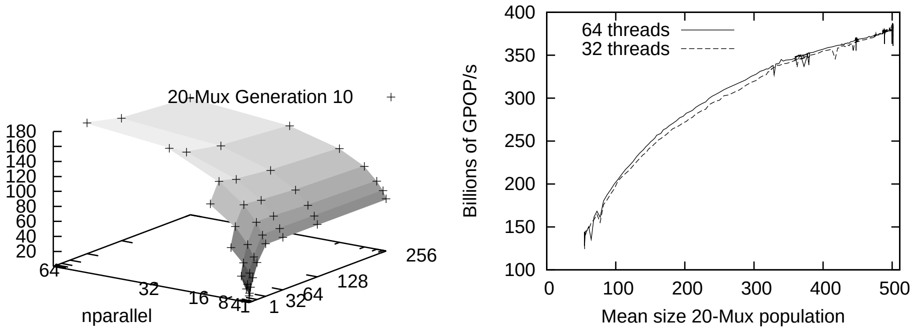

Fig. 6. Left: as Figure 5 but after ten generations. Right: Running all fitness cases in parallel (64 threads) is marginally faster than running each program twice in series with half the fitness cases in parallel (32 threads). Block_size= 64.

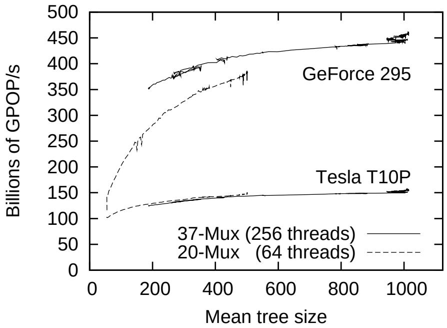

Fig. 7. Speed of CUDA Interpreter on GeForce 295 and early engineering 1.08GHz Tesla T10P v. average 20-Mux and 37-Mux tree sizes. The average speed, including selection crossover etc., for the 20-Mux is 96 (Tesla 68) and for 37-Mux it is 254 (Tesla 121) billion GP operation/second.

8 Theoretical Performance: Infinite Parallelism Model

For any configuration of the interpreter there is a certain amount of work that must be done. The programs must be transfered to the GPU and their fitness values returned to the host computer and they must be interpreted. The nVidia bandwidthTest program measures the data transfer speeds to and from the GPU. This allows us to estimate the time to copy a 20-Mux population to the GPU: time $= 512 \times 262\,144/2170$Mbytes per sec $=$ 62ms. Time to return the (4 byte) fitness values to the host PC: time $= 4 \times 262\,144/1433$Mbytes per sec $=$ 0.7ms. (Total 63ms.)

The time taken to transfer data internally within the GPU is very difficult to estimate. It will not only depend upon the number of times each program is executed but also on the degree of coalescing of reads from global memory. This is difficult to estimate. However taking the raw figures from bandwidthTest

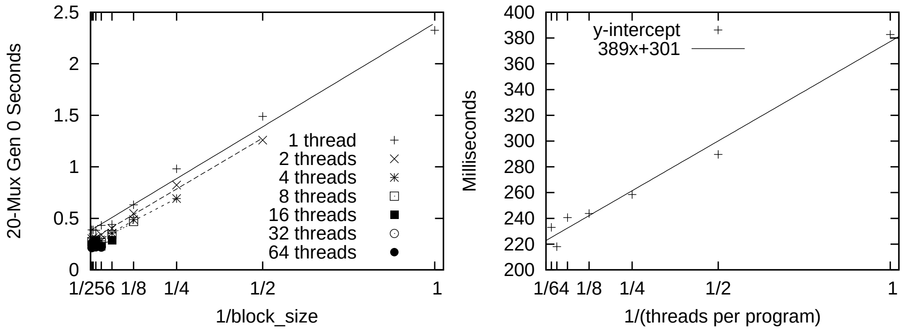

Fig. 8. Left: Elapse time for GeForce 295 GTX to interpret 1/4 million random 20-Mux programs. Right: Elapse time for infinite parallelism v. parallel threads per program. Time for infinite block_size (so 1/block_size=0) is estimated by Y-intercept of linear regression (left).

suggests it can be too short (> 0.2ms) to contribute and global memory latency is much more important.

We estimate the minimum calculation time from when the interpreter gets its best speed. This is when confirming the generality of the evolved 20-Mux solution (507 instructions). Here the whole GPU is devoted to a single program in constant memory, thus removing latency and divergence. Allowing for the time to transfer the answer back to the host gives a minimum calculation time of 53 milliseconds, corresponding to a maximum interpretation rate for the 20-Mux of 573 billion GP operations per second. This gives an estimated minimum time for the initial 20-Mux generation of 115 milliseconds on the 295 GTX.

Figure 8 plots the actual time for all the ways of interpreting the initial population in parallel. We see the wall clock time falls linearly with degree of parallelism. By fitting a least squared error linear regression lines to each we can estimate the infinite parallelism execution time for the initial generation. These seven times are plotted in the right of Figure 8. The vertical intercept of a final regression line says the infinite parallelism execution time would be 223 milliseconds. This is somewhat above our estimate of 115 milliseconds. This suggests that it is not possible to obtain 573 10^9 GP OP/s for the initial population (whose trees have on average only 55 instructions). Using our earlier estimate of transfer time but replacing using the solution (with 507 opcodes) with the new estimate of total time (223 ms) gives a new estimate of 188 billion GP operations per second for the infinite parallelism speed in the initial 20-Mux generation. The best configuration (see Figure 5) is 75% of this. I.e., on the 20-Mux initial population, the 295 GTX is within 25% of the best performance predicted if the interpreter worked with infinite parallelism.

9 Discussion

Modern high performance graphics hardware has a complex parallel hierarchy of memory and processing elements. CUDA exposes this to the programmer in a controlled and somewhat portable way. (I.e. between CUDA capable nVidia hardware.) In contrast other tools try to conceal this and provide a high level obscure view of the hardware. Programming GPUs using either is not easy. For the Mackey-Glass benchmark [15], CUDA is up to 92% faster than RapidMind 2 [21] on similar hardware.

Although we tried to get the best from the T10P Tesla's 192 cores, the CUDA code should run on any modern G80 GPU. In fact no changes to the kernel were needed to run on the GeForce 295 GTX.

For the largest of these problems, our results suggest the interpreter is already within 33% of the best that the current hardware (665 billion GP OP/sec) might deliver in practice.

The interpreter can be used in various models of parallelism. Naturally it is fastest when fitness testing is split across many threads. However when this is not possible individual GP trees can be tested by running fitness cases one after another but the hardware still permits many programs to be run in parallel. The interpreter also allows various intermediate combinations.

10 Conclusions

Ten years ago Koza *et al.* [2] said their Beowulf cluster delivered about a half peta-flop per day on genetic programming runs. We have presented a single office personal computer fitted with a top end graphics card which delivers not floating point but real GP operations at a sustained rate of 22 peta GP operations per day (254 billion GP operations per second). This is twenty times the best reported speed of the fastest previously published GP (obtained by running 14 workstations in parallel [8]) and more than sixty times that of the best reported performance of the next fastest single GPU genetic programming system [21].

The combination of powerful parallel processing in the form of a GPU card, sub-machine code GP, a reverse polish (RPN) interpreter and randomised sub-selection from a test suite has allowed us to solve using tree GP the Boolean 20-multiplexor problem. It has been estimated [24] that it would take more than 4 years. The GPU has consistently done it in less than an hour.

The 37-multiplexor benchmark has 137 billion fitness cases. It has never been attempted before. GP solves it in under a day.

Currently Tesla are available with up to 960 cores, running at up to 1.5 GHz, suggesting a further doubling of performance is possible immediately.

The single GPU code is available via FTP `cs.ucl.ac.uk` directory `genetic/gp-code/gp32cuda.tar.gz`

Acknowledgment

I am greatful for the assistance of Timothy Lanfear and Gernot Ziegler of nVidia, Simon Harding of Memorial, Sean Anderson and Stan Seibert. From King's, I would like to thank Graham Ashton, Don Lokuadassuriyage, Nikolay Korneyev and William Shaw. The initial early engineering Tesla was given by nVidia. Funded by EPSRC grant EP/D050863/1.

References

1. Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. *Genetic Programming – An Introduction.* Morgan Kaufmann, 1998.
2. Bennett III, F. H., Koza, J. R., Shipman, J., and Stiffelman, O. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In *GECCO-99*, W. Banzhaf, *et al.*, Eds., Morgan Kaufmann, pp. 1484–1490.
3. Chitty, D. M. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07*, pp. 1566–1573.
4. Comte, P. Design & implementation of parallel linear GP for the IBM cell processor. In *GECCO '09*, G. Raidl, *et al.*, Eds., ACM.
5. Ebner, M., Reinhardt, M., and Albert, J. Evolution of vertex and pixel shaders. *EuroGP-2005*, LNCS 3447, Springer, pp. 261–270.
6. Fok, K.-L., Wong, T.-T., and Wong, M.-L. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems 22*, 2 (2007), 69–78.
7. Harding, S., and Banzhaf, W. Fast genetic programming on GPUs. In *EuroGP-2007*, M. Ebner, *et al.*, Eds., LNCS 4445, Springer, pp. 90–101.
8. Harding, S. L., and Banzhaf, W. Distributed genetic programming on GPUs using CUDA. In *Wks Paral. Arch. and Bioinspired Algs.* (2009), I. Hidalgo, *et al.*
9. Juille, H., and Pollack, J. B. Massively parallel genetic programming. In *Advances in GP 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. MIT Press, ch. 17.
10. Koza, J. R. *Genetic Programming.* MIT press, 1992.
11. Langdon, W. B. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In *Par. & Dist. Comp. Intelligence.*
12. Langdon, W. B. *Genetic Programming and Data Structures.* Kluwer, 1998.
13. Langdon, W. B. A SIMD interpreter for genetic programming on GPU graphics cards. Tech. Rep. CSM-470, Computer Science, University of Essex, UK, 2007.
14. Langdon, W. B. A fast high quality pseudo random number generator for nVidia CUDA. In *CIGPU workshop at GECCO* (Montreal, 8 July 2009), G. Wilson, Ed., ACM, pp. 2511–2513.
15. Langdon, W. B., and Banzhaf, W. A SIMD interpreter for genetic programming on GPU graphics cards. In *EuroGP 2008*, LNCS 4971, Springer, pp. 73–85.
16. Langdon, W. B., and Harrison, A. P. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing 12*, 12 (2008), 1169–1183.
17. Langdon, W. B., and Poli, R. *Foundations of Genetic Programming.* 2002.
18. Lewis, T. E., and Magoulas, G. D. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. *GECCO '09* pp. 1379–1386.
19. Poli, R., and Langdon, W. B. Sub-machine-code genetic programming. In *Advances in GP 3*, L. Spector, *et al.*, Eds. MIT Press, 1999, ch. 13, pp. 301–323.
20. Poli, R., Langdon, W. B., and McPhee, N. F. *A field guide to genetic programming.* `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).
21. Robilliard, D., Marion-Poty, V., and Fonlupt, C. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines 10*, 4 (Dec. 2009), 447–471.
22. Teller, A., and Andre, D. Automatically choosing the number of fitness cases: The rational allocation of trials. In *GP 1997* (13-16 July), J. R. Koza, pp. 321–328.
23. Wilson, G., and Banzhaf, W. Linear genetic programming GPGPU on Microsoft's Xbox 360. *WCCI 2008*, IEEE.
24. Yanagiya, M. Efficient genetic programming based on binary decision diagrams. In *1995 IEEE Conf. Evolutionary Computation* (Perth, 1995), pp. 234–239.