Large Scale Bioinformatics Data Mining with Parallel Genetic Programming on Graphics Processing Units

William B. Langdon

Abstract The NCBI GEO GSE3494 breast cancer dataset contains hundreds of Affymetrix HG-U133A and HG-U133B GeneChip biopsies each with a million variables. Multiple genetic programming (GP) runs on a graphics processing unit (GPU) hardware, each with a population of five million programs both winnow (select) useful variables from the chaff and evolve small (three inputs) data models. The SPMD CUDA interpreter exploits the GPU's single instruction multiple data SIMD mode of parallel computing, even though the GP populations contain different programs. A 448 node nVidia Fermi C2050 Tesla graphics card delivers 8.5 giga GPops per second. In addition to describing our implementation, we survey current GPGPU work in Bioinformatics and genetic programming.

1 Introduction

Since they offer cheap high performance computing there is great interest in using mass market graphics hardware (GPUs) for scientific applications. For example the Chinese Tianhe-1A 2.566 petaflop supercomputer contains 7,168 nVidia Tesla M2050 general purpose GPUs. However a lot of scientific and engineering can be done with more modest computers and we will concentrate upon affordable personal computers or indeed laptop computers with one or more graphics cards or their Tesla compute only equivalents. More than 100 million GPUs have been sold [24]. This availability and their price/performance ratio has lead to the increasing use of essentially consumer gaming or entertainment hardware for research and engineering purposes. The field is often called general purpose computing on GPU (GPGPU) [84].

Until recently the doubling of the number of transistors in computer chips every eighteen months "Moore's Law" was a fact of life [79] and similar exponential rises occurred in processing speed and disk and memory storage capacity. The compound

Department of Computer Science,
University College, London

Preprint Massively Parallel Evolutionary Computation on GPGPUs, S. Tsutsui and P. Collet *Eds.* Chapter 15, pages 311-347 Springer, 2013
DOI:10.1007/978-3-642-37959-8_15

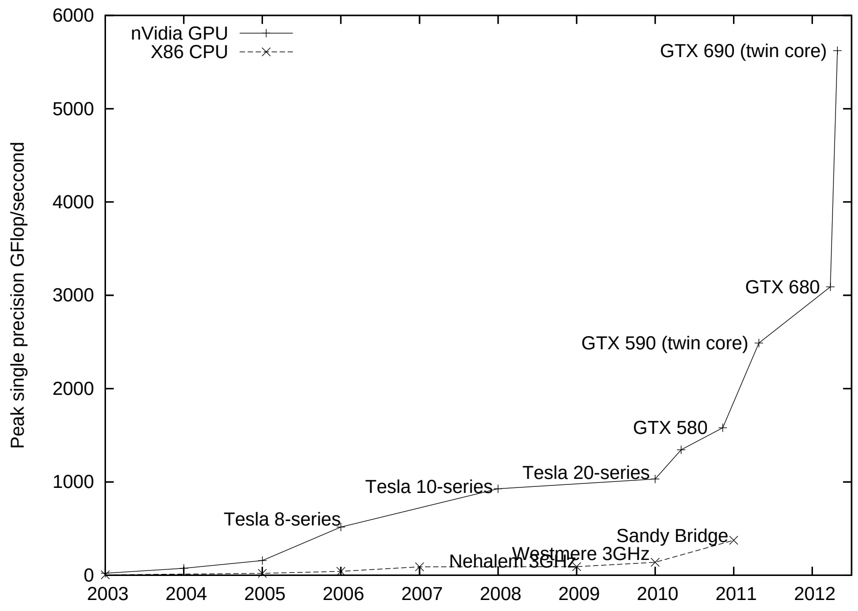


Fig. 1 Comparison of increase in speed of graphics cards (+ GPU) and CPU (× x86) (data supplied by nVidia). Similar trends hold for double precision and integer performance.

effect of Moore's Law has lead to literally million fold increases in hardware performance during careers in the software industry. Naysayers have frequently pointed out the impossibility of exponential grow continuing indefinitely, however today it looks like they are right in at least one important aspect and we have reached the end of Moore's law as it has been applied to processor speed. In commercial terms, the industry remains dominated by descendants of Intel's 8086 silicon chips yet for half a dozen years we have seen no major increase in CPU clock speed since the 3GHz Pentium (see lower plot in Figure 1). If clock speeds had continued to double every 1.5 years we would have 25GHz Pentium's on our desks and in our laptops. This has not happened. It looks like it will never happen.

In its original sense the manufactures of silicon chips continue to obey Moore's Law and the number of transistors per chip has continued to increase. Recently Izydorczyk and Izydorczyk [37] suggested Moore's Law will continue to hold for at least the next 22 years. However they appear to accept today's limit of about 3.5GHz on processor clocks.

The additional transistors packed evermore densely into chips have been used to create still bigger memory, particularly on chip cache memory, more exotic instruction sets (e.g. vector, parallel and special purpose instructions) and especially to build multiple CPU cores on the same chip. Dual and quad cores are now common place. Eight and even sixteen core Pentium computers are now on the horizon.

It looks like we are really seeing the parallel future which has been forecast even before the Transputer [1].

Since our initial results on the breast cancer survival prediction dataset GPU development has continued apace. For example, both AMD and nVidia have GPU which claim to deliver more than a teraflop at a cost of a few hundred dollars.

The next section will describe scientific and engineering computing on GPUs. Some successful applications of GPUs to Bioinformatics will be described in Section 3. In Section 4 we will summarise our original RapidMind work [57] in which genetic programming [55] is used to datamine a small number of indicative mRNA gene transcript signals from breast cancer tissue samples taken during surgery, each with more than a million variables, to predict long term survival. In [57] we described the medical problem and the way genetic programming [53] and a GPU simultaneously picked three of the million mRNA measurements available and found a simple non-linear combination of them which predicts long term outcomes at least as well as DLDA, SVM and KNN using seven hundred measurements [78]. Before concentrating upon using genetic programming [40, 4, 59, 86] in parallel on a GPU, Section 5 briefly describes the major hardware components of GPUs and programming them. Then Section 6 describes the new GP and CUDA code. We refer the interested reader to [57] for details of the data source and how they were obtained, checked and normalised. The experiments are repeated using the new CUDA kernel. (The results are summarised in Section 7.) The new system avoids many restrictions imposed by RapidMind and uses modern Tesla hardware (C2050) to deliver a more than ten fold speed up (Section 8). Finally in Section 9 we consider how successful our previous predictions about GPGPU have panned out and make new ones. We conclude (Section 10) that GPGPU will be one of the parallel techniques of the future but note it is still held back by development tools.

2 Using Games Hardware GPUs for Science

Owens *et al.* [84, 83] surveyed scientific and engineering applications running on mass market graphics cards. Today's GPUs can greatly exceed the floating point performance of their host CPU, see Figure 1. This speed comes at a price.

GPUs provide a restricted type of parallel processing, often referred to as single instruction multiple data (SIMD) or more precisely single program multiple data (SPMD). Each of the many processors simultaneously runs the same program on different data items (see Figure 2). Being tailored for fast real time production of interactive graphics, principally for the computer gaming market, GPUs are tailored to deal with rendering of pixels and processing of fragments of three dimensional scenes very quickly. Each is allocated a processor and the GPU program is expected to transform it into another data item. The data items need not be of the same type. For example the input might be a triangle in three dimensions, including its orientation, and the output could be a colour expressed as four floating point numbers (RGB and alpha).

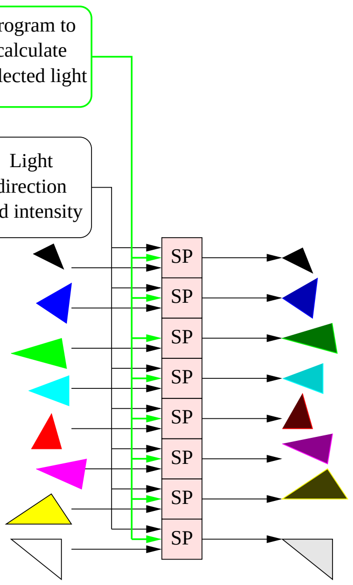

Program to calculate reflected light
Light direction and intensity
SP
SP
SP
SP
SP
SP
SP
SP

Fig. 2 An example of SIMD parallel processing. The stream processors (SP) simultaneously run the same program on different data and produce different answers. In this example the program has two inputs. One describes a triangle (position, colour, nature of its surface: matt, how shiny). The second input refers to a common light source and so all stream processors use the same value. Each stream processor calculates the apparent colour of its individual triangle. Notice, here, each output is independent of all the others and so they can all be calculated in parallel.

Typical GPUs are optimised so that programs can read data from multiple data sources (e.g. background scenes, placement of lights, reflectivity of surfaces) but generate one output. This parallel writing of data greatly simplifies and speeds the operation of the GPU. Even so both reading and writing from memory are still bottlenecks. This is true for the GPUs own memory but doubly so when data are transferred to/from the host PC and the GPUs.

The manufactures' continue to publish figures claiming enormous peak floating point performance. In practise such figures are not obtainable. A more useful statistic is often how much faster an application runs after it has been converted to run on a GPU. However, like FLOPS, the number of GP operations per second (GPops) allows easier comparison of different GP implementations.

Many scientific applications and in particular Bioinformatics applications are inherently suitable for parallel computing. In many cases data can be divided into almost independent chunks which can be acted upon almost independently. There are many different types of parallel computation which might be suitable for Bioinformatics. Applications where a GPU might be suitable are characterised by:

- Maximum dataset size $\approx 10^9$
- Maximum dataset data rate $\approx 10^9$ bytes/second
- Up to 10^{11} floating point operations per second (FLOPs)

- Applications which are dominated by small computationally heavy cores. I.e. a large number of computations per data item (known as arithmetic intensity).
- Core has simple data flow. Possibly a large fan-in and simple data stream output.

Naturally as GPUs continue to become more powerful these figures have continued to change.

3 GPUs in Bioinformatics and Computational Intelligence

As might be expected GPUs have been suggested for medical image processing applications for several years now. However we concentrate here on molecular Bioinformatics. We anticipate that after a few key algorithms are successfully ported to GPUs, within a few years Bioinformatics will adopt GPUs for many of its routine applications. As might be expected, early results were mixed.

Charalambous *et al.* successfully used a relatively low powered GPU to demonstrate inference of evolutionary inheritance trees (by porting RAxML onto an nVidia FX 5700) [9]. However a more conventional MPI cluster was subsequently used [98]. Recently a CUDA version of the alternative MrBayes tool was published [112].

Sequence comparison is the life blood of Bioinformatics. Weiguo Liu *et al.* ran the key Smith-Waterman algorithm on a high end GPU [68]. They demonstrated a reduction by a factor of up to sixteen in the look up times for most proteins. Smith-Waterman has also been ported to the Sony PlayStation 3 [106] and the GeForce 8800 (CUDA) [76]. Trapnell and Schatz also used CUDA to port another sequence searching tool (MUMmer) to another G80 GPU and obtained speed ups of up to 13 when matching short DNA strands against much longer sequences [99]. More recently Vouzis and Sahinidis [101] ported NCBI's Blast protein sequence alignment tool to CUDA but report only modest speed ups perhaps because of the large data volumes and their insistence on exactly emulating the original serial code. By breaking queries into GPU sized fragments, they were able to run short sequences (e.g. 50 DNA bases) against the complete human chromosome. Successful ports and CUDA implementations of sequence tasks include GBOOST [111] (40 fold), SOAP3 [67] (7.5–20$\times$ faster) and MrBayes [112] (19$\times$, more with a second GPU).

Bing Liu *et al.* used GPUs to model biomolecular pathways [66] (26–33$\times$) and Zhou *et al.* report speed ups of 12$\times$, 47$\times$ and 367$\times$ for Gillespie, LSODA and Euler-Maruyama using their cuda-sim Python package [113]. Kannan and Ganji [39] also report 10–47 fold speed up when porting AutoDock (a biomolecular drug discovery tool). Gobron *et al.* used OpenGL on a high end GPU to drive a cellular automata simulation of the human eye and achieved real-time processing of webcam input [25]. GPUs have also been used in medical engineering. E.g. a GeForce 8800 provided a 15-20 fold speedup, improving the haptic response of a real time interactive surgery simulation tool [69]. Dowsey *et al.* wrote 2D gel electrophoresis image registration code in Cg ("C for graphics") so that it could be off loaded onto an nVidia GPU [14].

The better GPU applications may claim speed ups of a factor of ten or more, however the distributed protein folding system folding@home obtains sixty times as much free computation per donated GPU as it does per donated CPU [84, p983]. The same authors also claim almost a 3600 fold speed up on a biomolecule dynamics simulation, albeit at the cost of using four FX 5600 GPUs [84, p995].

Computational intelligence applications of GPUs have included artificial neural networks (e.g. multi layer perceptrons [71, 91], self organising networks [88] and spiking neural networks [110]), fuzzy logic [34], genetic algorithms [22, 107, 80, 97, 72, 85, 95] and genetic programming [65, 70, 77, 18, 90, 29, 28, 33, 27, 10, 43, 57, 92, 102, 3, 15, 30, 62, 73, 93, 103, 7, 17, 16, 35, 36, 45, 46, 56, 74, 75, 94, 100, 104, 6, 8, 13, 19, 31, 32, 47, 49, 63, 64, 87, 96, 108]. Most GPGPU applications have only required a single graphics card, however Fan *et al.* have shown large GPU clusters are also feasible [20]. In 2008 the first computational intelligence on GPU special session (CIGPU-2008) was held in Hong Kong [105]. This has become an annual event. As Owens [83] makes clear games hardware has now broken out of the bedroom into scientific and engineering computing.

4 Gene Expression in Breast Cancer

We have previously [57] used genetic programming to data mine gene expression measurements provided by Miller *et al.* [78]. We will mostly be concerned with updating the original RapidMind code to CUDA and its improved performance. However we start by recapping the datamining problem. Miller *et al.* describe the collection and analysis of cancerous tissue from most of the women with breast tumours in the three years 1987–1989 in Uppsala in Sweden. Miller's primary goal was to investigate p53, a gene known to be involved in the regulation of other genes and implicated in cancers. In particular they studied the implications of mutations of p53 in breast cancer. The p53 genes of 251 women were sequenced so that it was known if they were mutant or not. Affymetrix GeneChips (HG-U133A and HG-U133B) were used to measure mRNA concentrations in each biopsy. Various other data were recorded, in particular if the cancer was fatal or not.

Each of the two types of GeneChips used contained more than half a million DNA probes arranged in a 712×712 square $(12.8\text{mm})^2$ array. (Current designs now exceed five million DNA probes on the same half inch square array.)

4.1 Uppsala Breast Cancer Affymetrix GeneChip Data Sets

As part of our large survey of GeneChip flaws [60] we had already down loaded all the HG-U133A and HG-U133B data sets in GEO [5] (6685 and 1815 respectively) and calculated a robust average for each probe. These averages across all these human tissues were used to normalise the 251 pairs of HG-U133A and HG-U133B

GeneChips and flag locations of spatial flaws [57]. R code to quantile normalise and detect spatial flaws is available via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/R. The value presented to GP is the probe's normalised value minus its average value from GEO. This gives an approximately normal distribution centred at zero. See Figure 3.

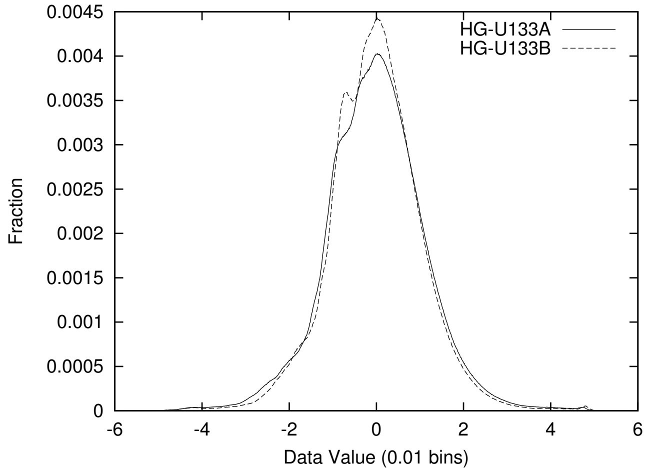


Fig. 3 Uppsala breast cancer distribution of log deviation from average value.

The GeneChip data created by [78] were obtained from NCBI's GEO (data set GSE3494). Other data, e.g. patients' age, survival time, if breast cancer caused death and tumour size, were also down loaded. Whilst [78] used the whole dataset: with more than a million inputs we were keen to avoid over fitting, therefore the data were split into independent training and verification data sets. See [57].

5 Summary of GPU Hardware and Programming

5.1 Main Hardware Components of GPUs

Figure 4 shows the major components of a C2050 Tesla card. It is typical of current top end GPUs. The card is connected to the host personal computer via the PC's PCI express bus. The effective speed of the PC–GPU connection varies both with GPU and with the mother board into which they both fit. Getting data into and out of the GPU via the PCI bus is one of the major design decisions in any GPU application. Although PCIe bus speeds have risen in recent years, it appears to have peaked. Recent top end systems have relied on a hierarchy of PCI interconnects which allow simultaneous parallel transfers along their various parts.

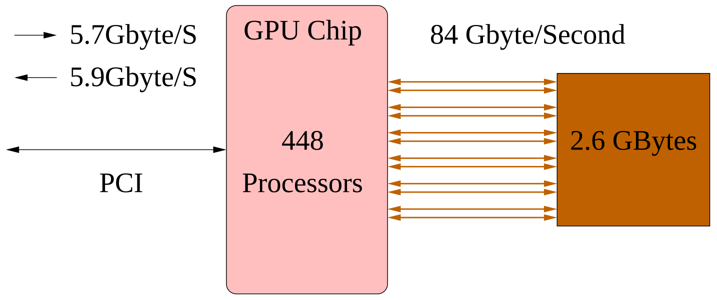


Fig. 4 Links from GPU chip to host computer via PCIe bus and to memory on the GPU board. Fermi C2050 (ECC memory checks turned on).

Typical GPU have space for several hundred megabytes or even a few gigabytes of data. The trend is still very much to increase the speed and size of onboard memory. Again deciding which application data (and when) are stored on board the GPU is an important design decision. The GPU chip is connected by a very high speed bus to its own high speed onboard RAM memory.

There are some two GPU system. Typically although there are two chips and two sets of RAM on the same board they are programmed as if they were two separate GPUs in the same PC.

It is typical for a single GPU chip to contain more than one multi-processor, see Figure 5. These have their own connections to the onboard RAM and act more or less independently in parallel. The number of multi-processors varies considerably between low end and older models and high end GPUs. The C2050 has fourteen multi-processors. There are already GPUs with 16 multi-processors and the trend is for the maximum number of multi-processors to increase whilst retaining low end GPUs with a single multi-processors.

The multiprocessors contain banks of stream processors (SP). These are where the essential SIMD nature of GPU computing arises. All the stream processors are locked together. They do the same calculation at the same time (albeit on different data). Thus a C2050 multiprocessor can take 32 data items, do 32 calculations and generate 32 answers in parallel. However when a program contains an if or branch instruction the 32 data items may cause the 32 stream processors to go in different directions. This they cannot do. Instead one branch direction is chosen and stream processors going in that direction are free to continue calculating. The rest are held. At some time, the freely running stream processors are held long enough for the others to run. It may be quite sometime later when all the stream processors return to a common instruction at the same time and all begin running at full speed in synchrony. In the mean time (when the stream processors' paths have diverged) the multiprocessor has been operating correctly but at reduced power. We shall use this property. It is important to remember that GPUs offer cheap computation, so its ok to waste some of it.

The number of stream processors varies between GPUs. nVidia multiprocessors contain multiples of eight. As with multiprocessors themselves, both the range and

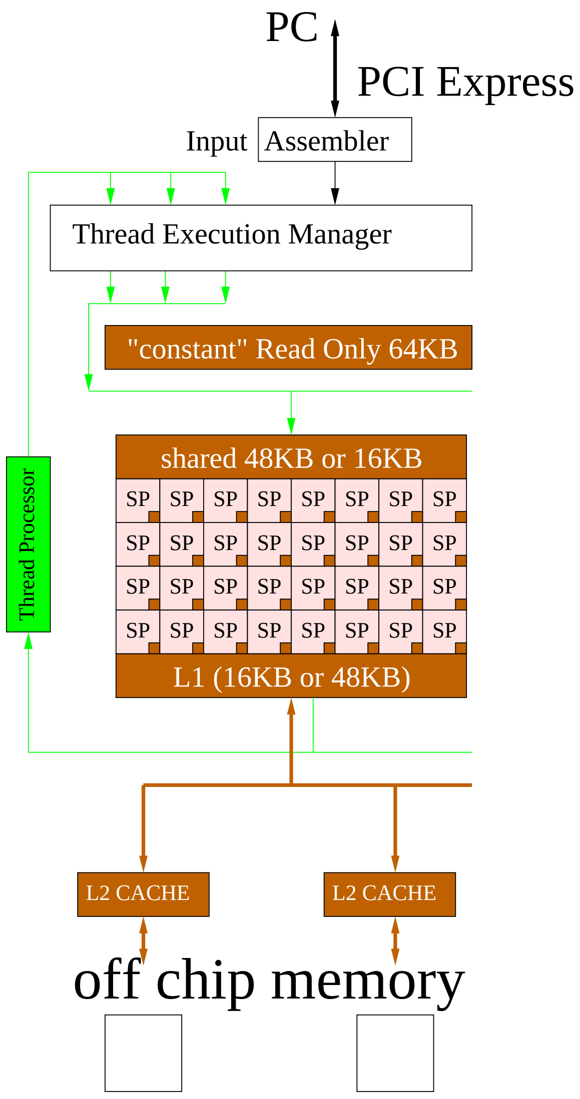


Fig. 5 nVidia GPU multi processor with 32 stream processors (SP). The C2050 contains 14 such multiprocessors, giving 448 SPs in total. Each stream processor obeys the same instruction at the same time. However each has its own registers and access to shared and constant memory. The L1 caches coalesce multiple separate accesses to off chip memory into a single access of 128 bytes each. In default operation each L1 cache occupies 16 Kbytes (giving 128 cache lines) however the 48 Kbytes shared memory can be reduced to 16 Kbytes to expand the L1 cache to 48 Kbytes.

maximum number of stream processors have increased and are likely to continue increasing. However the multi-processor clock speed have not increased and may even have fallen back a little. Typical clocks speeds are now 1.1–1.5 GHz and dramatic change is not likely.

The newer Fermi designs now include both per-multi processors (L1) read-write data caches and L2 read-write cache shared between the multi processors. Whereas older designs relied either on the application designing its own caches or read-only caches provided as part of graphics "texture" memory. The L2 cache also allows some limited communication between multi-processors via atomic operations. For sometime nVidia resisted the application developers' calls for caches but now implemented in the Fermi architecture they seem to be a great success. Future GPUs may see more and/or bigger caches.

5.2 Memory Latency – Efficiently Programming with Threads

An important thing in Figures 4 and 5 which we have not discussed is why caches are important. The fact that dominates GPU programming (even with caches) is that it can take hundreds of times longer to fetch data from the GPU's off chip memory than to calculate with it. Once data are in its registers, cache or shared memory, the multiprocessor can calculate with it blisteringly fast but an unfortunate application can perform badly simply by having the stream processors wait for data most of the time. Figure 6 is a schematic which shows the GPU hardware interleaving threads of execution (horizontal arrows) so that as threads are blocked (e.g. by waiting for off chip data to arrive) others are automatically released to run. If there are enough threads, the multiprocessor may still be busy when the data arrives, so keeping it fully loaded and enabling the application to efficiently use the GPU. However the number of active threads is limited.

Earlier nVidia GPUs limited the maximum number of threads to 512. The Fermi architecture has recently double this to 1024. However there is another limit. Each execution thread will need some registers. Unlike a preemptive scheduler on the host computer, when a thread stops, there is nowhere to save these registers when a new thread is scheduled. Thus even when a thread is blocked (e.g. waiting for data to arrive) it cannot release its registers. This enables extremely rapid context switching between threads but means all the multiprocessor's registers have to be shared by its active threads. (A C2050 multiprocessor has 32 768 registers, 1024 for each stream processor.) Although the CUDA nvcc compiler is very careful in how it allocates registers, it is possible, in complicated applications, for the number of active threads to be limited by the number of registers each thread requires before reaching the 1024 limit.

Although GPU and application dependent, Figure 7 shows typically a GPU starts to approach its maximum performance when there are more than about 18 threads per stream processor.

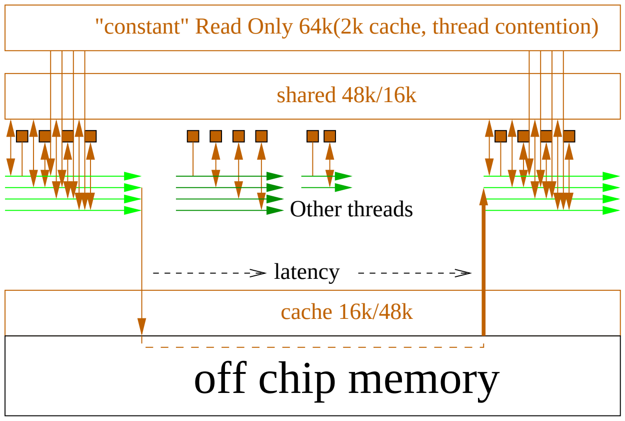


Fig. 6 nVidia CUDA mega threading (Fermi, compute level 2.0). Each thread in a warp (32 threads) executes the same instruction. When a program branches, some threads advance and others are held. This is known as thread divergence (Section 5.1). Later the other branches are run to catch up. Only the 32 768 registers (small squares) per block can be accessed at full processor speed. If threads in a warp are blocked waiting for off chip memory (i.e. local, global or texture memory) another warp of threads can be started. The examples assumes the requested data are not in a cache. Shared memory and cache can be traded, either 16 Kbytes or 48 Kbytes. Constant memory appears as up to 64 Kbytes via a series of small on chip caches [2].

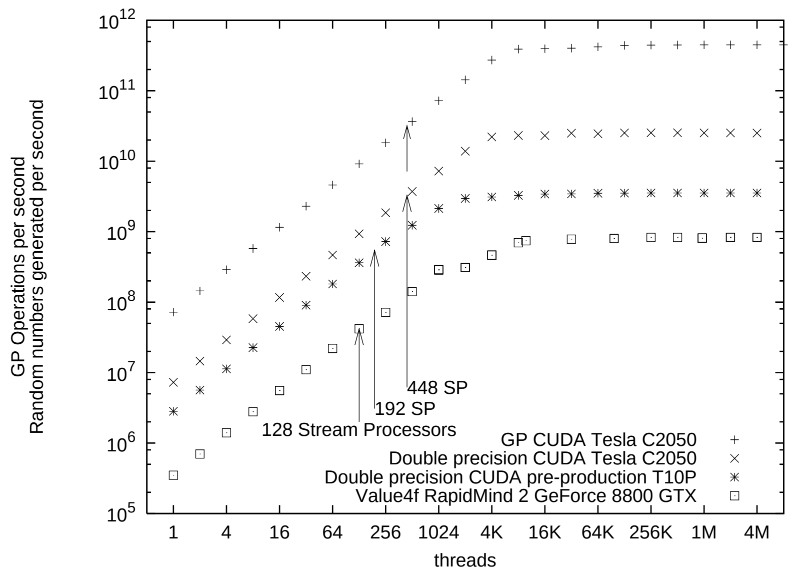


Fig. 7 Speed of genetic programming interpreter [46] and Park-Miller random numbers [44] (excluding host-GPU transfer time) versus number of parallel threads used on a range of nVidia GPUs. Top 3 plots refer to CUDA implementations and lowest one to RapidMind code. Code available via `ftp.cs.ucl.ac.uk /genetic/gp-code/`.

6 GeneChip Data Mining using Genetic Programming on a GPU

6.1 A CUDA Single Instruction Multiple Data Interpreter for GP

Section 3 has listed the previous experiments evolving programs with a GPU. Mostly these have either represented the programs as trees or as networks (Cartesian GP) [29] and used the GPU for fitness evaluation. Harding [29] compiled his networks into GPU programs before transferring the compiled code onto the GPU. However it turns out to be quite expensive to compile CUDA program and so it only makes sense when the program (in our case a GP individual) is to be run many times. (Harding showed the compiled approach can be improved by distributing the compilation across a local area network of workstations and obtained impressive results when each GP programs was run more than 100 million times [32].) Since we will be running each GP individual program on each training case (cancer patient) but we have at most only a few hundred training cases (actually only 91), it makes sense to avoid the compilation overhead and accept that interpreting the program may be slower than running compiled code but interpreting will be faster overall. Therefore we kept the traditional tree based GP and use an interpreter running on the GPU.

The host part of the program is a more-or-less traditional GP but with fitness evaluation transferred to the GPU. However it represents evolving genetic programming individuals as trees which are linearised into reverse polish expressions [53] so that the GPU can interpret them in straightforwardly in a single pass without recursive calls. The three mutation operations and crossover act directly on the reverse polish expressions. This enables them to be passed directly to the GPU without the need to change format between the host and the GPU. Next we shall recap how to interpret multiple programs simultaneously on a SIMD computer [42] before going into the details of the CUDA implementation (Sections 6.2–6.10). Section 6.11 describes how we use hundreds of GP runs to progressively refine the GeneChip data, how the largest ever GP populations are created and evolve under fitness selection, mutation and crossover. It also describes the non-panmictic fine grained distributed population and short evolution times use to maintain diversity. All these operations take place on the host PC and are implemented in C source code.

Essentially the interpreter trick is to recognise that in the SIMD model (Section 5.1) the "single instruction" belongs to the interpreter and the "multiple data" are the multiple GP trees. The single interpreter is used by millions of programs. It is quite small and needs to be compiled only once. It is loaded onto every stream processor within the GPU. Thus every clock tick, the GPU can interpret a part of up to 448 different GP trees. The guts of a standard interpreter is traditionally a n-way switch where each case statement executes a different GP opcode, however Figure 8 gives an alternative view in which the interpreter works on all possible opcodes and each GP program uses just those that it contains. The CUDA implementation is given in Figures 9–11.

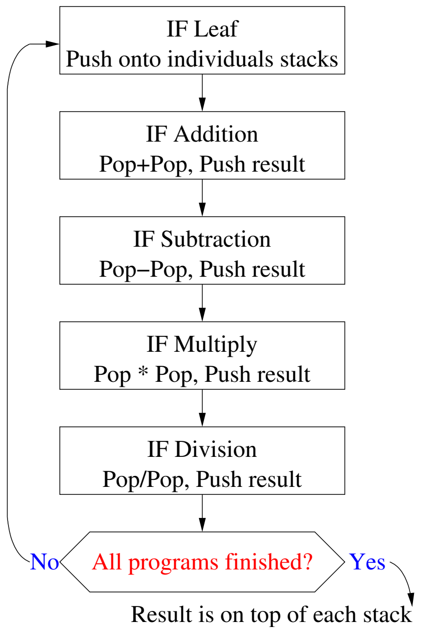


Fig. 8 The original idea for the SIMD interpreter was that it loop continuously through the whole genetic programming terminal and function sets with GP individuals select which operations they want as they go past and apply them to their own data and their own stacks. However this can be refined by noting that individual multi-processors act independently. If all 32 stream processors (SPs) in a warp run the same GP program they will be synchronised and the SIMD interpreter behaves more like a conventional interpreter acting in parallel 32 times. There is some lost in efficiency if they act on multiple GP individuals and lose synchronisation, since this may cause thread divergence (Section 5.1), however the GPU still performs well.

6.2 CUDA Interpreter for GP

The CUDA code is given in Figure 9. Potentially it could be improved further since 1) each program must end in a nop the for loop test `PC < LEN-1` could be removed. 2) the array indexing operation `Pop[PC]` could be replaced by using the pointer `Pop` directly and incrementing it by 4 bytes on each iteration of the loop, which would allow the variable `PC` to be removed.

6.3 CUDA Interpreter Stack for GP

The interpreter evaluates each GP tree as a reverse polish notation expression by pushing and popping intermediate values onto a stack (see Figure 8). Each expression needs its own stack. Each GPU thread works on its own expression and so needs its own stack.

```
int SP = 0;
for(unsigned int PC = 0; PC < LEN-1; PC++) {
  const optype OPCODE = Pop[PC];
  if(OPCODE==OPNOP) break;
  float d;
  if(OPCODE<= lastconst) {
    d = constants[OPCODE];
  } else if(OPCODE<= lastleaf) {
    d = d_Train0[(OPCODE-firstinput)*nexamples];
  } else {
    const float sp1 = stack(--SP);
    const float sp2 = stack(--SP);
    switch(OPCODE) {
    case OPADD:  d = sp2+sp1; break;
    case OPSUB:  d = sp2-sp1; break;
    case OPMUL:  d = sp2*sp1; break;
    case OPDIV:  d = sp2/sp1; break;
    }
  }
  push(d);
}
```

Fig. 9 GPU Reverse Polish Notation SIMD interpreter. The interpreter is invoked by every thread in the block (1001) in parallel and cycles through each the programs' instructions leaving the answer generated by each on the programs' stacks. (Fitness calculation in Figures 13–15.) Notice division is not protected [40]. `Pop` is a pointer to the start of the RPN program which is being evaluated on this stream processor. `d_Train0` points to the data for the current cancer victim (see Section 6.6).

```
extern __shared__ float shared_array[];
const int pStackMax = (MaxArity-1)*(pMaxDepth-1)+1;
#define stack(sp) shared_array[(sp)*blockDim.x+threadIdx.x]
#define push(x) {stack(SP) = x; SP++;}
```

Fig. 10 CUDA implementation of stack required by SIMD interpreter (given in Figure 9). The stack is placed in shared memory to ensure it remains on chip. CUDA allows indexed access to shared memory and so implementing a stack is much simpler than it was RapidMind (version 2.0) and using deeper stacks is also straight forward. Indexing by `threadIdx.x` ensures each thread accesses adjacent words of shared memory so there are no bank conflicts.

Since there is no communication between threads, with read-write caches, it might be possible to place the interpreter's stacks in per thread "local" memory. There is only a little shared memory whereas there is lots of local memory but if a cache line holding the stack was displaced, performance would be hit hard.

To avoid the possibility of any stack being moved to off chip memory we chose to put them in shared memory. (See code fragment in Figure 10.) Many GP systems restrict tree depth and function arity. E.g. our GP genetic operations ensure tree depth does not exceed eight (`pMaxDepth`) and Koza [40] enforces a depth limit of 17. If unusually deep trees are needed or the function set contains functions with more than just two inputs (our datamining trees use binary functions) more memory would be required. In which case the limited shared memory could start to restrict the number of threads that the interpreter can use.

Examining the PTX assembler produced by the nvcc compiler suggests that although accessing shared memory should be almost as fast as the threads' own registers, a surprisingly large number of PTX instructions are needed to implement push and pop. However its not clear why and also the mapping between PTX assembler and final machine code is far from straight forward. Even though efficient stack operations are vital, this makes further optimisation of the stack tricky.

6.4 Constants

In this application the GP system needs 1001 constants (with values between −5.0 and +5.0, every 0.01). To simplify the interpreter, the old RapidMind system precalculated these and loaded them as part of the training data. However pushing constants onto the stack is one of the most common operations and so to avoid reading them from global data (as the training data has to be), originally the new CUDA interpreter calculated them as required. This overhead was reduced by precalculating them once per multiprocessor and saving them in shared memory (see Figures 11 and 12). This only occupies 4004 bytes of shared memory but the speed up was modest.

It would also be possible to store them in constant memory, so avoiding calculating them on the GPU at all, but where two different programs cause the interpreter to simultaneously read different constants there is a surprising overhead [50, 61].

```
float* const constants = &shared_array[pStackMax*blockDim.x];
for(unsigned int i=threadIdx.x; i<=lastconst; i += blockDim.x){
  constants[i] = float(-5.0) + float(i) * float(0.01);
}
__syncthreads();
```

Fig. 11 Setting up GP constants in shared memory. The 1001 constants are stored immediately above the interpreter's stack (Figures 10 and 12). The calculation is spread across all the available threads. `__syncthreads()` prevents any thread moving on to interpret any program until all the constants have been initialised. As the calculation happens before any global data is read, `__syncthreads()` causes little overhead. Usually adjacent threads interpret the same GP individual so they will simultaneously read the same constant. This does not cause a bank conflict. Since there are multiple banks of shared memory, only occasionally will a delay occur as a bank conflict arises from threads in the same warp interpreting two different programs simultaneously.

6.5 Thread Lay Out

As we described under the heading of “The Computational Cube” in [47], one of the virtues of the SIMD GP interpreter is that it gives different ways to access the huge amount of parallelism inherent in having a population of individuals and multiple training cases which they need to be evaluated upon. As we showed in Figure 7 the efficient use of GPUs requires many active threads. While it will vary between applications, Figure 7 suggests even something as simple as generating random values will need at least 8000 threads to fully load a C2050. With this in mind we designed the thread layout to use as many of the 1024 threads per multiprocessor block as possible. However we decided to combine the fitness calculation with the interpreter into one CUDA kernel so all the threads interpreting one program must be in the same block and they are forced to synchronise when fitness is calculated. Also we decided to use one thread per GP program per test case. With 91 training cases, this means each block simultaneously interprets eleven programs using 1001 threads (98% of the 1024 maximum). See Figure 12. Giving a maximum of 14 014 active threads per C2050.

The interpreter threads are tightly packed, which means ignoring the 32 thread warp boundaries [93]. Thus ten of of our 32 warps will be interpreting two GP programs at once and so will suffer from divergence. For 91 training examples, we could have packed the thread into three warps (using 95% of the available threads) and allowing ten programs per block and up to 12 740 active threads per C2050. However tightly packing the programs into warps has the advantage that the number of training cases can be readily changed without detailed consideration of its impact. For example, the system worked well (without modification) with 41 training cases.

Other approaches are also possible. For example, all 91 fitness cases for one program could be interpreted by warps in the same block. This would simplify the across thread summations needed to calculate the program’s final fitness value and remove the need to use `__syncthreads()` in the fitness reduction (Section 6.8). Alternatively we could have used one thread per program, so avoiding the need for any data transfers between threads. This also avoids any idle threads. However as well as problems of the threads diverging (Section 5.1), having a large number of separate programs independently requesting uncorrelated data items would overwhelm the data caches.

A potential good compromise would be to allocate each program a whole warp (avoiding thread divergence), enabling it to read and use training data a cache line at a time. Having read and processed it, typically the program would not re-read it. With 91 training cases, each interpreter thread would would have to process the program between two and three times. (This also uses 95% of the available threads.)

As the computational cube approach makes clear, other compromises are possible. While their efficiency will vary, according to circumstances, many are viable

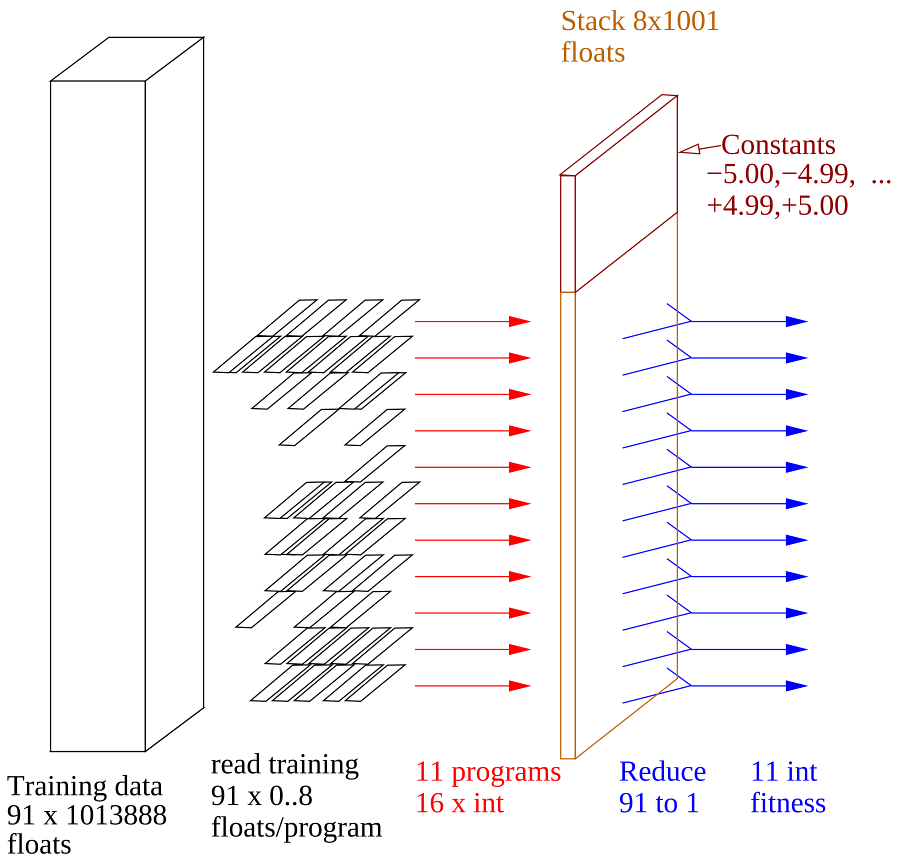

Stack 8x1001 floats
Constants −5.00,−4.99, ... +4.99,+5.00
Training data 91 x 1013888 floats
read training 91 x 0..8 floats/program
11 programs 16 x int
Reduce 91 to 1
11 int fitness

Fig. 12 On each of the 14 C2050 Tesla multiprocessors eleven GP programs of between 1 and 15 instructions (11 middle arrows) are interpreted in parallel each processing data for 91 of the breast cancer gene expression data sets. This uses $11 \times 91 = 1001$ of the 1024 available threads (97.8%). Each interpreter thread has its own stack in shared memory (slab between the two sets of arrows). Apart from warp divergence the 1001 threads act independently until fitness calculation. After comparing each program's output with the actual class, the CUDA kernel uses 7 reduce operations to sum the number of training cases which the program, got right and convert these to a fitness value which is written to global memory (11 arrows on right).

6.6 Training Data

Each training example has data from both HG-U133A and HG-U133B, i.e. $2 \times 712^2 = 1013888$ floats. The training data are not modified. (This is usual in machine learning applications.) They are stored in the GPU (left hand side of Figure 12) at the start of the run and then only read. This transfer happens only once so there would only be a marginal advantage in using non-paged ("pinned") memory on the host to speed up the transfer. Once loaded onto the GPU, the host does not use it again. Placing it in normal host memory allows the operating system to page it out to re-use the RAM it was occupying if need be.

When the training data are read in, they are effectively transposed so that all the data for the same GeneChip probe are placed consecutively. This enables probe data to be read into a few cache lines in a small number of operations (3 or 4 depending upon alignment).

6.7 Thread Divergence

Although all our reverse polish (RPN) flatten trees will start with pushing a data item, in a usual GP population the second, third, fourth and so on instruction will tend to be different. As the code in Figure 9 shows if a warp of threads is interpreting two different GP individuals their paths through the interpreter code will be different and only a small part (the top and bottom of the main loop) will be common. Since they cannot do this, we get "divergence" (Section 5.1). This means one set of threads proceed, with the others headed to different code, held up. Sometime later the first set of threads is held up and the second set allowed to run. At some later point all the threads in the warp resynchronise. Obviously this is slower than the usual case where the whole warp is interpreting the same GP program. From the computational point of view, we would expect such a warp to take a bit less than twice as long as a single program warp. Potentially more important is reading data.

Two different programs (even though adjacent in the GP population) will typically access different data. In the first sets of runs there is a huge volume of training data and reading different parts of it will probably mean they are not in the L1 cache, hence the threads will have to wait until it can be read into the GPU chip. Hopefully there will be other threads elsewhere on the same multiprocessor ready to run, but even so delays caused by reading data may be more important than thread divergence.

Unfortunately it is difficult to tune the code to get the best from the GPU and it could need re-tuning for other dataset and problems [50]. Nonetheless, while this may not be the absolute optimum code, we feel it is a good compromise.

6.8 Fitness Calculation

There are three stages to fitness calculation (arrows right hand side of Figure 12).

1. Each thread compares the sign of the value calculated by the GP individual with that desired. For the 21 positive cases it should be positive. For the 70 negative cases it should be not be positive. The value (0 or 1) is saved in shared memory, see Figure 13.

```
const unsigned int correct = (pos ^ (stack(0) <= 0)) & 1;
volatile unsigned int *sdata = (unsigned int*) shared_array;
sdata[threadIdx.x] = correct;
```

Fig. 13 Final part of `runprog()` (Figure 9). The value calculated by GP (on the top of the stack, Figure 10) is compared with the class of training example `pos`, converted into a Boolean (was it `correct` or not) and then saved in shared memory, overwriting part of the stack (which is no longer needed).

2. The 91 `correct` or not values are summed using a reduction technique to give the number of true negatives (`TN`) and number of true positives (`TP`) the GP individual scored. See Figure 14.
3. A single thread is used to convert (`TN`) and (`TP`) into a single fitness value which is stored in the GP individual's output (see Figure 15) for later transfer to the host.

Each thread always works on the same training case for each of the $\approx 34\,000$ GP programs it interprets each generation. Therefor `pos`[1] (Figure 13), like `d_Train0` (Figure 9) and the boundaries of the negative and positive cases (given by `start1` and `n` in Figure 15) are calculated once when the thread starts and then are reused.

6.9 *Fermi L1 Caches*

GP individuals are stored as 16 `unsigned int` (`LEN` = 16). Thus when the first thread interprets the first instruction it will actually cause the whole individual (`Pop`) to be loaded from off chip global memory into L1 cache and remain in cache on the multiprocessor until the interpreter finishes with it. Actually since each program occupies only half a cache line, the first instruction can also trigger the loading of `Pop` for the adjacent program. (A C2050 cache line covers 128 contiguous bytes). Since all the threads in a block work on 11 contiguous programs (Figure 12) they should fit into six cache lines. Eventually all of `Pop` will have to be read, but this is done efficiently and it does not have to be read more than once by that individual. Notice we also avoid explicitly caching the population in shared memory [93].

As mentioned in Section 6.6, the training data are organised to be adjacent to each other, so if one part of a training case is loaded into the multiprocessor L1 cache then 31 data items in the corresponding training cases are also loaded into the cache at the same time. It appears that with 91 training cases three cache lines per data item are needed. (Perhaps four, depending upon how the cache handles alignment.) Thus in the initial runs where there are thousands or indeed millions of data items, the L1 cache cannot hope to avoid reloading. However all the training data required to interpret each GP individual will be read efficiently into the multiprocessor and it will only be read once by that individual.

[1] `pos` is 1 for positive training cases and 0 for negative cases.

```
__device__
void reduce_sum(const unsigned int start, const unsigned int n){
//Ok to overlay on Stack as used syncthreads to ensure all done
volatile unsigned int *sdata = (unsigned int*) shared_array;
const unsigned int tid = threadIdx.x;
const unsigned int top = start+n;

// do reduction in shared mem
//__syncthreads() needed as operate across warp boundaries
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+128,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+64,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+32,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+16,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +8,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +4,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +2,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +1,top);
__syncthreads();
}
```

Fig. 14 Reduction code to add `n` items in $\log_2(\mathtt{n})$ steps. It calculates both the sum of `correct` (Figure 13) negative and the positive training examples simultaneously. `__device__` function `fsdata()` ensures the reduction code does not include data from threads running other programs, or indeed different classes for the same thread. Totals are left in shared memory index `start`. Will cope with up to 256 negative and 256 positive training cases. Clever use of templates and/or conditional compilation could eliminate operations which are not needed with fewer training cases. Atomic or barrier synchronisation might be an alternatives to `__syncthreads()`.

```
  reduce_sum(start1,n);
  __syncthreads();
  if(threadIdx.x==start) {
    volatile unsigned int *sdata = (unsigned int*) shared_array;
    const unsigned int TN = sdata[start];
    const unsigned int TP = sdata[start+nneg];
    const int penalty = (TP==0||TN==0)? 0 : 2*npos*nneg;
    *d_Output = 1 + penalty + TP*nneg + TN*npos;
  }
```

Fig. 15 `sumfit()` uses `reduce_sum()` (Figure 14) to give the number of `correct` negative (`TN`) and positive (`TP`) training examples. One thread per program calculates AUROC fitness without division (and keeping integer values) but keeping the same relative weighting of `TN`, `TP` and the penalty (Table 1). To aid debugging `1` is added to ensure fitness is never zero. Finally the thread writes fitness to global memory (`d_Output`).

In the final run, in which we interpret many millions of GP programs, they read only eight training cases. Since the L1 cache occupies 16 Kbytes, these 728 values of training data will fit into it and so should remain cached. (`Pop` still occupies 80 Mbytes and so now because the major data item.) The interpreter in the final run achieves only about 200 million more GPOP/second than it does on the large data training data runs. This hints that the kernel data I/O is working well and it is operating near the Fermi's computational limit.

6.10 CUDA Gives Improvements

Whereas RapidMind 2.0 imposed a 2^{22} bit addressing limit (i.e. no more than $\approx$4 million items per array) and no more than sixteen arrays per GPU, CUDA imposes no such limits. Instead all the GPU's memory is directly addressable. Thus originally the population of five million GP programs had to be split into 20 parts and the training data split into 8 or more arrays. Therefore 256 thousand GP programs were passed to the GPU (a GTX 8800) which, on average, took slightly less than a second to interpret them and return their fitness values. This had to be done with each of the twenty parts of the population. Now the whole population is a passed to the Tesla C2050 on one go, interpreted and 5 million fitness values returned to the host, in under a second.

Originally the multiple program outputs (required by splitting the training data into four separate arrays) were summed and combined into a single fitness value per GP individual by three additional GPU program, making a total of seven GPU programs. Now with the simplification allowed by bigger address ranges, the complete GP interpreter and fitness calculation is done by a single CUDA kernel.

6.11 GP for Large Scale Data Mining

We previously described using genetic programming to data mine GeneChip data [55]. Our intention was to automatically evolve a simple (possibly non-linear) classifier which uses few simple inputs to predict the future about ten years ahead. To ensure the solutions are simple (and for speed) the GP trees are limited to fifteen nodes. (Whilst this is obviously small, it is not unreasonable. For example, Yu *et al.* successfully evolved classifiers limited to only eight nodes [109].) Since many GPUs offer more that a gigabyte of onboard RAM both the population size and length of individuals could be increased. Indeed since GPUs can now directly access the host computer's RAM larger populations might be accommodated in large RAM 64 bit servers without explicit direct transfer to the Tesla. Undoubtedly there will be performance implications but assuming reasonable locality, so the data caches now available are not overwhelmed, this might be quite a successful approach. However we have not tried this as yet and instead have kept the explicit data transfer.

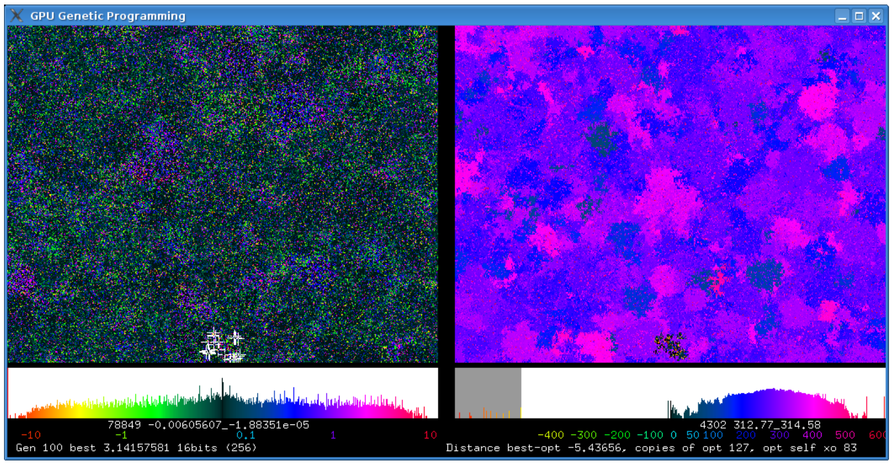
Typically this takes 55 milliseconds (PCIe bandwidth 5.7 Giga bytes/second). Explicit transfer of the 5 million fitness values in the other direction back to the host server takes 3 milliseconds (PCIe bandwidth 5.9 GBytes/second). Both transfers are to/from non-paged ("pinned") host memory. These large data transfers make the best use of the PCIe bus. If they were replaced by the GPU directly accessing the host ("zero copy" transfers) presumably they would be replaced by data transfers limited to the width of the GPU cache which might be less efficient. However they would seamlessly allow the GPU to overlap data transfers and computation, whilst we have not attempted such asynchronous use of the GPU.

Previously [55] we had demonstrated GP on datasets with more than seven thousand inputs (created by pre-processed raw data). Now we have more than a million individual probe values (and compute power to use them). Therefore we asked GP to evolve combinations of the probe values rather than use Affymetrix or other human designed combinations of them. In our approach the first step is to use GP as its own feature selector.

Essentially the idea is to use Price's theorem [89]. Price showed the number of fit genes in the population will increase each generation and the number of unfit genes will decrease. We run GP one hundred times. We ignore the performance of the best of run individual and instead look at the genes it contains. Thus the first pass starts with a million inputs and we select in the region of 10 000 for the second pass and so on until we get down to a reasonable number. Finally GP is run with a much enriched terminal set containing only inputs which had showed themselves to be highly fit in previous GP runs. See Section 7.

The question of how big to make the GP population can be solved by considering the coupon collector problem [21, p284]. On average $n(\log(n)+0.37)$ random trials are needed to collect all of n coupons. Since we are using GP to filter inputs, we insist that the initial random population contains at least one copy of each input. That is we treat each input as a coupon (so $n = 1\,013\,888$) and ask how many randomly chosen inputs must we have in the initial random population to be reasonably confident that we have them all. The answer is 14 million. The spread in the distribution of answers to the coupon collector problem is of the order of square root of n. Therefore if we overshoot by a few thousands, we are sure to get all the inputs (GP tree leafs) into the initial population. Since a program of fifteen nodes has eight leafs and half of these are constants we need at least $\frac{1}{4}(14 \text{ million}) = 3.6$ million random trees. An initial population of five million ensures this.

At the end of the first pass, we want of the order of 100 000 inputs to chose from. This means we need about 25 000 good programs (each with about four inputs). We do not want to run our GP 25 000 times. The compromise was to use overlapping fine grained demes [41] to delay convergence of the population, see Figure 16. The GP population is laid out on a rectangular 2560×2048 grid (See Figure 17). This was divided into eighty 256×256 squares. At the end of the run, the genetic composition of the best individual in each square was recorded. Note to prevent the best of one square invading the next, parents were selected to be within 10 grid points of their offspring. Thus genes can travel at most 100 grid points in ten generations. The GP parameters are summarised in Table 1.



```
78849 -0.00605607_-1.88351e-05
Gen 100 best 3.14157581 16bits (256)
4302 312.77_314.58
Distance best-opt -5.43656, copies of opt 127, opt self xo 83
```

Fig. 16 Screen shot of a 512 × 400 GP population, i.e. 204 800 programs (from run approximating π [53]) evolving under selection, crossover and subtree mutation after 100 generations. Colour indicates fitness (left) and syntax (right). Below are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. (Colour scales below each histograms.) Local convergence and the production of species is visible (especially right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation.

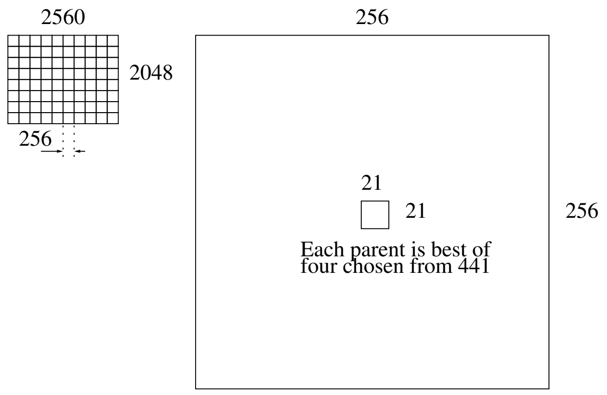


Fig. 17 Left: The GP population of five million programs is arranged on a 2560 × 2048 grid, which does not wrap around at the edges. At the end of the run the best in each 256 × 256 tile is recorded. Right: (note different scale) parents are drawn by 4-tournament selection from within a 21 × 21 region centred on their offspring.

Table 1 GP Parameters for Datamining Uppsala Breast Tumour Biopsies

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	712^2 Affymetrix HG-U133A and 712^2 HG-U133B probe mRNA concentrations. 1001 Constants -5, -4.99, -4.98, ... 4.98, 4.99, 5
Fitness:	Area under ROC curve (AUROC) = $\left(\frac{1}{2}\frac{\text{TP}}{\text{No. pos}} + \frac{1}{2}\frac{\text{TN}}{\text{No. neg}}\right)$ less 1.0 penalty if either all the positive cases or all the negative cases are wrong (TP=0 or TN=0) [54].
Selection:	tournament size 4 in overlapping fine grained 21×21 demes [41], non elitist, Population size 2560×2048
Initial pop:	ramped half-and-half 1:3 (50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 15, no tree depth limit.
Termination:	10 generations

Table 2 Eight Affymetrix probes used most in 8000 best of generation 10 second pass RapidMind GP programs which were used in final RapidMind run [57]. See Figure 18. Second column gives rank in these experiments.

		Used	X,Y	chip	Affy id	NetAffx Gene Title
1	2	579	350,514	A	200903_s_at.mm8	S-adenosylhomocysteine hydrolase
2	10	493	325,511	A	219260_s_at.pm7	C17orf81. chromosome 17 open reading frame 81
3	6	363	254,667	A	201893_x_at.pm2	decorin
4	1	291	392,213	A	219778_at.pm4	zinc finger protein, multitype 2
5	4	286	366,310	B	230984_s_at.mm10	230984_s_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 17 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster.*
6	3	265	324,484	A	216593_s_at.mm9	phosphatidylinositol glycan anchor biosynthesis, class C
7	19	263	542,192	B	233989_at.mm4	EST from clone 35214, full insert. UniGene ID Build 201 (01 Mar 2007) Hs.594768 NCBI
8	41	245	269,553	B	223818_s_at.pm2	remodeling and spacing factor 1

7 Experimental Results

The new CUDA system is much faster but, as expected, the results are similar. GP was run one hundred times with all inputs taken from the 91 training examples using the parameters given in Table 1. After ten generations the best program in each of the eighty 256×256 squares was recorded. The distribution of inputs used by these 100×80 programs is given in Figure 18. Most probes were not used by any of the 8000 programs, 24 892 were used by only one, 2 029 by two, and so on.

The 28 305 probes which appeared in any of the 8000 best of generation ten programs were used in a second pass. In the second pass GP was also run 100 times. (The GP parameters were kept the same).

Eight probes which appeared in more than 200 of the best 8000 programs of second pass were the inputs to a final GP run. (The GP parameters were again kept the same). See also upper trace in Figure 18 and Table 2.

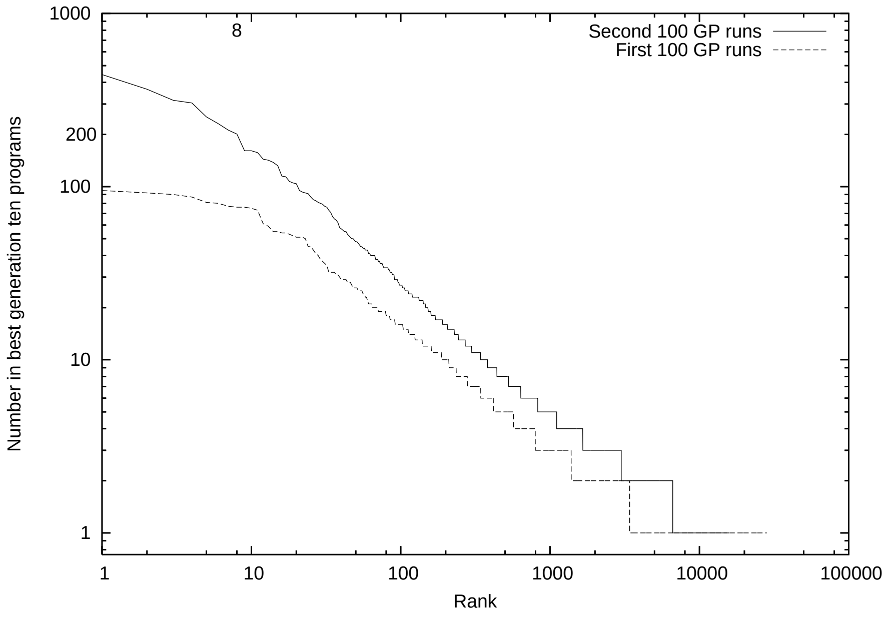
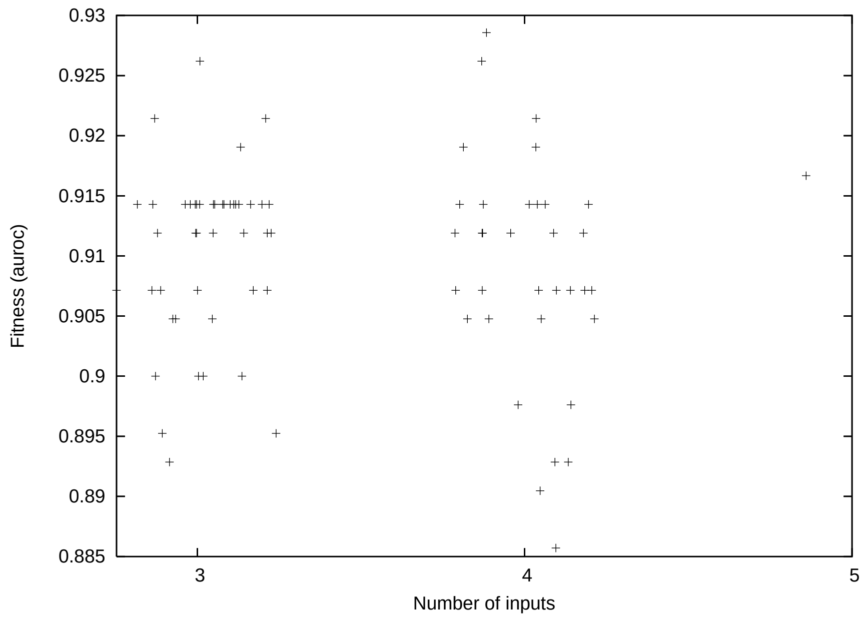


Fig. 18 Distribution of usage of Affymetrix probe in 8000 best of generation 10 GP programs. Both distributions are almost a straight lines (note log scales, cf. Zipf's law [114]) and closely agree with earlier runs [57].

As Figure 19 shows, GP finds many good matches to the 91 training examples, most of the 80 score above 90% and several scoring more than 92%. Ever mindful of over fitting [12], in the original RapidMind runs as a solution we chose one with the fewest inputs (three). GP found a non-linear combination of two PM probes and one MM probe from near the middle of HG-U133A, see Figure 20 and Table 2. The evolved predictor is the sum of two non-linear combination of two human genes, see Figure 21). Both sub-expressions have some predictive ability. The three probes chosen by GP are each highly correlated with all PM probes in their probeset [58] and so can be taken as a true indication of the corresponding gene's activity. The gene names used in Figure 20 where given by the manufacturer's netaffx www pages. Possibly terms like decorin/C17orf81 are simply using division as a convenient way to compare two probe values. Indeed the sign indicates if two values are both above or both below average. (Division appears in all 80 of the best of generation ten programs, slightly more than $+$ and $-$, but much more often than multiplication: $/$ 80, $+$ 72, $-$ 71, $\times$ 27.)

The chosen solution compares well with that produced by Miller *et al.* [78], which used with more than 704 data items compared to GP's three. We also showed in [57] the RapidMind interpreted 535 million GP operations per second (535 MGPop/S). This corresponded to a 7.59 speed up compared to an Intel 2.40GHz CPU.



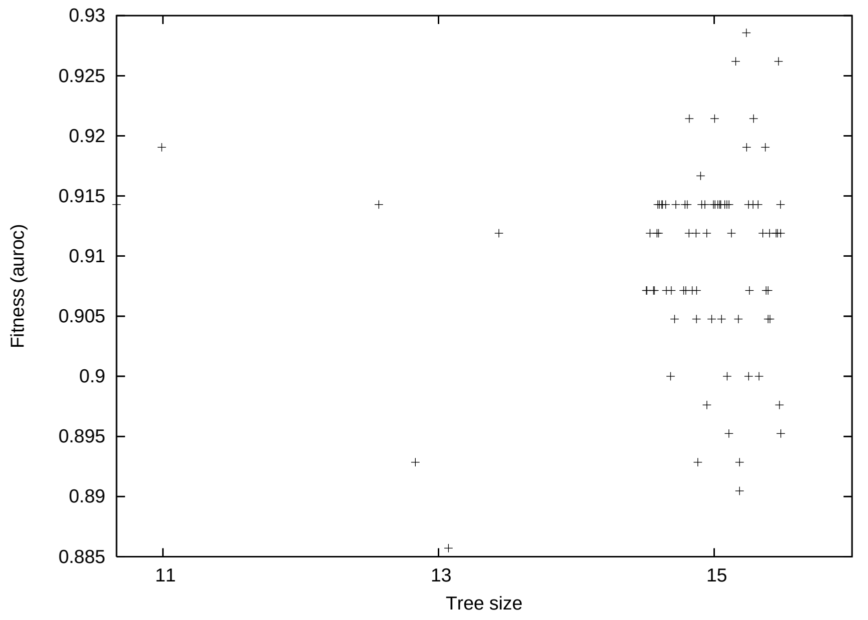


Fig. 19 Spread of performance on training data v. program size. 80 best of generation 10 programs in final CUDA GP run with 8 inputs. Size is given in top graph by the number of different different inputs and by the number of GP instructions in the bottom graph. Noise added to spread data horizontally. Whilst most of these high fitness predictors are of the maximum size (15) most use only 3 or 4 of the eight available inputs.

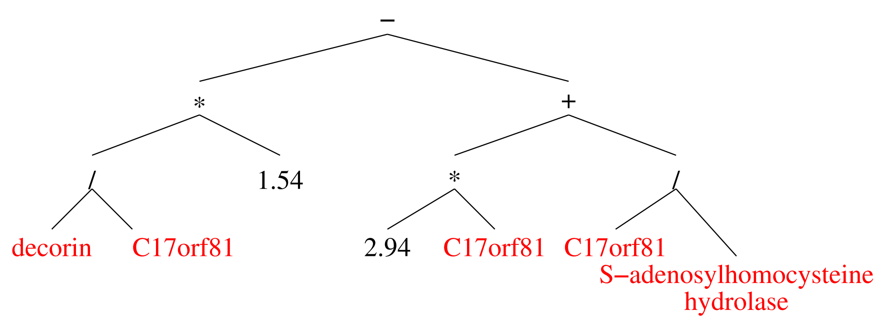


Fig. 20 GP evolved three input classifier. The figure uses gene names. We can also use Affymetrix probe names. Using probe names, the evolved tree says survival is predicted if $1.54\frac{201893\_x\_at.2pm}{219260\_s\_at.7pm} - 2.94\,219260\_s\_at.7pm - \frac{219260\_s\_at.7pm}{200903\_s\_at.8mm} < 0$.

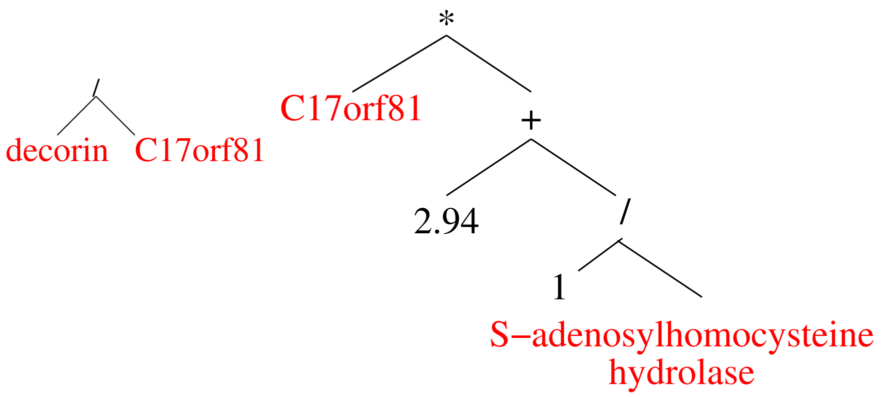


Fig. 21 GP classifier (Figure 20) is the weighted addition of two input classifiers (left and right).

8 Genetic Programming Interpreter Speed on Tesla C2050 GPU

On average across the 201 GP runs the C2050 processes 8.5 billion GP primitives per second. This is fairly consistent, even on the last run, where there are only 8 inputs (effectively 3 Kbytes of global training data). The server has two C2050 Tesla, so the 100 runs of each phase can be split into two and 50 run on each one. On the 4 core server there is little interaction between them and so the combined speed of fitness evaluation using two C2050 is 17 billion GPop/S.

The GPU interpreter's performance 8.5 gigaGPOP/S (line marked ^{o} in Table 3) is very good. It is by far the fastest for a floating point GP data mining application, being surpassed only by our Boolean multiplexor benchmarks [46], graphics applications [32] and a special bench mark [64]. The number of successful applications has expanded in recent years. Where GP operations rates (rather than just speed up ratios) were given, the result is included in Table 3. Interpreter performance is expected to vary somewhat with size of the terminal and functions sets (columns 2–4)

Table 3 Genetic Programming Primitives Interpreted Per Second. Unless otherwise noted the GPU was an nVidia GeForce 8800 GTX

Experiment	No. of Terminals Inputs+ Consts	Funcs	Pop size	Prog size	Stack depth	Test cases	Speed 10^6 OP/S	GPU
Mackey-Glass	8+128	4	204 800	11.0	4	1200	895	
Mackey-Glass	8+128	4	204 800	13.0	4	1200	1056	
Mackey-Glass[a]	8+128	4	204 800	10.2	4	1200	1720	
Protein	20+128	4	1 048 576	56.9	8	200	504	
Laser	3+128	4	18 225	55.4	8	151 360	656	
Laser	9+128	4	5 000	49.6	8	376 640	190	
Sextic[b]	1+0	4	100	16	n/a	200	.5	XBox 360
Sextic[c]	1+0	8	12 500	70.0	17	100 000	4 073	
Image processing[d]	9+na	?	2 048	2048	n/a	$\approx 10^8$	26 200	28×8200
TMBL	?+?	4	120	300	n/a	65 536	191 724[e]	260 GTX
Multiplexor-6[f]	6+0	4	12 500	120.6	17	64	47	
Multiplexor-11[g]	11+0	4	12 500	156.2	17	2 048	501	
Multiplexor-20[h]	20+0	4	262 144	428.5	15	2 048[i]	254 000	295 GTX
Multiplexor-37[j]	37+0	4	262 144	915.6	15	8 192[k]	665 000	295 GTX
GeneChip	47+1001	6	16 384	$\leq$63.0	8	200[l]	314	
Cancer	1 013 888+1001	4	5 242 880	$\leq$15.0	4	128	535[m]	
Cancer[n]	1 013 888+1001	4	5 242 880	12.9	4	91	1 352	C2050
Cancer	1 013 888+1001	4	5 242 880	12.9	4	91	8 517[o]	C2050
Cancer[p]	1 013 888+1001	4	5 242 880×24	12.9	4	91	7 140	M2090
Cancer	1 013 888+1001	4	5 242 880×24	12.9	4	91	9 943[q]	M2090

[a] [93] clusters of ten programs per CUDA block

[b] $x^6 - 2x^4 + x^2$ [104]

[c] $x^6 - 2x^4 + x^2$ [93]

[d] [32] "emboss" image filter evolved with Cartesian GP with distributed nvcc compilation on up to 28 nodes

[e] [64] PTX evaluation only

[f] [93]

[g] [93]

[h] [46]

[i] The 2 048 test cases used were randomly sampled from 1 048 576 available every generation

[j] [46]

[k] The 8 192 test cases used were randomly sampled from 137 438 953 472 available every generation

[l] The 200 test cases used were randomly sampled from 300 000 available every generation

[m] Interpreter speed only

[n] These runs

[o] These runs. Interpreter speed only

[p] [51] Three 8 GPU nodes of the STFC Rutherford Appleton Laboratory Emerald GPU supercomputer. I.e. Total 24×M2090 tesla.

[q] Same runs as [p]. Mean interpreter speed on each of twenty four M2090. I.e. the same kernel runs 40% faster on a shared supercomputer M2090 than it does on our server mounted C2050s. Over all 201 runs (including data transfers, host operations, etc.) the emerald supercomputer has interpreted 33.8×10^9 GP operations per second. This used up to 80 M2090, however the average performance was reduced by significant scheduling delays due to sharing emerald with other users [52].

[38]. The performance of compiled GPs on GPUs can varied widely, e.g. with program size (column 6) and number of training examples (column 8). Table 3 gives the maximum speeds, see the individual references for details of which factors effect speed.

Run time in most genetic programming systems is dominated by the time to calculate fitness, now this is done by the GPU the remaining operations (still done on the host) become more important. Our host code is almost identical to the original RapidMind experiments and has not been optimised. As fitness evaluation speeds up, it may become necessary to parallelise these other parts of the evolutionary process.

The earliest evolutionary computation GPGPU [22] implemented both genetic operations and fitness evaluation on the GPU. More recently Pospichal *et al.* [87] ported both genetic operations and fitness of grammatical evolution onto a GTX 480 with CUDA.

9 The Future of General Purpose GPU Computing

It is gratifying to note that some of our earlier predictions [57] have already come about. For example, we see more and more on chip transistors being used to introduce on chip caches and more on chip memory. We also see routine support for double precision, removal of 22 bit limit on data sizes, direct access to host PC RAM, routine support for 64 bit addressing and direct transfer of data between GPU in the same host computer. The concept of GPGPU has broaden out and is directly supported by nVidia's Tesla range (of non-graphics card GPUs). GPGPU continues to grow.

Like the x86 processor range, modern GPU chips are gathering functionality with great reluctance to remove transistors designed to support older graphics applications such as anti-aliasing. This hardware in unlikely to be useful for scientific computing and so represents an overhead.

Published GPGPU computation has been dominated by nVidia. Initial publications were by people programming scientific applications using graphics tools (e.g. Cg). There was then a move to nVidia's CUDA. In the last couple of years there has been a little interest in OpenCL applications on nVidia cards. OpenCL offers the possibility of porting applications between different graphics hardware. Indeed recently some new GPGPU applications [26] have been coded to use OpenCL on ATI cards.

It is clear that GPU programming is aimed squarely at the high level language programmer. Even the CUDA assembler language PTX is remote from the machine code that the GPU actually runs. Both PTX and high level language sources must be compiled before the GPU can use them. The compilation tools are aimed at one programmer working on one (or a few) program at a time, and aim to produced the very best machine code for the GPU and do not worry about how long it will take to compile. This is fundamentally not suitable for populations of programs.

We have worked around this problem. Harding [32] (and now more recently others) ran the CUDA compiler multiple times in parallel. Lewis showed evolving PTX can reduce the compilation overhead by more than 20% [64]. We have taken the approach of not compiling the GP programs but instead interpreting them. Whilst this allows a single GPU to run millions of programs simultaneously, an interpreter will always be slower than machine code. Nordin [81] was the first to recognise this and built GP systems that both genetically manipulated and ran first SUN machine code and later Intel x86 code. Indeed his x86 system is now the basis of a successful commercial GP system [23]. Whilst GPU machine code is not straight forward [64], we anticipate soon someone will bite the bullet and remove the compiler/interpreter bottleneck by implementing a GP system which evolves GPU machine code directly.

Despite improving tools, both debugging (see [48] and Chapter 2) and performance tuning [50] remain difficult. There is still a risk that if GPUs remain difficult to use, they will remained limited to specialised niches. To quote John Owens "Its the software, stupid" [82].

10 Conclusions

Previously [57] we took a large GeneChip breast cancer biopsy dataset with more than a million inputs and demonstrated genetic programming running in parallel on an nVidia GeForce 8800 GTS and showed a 7.6 speed up compared to a single core PC. The compute intensive fitness evaluation has now be recoded in CUDA and run on modern hardware C2050 Tesla. With the new kernel a single nVidia C2050 delivers about 8.5 billion GP operations per second. I.e. sixteen times faster than the old code with a 8800 GTS, even though in simple terms of peak floating point (single precision) performance the C2050 is just 2.5 times faster.

Two C2050 can deliver 17 GigaGPOP/s. This includes interaction times between host and GPU but not selection, crossover and mutation, which are still done by the original C++ code on the host.

In some ways a genetic programming interpreter is an ideal GPU application. The cross product of the GP population and training case sizes is already huge. If we also include running multiple GP runs in parallel, in these experiments we have 48 billion almost independent calculations which could be done in parallel. Sometimes highly parallel applications can give disappointing results on GPUs because there is little computation per data item and so more time is spent moving data than computing with it. We estimate very roughly 30 machine instructions are needed to interpret each GP primitive. Which gives an "arithmetic intensity" (i.e. the ratio of calculations per data item) of about twenty. This puts the GP interpreter in the upper part of the typically range of arithmetic intensities of 4–64 FLOP/TDE for successful parallel applications [11, p206].

Sections 2 and 3 showed that general purpose computation on graphics processing units is becoming established and there are an expanding range of GPGPU applications, particularly in Bioinformatics. Today GPGPU is dominated by nVidia's

GPUs and CUDA. It may be OpenCL will soon open the way, not to portable GPGPU applications but to more use of ATI and Intel GPU hardware. Undoubtedly the 3GHz ceiling on CPU clocks will mean that the future of computing is parallel and GPGPU will be one of the popular approaches whereby desktop and other applications will exploit parallel hardware.

C++ Source Code

CUDA code can be down loaded via anonymous ftp from `ftp.cs.ucl.ac.uk` or via `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/gpu_gp_cuda.tar.gz` The large data set GSE3494 can also be down loaded from the UCL ftp site `ftp.cs.ucl.ac.uk/genetic/gp-code/GSE3494/`.

Acknowledgements I would like to thank Shigeyoshi Tsutsui, Stan Seibert, Neil Daeche (UCL) and Derek Ross (STFC Rutherford Appleton Laboratory). The two C2050 were donated by nVidia as part of the GISMO EPSRC project.

References

1. Hamid R. Arabnia and Martin A. Oliver. A transputer network for the arbitrary rotation of digitised images. *The Computer Journal*, 30(5):425–432, 1987.
2. Ali Bakhoda, *et al.* Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pp163–174, Boston, MA, USA, 26-28 April 2009. IEEE.
3. Wolfgang Banzhaf, *et al.* Accelerating genetic programming through graphics processing units. In Rick L. Riolo, *et al.*, eds., *GPTP VI*, chpt. 15, pp229–249. Springer, Ann Arbor, 15-17 May 2008.
4. Wolfgang Banzhaf, *et al. Genetic Programming*. Morgan Kaufmann, 1998.
5. Tanya Barrett, *et al.* NCBI GEO: mining tens of millions of expression profiles–database and tools update. *Nucleic Acids Research*, 35(Database issue):D760–D765, January 2007.
6. Carlos Ivan Camargo Bareno, *et al.* Intrinsic evolvable hardware for combinatorial synthesis based on soC+FPGA and GPU platforms. In Natalio Krasnogor, *et al.*, eds., *GECCO companion*, pp189–190, Dublin, 12-16 July 2011. ACM.
7. Alberto Cano, *et al.* Solving classification problems using genetic programming algorithms on GPUs. In Emilio Corchado, *et al.*, eds., *Hybrid Artificial Intelligence Systems*, *LNCS 6077*, pp17–26, San Sebastian, Spain, June 23-25 2010. Springer.
8. Alberto Cano, *et al.* Speeding up the evaluation phase of GP classification algorithms on GPUs. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2011.
9. Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In Panayiotis Bozanis and Elias N. Houstis, eds., *Advances in Informatics, 10th Panhellenic Conference on Informatics, PCI 2005*, *LNCS 3746*, pp415–425, Volos, Greece, 11-13 November 2005. Springer.
10. Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens, *et al.*, eds., *GECCO*, vol 2, pp1566–1573, London, 7-11 July 2007. ACM Press.
11. Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science - Research and Development*, 26(3):205–210, 2011.

12. David Peter Alfred Corney. *Intelligent Analysis of Small Data Sets for Food Design*. PhD thesis, University College, London, 2002.
13. Leandro F. Cupertino, *et al.* Evolving CUDA PTX programs by quantum inspired linear genetic programming. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp399–406, Dublin, 12-16 July 2011. ACM.
14. Andrew W. Dowsey, Michael J. Dunn, and Guang-Zhong Yang. Automated image alignment for 2D gel electrophoresis in a high-throughput proteomics pipeline. *Bioinformatics*, 24(7):950–957, 2008.
15. Marc Ebner. Engineering of computer vision algorithms using evolutionary algorithms. In Jacques Blanc-Talon, *et al.*, eds., *ACIVS*, *LNCS 5807*, pp367–378, Bordeaux, France, September 28-October 2 2009. Springer.
16. Marc Ebner. Towards automated learning of object detectors. In Cecilia Di Chio, *et al.*, eds., *EvoIASP*, *LNCS 6024*, pp231–240, Istanbul, 7-9 April 2010. Springer.
17. Marc Ebner. Evolving object detectors with a GPU accelerated vision system. In Gianluca Tempesti, *et al.*, eds., *ICES*, *LNCS 6274*, pp109–120, York, September 6-8 2010. Springer.
18. Marc Ebner, *et al.* Evolution of vertex and pixel shaders. In Maarten Keijzer, *et al.*, eds., *EuroGP*, *LNCS 3447*, pp261–270, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.
19. Wes Faler. Automatic algorithm invention with GPU. In *28th Chaos Communication Congress*, page ID 4764, Berlin, 27-30 December 2011.
20. Zhe Fan, *et al.* GPU cluster for high performance computing. In *Proceedings of the ACM/IEEE SC2004 Conference Supercomputing*, 2004.
21. William Feller. *An Introduction to Probability Theory and Its Applications*, vol 1. John Wiley and Sons, New York, 2 edition, 1957.
22. Ka-Ling Fok, *et al.* Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, March-April 2007.
23. Frank D. Francone. *Discipulus Owner's Manual*. 11757 W. Ken Caryl Avenue F, PBM 512, Littleton, Colorado, 80127-3719, USA, version 3.0 draft edition, 2001.
24. Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.
25. Stephane Gobron, Francois Devillard, and Bernard Heit. Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, 18(6):331–342, December 2007.
26. Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *GPGPU*, Newport Beach, CA, USA, 2011. ACM.
27. Simon Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In Jun Wang, ed., *WCCI*, pp1921–1928, Hong Kong, 1-6 June 2008. IEEE Press.
28. S. L. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *HPCS*, page 2, Canada, 2007. IEEE Computer Society.
29. Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner, *et al.*, eds., *EuroGP*, *LNCS 4445*, pp90–101, Valencia, Spain, 11-13 April 2007. Springer.
30. Simon L. Harding and Wolfgang Banzhaf. Distributed genetic programming on GPUs using CUDA. In Ignacio Hidalgo, *et al.*, eds., *Workshop on Parallel Architectures and Bioinspired Algorithms*, pp1–10, Raleigh, NC, USA, 13 September 2009. Universidad Complutense de Madrid.
31. Simon Harding and Wolfgang Banzhaf. Implementing cartesian genetic programming classifiers on graphics processing units using GPU.NET. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp463–470, Dublin, 12-16 July 2011. ACM.
32. Simon L. Harding and Wolfgang Banzhaf. Hardware acceleration for CGP: Graphics processing units. In Julian F. Miller, ed., *Cartesian Genetic Programming*, chpt. 8, pp231–253. Springer, 2011.

33. Simon L. Harding, *et al.* Self-modifying cartesian genetic programming. In Dirk Thierens, *et al.*, eds., *GECCO*, vol 1, pp1021–1028, London, 7-11 July 2007. ACM Press.
34. Nicholas Harvey, Robert Luke, James M. Keller, and Derek Anderson. Speedup of fuzzy logic through stream processing on graphics processing units. In Jun Wang, ed., *WCCI*, pp3809–3815, Hong Kong, 1-6 June 2008. IEEE Press.
35. Andrew Howlett, *et al.* Evolving pixel shaders for the prototype video game subversion. In *The Thirty Sixth Annual Convention of the Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB'10)*, De Montfort University, Leicester, UK, 30th March 2010. AI & Games Symposium.
36. Ting Hu, *et al.* Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2):205–225, June 2010.
37. Jacek Izydorczyk and Michael Izydorczyk. Microprocessor scaling: What limits will hold? *IEEE Computer*, 43(8):20–26, August 2010.
38. Hugues Juille and Jordan B. Pollack. Parallel genetic programming and fine-grained SIMD architecture. In E. V. Siegel and J. R. Koza, eds., *Working Notes for the AAAI Symposium on Genetic Programming*, pp31–37, MIT, 10–12 November 1995. AAAI.
39. Sarnath Kannan and Raghavendra Ganji. Porting Autodock to CUDA. In Pilar Sobrevilla, ed., *WCCI*, pp3815–3822, Barcelona, 18-23 July 2010. IEEE.
40. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
41. William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, Boston, 1998.
42. W. B. Langdon. A SIMD interpreter for genetic programming on GPU graphics cards. Technical Report CSM-470, Department of Computer Science, University of Essex, Colchester, UK, 3 July 2007.
43. W. B. Langdon. Evolving GeneChip correlation predictors on parallel graphics hardware. In Jun Wang, ed., *WCCI*, pp4152–4157, Hong Kong, 1-6 June 2008. IEEE Press.
44. W. B. Langdon. A fast high quality pseudo random number generator for nVidia CUDA. In Garnett Wilson, ed., *CIGPU workshop at GECCO*, pp2511–2513, Montreal, 8 July 2009. ACM.
45. W. B. Langdon. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In Francisco Fernandez de Vega and Erick Cantu-Paz, eds., *Parallel and Distributed Computational Intelligence*, chpt. 5, pp113–141. Springer, January 2010.
46. W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna Isabel Esparcia-Alcazar, *et al.*, eds., *EuroGP*, *LNCS 6021*, pp146–158, Istanbul, 7-9 April 2010. Springer.
47. W. B. Langdon. Graphics processing units and genetic programming: An overview. *Soft Computing*, 15:1657–1669, August 2011.
48. William B. Langdon. Debugging CUDA. In Simon Harding, W. B. Langdon, Man Leung Wong, Garnett Wilson, and Tony Lewis, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp415–422, Dublin, 13 July 2011. ACM.
49. William B. Langdon. Generalisation in genetic programming. In Natalio Krasnogor, *et al.*, eds., *GECCO*, page 205, Dublin, 12-16 July 2011. ACM.
50. W.B. Langdon. Creating and debugging performance CUDA C. In Francisco Fernandez de Vega, *et al.*, eds., *Parallel Architectures and Bioinspired Algorithms*, chpt. 1, pp7–50. Springer, 2012.
51. W. B. Langdon. Initial experiences of the emerald: e-infrastructure south GPU supercomputer. Research Note RN/12/08, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 17 June 2012.
52. W. B. Langdon. Distilling GeneChips with genetic programming on the emerald GPU supercomputer. *SIGEvolution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation*, 6(1):15–21, 25 July 2012.

53. William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In Michael O'Neill, *et al.*, eds., *EuroGP*, *LNCS 4971*, pp73–85, Naples, 26-28 March 2008. Springer.
54. W. B. Langdon and S. J. Barrett. Genetic programming in data mining for drug discovery. In Ashish Ghosh and Lakhmi C. Jain, eds., *Evolutionary Computing in Data Mining*, chpt. 10, pp211–235. Springer, 2004.
55. W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257, September 2004.
56. W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, ed., *WCCI*, pp2376–2383, Barcelona, 18-23 July 2010. IEEE.
57. W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, October 2008.
58. W. B. Langdon, A. P. Harrison, and Olivia Sanchez Graillet. RNAnet a map of human gene expression. In *EMBO-2008*, Heidelberg, 15-18 Nov 2008. Abstract presented.
59. W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
60. W. B. Langdon, G. J. G. Upton, R. da Silva Camargo, and A. P. Harrison. A survey of spatial defects in Homo Sapiens Affymetrix GeneChips. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(4):647–653, oct.-dec 2009.
61. W. B. Langdon, Shin Yoo, and M. Harman. Formal concept analysis on graphics hardware. In Amedeo Napoli and Vilem Vychodil, eds., *The Eighth International Conference on Concept Lattices and Their Applications*, pp413–416, Nancy, France, 17-21 October 2011. INRIA Nancy and LORIA.
62. Tony E. Lewis and George D. Magoulas. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In Guenther Raidl, *et al.*, eds., *GECCO*, pp1379–1386, Montreal, 8-12 July 2009. ACM.
63. Tony E. Lewis and George D. Magoulas. Identifying similarities in TMBL programs with alignment to quicken their compilation for GPUs. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp447–454, Dublin, 12-16 July 2011. ACM.
64. Tony E. Lewis and George D. Magoulas. TMBL kernels for CUDA GPUs compile faster using PTX. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp455–462, Dublin, 12-16 July 2011. ACM.
65. Fredrik Lindblad, *et al.* Evolving 3D model interpretation of images using graphics hardware. In David B. Fogel, *et al.*, eds., *CEC*, pp225–230. IEEE Press, 12-17 May 2002.
66. Bing Liu, *et al.* Approximate probabilistic analysis of biopathway dynamics. *Bioinformatics*, 28(11):150–1516, 2012.
67. Chi-Man Liu, *et al.* SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.
68. Weiguo Liu, *et al.* Bio-sequence database scanning on a GPU. In *IPDPS*, Rhodes, Greece, 25-29 April 2006. IEEE Press.
69. Youquan Liu and De Suvranu. CUDA-based real time surgery simulation. *Studies in Health Technology and Informatics*, 132:260–262, 2008.
70. Jörn Loviscach and Jennis Meyer-Spradow. Genetic programming of vertex shaders. In M. Chover, *et al.*, eds., *Proceedings of EuroMedia 2003*, pp29–31, University of Plymouth, Plymouth, UK, April 14-16 2003.
71. Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *IJCNN*, vol 1, pp622–626, 31 July-4 Aug 2005. IEEE.
72. The Van Luong, Nouredine Melab, and El-Ghazali Talbi. Parallel hybrid evolutionary algorithms on GPU. In Pilar Sobrevilla, ed., *WCCI*, pp2734–2741, Barcelona, 18-23 July 2010. IEEE.

73. Ogier Maitre, *et al.* Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In Guenther Raidl, *et al.*, eds., *GECCO*, pp1403–1410, Montreal, 8-12 July 2009. ACM.
74. Ogier Maitre, *et al.* Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In Anna Isabel Esparcia-Alcazar, *et al.*, eds., *EuroGP*, *LNCS 6021*, pp301–312, Istanbul, 7-9 April 2010. Springer.
75. Ogier Maitre, *et al.* EASEA parallelization of tree-based genetic programming. In Pilar Sobrevilla, ed., *WCCI*, pp1997–2004, Barcelona, 18-23 July 2010. IEEE.
76. Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.
77. Jennis Meyer-Spradow and Jörn Loviscach. Evolutionary design of BRDFs. In M. Chover, *et al.*, eds., *Eurographics 2003 Short Paper Proceedings*, pp301–306, 2003.
78. Lance D. Miller, *et al.* An expression signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. *PNAS*, 102(38):13550–5, Sep 20 2005.
79. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.
80. Asim Munawar, *et al.* 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, December 2009.
81. Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., ed., *Advances in Genetic Programming*, chpt. 14, pp311–331. MIT Press, 1994.
82. John Owens. Experiences with GPU computing, 2007. presentation slides.
83. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
84. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. Invited paper.
85. Martin Pedemonte, *et al.* Bitwise operations for GPU implementation of genetic algorithms. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp439–446, Dublin, 12-16 July 2011. ACM.
86. Riccardo Poli, William B. Langdon and Nicholas Freitag 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).
87. Petr Pospichal, *et al.* Acceleration of grammatical evolution using graphics processing units: computational intelligence on consumer games and graphics hardware. In Simon Harding, *et al.*, eds., *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pp431–438, Dublin, 12-16 July 2011. ACM.
88. Raghavendra D. Prabhu. SOMGPU: an unsupervised pattern classifier on graphical processing unit. In Jun Wang, ed., *WCCI*, pp1011–1018, Hong Kong, 1-6 June 2008. IEEE Press.
89. George R. Price. Selection and covariance. *Nature*, 227, August 1:520–521, 1970.
90. J. Reggia, *et al.* Development of a large-scale integrated neurocognitive architecture - part 2: Design and architecture. Technical Report TR-CS-4827, UMIACS-TR-2006-43, University of Maryland, USA, October 2006.
91. Bernardete Ribeiro, Noel Lopes, and Catarina Silva. High-performance bankruptcy prediction model using graphics processing units. In Pilar Sobrevilla, ed., *WCCI*, pp2210–2216, Barcelona, 18-23 July 2010. IEEE.
92. Denis Robilliard, *et al.* Population parallel GP on the G80 GPU. In Michael O'Neill, *et al.*, eds., *EuroGP*, *LNCS 4971*, pp98–109, Naples, 26-28 March 2008. Springer.
93. Denis Robilliard, *et al.* Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, December 2009.

94. Marjan Rouhipour, *et al.* Systemic computation using graphics processors. In Gianluca Tempesti, *et al.*, eds., *ICES*, *LNCS 6274*, pp121–132, York, September 6-8 2010. Springer.
95. Mikiko Sato, Yuji Sato, and Mitaro Namiki. Acceleration experiment of genetic computations for sudoku solution on multi-core processors. In Christian Blum, ed., *GECCO Late breaking abstracts*, pp823–824, Dublin, 12-16 July 2011. ACM.
96. Pitchaya Sitthi-amorn, *et al.* Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):article:152, December 2011. Proceedings of ACM SIGGRAPH Asia 2011.
97. Nicolas Soca, *et al.* PUGACE, a cellular evolutionary algorithm framework on GPUs. In Pilar Sobrevilla, ed., *WCCI*, pp3891–3898, Barcelona, 18-23 July 2010. IEEE.
98. A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, Nov 1 2006.
99. Cole Trapnell and Michael C. Schatz. Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing*, 35(8-9):429–440, 2009.
100. Tatsuo Unemi. SBArt4 – Breeding abstract animations in realtime. In *WCCI*, Barcelona, Spain, 18-23 July 2010. IEEE Press.
101. Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.
102. Garnett Wilson and Wolfgang Banzhaf. Linear genetic programming GPGPU on Microsoft's Xbox 360. In Jun Wang, ed., *WCCI*, pp378–385, Hong Kong, 1-6 June 2008. IEEE Press.
103. Garnett Carl Wilson and Wolfgang Banzhaf. Deployment of CPU and GPU-based genetic programming on heterogeneous devices. In Anna I. Esparcia, *et al.*, eds., *GECCO Workshop on Computational intelligence on consumer games and graphics hardware (CIGPU-2009)*, pp2531–2538, Montreal, 8-12 July 2009. ACM.
104. Garnett Wilson and Wolfgang 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.
105. Garnett Wilson and Simon Harding. WCCI 2008 special session: Computational intelligence on consumer games and graphics hardware (CIGPU-2008). *SIGEvolution*, 3(1):19–21, Spring 2008.
106. Adrianto Wirawan, Chee Kwoh, Nim Hieu, and Bertil Schmidt. CBESW: sequence alignment on the playstation 3. *BMC Bioinformatics*, 9(1):377, 2008.
107. Man Leung Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *GECCO*, pp2515–2522, Montreal, 8-12 July 2009. ACM.
108. Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. Research Note RN/12/03, Department of Computer Science, University College, London, UK, 8 May 2012.
109. Jianjun Yu, *et al.* Feature selection and molecular classification of cancer using genetic programming. *Neoplasia*, 9(4):292–303, April 2007.
110. Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. GPU-based implementation of real-time system for spiking neural networks. In Pilar Sobrevilla, ed., *WCCI*, pp2143–2150, Barcelona, 18-23 July 2010. IEEE.
111. Ling Sing Yung, Can Yang, Xiang Wan, and Weichuan Yu. GBOOST: a GPU-based tool for detecting gene-gene interactions in genome-wide case control studies. *Bioinformatics*, 27(9):1309–1310, 2011.
112. Jianfu Zhou, Xiaoguang Liu, Douglas S. Stones, Qiang Xie, and Gang Wang. MrBayes on a graphics processing unit. *Bioinformatics*, 27(9):1255–1261, 2011.
113. Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael P. H. Stumpf, and Chris Barnes. GPU accelerated biochemical network simulation. *Bioinformatics*, 27(6):874–876, 2011.
114. George Kingsley Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press Inc., 1949.