Soft Computing (2011) 15:1657–1669 DOI: 10.1007/s00500-011-0695-2
Preprint

Graphics Processing Units and Genetic Programming: An overview

W. B. Langdon

CREST centre, Department of Computer Science, University College, London, Gower Street, London, WC1E 6BT, UK

Abstract A top end graphics card (GPU) plus a suitable SIMD interpreter, can deliver a several hundred fold speed up, yet cost less than the computer holding it. We give highlights of AI and computational intelligence applications in the new field of general purpose computing on graphics hardware (GPGPU). In particular we survey genetic programming (GP) use with GPU. We give several applications from Bioinformatics and show how the fastest GP is based on an interpreter rather than compilation. Finally using GP to generate GPU CUDA kernel C++ code is sketched.

1 Introduction

Throughout my life computing has been dominated by Moore's Law [1]. The doubling of the number of components available on integrated circuit chips every eighteen months has been taken as an obvious fact of life and similar exponential rises have occurred in processing power and storage capacity. The compound 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 decade we have seen no major increase in CPU clock speed since the 3GHz Pentium. (See lower plot in Figure 1.) If the $1\frac{1}{2}$ year doubling had continued we would have 20-25GHz Pentium's on our desks and in our laptops. This has not happened. It looks like it will never happen.

In the last century hardware manufactures were able to help the software industry out of the hole it continues to create for itself by continually increasing software

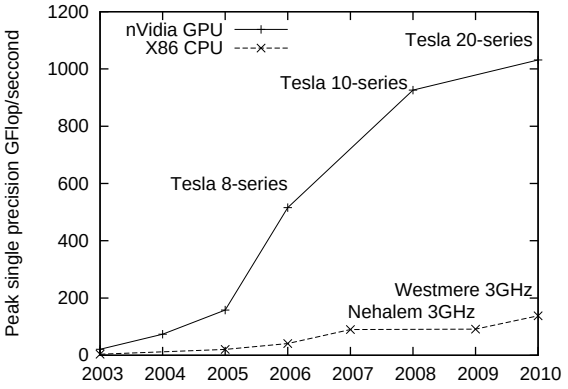

Fig. 1 Comparison of increase in speed of graphics cards (+ GPU) and CPU (× X86). (Data supplied by NVIDIA.) Similar trends hold for double precision and integer performance.

complexity and so processing load. They did this by repeatedly doubling both memory capacity and individual CPU power. Those days are gone. However, in its original sense the integrated circuit manufactures have obeyed Moore's Law and the number of transistors per chip has continued to increase. Indeed in a recent article, Izydorczyk and Izydorczyk [2] argue heat dissipation will remain a major limit on processing power nevertheless they suggest 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 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 since even before the Transputer [3].

Operating systems have been successfully adapted to multi-core architecture but it continues to be only spe-

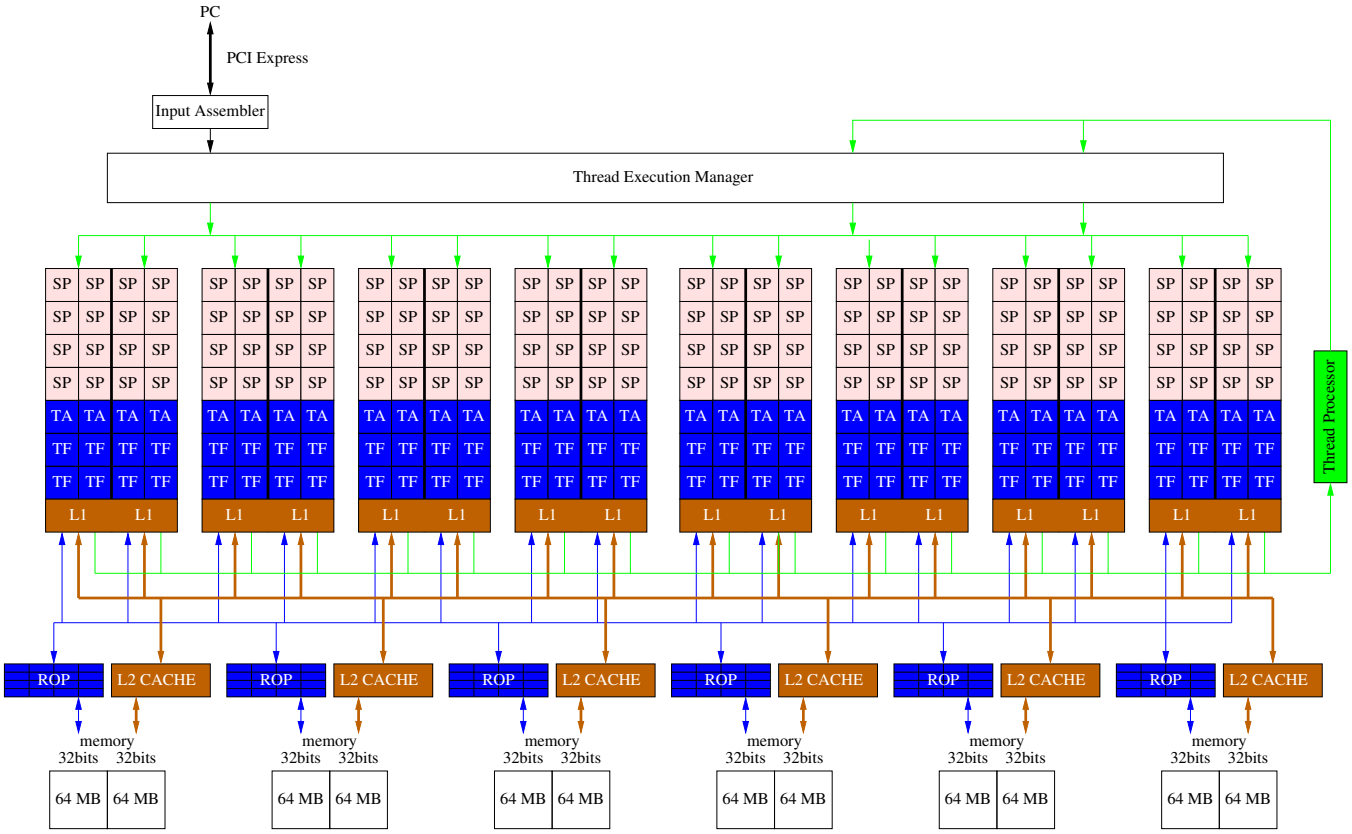

Fig. 2 NVIDIA 8800 Block diagram. The 128 1360 MHz Stream Processors are arranged in sixteen blocks of eight. Blocks share 16 KB memory (not shown), an 8/1 KB L1 cache, four Texture Address units and eight Texture Filters. The 6×64 bit bus (brown) links off chip RAM at 900 (1800) MHz. There are 6 Raster Operation Partitions (ROP). In more recent high end NVIDIA GPUs the multiprocessor blocks operate at similar clock speeds but can contain 24 or even 32 stream processors and there may be ten or even fourteen multiprocessors. Also newer designs tend to have higher speed off chip memory connected by higher bandwidth buses to the GPU chip with improved data caches.

cialised software that really takes advantage of parallel hardware. A good parallelism strategy is simply to run different applications on different cores. This can be effective where many user's share a computer but it is common place for each multi-core machine to have a single user, which limits the number of different active applications required at a time. The most useful aspect of multi-core computers remains to have a spare core free to allow the user to intervene, without crashing the whole system, when a rogue application locks up its core.

The pedestrian pace with which the CPU manufactures have exploited the riches Moore's Law continues to shower on the world, is in stark contrast with the attitude of the manufactures of computer games. They have been much more aggressive. Whilst four core CPUs are now common, PlayStation Cell processor's contain eight cores and state-of-the art graphics cards contain hundreds of processors. (See upper plot in Figure 1 and Figure 2.)

While the balance between off-chip and on-chip memory may change the basic GPU architecture seems to be well able to take advantage of more transistors. In the near term we would expect to see Moore's law doubling of transistors leading to continued rapid increase in number of processing cores per GPU. Thousands of cores rather than today's hundreds appears feasible. Also some chip area might be used to continue the trend towards more functionality per core and larger on chip memory and extended data and instruction caches. However Section 4.4 will describe some alternatives to GPUs and it may be business rather than technological reasons that eventually limit GPUs.

To be commercially successful games consoles and GPUs have to be affordable. More than 100 million GPUs have been sold [4], compared to earlier specialised parallel machines, such as MasPar's MP-2, where about 250 were built. Software engineers have been alive to the possibilities of exploiting GPUs with teraflop performance for purposes other than entertainment. In recent years the field of general purpose computing on GPU (GPGPU) has become established [5]. Surprisingly software researchers were a little slower to recognise the potential of GPUs. Nevertheless the number of uses of GPGPU in computer science continues to rise and as we shall see GPGPU is increasingly being demonstrated in artificial intelligence (AI). For example this summer

(2010) we saw GPGPU competitions at the GECCO conference and the third CIGPU (held as a hybrid special session of the WCCI conference). Together these two conferences published approximately 25 AI papers on GPGPU.

The use of GPUs with genetic programming is relatively new and perhaps as a consequence its development so far has been a little uneven. However, we are now at a good point. As we shall see, the basic frame work is established and shortly we will see rapid infilling of as yet sparse research areas. Early applications combining GPUs and GP were concentrated in Bioinformatics. Again, we expect there will soon be progress in other application areas.

The next section will briefly describe current research in AI techniques which have exploited GPUs and then Section 3 will concentrate upon one such technique, genetic programming (GP). Before considering the way ahead (Section 4), Section 3.5 will summarise applications using GPU and GP.

2 Computational Intelligence on GPU

It appears that some of the interactive game manufactures have considered using the GPU to reduce the load on the CPU caused by the artificial intelligence required to drive sophisticated autonomous agents which the computer provides to play against the user. However it is not clear as yet how widespread or successful this is. Gamers would not want to see clever AI opponents causing the GPU to slow down or otherwise degrade on screen performance. Nevertheless there are quite a few open academic publications where non-traditional AI techniques have been ported to GPUs. Often considerable speed ups are reported.

2.1 Neural Networks

There are many flavours of artificial neural network, however typically they have many co-operating elements and so are potential candidates for exploiting parallel hardware. Typical applications do not require enormous local interconnectivity and so neural network implementations may be able to exploit the relatively small amount of very fast memory associated directly with each GPU stream processing element. (Figure 2.)

Oh and Jung [6] and Luo *et al.* [7] both treat neural networks as matrix operations and report considerable speed ups by using a GPU when the neural network is used on compute intensive image processing tasks. For example, Ribeiro *et al.* [8] demonstrated speed ups in the region of 170 fold when using Multiple Back-Propagation to predict bankruptcy. As well as multi-layer perceptrons (MLPs), Luo *et al.* [7] and Prabhu [9] have demonstrated Kohonen's self organising maps (SOMs) on NVIDIA GPUs.

More biologically realistic simulations of neural networks tend to be more computationally demanding, consequently there has been interest in exploiting GPUs to speed up spiking neural networks [10,11,12,13,14,15,16,17,18]. Whilst Taha *et al.* [19] used a cluster of Sony PlayStation 3s (each containing an STI CELL processor) rather than GPUs.

2.2 Fuzzy Systems

GPUs have been used to improve both rule construction and inference [20] and speed up fuzzy clustering [21,22]. For example Harris and Haines [21] used NVIDIA's Cg language to implement a fuzzy system. With an early GPU (an NVIDIA GeForce 5900 Ultra which had two fragment processors) they report a consistent doubling of speed when finding the centres of clusters in 2D images. Harvey *et al.* [20] implemented a fuzzy system in Cg to be used for video based health monitoring of the elderly. They report a speed up in excess of 150 times from a single 8800 GTX. Similarly Anderson *et al.* [22] used Cg but tested three NVIDIA GPUs (7800, Quadro FX 2500M, as well as a 8800 GTX). They report relative performance improving both with the number of processing elements in the GPU but also with the size and complexity (number of fuzzy clusters) of the data.

2.3 Cellular Automata

Cellular automata (such as Conway's Game of Life) are natural candidates for parallel hardware. Early GPU work includes John Tran's poster at SIGGRAPH 2004 [23]. Whilst Gobron *et al.* [24] use openGL to implement a nice GPU/cellular automata combination which speeds up an image processing system inspired by the human eye.

2.4 Particle Swarm Optimization (PSO)

Swarm intelligence is a population based computational intelligence technique which naturally benefits from parallel hardware. Mussi *et al.* have demonstrated using GPUs to enable PSOs to automatically detect and categorise traffic road signs in real time [25].

2.5 Ant Colony Optimisation (ACO)

Surprisingly only modest speed up were reported initially [26]. GPU work using artificial ant pheromone trails appears to have concentrated upon NP-hard problems like the travelling salesman problem. However recently Sinnott-Armstrong *et al.*'s prize winning work [27] claims "substantial" speed ups in a Bioinformatics application.

2.6 Evolutionary Computation

Ka-Ling Fok *et al.* [28,29] were perhaps the first to implement a complete genetic algorithm on a GPU. Their system included not only evaluating fitness but included storing the population on the GPU and the genetic operations used to introduce changes and so search for improved solutions. They implemented mutation specifically for the GPU but decided not to provide crossover, fearing it would prove difficult and expensive on their GPU. In contrast, on a more modern GPU and using CUDA, Arora *et al.* [30] were able to implement both binary and real coded genetic algorithms including crossover. Whilst Kannan and Ganji [31] use a GPU to apply a GA to three dimensional drug modelling and Tsutsui and Fujimoto [32] used an NVIDIA 285 GTX to solve large quadratic assignment problems.

Now-a-days many other types of evolutionary algorithm have been implemented on GPUs, including: multi-objective [33], Cellular Evolutionary Algorithms [34], Differential Evolution [35], Bayesian Optimization Algorithm (BOA) [36], Classifier Systems (LCS) [37] and hybrid GA with local search [38]. However, as demonstrated by Clayton *et al.* [39], GPU can provide substantial benefits to all kinds of evolutionary computation (they considered: an Estimation of Distribution Algorithm (EDA), PSO and several types of GA) by simply speeding up fitness evaluation. Similarly Chwatal *et al.* [40] showed an eight fold speedup in fitness evaluation for when detecting exoplanets with Evolution Strategies.

3 Genetic Programming on GPU

Genetic programming (GP) [41] is a branch of evolutionary computation in which the population of potential solutions are executable programs whose performance (fitness) is given by running them and comparing the result they calculate with the desired answer. Typically each individual member of the evolving population is run several times in a sequence of fitness test cases. After each run, its answer is compared with the ideal answer for the current test case. Often the overall fitness is given by the sum of the errors made on all of the test cases. As with other genetic algorithms (Section 2.6), new populations are formed by selecting the better programs from the previous generation and mutating them. There is even an analogue of sexual recombination (crossover), whereby parts of two high fitness programs are spliced together to form a new child.

Typically in GPU implementations the selection and genetic operations (mutation and cross-over/recombination) are done by the host PC and the GPU is only used to run the evolved program in order to determine their fitness. This is not fundamental. Other operations might also be run on the GPU. (Ka-Ling Fok *et al.* [29] and others have shown this can be done for linear genetic algorithms.) However since typically almost all the computational effort is taken by fitness evaluation, there is little incentive to speed up the remaining part of the code.

The earliest uses of GPUs with genetic programming, used the GPU as it was intended: to render 3D images. Lindblad *et al.* [42] gave genetic programming the (inverse) task of generating virtual 3D models whose rendered appearance matched as closely as possible real objects as seen by a robot. Meyer-Spradow and Loviscach (e.g. [43]) were the first to run evolved programs on the GPU. Now-a-days CUDA and OpenCL offer more sophisticated GPGPU facilities, however Meyer-Spradow and Loviscach successfully evolved short linear assembly language program pixel shaders and used them for interactive real-time rendering. (At present CUDA is the system of choice and there are no GP system implemented in OpenCL. Doubtless this will soon change.) Ebner *et al.* [44] use a similar approach to evolve new pixel shaders for use within interactive evolution. However, rather than manipulating assembler directly, they use genetic programming to create shaders written in the high level graphics language Cg.

3.1 Compiled, Parallel Testing

Excluding the use of GPUs for graphics rendering (their original designed purpose, see previous section) Harding and Banzhaf were the first to run evolved programs on GPUs. In [45] Microsoft's accelerator research tool was used to compile Cartesian GP graph individuals into programs and run them on a GPU. Note the parallelism comes from the GPU's ability to run the same program simultaneously multiple times on different data. This approach works best when each GP individual must be run many times in order to determine whether it will be used to create the next generation or not. (I.e. its fitness.) Chitty [46] used NVIDIA's Cg rather than accelerator. He also ran a single GP program at a time on the GPU. Again the GPU's advantage only comes when many fitness cases can be run in parallel.

3.2 Interpreted, Parallel Populations

Genetic programming and other population based approaches have been called "embarrassingly parallel" from the ease with which they can be run on traditional parallel computers and their ability to make good use of the hardware. Since each member of the population can be tested on its own, the whole population can be run in parallel. (GP populations range from a few hundred individuals up to a few millions.)

To get the best of modern GPU's containing hundreds of stream processors it is necessary to run thousands or tens of thousands of parallel computation threads. There are two main causes for this. 1) The depth of the computation hardware pipeline in each stream processor.

Depending on the type of the instructions, a stream processor may be capable of overlapping the execution of up to four instructions simultaneously. 2) Threads are often paused because they are waiting for off-chip data to arrive. If other threads are ready to execute (because their data has arrived), the stream processors can start them immediately, rather than becoming idle. While somewhat application and GPU dependent it seems at least twenty computation threads per stream processor are needed to get near to the GPU's best performance. (NVIDIA's G80 architecture allows up to 512 threads per multi-processor block of eight or more stream processors. Version 2.0 allows up to 1024 threads per block.)

Apart from waiting for off-chip data, another major cause for GPUs not delivering their peak performance in practice, is that the stream processors in a multi-processor block are inherently locked in step. Each of them execute exactly the same instruction at the same time, albeit on different data items. This is known as the Single Instruction Multiple Data (SIMD) parallelism [47]. SIMD parallelism works well for many graphics applications, where exactly the same transformations are needed for many different data items (e.g. pixels). The GPU architecture causes a problem where a program contains branches (if statements, switch, case, etc.) and different threads take different branches. This is known as thread divergence. The GPU hardware will pause the divergent thread, and run the others. At some point, it will pause them and restart the divergent threads. Only when all threads reach the end of the branch and return to running the same code in lock-step will the GPU be able to deliver its full power.

In many of the cases described in Section 2 the algorithms do indeed perform the same action on each datum and so threads can operate in lock-step and not diverge. This is also sometimes true of the early GP approaches in Sections 3 and 3.1. In these approaches typically a single program is run in parallel threads which simultaneously act upon different parts of the training set. However even threads running a single GP program can diverge. For example: 1) because IF-like branches are included in the function set. 2) Optimisations allow short cuts. For example, when implementing AND and OR, it is common to evaluate only one argument, if by doing so, it is known that the other argument can have no effect. E.g. AND(x=0,y) will be false, so there is no need to evaluate y. Thus a thread where x=0 will take a different path to one where x=1. (The same thing could even happen in symbolic regression problems using the traditional four functions: $+$, $-$, $\times$ and $\div$, since $\div$ is usually "protected" [41] by treating divide by zero as a special case. However the GPU floating point hardware can recognise divide by zero and generate and propagate special values NaN. So on GPUs, the GP function set may not be explicitly "protected".) Finally 3) even if the GP program itself does not diverge, threads may diverge during fitness evaluation if the fitness function involves running something else which diverges. E.g. fitness is calculated by running a simulator.

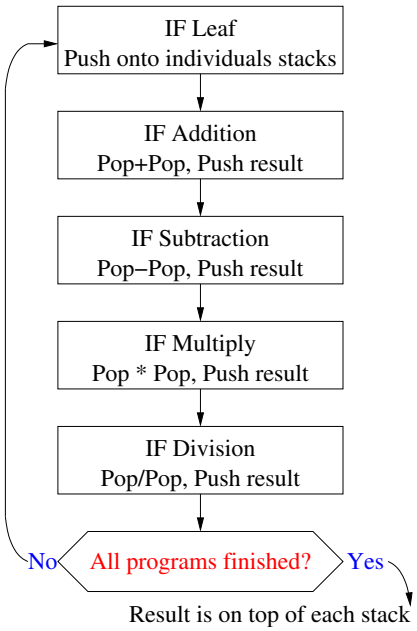

Fig. 3 The Reverse Polish Notation (RPN, postfix) SIMD interpreter loops continuously through the whole genetic programming instruction set for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks.

In some cases it is possible to avoid divergence by having each thread perform both sides of the branch and using a data indexing operation to select the appropriate branch's answer. In principle avoiding divergence could have benefits, like allowing global memory access to be coalesced and keeping instruction pipelines full, which could more than pay back for the extra computation. However the hardware scheduling and resumption of stalled threads appears to be sufficiently smooth that there may be little to be gained [48]. However Arora *et al.* [30, p3683] advocates replacing a simple IF with a more complicated algebraic expression (which effectively does both branches of the IF).

Regardless of whether the program diverges, the GP approaches in Sections 3 and 3.1 test one compiled GP program at a time. This restricts the number of simultaneous threads they can use, typically to the number of fitness test cases. With more powerful GPUs with more stream processors per multi-processor block this becomes more serious.

Our GPU SIMD interpreter approach [49] took the opposite approach. It avoids the (surprisingly high) cost of compiling GP individuals by using an interpreter and secondly it processes the whole population in parallel. (See Figure 3.) You will notice the main SIMD interpreter loop contains an IF statement for every instruction that can appear in the GP language. So even though we can have a huge number of threads (at least one per member of the population) the interpreter can suffer

from divergence. Juille and Pollack [50] originally proposed a SIMD GP interpreter for the MasPar MP-2. They recognised on the MP-2 [50, Sec. 17.2.3] the SIMD architecture lead to an overhead "directly related to the size of the instruction set interpreted by the virtual processor". However the GPU SIMD interpreter approach primarily wins because it allows many threads which in turn allows the GPU to better overlap reading its own memory.

It turns out moving data around within the GPU can be much more expensive than computing with it after it has arrived. The size of the instruction set depends upon the application but in GP between five and ten is common. However rather than thread divergence imposing a 5–10 fold overhead, this is masked by overlapping other GPU operations (such as reading memory). Therefore an overhead of about 2.5 is more usual. Where multiple test cases are needed the divergence overhead can be made negligible (see next section).

3.3 Compilation v. Interpreter

In computer science we are used to the trade offs between compiling and interpreting code. (While assembler, Java byte code and even machine code have been evolved with genetic programming, there has been little work on evolving either GPU machine code or PTX assembler.) Despite the introduction of Java and other interpreted languages, we expect compiled code to be faster than interpreting the code. Normally the compiler is preferred because we expect the sum of the run times of the program to exceed the compilation time. However GP programs tend to be lightweight and possibly run few times, so it is common to use a purpose built interpreter.

One way to reduce the compilation overhead is to convert the whole population into a single program and compile it [51]. The complete population program contains (non-evolved) stub code to run each of its component individuals and record their fitness. By running the compiler only once, rather than once per member of the population, considerable savings are made and typically the compiled code runs much faster than interpreting it.

It turns out that the NVIDIA nvcc CUDA compiler is particularly slow and so interpreting usually wins hands down. However Harding and Banzhaf [52] describe a way of reducing the compilation overhead by running the compiler in parallel across a cluster of workstations. nvcc is a wrapper which encompasses many stages. Significant savings may be possible by disabling some options and/or by having GP operate at lower levels, i.e. closer to the executable instructions.

3.4 Recent Interpreters

In [53] we argued that the fitness evaluation of the GP population can be treated as a cube of work to be done. The cube has dimensions parallel to the length of the programs, running across the members of the populations, and another through the test cases to be run. The GPU SIMD interpreter approach allows the cube to be sliced in many ways. The slices can be stacked together and divided between the stream processors in ways that suit each application (However, since a great deal of check pointing would be required, at present, it does not make sense to slice at right angles to the programs' lengths.)

In [54] we integrated three levels of parallelism to show a GPU SIMD interpreter is capable of delivering a sustain average of 254 billion GP operations per second. At present, this is about twenty times faster than the best compiled approach and sixty times that of the next interpreter. Doubtless it could be improved further. (Equally well, we can expect improvements in other approaches.) 254 10^9 GPops/sec is a 10,000 fold speed up compared to TinyGP [41] running on the 2.6 GHz PC used to host the NVIDIA 295 GTX. (TinyGP is efficiently written in C but does not use all the tricks. In particular it does not use bit level parallelism [55]. It is these multiple layers of parallelism that give our GPU SIMD interpreter its enormous advantage. Nevertheless it should be possible for you to use these techniques in your own application.)

As expected, in [54] we found the best performance comes from slicing the computational cube both along the per individual and along the per test case dimensions [56]. (Robilliard *et al.* [57,58,59] later called their similar approach "blockGP".) This gives millions of active threads. The interpreter allows all the per test case threads associated with a single program to be run on the same multiprocessor. Thus, despite the IF branches shown in Figure 3, all the threads in that multiprocessor run exactly the same instruction at the same time. Hence in this case there is no divergence. (As mentioned in Section 3.2, other instruction sets could introduce divergence.) However [54] again showed that, as long as there are sufficient threads, the GPU SIMD interpreter's performance is fairly stable and falls only by a modest fraction (between 2 and 3) even when different GP programs from the same population are simultaneously interpreted by the same block of stream processors.

A recent innovation is to extend the EASEA evolutionary algorithm framework with a `-cuda` switch to cause it to run on GPUs [60]. In [61] Maitre *et al.* demonstrate EASEA's population parallel SIMD GP interpreter. They show impressive speed ups on an NVIDIA 295 GTX. These include learning aircraft control using a population of 40 960 and 51 000 training values obtained from flight data.

3.5 Bioinformatics Applications

For many practical and public policy reasons Bioinformatics is a rapidly expanding area. Many of its fundamental algorithms are inherently parallel and so quite a few Bioinformatics groups have been amongst the first adaptors of GPUs for non-graphics work. (The next paragraph describes just two non-GP examples.) Some of these groups also have an interest in genetic programming. It is for these practical reasons that there are substantially more publications on real GP applications using GPUs in Bioinformatics. These are described in the rest of this section.

The Smith-Waterman algorithm lies behind many Bioinformatics applications, such as sequence alignment. It is highly parallel with only local interactions. However since it works across the trailing diagonal of square matrices, this makes good implementations of it slightly tricky, nonetheless Liu *at al* [62] reported a 16 fold speed up. (More recent work includes that of Manavski and Valle [63]. While Wirawan *et al.* [64] accelerate the Smith-Waterman algorithm by running it on PlayStation games consoles.)

In the aftermath of the human genome project, modern Biology has become awash with huge datasets (such as the GeneChip data used in the next two sections). GPUs have been demonstrated in data mining. For example, Sinnott-Armstrong *et al.* [65] show a single computer with three GTX 260 GPUs has similar performance to a 150 node Beowulf cluster when using multifactor dimensionality reduction (MDR) on a genetic (SNP) dataset.

3.5.1 Predicting GeneChip Probe Performance While Sinnott-Armstrong *et al.* [65] analysed a 1600×1000 discrete single point mutation dataset, GPUs have also been used to analyse continuous whole human gene expression levels. In [66] we were interested in correcting faults in GeneChip gene expression measurement data. Using a population of 16 384, a single GPU containing 128 stream processors was able to find predictive patterns in a $5\,310\,652 \times 6685$ RNA gene expression correlation dataset. This GPU work gave an early indication of problems in the GeneChip data. These where taken up by a grammar approach [67] which discovered a Cytosine and Guanine rich motif (`GGGG|CGCC|G(G|C){4}|CCC`) that indicates potentially poor data.

3.5.2 Genetic Factors in Breast Cancer Survival Between 1987 and 1989 gene expression was measured in most of the breast cancers surgically removed in Uppsala (Sweden). (This data was gathered because a genetic link to cancer was suspected.) This gave a dataset of $1\,013\,888 \times 251$ continuous readings. In [56] we describe how populations of 5 242 880 small GP programs were run on a single GPU. Multiple GP runs were used to progressively filter the dataset. The resulting small predictive model is shown in Figures 4 and 5.

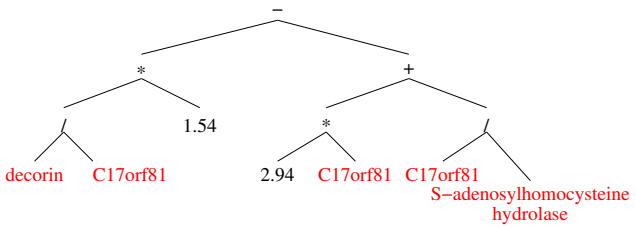

Fig. 4 GP evolved Breast cancer outcome predictor based on just three gene expression measurements. Survival is predicted if $1.54\frac{201893\_x\_at.2pm}{219260\_s\_at.7pm} - 2.94\,219260\_s\_at.7pm - \frac{219260\_s\_at.7pm}{200903\_s\_at.8mm} < 0$ (In the equation we have used the Affymetrix probe names corresponding to the gene names used in the picture.) Figure 5 shows this can be further simplified.

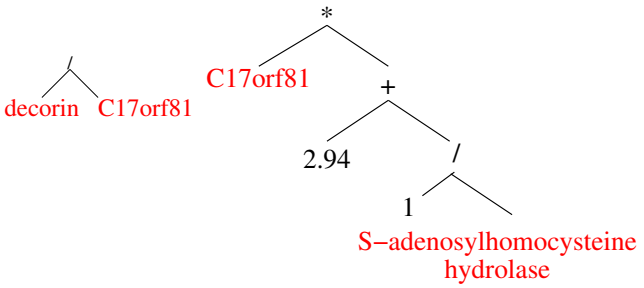

Fig. 5 The GP classifier (Figure 4) is the weighted addition of two 2 input classifiers (left and right).

Huang *et al.* [68] demonstrated a GPU based breast cancer prognosis system on 15 biopsies. However their image processing system is totally different and used optical microscopes to probe the appearance of potential tumours rather than using GeneChips to measure gene expression.

4 Discussion

4.1 Future of Genetic Programming on GPU

Mostly we have talked about running GP programs on GPU cards, however, as we saw in Section 3, GPUs can also be used to speed up calculations required by the fitness function. Particularly where a GP individual's fitness requires a simulation to be run, this can be very computationally demanding. A GPU may be the solution. For example, Rieffel *et al.* [69] showed highly complex physical interactions inherent in soft body interactions could be effectively calculated using PhysX on a GPU.

While it is possible to mount multiple GPUs in a single workstation, the practical problems of motherboard connections, physical space and power consumption mean more than two GPUs per PC are rare. In contrast, systems with three or four Tesla[1] GPUs mounted

[1] Tesla are high end NVIDIA GPUs dedicated to computation rather than generating graphical images.

outside the host PC's own box are in use. Lewis [70] demonstrates a nice multi-threaded twin GPU system where operations on the host neatly dovetail with fitness evaluation on the GPUs.

Despite the fact that AMD's ATI series of GPU appear to be competitive with NVIDIA's almost all recent GPGPU work has used NVIDIA's CUDA. There are good reasons for using CUDA. For example, in [49] we used an early version of RapidMind (version 2), Robilliard *et al.* [59] reimplemented it using an up to date version of CUDA and reported an (up to) 92% speed increase.

4.2 Genetic Programming with MIMD GPUs

In the near term, it is clear that GPUs will continue to increase in performance and capability. For example, NVIDIA's Fermi architecture not only includes many more processing elements but also aspects of multi-instruction multiple-data (MIMD) processing, which might be useful for genetic programming. Typically current applications have a single GP kernel on the GPU. Multi-program multiple-data MPMD processing allows much more efficient mixing of multiple kernels. GPers might take advantage of this by splitting fitness evaluation and program interpreters into multiple GPU kernels. For example, a large physics simulation used as part of the fitness function might be composed of multiple kernels as well as separate GP interpreter kernels. The newer architectures both reduces the overhead of context switching between these and provides better overlap when they can be run in parallel.

Modern version of CUDA are less restrictive in their implementation of C and support recursion and function pointers. The lack of recursion previously forced GP interpreters to be based on Reverse Polish Notation (RPN) (Section 3.2). GP interpreters, such as gpquick are able to squeeze high performance from single core personal computers by linearising each Lisp evaluation tree into an equivalent list of C function pointers [71]. CUDA 3.2 should now allow this on GPUs as well.

Depending on the application (particularly number of fitness tests) the newer MIMD architecture could lead to much reduced divergence overhead in GP interpreters (Section 3.2).

Another potential benefit is the use of a single memory addressing scheme for all the memory that the GPU can access. This includes the host PC's memory. At present the PCI bus still imposes a large overhead on transfers to and from the GPU but the newer architecture potentially allows both easier programming and also access to potentially much more memory. In future, tighter hardware integration between host memory and GPU may greatly ease the host-GPU transfer bottleneck.

Currently, Reverse Polish interpreters often explicitly build their stacks in shared memory, even though each stack is independent. A single addressing scheme should allow the stack to be anywhere, including in ultra-fast registers. Also newer architectures will include more memory, allowing deeper stacks (and so less restrictions on evolved GP individuals) and more and bigger kernels.

The newer architectures have more sophisticated caches, which allow much more rapid data transfer between previously isolated parallel computations. Currently GP exploits the GPU well because it does not need such synchronisation or data passing. However future GPs may have more functionality on the GPU (other than fitness evaluation) and so may also exploit this aspect of future GPU chips.

It may be possible to write a traditional prefix interpreter for the new GPUs using a Lisp like depth first recursive evaluation. However, it may be that such GPUs will only lightly conceal their true SIMD roots and Reverse Polish interpreters will continue to give the best GP performance. Which is best can only be evaluated by implementation and benchmarking.

4.3 GPU Tools

Harding [45] gives a nice summary of the general purpose GPU (GPGPU) tools available in 2007. However many have fallen out of use. The software side of GPU computing has proved less stable than the underlying GPU architectures. In contrast, despite Moore's law type improvements in hardware performance, GPU architectures have been relatively stable over the last half decade. This raises the question of how can we pick long lasting GPGPU software tools and avoid getting locked into dead ends.

Even now GPGPU tools are limited with only basic emulation, debugging and performance monitoring available. Undoubtedly NVIDIA and others will provide improve tools, such as nsight, however newer architectures which include multi-instruction multiple-data (MIMD) aspects build on previous SIMD architectures. The improved tools and increasingly baroque instruction sets have not removed the steep learning curve associated with moving from sequential to parallel coding. How do you debug 10 000 simultaneous threads? This is made even harder by the GPU's strange SIMD model of parallelism. Despite the overwhelming price advantage of GPUs, it is this per person cost, that continues to limit uptake of general purpose use of GPUs.

In [72] we show how evolutionary computing might help by using GP to automatically create an NVIDIA CUDA kernel for the compute intensive part of file compression and use it within the Unix gzip utility. [72] is a very early demonstration and much work still needs to be done. There are also more traditional "auto-paral lelisation" approaches, such as Baskaran *et al.* [73], which can be applied in special cases.

4.4 Alternatives to GPU

While the Cell processor [74] and games consoles [75] are viable hardware platforms they also suffer the steep human programmer learning curve of GPUs. (Which we described in the previous section.) Their architectures also seem less scalable and we have not seen the enormous Moore's law performance gains that have been delivered by GPUs in recent years.

There has been talk of integrating the GPU into the CPU and certainly n-core Pentium or Pentium like CPUs are with us and with "n" set to grow. An alternative view of such integration might be that the chip holds many (perhaps hundreds) of fully functioning processors. Most of these would operate in parallel like processors in today's GPUs and also be fully capable CPUs in their own right. However a small part of the chip would be reserved for a few processors which would actually run the operating system, including driving the file, network and user interfaces. That is, a small part of the GPU chip acts as its operating system (OS) server. Of course there are substantial technical problems to overcome, not least the power barrier [2] associated with getting power to and removing heat from the literally billions of high speed transistors which would have to be packed into a small area. Nevertheless, if such approaches came with software development tools that reduce the GPU learning curve, we would expect their model of parallelism to succeed at the expense of GPGPU approaches.

Apart from some early work using the Edinburgh super computers [76], super computing has had little impact on GP. Much more successful have been loosely connected distributed systems, such as Beowulf clusters and Ethernet based local area networks connected by PVM, MPI or even NFS. There have been some planet wide distributed system [77,78] but this has yet to achieve SETI@Home notoriety. Nonetheless work continues. Recent successes include work by Cole [79], Desell [80] and their co-workers. They showed evolutionary algorithms, such as Differential Evolution, can be run across the Internet using Boinc. The Boinc framework successfully allows individuals to donate their computer's spare time for scientific research.

Cloud computing, for example Amazon's EC2, is now a commercial reality and is likely to be a natural home for short term but compute hungry applications. Perhaps off-site computers rented by the hour will displace loosely coupled computer clusters, which are today's favoured mode of parallel genetic programming.

4.5 GPGPU and GP Information Sources

Although NVIDIA is a commercial company they make CUDA and a huge amount of documentation freely available. They also host a (confusingly) large number of on line developer forums[2].

OpenCL is an alternative to CUDA and is also freely available. Whilst supposedly able to support computing across a wide range of parallel hardware, currently active use appears to be concentrated upon NVIDIA GPUs. However Apple (MAC OS), IBM, AMD ATI as well as NVIDIA have publicly committed to it.

Other useful Internet based sources of information include gpgpu.org[3] and Simon Harding's gpgpgpu.com[4].

The Field Guide to Genetic Programming [41] is freely available and covers almost all aspects of GP. The free PDF contains hyperlinks to Internet resources. The Field Guide has a section on GPUs.

The genetic programming bibliography[5] contains most papers on GP. In most cases it also has links to the PDF or post-script of the paper. (Papers relevant to graphics cards are marked "GPU".)

5 Conclusions

We have sketched the threat offered to the software industry by the many times announced failure of Moore's law, and how the also many times announced age of parallel computing may (this time) actually be upon us. We have concentrated upon a particularly cheap form of computing in which consumer electronics is subverted for use in science and engineering. This misuse of computer games and particularly computer screen VDU drivers is now being actively encouraged by manufactures such as NVIDIA.

The real time interactive response demanded by users of fantasy games has yielded tera flop performance in devices smaller than a laptop and at lower cost. This mass market has created a leap-frogging cycle in which the two principle manufactures have created ever more powerful or cheaper products than each other and in the process they have driven out all other competitors. (Similarly Intel dominates the market for low end GPUs.) With hundreds of millions of devices being sold, the remaining companies are able to spread their development costs thinly and so keep the cycle moving forwards.

With what used to be super computer performance available for less than the cost a conference registration, computer engineers and computer scientists have founded a new field, GPGPU (General-Purpose use of GPUs).

Section 2 gave a whistle stop tour of modern artificial intelligence using GPGPU. The next section concentrated upon one such technique, genetic programming.

[2] `http://forums.nvidia.com/`

[3] General-Purpose Computation on Graphics Hardware `http://gpgpu.org/`

[4] Genetic Programming on General Purpose Graphics Processing Units `http://gpgpgpu.com/`

[5] `http://www.cs.bham.ac.uk/~wbl/biblio`

We included a couple of examples where the GPU's supposedly single instruction multiple data (SIMD) model of parallelism has been successfully subverted to allow sixteen thousand or (in the breast cancer example) five million different programs, to be run simultaneously.

Section 4 looked at the downside and whether the future is really based on fantasy hardware. We also considered potential alternatives to GPU. Finally we included a few hints on getting started with programming GPUs and on genetic programming.

Acknowledgements

Work started in Memorial University, Newfoundland (with Wolfgang Banzhaf) and continued at Essex University and King's College, London prior to UCL. Some of it using pre-release hardware donated by NVIDIA.

I would like to thank Gernot Ziegler of NVIDIA and Lidia Yamamoto.

Funded by EPSRC grant EP/G060525/2

References

1. Moore, G. E. (1965) Cramming more components onto integrated circuits. *Electronics*, **38**, 114–117.
2. Izydorczyk, J. and Izydorczyk, M. (2010) Microprocessor scaling: What limits will hold? *IEEE Computer*, **43**, 20–26.
3. Arabnia, H. R. and Oliver, M. A. (1987) A transputer network for the arbitrary rotation of digitised images. *The Computer Journal*, **30**, 425–432.
4. Del Rizzo, B. (2008). Dice puts faith in nvidia PhysX technology for Mirror's Edge. NVIDIA Corporation press release.
5. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008) GPU computing. *Proceedings of the IEEE*, **96**, 879–899. Invited paper.
6. Oh, Kyoung-Su and Jung, Keechul (2004) GPU implementation of neural networks. *Pattern Recognition*, **37**, 1311–1314.
7. Luo, Zhongwen, Liu, Hongzhi, and Wu, Xincai (2005) Artificial neural network computation on graphic process unit. *International Joint Conference on Neural Networks, IJCNN '05*, Montreal, 31 July-4 Aug., pp. 622–626.
8. Ribeiro, B., Lopes, N., and Silva, C. (2010) High-performance bankruptcy prediction model using graphics processing units. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 2210–2216.
9. Prabhu, R. D. (2008) SOMGPU: an unsupervised pattern classifier on graphical processing unit. In Wang, J. (ed.), *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June, pp. 1011–1018.
10. Bernhard, F. and Keriven, R. (2006) Spiking neurons on GPUs. In Alexandrov, V. N., van Albada, G. D., Sloot, P. M. A., and Dongarra, J. (eds.), *Proceedings of the 6th International Conference on Computational Science, ICCS 2006, Part IV*, Reading, UK, May 28-31, Lecture Notes in Computer Science, **3994**, pp. 236–243. Springer.
11. Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009) A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, **22**, 791–800.
12. Fidjeland, A. K., Roesch, E. B., Shanahan, M. P., and Luk, Wayne (2009) NeMo: A platform for neural modelling of spiking neurons using GPUs. *20th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2009*, Rennes, France, 7-9 July, pp. 137–144.
13. Han, Bing and Taha, Tarek M. (2010) Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors. *Applied Optics*, **49**, B83–B91.
14. Bhuiyan, M. A., Pallipuram, V. K., and Smith, M. C. (2010) Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In Karypis, G. (ed.), *Ninth IEEE International Workshop on High Performance Computational Biology*, Atlanta, USA, 19 April.
15. Yudanov, D., Shaaban, M., Melton, R., and Reznik, L. (2010) GPU-based implementation of real-time system for spiking neural networks. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 2143–2150.
16. Fidjeland, A. K. and Shanahan, M. P. (2010) Accelerated simulation of spiking neural networks using GPUs. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 536–543.
17. Han, Bing and Taha, T. M. (2010) Neuromorphic models on a GPGPU cluster. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3050–3057.
18. Nowotny, T. (2010) Parallel implementation of a spiking neuronal network model of unsupervised olfactory learning on NVidia CUDA. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3238–3245.
19. Taha, T. M., Yalamanchili, P., Bhuiyan, M., Jalasutram, R., Chen, Chong, and Linderman, R. (2010) Neuromorphic algorithms on clusters of PlayStation 3s. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3040–3049.
20. Harvey, N., Luke, R., Keller, J. M., and Anderson, D. (2008) Speedup of fuzzy logic through stream processing on graphics processing units. In Wang, J. (ed.), *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June, pp. 3809–3815.
21. Harris, Chris and Haines, K. (2005) Iterative solutions using programmable graphics processing units. *The 14th IEEE International Conference on Fuzzy Systems, FUZZ '05*, Reno, Nevada, USA, 22-25 May, pp. 12–18. IEEE.
22. Anderson, D. T., Luke III, R. H., and Keller, J. M. (2008) Speedup of fuzzy clustering through stream processing on graphics processing units. *IEEE Transactions on Fuzzy Systems*, **16**, 1101–1106.
23. Tran, J., Jordan, D., and Luebke, D. (2004) New challenges for cellular automata simulation on the GPU. *SIGGRAPH*, Los Angeles. ACM. Poster.

24. Gobron, S., Devillard, F., and Heit, B. (2007) Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, **18**, 331–342.
25. Mussi, L., Cagnoni, S., and Daolio, F. (2009) GPU-based road sign detection using particle swarm optimization. *Ninth International Conference on Intelligent Systems Design and Applications, ISDA 2009*, Pisa, Italy, November 30-December 2, pp. 152–157. IEEE Computer Society.
26. Bai, Hongtao, OuYang, Dantong, Li, Ximing, He, Lili, and Yu, Haihong (2009) MAX-MIN ant system on GPU with CUDA. *Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009*, Kaohsiung, Taiwan, 7-9 Dec., pp. 801–804. IEEE.
27. Sinnott-Armstrong, N. A., Greene, C. S., and Moore, J. H. (2010) Fast genome-wide epistasis analysis using ant colony optimization for multifactor dimensionality reduction analysis on graphics processing units. In Pelikan, M. and Branke, J. (eds.), *Genetic and Evolutionary Computation Conference, GECCO 2010*, Portland, Oregon, USA, July 7-11, pp. 215–216. ACM.
28. Wong, Man-Leung, Wong, Tien-Tsin, and Fok, Ka-Ling (2005) Parallel evolutionary algorithms on graphics processing unit. In Corne, D., Michalewicz, Z., McKay, B., Eiben, G., Fogel, D., Fonseca, C., Greenwood, G., Raidl, G., Tan, K. C., and Zalzala, A. (eds.), *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, Edinburgh, 2-5 September, pp. 2286–2293.
29. Fok, Ka-Ling, Wong, Tien-Tsin, and Wong, Man-Leung (2007) Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, **22**, 69–78.
30. Arora, R., Tulshyan, R., and Deb, K. (2010) Parallelization of binary and real-coded genetic algorithms on GPU using CUDA. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3680–3687.
31. Kannan, S. and Ganji, R. (2010) Porting Autodock to CUDA. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3815–3822.
32. Tsutsui, S. and Fujimoto, N. (2010) An analytical study of GPU computation for solving QAPs by parallel evolutionary computation with independent run. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 889–889.
33. Wong, Man Leung and Cui, Geng (2010) Data mining using parallel multi-objective evolutionary algorithms on graphics hardware. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3815–3822.
34. Soca, N., Blengio, J. L., Pedemonte, M., and Ezzatti, P. (2010) PUGACE, a cellular evolutionary algorithm framework on GPUs. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 3891–3898.
35. de P. Veronese, L. and Krohling, R. A. (2010) Differential Evolution algorithm on the GPU with C-CUDA. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 1878–1884.
36. Munawar, A., Wahib, M., Munawar, A., and Wahib, M. (2009) Theoretical and empirical analysis of a GPU based parallel Bayesian Optimization Algorithm. *International Conference on Parallel and Distributed Computing, Applications and Technologies, 2009*, Higashi, Hiroshima, 8-11 Dec, pp. 457–462. IEEE.
37. Franco, M. A., Krasnogor, N., and Bacardit, J. (2010) Speeding up the evaluation of evolutionary learning systems using GPGPUs. In Pelikan, M. and Branke, J. (eds.), *Genetic and Evolutionary Computation Conference, GECCO 2010*, Portland, Oregon, USA, July 7-11, pp. 1039–1046. ACM.
38. Luong, The Van, Melab, N., and Talbi, E.-G. (2010) Parallel hybrid evolutionary algorithms on GPU. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 2734–2741.
39. Clayton, T. F., Patel, L. N., Leng, Gareth, Murray, A. F., and Lindsay, I. A. (2008) Rapid evaluation and evolution of neural models using graphics card hardware. In Keijzer, M., Antoniol, G., Congdon, C. B., Deb, K., Doerr, B., Hansen, N., Holmes, J. H., Hornby, G. S., Howard, D., Kennedy, J., Kumar, S., Lobo, F. G., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Pollack, J., Sastry, K., Stanley, K., Stoica, A., Talbi, E.-G., and Wegener, I. (eds.), *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, GA, USA, 12-16 July, pp. 299–306. ACM.
40. Chwatal, A. M., Raidl, G. R., and Zöch, M. (2010) Fitting multi-planet transit models to photometric time-data series by evolution strategies. In Pelikan, M. and Branke, J. (eds.), *Genetic and Evolutionary Computation Conference, GECCO 2010*, Portland, Oregon, USA, July 7-11, pp. 377–384. ACM.
41. Poli, R., Langdon, W. B., and McPhee, N. F. (2008) *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).
42. Lindblad, F., Nordin, P., and Wolff, K. (2002) Evolving 3D model interpretation of images using graphics hardware. In Fogel, D. B., El-Sharkawi, M. A., Yao, X., Greenwood, G., Iba, H., Marrow, P., and Shackleton, M. (eds.), *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, 12-17 May, pp. 225–230. IEEE Press.
43. Meyer-Spradow, J. and Loviscach, J. (2003) Evolutionary design of BRDFs. In Chover, M., Hagen, H., and Tost, D. (eds.), *Eurographics 2003 Short Paper Proceedings*, pp. 301–306.
44. Ebner, M., Reinhardt, M., and Albert, J. (2005) Evolution of vertex and pixel shaders. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J. I., and Tomassini, M. (eds.), *Proceedings of the 8th European Conference on Genetic Programming*, Lausanne, Switzerland, 30 March - 1 April, Lecture Notes in Computer Science, **3447**, pp. 261–270. Springer.
45. Harding, S. and Banzhaf, W. (2007) Fast genetic programming on GPUs. In Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., and Esparcia-Alcázar, A. I. (eds.), *Proceedings of the 10th European Conference on Genetic Programming*, Valencia, Spain, 11-13 April, Lecture Notes in Computer Science, **4445**, pp. 90–101. Springer.
46. Chitty, D. M. (2007) A data parallel approach to genetic programming using programmable graphics hardware. In Thierens, D., Beyer, H.-G., Bongard, J., Branke,

J., Clark, J. A., Cliff, D., Congdon, C. B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K. O., Stutzle, T., Watson, R. A., and Wegener, I. (eds.), *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, 7-11 July, pp. 1566–1573. ACM Press.

47. Flynn, M. J. (1972) Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, **C-21**, 948–960.
48. Langdon, W. B. (2009) A CUDA SIMT interpreter for genetic programming. Technical Report TR-09-05. Department of Computer Science, King's College London, Strand, WC2R 2LS, UK. 18 June 2009.
49. Langdon, W. B. and Banzhaf, W. (2008) A SIMD interpreter for genetic programming on GPU graphics cards. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E. (eds.), *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, Naples, 26-28 March, Lecture Notes in Computer Science, **4971**, pp. 73–85. Springer.
50. Juille, H. and Pollack, J. B. (1996) Massively parallel genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E. (eds.), *Advances in Genetic Programming 2*, chapter 17, pp. 339–358. MIT Press.
51. Harris, Christopher. (1997) An investigation into the Application of Genetic Programming techniques to Signal Analysis and Feature Detection. PhD thesis University College, London.
52. Harding, S. L. and Banzhaf, W. (2009) Distributed genetic programming on GPUs using CUDA. In Hidalgo, I., Fernandez, F., and Lanchares, J. (eds.), *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, September 13.
53. Langdon, W. B. (2010) Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In Fernandez de Vega, F. and Cantu-Paz, E. (eds.), *Parallel and Distributed Computational Intelligence*, chapter 5, January, Studies in Computational Intelligence, **279**, pp. 113–141. Springer.
54. Langdon, W. B. (2010) A many threaded CUDA interpreter for genetic programming. In Esparcia-Alcazar, A. I., Ekart, A., Silva, S., Dignum, S., and Uyar, A. S. (eds.), *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, Istanbul, 7-9 April, LNCS, **6021**, pp. 146–158. Springer.
55. Poli, R. and Langdon, W. B. (1999) Sub-machine-code genetic programming. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J. (eds.), *Advances in Genetic Programming 3*, chapter 13, pp. 301–323. MIT Press.
56. Langdon, W. B. and Harrison, A. P. (2008) GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, **12**, 1169–1183. Special Issue on Distributed Bioinspired Algorithms.
57. Robilliard, D., Marion-Poty, V., and Fonlupt, C. (2008) Population parallel GP on the G80 GPU. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E. (eds.), *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, Naples, 26-28 March, Lecture Notes in Computer Science, **4971**, pp. 98–109. Springer.
58. Robilliard, D., Marion, V., and Fonlupt, C. (2009) High performance genetic programming on GPU. In Folino, G., Krasnogor, N., Mastroianni, C., and Zambonelli, F. (eds.), *Proceedings of the 2009 workshop on Bioinspired algorithms for distributed systems*, Barcelona, Spain, June 15-19, pp. 85–94. ACM. paper invited for the FGCS special issue.
59. Robilliard, D., Marion-Poty, V., and Fonlupt, C. (2009) Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, **10**, 447–471. Special issue on parallel and distributed evolutionary algorithms, part I.
60. Maitre, O., Baumes, L. A., Lachiche, N., Corma, A., and Collet, P. (2009) Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In Raidl, G., Rothlauf, F., Squillero, G., Drechsler, R., Stuetzle, T., Birattari, M., Congdon, C. B., Middendorf, M., Blum, C., Cotta, C., Bosman, P., Grahl, J., Knowles, J., Corne, D., Beyer, H.-G., Stanley, K., Miller, J. F., van Hemert, J., Lenaerts, T., Ebner, M., Bacardit, J., O'Neill, M., Di Penta, M., Doerr, B., Jansen, T., Poli, R., and Alba, E. (eds.), *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Montreal, Québec, Canada, 8-12 July, pp. 1403–1410. ACM.
61. Maitre, O., Querry, S., Lachiche, N., and Collet, P. (2010) EASEA parallelization of tree-based genetic programming. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 1997–2004. IEEE.
62. Liu, Weiguo, Schmidt, B., Voss, G., Schroder, A., and Muller-Wittig, W. (2006) Bio-sequence database scanning on a GPU. *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, Rhodes, Greece, 25-29 April. IEEE Press.
63. Manavski, S. and Valle, G. (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, **9**, S10.
64. Wirawan, Adrianto, Kwoh, Chee, Hieu, Nim, and Schmidt, Bertil (2008) CBESW: sequence alignment on the playstation 3. *BMC Bioinformatics*, **9**, 377.
65. Sinnott-Armstrong, N. A., Greene, C. S., Cancare, F., and Moore, J. H. (2009) Accelerating epistasis analysis in human genetics with consumer graphics hardware. *BMC Research Notes*, **2**, 147.
66. Langdon, W. B. (2008) Evolving GeneChip correlation predictors on parallel graphics hardware. In Wang, J. (ed.), *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June, pp. 4152–4157.
67. Langdon, W. B. and Harrison, A. P. (2009) Evolving DNA motifs to predict GeneChip probe performance. *Algorithms in Molecular Biology*, **4**.
68. Huang, Chao-Hui, Racoceanu, D., Roux, L., and Putt, T. C. (2010) Bio-inspired computer visual system using GPU and visual pattern assessment language (ViPAL): application on breast cancer prognosis. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 1103–1110.
69. Rieffel, J., Saunders, F., Nadimpalli, S., Zhou, Harvey, Hassoun, S., Rife, J., and Trimmer, B. (2009) Evolving

soft robotic locomotion in PhysX. *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, Montreal, Québec, Canada, 8-12 July, pp. 2499–2504. ACM.

70. Lewis, T. E. and Magoulas, G. D. (2009) Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In Raidl, G., Rothlauf, F., Squillero, G., Drechsler, R., Stuetzle, T., Birattari, M., Congdon, C. B., Middendorf, M., Blum, C., Cotta, C., Bosman, P., Grahl, J., Knowles, J., Corne, D., Beyer, H.-G., Stanley, K., Miller, J. F., van Hemert, J., Lenaerts, T., Ebner, M., Bacardit, J., O'Neill, M., Di Penta, M., Doerr, B., Jansen, T., Poli, R., and Alba, E. (eds.), *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Montreal, 8-12 July, pp. 1379–1386. ACM.
71. Keith, M. J. and Martin, M. C. (1994) Genetic programming in C++: Implementation issues. In Kinnear, Jr., K. E. (ed.), *Advances in Genetic Programming*, chapter 13. pp. 285–310. MIT Press.
72. Langdon, W. B. and Harman, M. (2010) Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 2376–2383.
73. Baskaran, M. M., Ramanujam, J., and Sadayappan, P. (2010) Automatic C-to-CUDA code generation for affine programs. In Gupta, R. (ed.), *9th International Conference Compiler Construction, CC 2010*, Paphos, Cyprus, March 20-28, Lecture Notes in Computer Science, **6011**, pp. 244–263. Springer. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010.
74. Comte, P. (2009) Design & implementation of parallel linear GP for the IBM cell processor. In Raidl, G., Rothlauf, F., Squillero, G., Drechsler, R., Stuetzle, T., Birattari, M., Congdon, C. B., Middendorf, M., Blum, C., Cotta, C., Bosman, P., Grahl, J., Knowles, J., Corne, D., Beyer, H.-G., Stanley, K., Miller, J. F., van Hemert, J., Lenaerts, T., Ebner, M., Bacardit, J., O'Neill, M., Di Penta, M., Doerr, B., Jansen, T., Poli, R., and Alba, E. (eds.), *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Montreal, 8-12 July. ACM.
75. Wilson, G. and Banzhaf, W. (2010) Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms. *Genetic Programming and Evolvable Machines*, **11**, 147–184.
76. Openshaw, S. and Turton, I. (1994) Building new spatial interaction models using genetic programming. In Fogarty, T. C. (ed.), *Evolutionary Computing, AISB workshop*, Leeds, UK, 11-13 April (unpublished).
77. Chong, Fuey Sian and Langdon, W. B. (1999) Java based distributed genetic programming on the internet. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13-17 July 1229. Morgan Kaufmann. Full text in technical report CSRP-99-7.
78. Klein, J. and Spector, L. (2007) Unwitting distributed genetic programming via asynchronous JavaScript and XML. In Thierens, D., Beyer, H.-G., Bongard, J., Branke, J., Clark, J. A., Cliff, D., Congdon, C. B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K. O., Stutzle, T., Watson, R. A., and Wegener, I. (eds.), *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, 7-11 July, pp. 1628–1635. ACM Press.
79. Cole, N., Desell, T., Lombrana Gonzalez, D., Fernandez de Vega, F., Magdon-Ismail, M., Newberg, H., Szymanski, B., and Varela, C. (2010) Evolutionary algorithms on volunteer computing platforms: The milkyway@home project. In Fernandez de Vega, F. and Cantu-Paz, E. (eds.), *Parallel and Distributed Computational Intelligence*, chapter 4, pp. 63–90. Springer.
80. Desell, T., Anderson, D. P., Magdon-Ismail, M., Newberg, H., Szymanski, B., and Varela, C. A. (2010) An analysis of massively distributed evolutionary algorithms. In Sobrevilla, P. (ed.), *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July, pp. 873–880.