Preprint Parallel and Distributed Computational Intelligence, Francisco Fernandez de Vega and Erick Cantu-Paz *Eds.* Chapter 5, pages 113-141 Springer, 2010 DOI

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

W. B. Langdon

Abstract A suitable single instruction multiple data GP interpreter can achieve high (Giga GPop/second) performance on a SIMD GPU graphics card by simultaneously running multiple diverse members of the genetic programming population. SPMD dataflow parallelisation is achieved because the single interpreter treats the different GP programs as data. On a single 128 node parallel nVidia GeForce 8800 GTX GPU, the interpreter can out run a compiled approach, where data parallelisation comes only by running a single program at a time across multiple inputs.
The RapidMind GPGPU Linux C++ system has been demonstrated by predicting ten year+ outcome of breast cancer from a dataset containing a million inputs. NCBI GEO GSE3494 contains hundreds of Affymetrix HG-U133A and HG-U133B GeneChip biopsies. Multiple GP runs each with a population of five million programs winnow useful variables from the chaff at more than 500 million GPops per second. Sources available via FTP.

1 Introduction

Due to their speed, price and availability, there is increasing interest in using mass market graphics hardware (GPUs) for scientific applications. Since our initial experiments GPU development has continued apace. For example, AMD has launched its 800 $\times$ 750MHz processor ATI Radeon HD 4870. Whilst almost simultaneously nVidia launched its 240 $\times$ 1296MHz GTX 280 GPU. Both claim to deliver about one Tetraflop at a cost of a few hundred dollars.

The next section will describe scientific and engineering computing on GPUs. (Known as GPGPU). So far there are a few reported successful applications of GPUs to Bioinformatics. These will be described in Section 3. In Section 4 we will describe one where genetic programming [24] is used to datamine a small number

W. B. Langdon
King's College, London

of indicative mRNA gene transcript signals from breast cancer tissue samples taken during surgery. Section 5 describes how we run GP [18, 2, 27, 45] in parallel on a GPU. Whilst the rest of Section 5 (i.e. 5.1 and 5.2) and Section 6 describe the medical problem and the way a powerful GPU [29, 26] simultaneously picks three of the million mRNA measurements available and finds 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 [37].

2 Using Games Hardware GPUs for Science

Owens *et al.* have recently surveyed scientific and engineering applications running on mass market graphics cards (known as general purpose computing on GPUs, i.e. GPGPU) [42, 43]. Whilst there is increasing interest, so far both Bioinformatics and computational intelligence are under represented. As with other GPGPU applications, the drivers are: locality, convenience, cost and concentration of computer power. Indeed the principle manufactures (nVidia and ATI) claim faster than Moore's Law increase in performance (e.g. [11, p4]). They suggest that GPU floating point performance will continue to double every twelve months, rather than the 18-24 months observed for electronic circuits in general [38] and personal computer CPUs in particular. Indeed the apparent failure of PC CPUs to keep up with Moore's law in the last few years [42, p890]. makes GPU computing even more attractive. Even today's top of the range GPU greatly exceed the floating point performance of their host CPU. This speed comes at a price.

GPUs provide a restricted type of parallel processing, often referred to a 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 1. 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). Indeed vectors of four floats can be thought of as the native data type of current GPUs. RapidMind's software translates other data types to floats when it transfers it from the CPU's memory to the GPU and back again when results are read back. Note integer precision may only be 24 bits, however GPUs will soon support 64 bits.

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 exactly 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 even though GPUs usually come with their

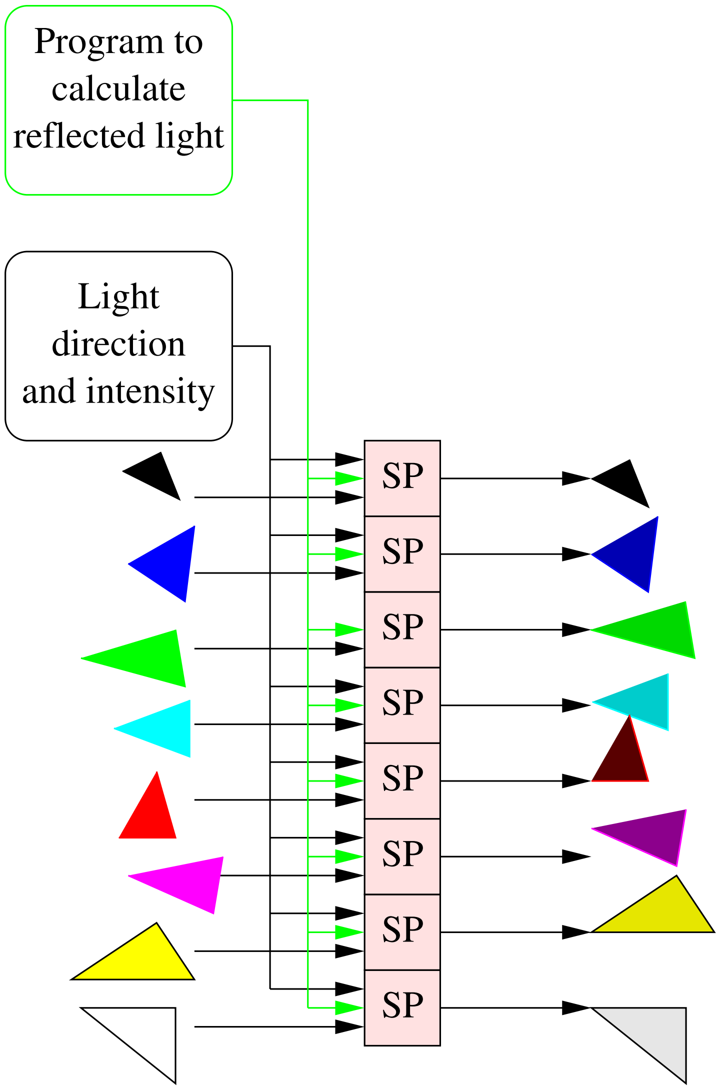

Program to calculate reflected light
Light direction and intensity
SP
SP
SP
SP
SP
SP
SP
SP

Fig. 1 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 each programs 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 SP use the same value. Each SP calculates the apparent colour of its triangle. Each calculation is complex. The stream processors use the colour of the light, angles between the light and its triangle, direction of its triangle, colour of its triangle, etc.

own memory and memory caches. (The nVidia 8800 comes with 768Mbytes). Additionally data must be transfered to and from the GPU. Even when connected to the CPU's RAM via PCI, this represents an even narrower bottle neck. Faster hardware (e.g. PCI Express x16) is available for some PC/GPU combinations. However this does not remove the bottle neck. CPU–GPU communication can also be delayed by the operating system check pointing and rescheduling the task.

The manufactures' 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 the number of GP operations per second (GPops) should allow 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 $\approx 10^8$
- Maximum dataset data rate $\approx 10^8$ 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.
- Core has simple data flow. Large fan-in (but less than sixteen) and simple data stream output (no fan-out).

Naturally as GPUs become more powerful these figures will change.

In some cases, it might be possible to successfully apply GPUs to bigger problems. For example, a large dataset might be broken into smaller chunks, and then each chunk is loaded one at a time onto the GPU. When the GPU has processed it, the next chunk is loaded and so on, until the whole dataset has been processed. The time spent loading data into (and results out of) each GPU may be important. If the application needs a data rate of 100Mbyte/second we must consider how the data is to be loaded into a personal computer at this rate in the first place. Alternatively it may be possible to load data from a scientific instrument directly connected to the GPU.

nVidia say their GeForce 8800 (Fig. 2) has a theoretical upper limit of 520 GFLOPS [39, p36], however we obtained about 30 GFLOPS in practice. Depending on data usage (cf. Section 7), it appears that 100 GFLOPS might be reached in practise. While tools to support general purpose computation on GPUs have been greatly improved, getting the best from a GPU is still an art. Indeed some publications claim a speed up of only 20% (or even less than one) rather than 7+, which we report.

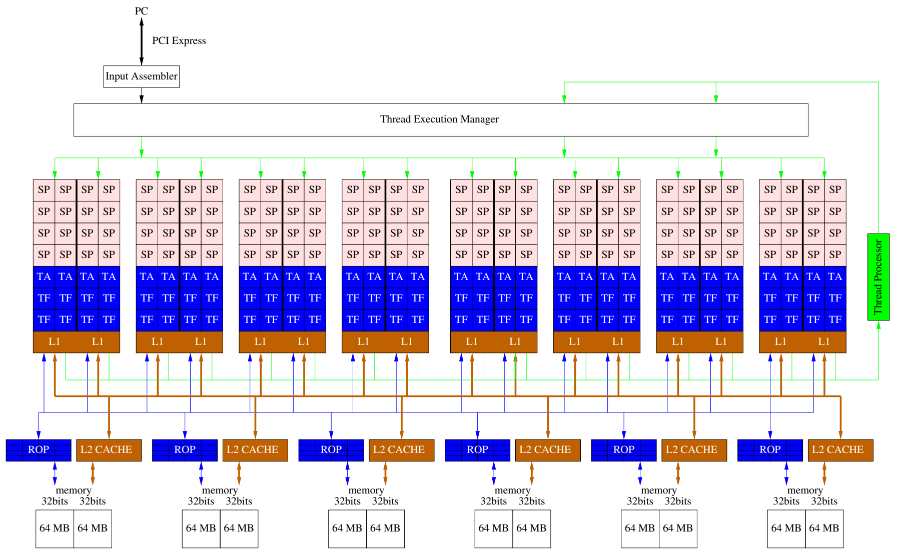


Fig. 2 nVidia 8800 Block diagram. The 128 1360 MHz Stream Processors are arranged in sixteen blocks of eight. Blocks share 16 KB memory (not shown), an 8/1 KB L1 cache, four Texture Address units and eight Texture Filters. The 6×64 bit bus (brown) links off chip RAM at 900 (1800) MHz [39, 40]. There are 6 Raster Operation Partitions (ROP).

3 GPUs in Bioinformatics and Computational Intelligence

As might be expected GPUs have been suggested for medical image processing applications for a few 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) [4]. However a more conventional MPI cluster was subsequently used [50]. Sequence comparison is the life blood of Bioinformatics. Liu *et al.* ran the key Smith-Waterman algorithm on a high end GPU [31]. 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 [54] and the GeForce 8800 (CUDA) [35]. Schatz *et al*. also used CUDA to port another sequence searching tool (MUMmer) to another G80 GPU and obtained speed ups of 3–10 when matching short DNA strands against much longer sequences [49]. By breaking queries into GPU sized fragments, they were able to run short sequences (e.g. 50 bases) against a complete human chromosome. 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 [13]. 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 [32]. 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 [7].

The better GPU applications may claim speed ups of a factor of ten or so, however the distributed protein folding system folding@home obtains sixty times as much free computation per donated GPU as it does per donated CPU [42, 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 [42, p995].

Computational intelligence applications of GPUs have included artificial neural networks (e.g. multi layer perceptrons and self organising networks [34]), genetic algorithms [12] and a few genetic programming experiments [30, 33, 36, 8, 47, 16, 14, 17, 15, 5, 21, 26, 48, 52, 1].

Most GPGPU applications have only required a single graphics card, however Fan *et al.* have shown large GPU clusters are also feasible [9]. In 2008 the first computational intelligence on GPU special session (CIGPU-2008) was held in Hong Kong [53]. It is anticipated that this will become an annual event. As Owens [43] makes clear games hardware is now breaking out of the bedroom into scientific and engineering computing.

4 Gene Expression in Breast Cancer

Miller *et al.* describes the collection and analysis of cancerous tissue from most of the women with breast tumours from whom samples were taken in the three years 1987–1989 in Uppsala in Sweden [37]. 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.

Affymetrix GeneChips estimate the concentration of strands of messenger RNA by binding them to complementary DNA itself tied to specially treated glass slides. GeneChips are truly amazing. When working well they can measure the activity, in terms of mRNA concentration, of almost all known human genes in one operation. 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.) Obviously such tiny measuring devices are very subject to noise and so between eleven and twenty readings are taken per gene. In fact each reading is duplicated with a control which differs only by its central DNA base. These controls are known as mismatch MM probes.

There has been considerable debate about the best way of converting each of the eleven or more pairs of readings into a single value to represent the activity of a gene.

Also in more recent designs (e.g. exon arrays), Affymetrix have replaced the MM probes of each pair with general area control probes. Miller *et al.* used Affymetrix' MAS5 program. MAS5 uses outlier detection etc. to take a robust average of the twenty two or more data. The academic community has also developed its own tools. These have tended to replace the manufacturer's own analysis software. Such tools also use outlier detection and robust averaging. Some, such as GCRMA [55], ignore the control member of each pair.

Miller *et al.* separately normalised the natural log of the HG-U133A and HG-U133B values and then used MAS5 to calculate 44 928 gene expression values for each pair patient [37]. (Normalisation is needed to avoid the need to carefully control the amount of mRNA used and since Biologists are usually more interested in the relative strengths of gene activity, rather than absolute values.) Between 125 and 5 000 of the most variable were selected for further analysis. They used diagonal linear discriminant analysis to fit the *whole* data set. They say DLDA gave better results than k nearest neighbours and support vector machines. The DLDA p53 classifier used 32 genes.

Recently we have surveyed defects in more than ten thousand Affymetrix Gene Chips using a new technique [25, 28]. While [37] claims GeneChips with "visible artefacts" were re-run, we found spatial flaws in all their data. GeneChips should have an almost random speckled pattern due to the pseudo random placement of gene probes. The large light gray areas in Figure 3 indicate spatial flaws. Spatial flaws occur most often towards the edges of GeneChips. Figures 4 and 5 shows the location and density of known errors in some data used for training GP and subsequent testing.

5 GeneChip Data Mining using Genetic Programming on a GPU

Section 3 has listed the previous experiments evolving programs with a GPU. These have either represented the programs as trees or as networks (Cartesian GP) [16] and used the GPU for fitness evaluation. Harding compiled his networks into GPU programs before transferring the compiled code onto the GPU. We retain the traditional tree based GP and use an interpreter running on the GPU. Next we shall briefly recap how to interpret multiple programs simultaneously on a SIMD computer [20] and then detail tricks needed to address 512MBytes on a GPU.

Essentially the interpreter trick is to recognise that in the SIMD model 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 128 different GP trees. The guts of a standard interpreter is traditionally a n-way switch where each case statement executes a different GP opcode. A SIMD machine cannot (in principle) execute multiple different operations at the same time. However they do provide a `cond` statement.

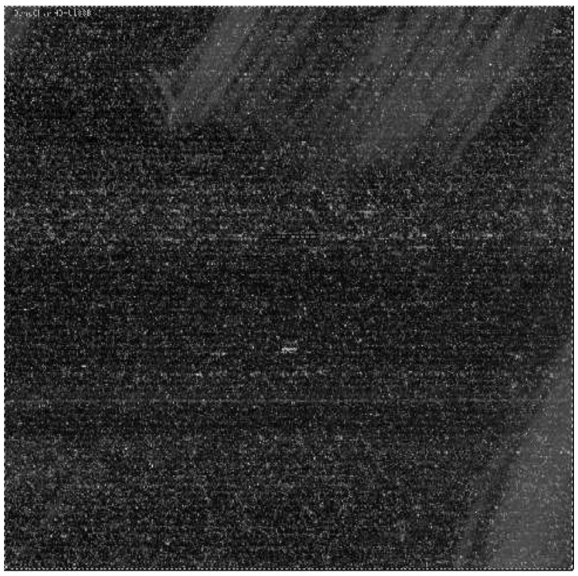


Fig. 3 First HG-U133B Breast Cancer GeneChip. Data have been quantile normalised. (This is like converting to a standard score and effectively replaces data by its logged value). Large spatial flaws can be seen at the top and lower right hand corners.

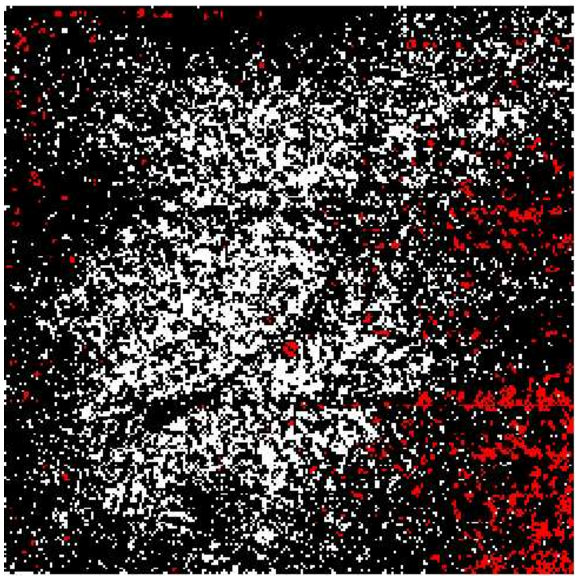


Fig. 4 Density of spatial flaws in 98 HG-U133A Breast Cancer GeneChips. Red more than twenty of 98 GeneChips are flawed (Black at least one).

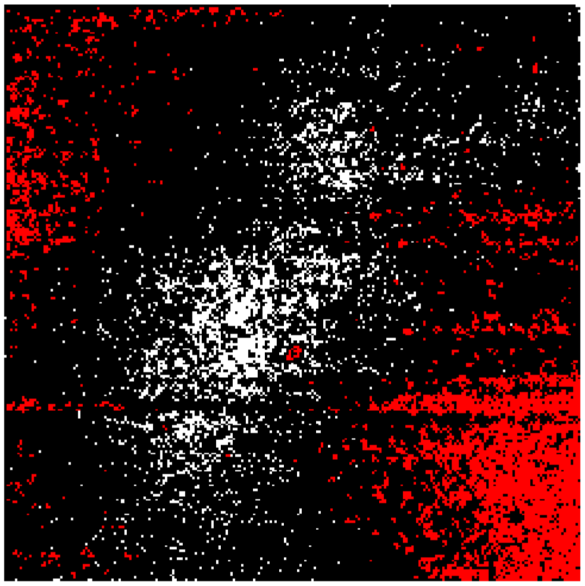


Fig. 5 Density of spatial flaws in 98 HG-U133B Breast Cancer GeneChips showing HG-U133B have more spatial errors than HG-U133A, c.f. Fig. 4.

A `cond` statement has three arguments. The first is the control. It decides which of the other two arguments is actually used. `cond` behaves as if the calculations needed by its second and third arguments are both performed, but only one is used. Which one depends upon the `cond`'s first argument.

We use conditional statements like `x=cond(opcode=='+', a+b, x)` to perform an operation only if required. If the current instruction is + `cond` sets `x` to `a+b`. Otherwise it does nothing (by setting `x` to itself). (See Figures 6, 7 and 8.) Note the SIMD interpreter executes every `cond` for every instruction in the program. (In a normal interpreter a switch statement would direct the interpreter to execute just the code needed for the current instruction.) If there are five opcodes, this means for every leaf and every function in the program, the opcode at that point in the tree will be obeyed once but so too will four `cond` no-ops. As we showed in [26] the no-ops and indeed the functions cost almost nothing. It is reading the inputs from the training data which is expensive.

GPUs, at present, cannot imagine anyone having a screen bigger than 2048×2048 and therefore do not support arrays with more than 2^{22} elements. Each training example has data from both HG-U133A and HG-U133B, i.e. $2 \times 712^2 = 1\,013\,888$ floats. Therefore we pack four training examples per array. Since we split the available data into more or less equal training and holdout sets, the GPU fitness evaluation code need process at most only half the 251 patients' data at a time. The code allows 32 arrays (i.e. upto 128 patients). This occupies 512MB. All data transfers and data conversions are performed automatically by RapidMind's package. RapidMind keeps track of when data are used and modified. Since the training data are not

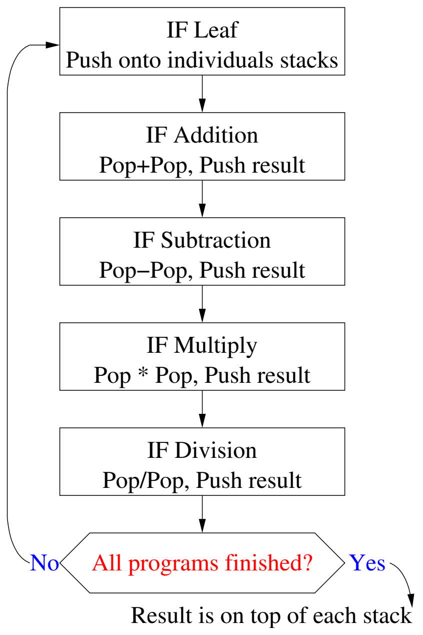


Fig. 6 The SIMD interpreter loops continuously through the whole genetic programming terminal and function sets for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks.

```
#define OPCODE(PC) ::PROG[PC+(prog0*LEN)]
PC=0;
FOR(PC,PC<(LEN-1),PC++) {
  //if leaf push data onto stack
  top = cond(OPCODE(PC)=='+', stack(1)+stack(0), top);
  top = cond(OPCODE(PC)=='-', stack(1)-stack(0), top);
  top = cond(OPCODE(PC)=='*', stack(1)*stack(0), top);
  top = cond(OPCODE(PC)=='/', stack(1)/stack(0), top);
  //remaining stack operation not shown
 } ENDFOR
```

Fig. 7 GPU Reverse Polish Notation SIMD interpreter. `prog0` indicates which RPN program is being evaluated on which stream processor. The central loop cycles through all operations on all stream processors. Each individual program uses `cond` statements to execute only those operations it needs.

modified, they are stored in the GPU at the start of the run. Each generation, only the data which has changed, i.e. the GP individuals and their fitness's, are transfered between the host computer and the GPU. The architecture is shown shown in Figure 9.

The interpreter has to be structured to work within another GPU restriction. Like most other GPUs, the nVidia 8800 allows each GPU program at most sixteen inputs. I.e. the interpreter cannot access all 32 training data arrays simultaneously. Since it

```
Value<8,float> stack;

#define PUSH(V) \
join(join(V,stack(0,1,2)),stack(3,4,5,6))

//conditionally POP stack (fake by using rotation)
#define OP3(XCODE,OP) \
stack = cond(XCODE==OPCODE, \
             join(OP,stack(2,3,4),stack(5,6,7,1)), \
             stack);
```

Fig. 8 Partial implementation for GPU stack operations. Since RapidMind does not support index operations on writing to arrays the whole stack is updated. On PUSH the eight element stack is shuffled to the left using nested `join()` and the value is placed in `stack(0)`. The upper most element is lost. GP genetic operations ensure tree depth does not exceed eight and so there can be no stack overflow. (However GP can evolve solutions which happily cause stack over run. Nature will find a way.) `OP3` uses `cond` so that the operation `OP` on the two elements on top of the stack only takes place if the current instruction `OPCODE` is the right one. Then the stack is shifted down one place and the result of the operation is put in `stack(0)`.

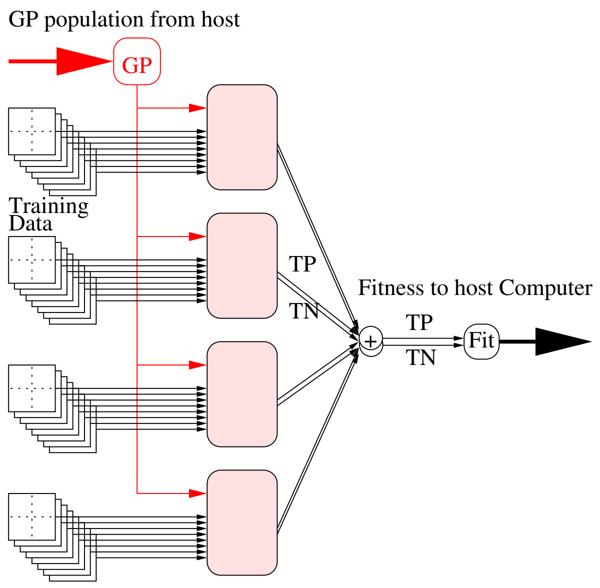


Fig. 9 GPU software architecture needed to overcome 2^{22} and no more than sixteen arrays GPU limits in order to access 512MB of training data and a population of five million GP programs. The population is split into twenty 256k parts by the host CPU. Twenty times per generation 256 thousand GP programs are passed to GPU (red) and interpreted by it. On average, the GPU takes slightly less than a second to interpret them and return their fitness values. There are four parameterised instances of the SIMD interpreter (pink). Each deals with upto 32 training cases. Each uses 1+8+2 arrays (plus others for control and debug, not shown, total 12 or more). Each instance is limited to sixteen arrays. We pack four sets of patient data ($4 \times 1\,013\,888$) per array. Four groups of eight arrays allows 512M of training data. After running each group of $\frac{1}{4}$ million programs, $\frac{1}{4}$ million fitness values are returned to the host PC.

must access other data arrays (programs, fitness, debugging, etc.) as well as the training arrays, the interpreter was split into four equal parts, each of which deals with eight arrays (i.e. upto 32 patients). A parameterised C++ macro is used to define the interpreter code for one array. To access the 32 arrays of training data, the macro is used eight times in each of the four programs.

The four sets of outputs are summed and combined into a single fitness value per GP individual. For convenience the summation and fitness calculation are done by three auxiliary GPU programs. Only the final result is transfered to the host computer. RapidMind's optimising compiler deals with all seven GPU programs as one unit and therefore can, in principle, optimise across their boundaries. C++ code to invoke the GPU via RapidMind is shown in Figure 10.

As described in [29] the interpreter represents the GP trees as linearised reverse polish expressions. By using a stack these can be evaluated in a single pass. For simplicity, the expressions are all the same length. Smaller trees are simply padded with no-ops. Because of the enormous number of inputs, it is no longer possible to code each opcode into a byte [29] instead at least 20 bits are needed. In fact we use a full word per opcode. This means a population of five million fifteen node programs can be stored in 320Mbyte on the PC. Here we again run into the 2^{22} GPU addressing limit. Since each program occupies sixteen words (fifteen, plus one for a stop code), the population is broken into twenty 256k units.

It takes slightly less than a second to evaluate all 262 144 programs. This fits tolerably well with our earlier finding [29] that, to get the best from the GPU, its work should be fed into the GPU in units of between 1 and 10 seconds.

5.1 GP for large scale data mining

We have previously described using genetic programming to data mine GeneChip data [24]. Our intention is 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 [56].)

In our earlier work we had only one GeneChip for each of the 60 patients (and that was an older design). Also the data set did not include the probe values but only 7129 gene expression values [24]. We now have the raw probe values (and compute power to use them). Therefore we will ask GP to evolve combinations of the probe values rather than use Affymetrix or other human designed combinations of them. This gives us more than a million inputs. The first step is to use GP as its own feature selector.

Essentially the idea is to use Price's theorem [46]. 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 several times. We ignore the performance of the best of run individual and instead look at the genes it contains. The intention was the first

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace std;
using namespace rapidmind;

const int NP = 2560*2048; //NP is Number of programs in Population
const int LEN =15+1;      //Max GP individual length, allow stop code

//Number gp individual Programs loaded onto GPU
const int GPU_NP = 4*1024*1024/LEN; //22bit limit

//virtual array prog0 is used to simulate indexOf
Array<1,Value1i> prog0 = grid(GPU_NP);

for(int n=0;n<(NP/GPU_NP);n++) {
  // Access the internal arrays where the data is stored
  unsigned int* in_PROG = PROG.write_data();
  memcpy(in_PROG,&Pop[n*GPU_NP*LEN],LEN*GPU_NP*opsize);

  Array<1,Value1i> TP0;
  Array<1,Value1i> TN0;
  Array<1,Value1i> TP1;
  Array<1,Value1i> TN1;
  Array<1,Value1i> TP2;
  Array<1,Value1i> TN2;
  Array<1,Value1i> TP3;
  Array<1,Value1i> TN3;
  Array<1,Value1i> TP;
  Array<1,Value1i> TN;

  Array<1,Value1f> F;

  bundle(TP0,TN0) = gpu->m_update0(prog0);
  bundle(TP1,TN1) = gpu->m_update1(prog0);
  bundle(TP2,TN2) = gpu->m_update2(prog0);
  bundle(TP3,TN3) = gpu->m_update3(prog0);
  TP              = gpu->sum(TP0,TP1,TP2,TP3);
  TN              = gpu->sum(TN0,TN1,TN2,TN3);
  F               = gpu->fitness(TP,TN);

  const float* fit = F.read_data();
  memcpy(&output[n*GPU_NP],fit,GPU_NP*sizeof(float));
}//endfor each GPU sized element of Pop
```

Fig. 10 Part of C++ code to run GP interpreter on the GPU twenty times (`NP/GPU_NP`) per generation. At the start of the loop the next fragment of `Pop` is copied into RapidMind variable `PROG`. `PROG`'s address is given by `write_data()`. RapidMind variables `TP0` to `TN` are used to calculate fitness, cf. Figure 9. They are not used by the host CPU and are never transfered from the GPU to the CPU. The four `m_update*(prog0)` programs each run the GP interpreter on 256k programs on 32 patients' data. They are identical, except they are parameterised to run on different quarters of 128 training cases. The RapidMind `bundle()` provides a way that is compatible with C++ syntax for a GPU program to return two or more values. All evaluation is run on the GPU until `read_data()` is called. `read_data()` not only transfers the fitness values, in `F`, but also resynchronises the GPU and CPU.

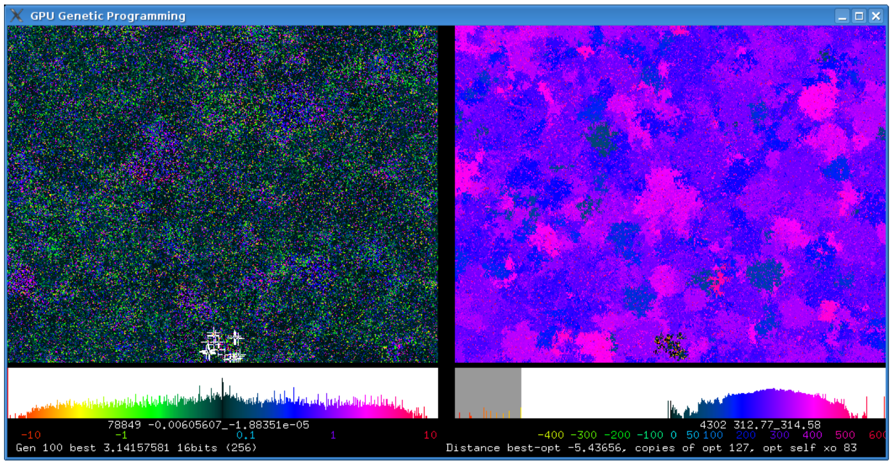
pass would start with a million inputs and we would select in the region of 10 000 for the second pass. Then we would select about 100 from it for the third pass. Finally a GP run would be started with a much enriched terminal set containing only inputs which had showed themselves to be highly fit in GP runs. However we found only two selection passes were needed, cf. Section 6.

The question of how big to make the GP population can be solved by considering the coupon collector problem [10, 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. If we overshoot by a few thousands, we are sure to get all the 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.

In [29] we used a fairly gentle selection pressure. Here we need our programs to compete, so the tournament size was increased to four. However we have to be cautious. 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 [19] to delay convergence of the population, cf. Figure 11. The GP population is laid out on a rectangular 2560×2048 grid (cf. Figure 12). 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.

Table 1 GP Parameters for Uppsala Breast Tumour Biopsy

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:	AUROC $\left(\frac{1}{2}\frac{\text{TP}}{\text{No. pos}} + \frac{1}{2}\frac{\text{TN}}{\text{No. neg}}\right)$
	less 1.0 if number of true positive cases (TP=0) or number of true negative cases (TN=0) [23].
Selection:	tournament size 4 in overlapping fine grained 21×21 demes [19], 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



```
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. 11 Screen shot of a 512×400 GP population, i.e. 204 800 programs (from run approximating π [29]) 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 (esp. right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation.

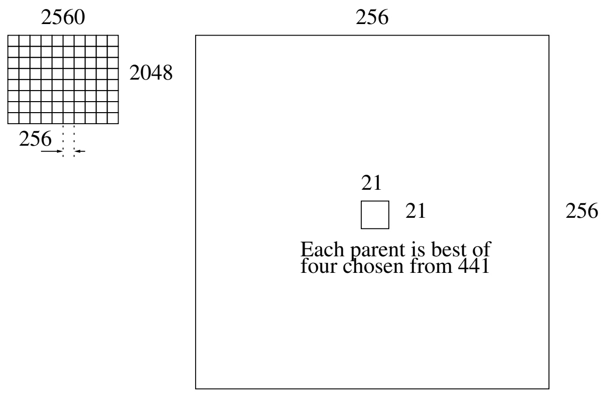


Fig. 12 Left: The GP population of five million programs is arranged on a 2560×2048 grid, which does not wrap around at the edges. At the end of the run the best in each 256×256 tile is recorded. Right: (note different scale) parents are drawn by 4-tournament selection from within a 21×21 region centred on their offspring.

5.2 Data Sets

As part of our large survey of GeneChip flaws [28] we had already down loaded all the HG-U133A and HG-U133B data sets in GEO [3] (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. (Cf. Figures 3–5. R code to quantile normalise and detect spatial flaws is available via http://bioinformatics. essex.ac.uk/users/wlangdon.) 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. Cf. Figure 13.

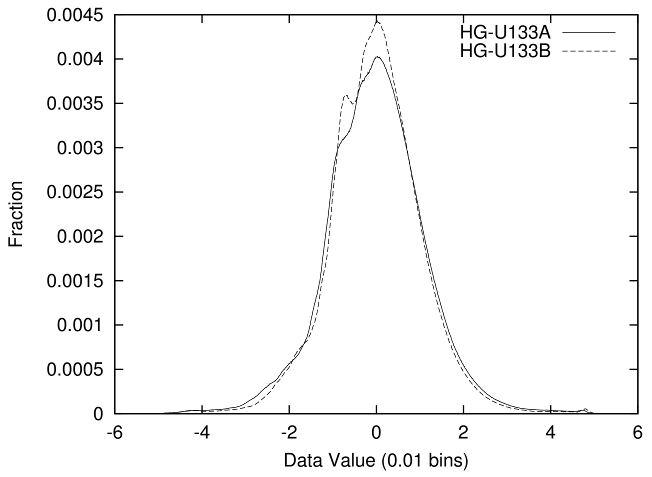


Fig. 13 Uppsala breast cancer distribution of log deviation from average value.

The GeneChip data created by [37] 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 [37] 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.

Initially 120 GeneChip pairs were randomly chosen for training but results on the verification set were disappointing. Accordingly we redesigned our experiment to chose training data in a more controlled fashion. To reduced scope for ambiguity we excluded patients who: a) survived for more than 6 years before dying of breast cancer, b) survived for less than 9.8 years before dying of some other cause, c) patients where the outcome was not known. We split the remaining data as evenly as possible into training (91) and verification (90) sets.

It is known that age plays a prominent role in disease outcomes but the patients were from 28 to 83 years old. So we ordered the data to ensure both datasets had the same age profile. We also balanced as evenly as possible outcome (140 v. 41), tumour size, estrogen receptor (ER) status and progesterone receptor (PgR) status.

6 Results

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 14. Most probes were not used by any of the 8000 programs. 24 810 were used by only one. 2091 by two, and so on.

The 3422 probes which appeared in more than one of the 8000 best of generation ten programs were used in a second pass. In the second pass GP was also run 100 times.

Eight probes appeared in more than 240 of the best 8000 programs of the second pass. These were the inputs to a final GP run. (The GP parameters were again kept the same).

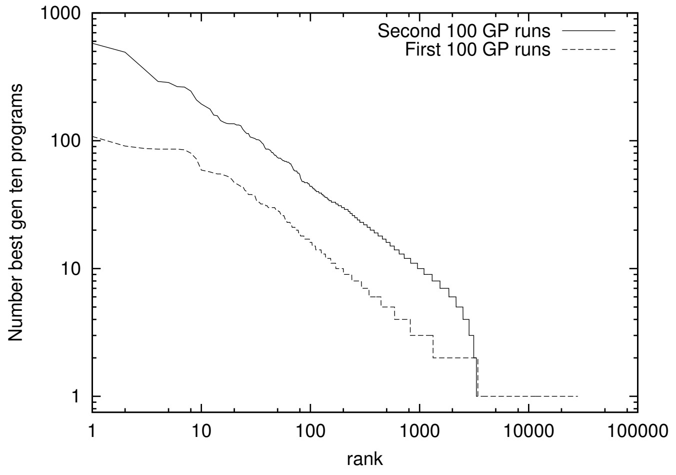


Fig. 14 Distribution of usage of Affymetrix probe in 8000 best of generation 10 GP programs. Both distributions are almost a straight lines (note log scales). Cf. Zipf's law [57].

The GP found several good matches to the 91 training examples. Ever mindful of overfitting [6], 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, cf. Figure 15 and Table 2. The evolved predictor is the sum of two non-linear combination of two genes (decorin/C17orf81 and C17orf81$(2.94 + 1/$S-adenosylhomocysteine hydrolase), cf. Figure 16). Both sub-expressions have some predictive ability. The three probes chosen by GP are each highly correlated

Table 2 Top twenty Affymetrix probes used most in 8000 best of generation 10 second pass GP programs. Cf. Figure 14. The top eight were used in the final GP run.

	Used	X,Y	chiptype	Affy id		NetAffx Gene Title
1	579	350,514	A	200903_s_at	8.mm	S-adenosylhomocysteine hydrolase
2	493	325,511	A	219260_s_at	7.pm	C17orf81. chromosome 17 open reading frame 81
3	363	254,667	A	201893_x_at	2.pm	decorin
4	291	392,213	A	219778_at	4.pm	zinc finger protein, multitype 2
5	286	366,310	B	230984_s_at	10.mm	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	265	324,484	A	216593_s_at	9.mm	phosphatidylinositol glycan anchor biosynthesis, class C
7	263	542,192	B	233989_at	4.mm	EST from clone 35214, full insert. UniGene ID Build 201 (01 Mar 2007) Hs.594768 NCBI
8	245	269,553	B	223818_s_at	2.pm	remodeling and spacing factor 1
9	209	416,107	B	226884_at	10.pm	leucine rich repeat neuronal 1
10	194	613,230	B	235262_at	6.mm	Zinc finger protein 585B. 235262_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 7 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster.*
11	185	61,573	A	221773_at	4.pm	ELK3, ETS-domain protein (SRF accessory protein 2)
12	177	619,316	B	235891_at	6.mm	235891_at was annotated using the Genome Target Overlap based pipeline to a UCSC Genes,ENSEMBL ncRNA identifier using 2 transcript(s).
13	159	531,613	A	NA		
14	157	426,349	A	213706_at	11.pm	glycerol-3-phosphate dehydrogenase 1 (soluble)
15	144	57,434	B	242689_at	10.mm	Ral GEF with PH domain and SH3 binding motif 1. 242689_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 5 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster.*
16	140	15,353	A	213071_at	4.pm	dermatopontin
17	137	65,606	B	229198_at	6.mm	ubiquitin specific peptidase 35
18	136	107,597	A	202995_s_at	4.pm	fibulin 1
19	136	108,393	A	209615_s_at	5.pm	p21/Cdc42/Rac1-activated kinase 1 (STE20 homolog, yeast)
20	136	135,279	A	202995_s_at	2.pm	fibulin 1

with all PM probes in their probeset and so can be taken as a true indication of the corresponding gene's activity. The gene names 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.

The evolved classifier gets 70% of the verification set correct. If we use the three input predictor on the whole Uppsala dataset (excluding the fifteen cases where the outcome is not known), it gets right 184 out of 236 (78%). Figure 17 shows this non-

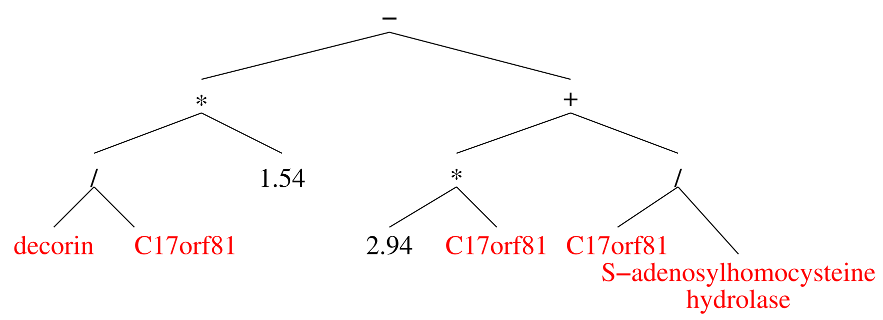


Fig. 15 GP evolved three input classifier. (Using Affymetrix probe names) survival is predicted if $1.54\frac{201893\_x\_at.2pm}{219260\_s\_at.7pm} - 2.94\,219260\_s\_at.7pm - \frac{219260\_s\_at.7pm}{200903\_s\_at.8mm} < 0$

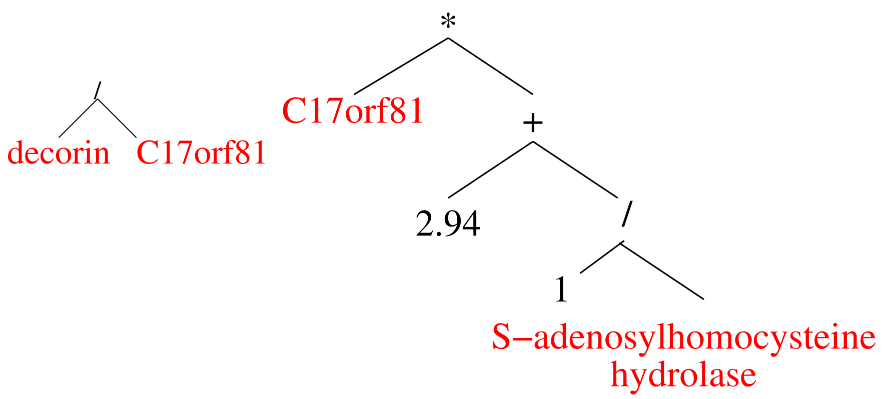


Fig. 16 The GP classifier (Figure 15) is the weighted addition of two two input classifiers (left and right).

linear classifier gives a bigger separation between the two outcomes than a 32-gene model requiring non-linear calculation of more than seven hundred probe values [37, Fig. 3 B].

We tried applying our evolved classifier to a different Breast tumour dataset [44]. Unfortunately we have less background data and no details of follow up treatment for the second group of patients. Also they were treated in another hospital a decade later. Undoubtedly cancer treatment has changed since our data was collected. These, and other differences between the cohorts, may have contributed to the fact that our classifier did less well on the second patient cohort. For example, the Kaplan survival plot to eight years [25, Figure 6] is less well separated than in Figure 17 for twelve years.

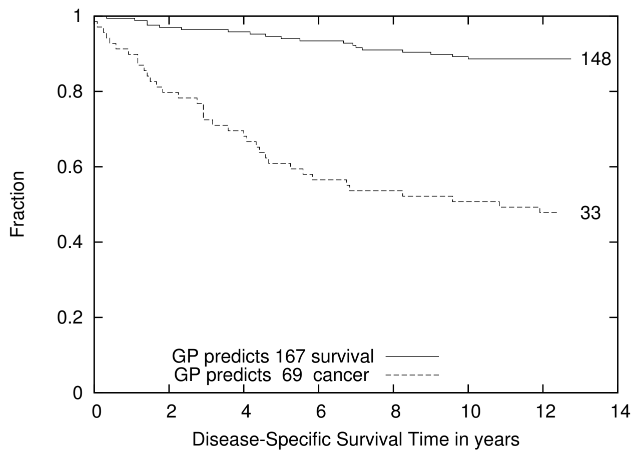


Fig. 17 Kaplan-Meier survival plots, such as the one above, are often used to measure the fraction of patients surviving a certain time after treatment (in this case breast cancer surgery). The three input GP classifier (given in Figure 15) predicts 167 survivors and 69 breast cancer fatalities. The right end of the top line shows that 148 of the 167 predicted to survive lived for more than 12 years. In contrast the lower curve refers to the 67 patients whose gene expression values suggested they would not survive ten years (However 33 of the 67 lived at least $12\frac{1}{2}$ more years).

7 Discussion/Practicalities

In [26] we present detailed timing arguments which show the GP RapidMind interpreter is limited not by the calculations need to interpret the millions of programs but the time taken to fetch their inputs from the GPU's own memory, cf. Figure 18. So replacing interpreted code by compiled code, without addressing the memory bottle neck, would give negligible speed up. Indeed the interpreter is already faster than some compiled GPU approaches.

To a first approximation, any artificial intelligence supervised learning technique, which used this training data in the same way will take about a second or more to test $\frac{1}{4}$ million random classifiers; be they rules, artificial neural networks or programs.

7.1 Speed Up

For this application, the GP interpreter's runs 535 million GP operations per second. 535 MGPop/S is only slightly less than we measured previously [29] with training sets containing ten times as many examples but only about 5kB of training data in total.

To determine speed up, the RapidMind C++ GPU interpreter was converted into a normal C++ GP interpreter and run on the same CPU as was used to host the GPU. I.e. an Intel CPU 6600 2.40GHz. Within the differences of floating point rounding,

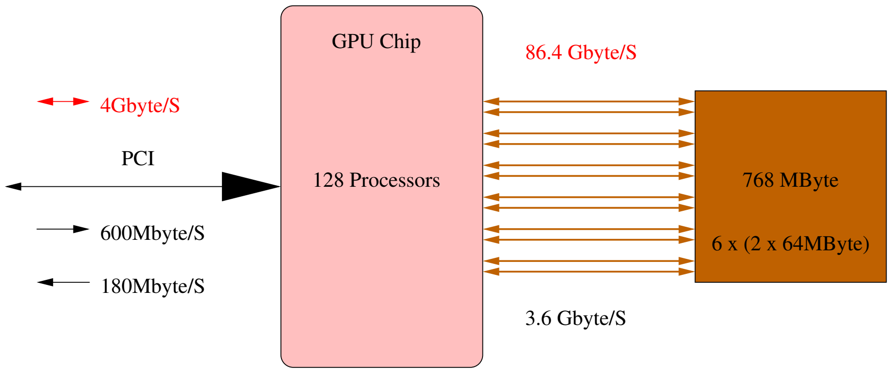


Fig. 18 nVidia 8800 Block diagram. The 128 Stream Processors are connected to the host PC via its PCI express bus. Measurements show RapidMind data both into the GPU and back to the host are efficient (600 and 180 Megabytes per second, i.e. about three quarters of the maximum possible with this PCI). 4 Gigabytes per second is likely to be available soon. The 8800 has twelve 64 megabyte RAM chips. These are paired to give six 16 million $\times$ 64 bit words of storage. Each is connected to the GPU main silicon die by its own 64 bit wide bus. In principle this gives 86.4 GBytes per second of on board memory I/O, however in practise with RapidMind it is impossible to use more than one the six buses simultaneously. Nevertheless it appears that multi-threading of 32-bit access enables the 128 stream processors to obtain about 3.6 GBytes per second.

the GPU program and the new program produced the same answers but in terms of the fitness evaluation the GPU ran 7.59 times faster.

On a different example with more training examples but each containing much less data we obtained a GPU speed up of 12.6 [29]. The GPU interpreter's performance on a number of problems has been in the region $\frac{1}{2}$ to 1 giga GPops, cf. Table 3. In contrast the performance of compiled GPs on GPUs has varied widely, e.g. with number of training examples and program size.

Table 3 Nvidia GeForce 8800 GTX. Genetic Programming Primitives Interpreted Per Second

Experiment	No. of Terminals Inputs+Consts	Functions	Population size	Program size	Stack depth	Test cases	Speed 10^6 OP/S
Mackey-Glass	8+128	4	204 800	11.0	4	1200	895
Mackey-Glass	8+128	4	204 800	13.0	4	1200	1056
Protein	20+128	4	1 048 576	56.9	8	200	504
Laser$_a$	3+128	4	18 225	55.4	8	151 360	656
Laser$_b$	9+128	4	5 000	49.6	8	376 640	190
Cancer	1 013 888+1001	4	5 242 880	$\leq$15.0	4	128	535
GeneChip	47+1001	6	16 384	$\leq$63.0	8	200[a]	314
Sextic[b]	1+na	8	12 500	66.0	17	1024	650[c]

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

[b] $x^6 - 2x^4 + x^2$ approximated by a CUDA system [48] using an optimised RPN interpreter

[c] If we excluded Java code running on the host PC and considered only fitness evaluation on the GPU 1300 MGPop/S was achieved.

7.2 Computational Cube

In genetic programming fitness evaluation, which usually totally dominates run time, can be thought of along three dimensions: 1) the population 2) the training examples and 3) the programs or trees themselves. While it need not be the case, often the GP uses a generational population. Meaning:

1. the whole population is evaluated as a unit before the next generation is created.
2. Often either the whole of the training data, or the same subset of it, is used to calculate the fitness of every member of the population. (Sometimes, in other work, between generations we change which subset is in use.)
3. In many, but by no means all, cases the programs to be tested have a maximum size and do not contain dynamic branches, loops, recursion or function calls. Even for trees, this means the programs can be interpreted in a single pass through a maximum number of instructions. (Shorter programs could, in principle, be padded with null operations.)

We can think of these three dimensions as forming a cube of computations to be done. See Figure 19.

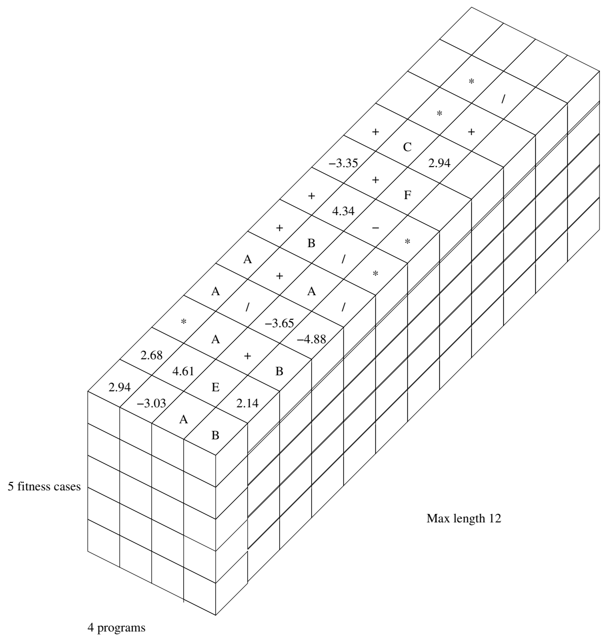


Fig. 19 Evaluating a GP population of four individuals each on the same five fitness cases. There are upto $4 \times 5 \times 12$ GP operations to be performed by, in principle, 240 GPU threads. Each cube needs the opcode to be interpreted, the fitness test case (program inputs) and the previous state of the program (i.e. the stack).

In our implementation (Section 5) the computational cube is sliced vertically (Figure 19) with one GPU thread for each program and each thread looking after all the fitness cases for an individual program. Explicit code in the thread works along the length of the program and processes all the fitness cases for that program. We believe this model of parallel processing works well generally.

Recently we have implemented horizontal slicing. That is, each fitness case has its own GPU thread. The fundamental switch in the GP interpreter makes little difference to the GPU and is readily implemented. Indeed in this respect the GPU is quite flexible. It is relatively straightforward to radically re-arrange the way in which the GPU parallel hardware is used. We have not as yet tried slicing the computational cube along the programs' lengths.

In principle it is possible for each GP instruction to be executed in a different computational thread. In normal programs this would not be contemplated since the complete computational state would have to be passed through each thread. However the complete state for many GP applications is purely the stack. In many cases this is quite small. Therefore executing each function and each GP terminal in a separate GPU thread could be considered. This dimension, also requires dealing with programs that are of different lengths. It is also unattractive since variable data needs to be passed, whilst the corresponding data along the other dimensions are not modified, which saves writing them back to memory.

The efficient use of current GPUs requires many active threads, perhaps upto sixty four per stream processor. With a powerful GPU this means thousands of threads must be kept active to get the best from the hardware, cf. Figure 20. While the computational cube is an attractive idea it is easy to see that far from having too few threads it would be easy to try to divide a GP fitness computation into literally millions of parallel operations, which could not be efficiently implemented. However dividing it along two of the possible three planes is effective.

7.3 *Tesla and the Future of General Purpose GPU Computing*

Unsurprisingly a large fraction of the $618\,10^6$ transistors of the GPU chip are devoted to graphics operations, such as anti-aliasing. This hardware in unlikely to be useful for scientific computing and so represents an overhead. It appears the newly introduced Tesla cards retain this overhead. However if Tesla makes money, the next generation of GPGPU may trade transistors to support graphics operations for transistors to support more scientific data manipulation. E.g. for bigger on chip caches.

7.4 *Absence of Debugger and Performance Monitoring Tools*

RapidMind allows C++ code to be moved between the CPU, the GPU and CELL processors without recompilation. Their intention is the programmer should debug

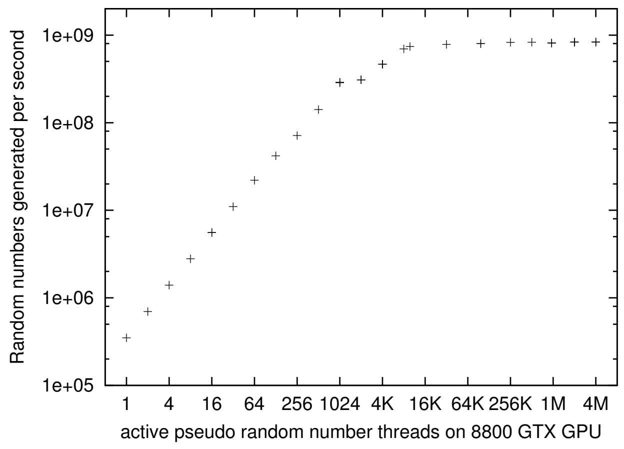


Fig. 20 Park-Miller random numbers per second (excluding host-GPU transfer time) on nVidia 8800 GTX. In the test environment the rate depends upon how effectively the 128 parallel stream processors can be used. Only when there are more than 8192 separate threads do the 128 stream processors effectively saturate [22].

C++ code on the CPU. This allows programmers to use their favourite programming environment (IDE), including compiler and debug tools. Recently RapidMind has introduced a "debug backend" but it too actually runs the code being debugged on the host CPU. Linux GNU GCC/GDB and Microsoft visual C++ are both supported. Owens *et al.* say Google's PeakStream, which has some similarities with RapidMind but was inspired by Brook, "is the first platform to provide profiling and debugging support" [42, p886].

The RapidMind performance log can be configured to include details about communication between the CPU and the GPU. Details include, each transfer, size of transfer, automatic data conversion (e.g. unsigned byte to GPU float) and representation used on the GPU. (E.g. texture size, shape and data type.) However for the internal details of GPU performance and location of bottle necks, one is forced to try and infer them by treating the GPU as a black box.

Recent software advances under the umbrella term of general purpose computing on GPUs (GPGPU) have considerably enhanced the use of GPUs. Nevertheless, GPU programming tools for scientific and/or engineering applications are primitive and getting the best out of GPUs "remains something of a black art" [42, p896,p897]. This is exacerbated by 1) the small number and consequent instability of hardware and software vendors in the GPGPU market. 2) Hardware specific program interfaces (APIs) which have been much more likely to require modification to existing programs to take advantage of new hardware than the corresponding interfaces in CPUs. 3) Lack of vendor independent APIs [42].

For GPU manufactures GPGPU remains an add-on to their principal market: games. Accompanying the rapid development in hardware they make corresponding changes in their software. This means the manufacturer's APIs tend to tested and optimised for a few leading games. This can have unfortunate knock effects on GPGPU applications [42]. Potentially GPU developers can isolates themselves from this by using higher level tools or languages, like RapidMind.

Despite their undoubted speed, if GPUs remain difficult to use, they will remained limited to specialised niches. To quote John Owens “Its the software, stupid” [41].

7.5 C++ Source Code

C++ code can be down loaded via anonymous ftp or `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/gpu_gp_2.tar.gz` Also `gpu_gp_1.tar.gz` has a small introductory example [29]. Whereas `random-numbers/gpu_park-miller.tar.gz` is for generating random numbers [22].

8 Conclusions

We have taken a large GeneChip breast cancer biopsy dataset with more than a million inputs to demonstrate a successful computational intelligence application running in parallel on GPU mass market gaming hardware (an nVidia GeForce 8800 GTS). We find a 7.6 speed up.

Initial analysis of the GPU suggests that the major limit is access to its 768Mbytes where the training data is stored. Indicating that, if other computational intelligence techniques, access the training data in similar ways, they would suffer the same bottle neck.

Whilst primarily interested in mutation of the p53 gene, Miller *et al.* tried support vector machines and k nearest neighbour but say diagonal linear discriminant analysis worked better for them [37]. They used DLDA to construct a non-linear model with more than 704 data items per patient. The non-linear model evolved by genetic programming uses only three. It has been demonstrated on a separated verification dataset. As Figure 17 shows, on all the available labelled data (236 cases), the classifier evolved using a GPU gives a wider separation in the survival data.

Acknowledgements

I would like to thank Denis Robilliard, Derek Foster, John Owens, Lance Miller, Simon Harding, Nolan White, Paul Price, Joanna Armbruster and Nick Holby.

References

1. Wolfgang Banzhaf, Simon Harding, William B. Langdon, and Garnett Wilson. Accelerating genetic programming through graphics processing units. In *Genetic Programming Theory and Practice VI*, chapter 15. Springer, Ann Arbor, 15-17 May 2008.
2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann, 1998.
3. Tanya Barrett, Dennis B. Troup, Stephen E. Wilhite, Pierre Ledoux, Dmitry Rudnev, Carlos Evangelista, Irene F. Kim, Alexandra Soboleva, Maxim Tomashevsky, and Ron Edgar. NCBI GEO: mining tens of millions of expression profiles–database and tools update. *Nucleic Acids Research*, 35(Database issue):D760–D765, January 2007.
4. Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In Panayiotis Bozanis and Elias N. Houstis, editors, *Advances in Informatics, 10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece, November 11-13, 2005, Proceedings*, *LNCS 3746*, pages 415–425. Springer, 2005.
5. Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens *et al.*, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 7-11 July 2007. ACM Press.
6. David Peter Alfred Corney. *Intelligent Analysis of Small Data Sets for Food Design*. PhD thesis, University College, London, 2002.
7. 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.
8. Marc Ebner, Markus Reinhardt, and Jürgen Albert. Evolution of vertex and pixel shaders. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, *LNCS 3447*, pages 261–270, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.
9. Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the ACM/IEEE SC2004 Conference Supercomputing*, 2004.
10. William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley and Sons, 2 edition, 1957.
11. Randy Fernando. GPGPU: general general-purpose purpose computation on GPUs. NVIDIA Developer Technology Group, 2004. Slides.
12. Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, March-April 2007.
13. Stephane Gobron, Francois Devillard, and Bernard Heit. Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, 2007.
14. S. L. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, page 2, Canada, 2007. IEEE Press.
15. Simon Harding. Evolution of image filters on graphics processor units using Cartesian genetic programming. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Press.
16. Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner *et al.*, editors, *Proceedings of the 10th European Conference on Genetic Programming*, *LNCS 4445*, pages 90–101, Valencia, Spain, 11 - 13 April 2007. Springer.
17. Simon L. Harding, Julian F. Miller, and Wolfgang Banzhaf. Self-modifying Cartesian genetic programming. In Dirk Thierens *et al.*, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 1021–1028, London, 7-11 July 2007. ACM Press.

18. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
19. William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, 1998.
20. 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.
21. W. B. Langdon. Evolving GeneChip correlation predictors on parallel graphics hardware. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 4152–4157, Hong Kong, 1-6 June 2008. IEEE Press.
22. W. B. Langdon. A fast high quality pseudo random number generator for graphics processing units. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 459–465, Hong Kong, 1-6 June 2008. IEEE Press.
23. W. B. Langdon and S. J. Barrett. Genetic programming in data mining for drug discovery. In Ashish Ghosh and Lakhmi C. Jain, editors, *Evolutionary Computing in Data Mining*, volume 163 of *Studies in Fuzziness and Soft Computing*, chapter 10, pages 211–235. Springer, 2004.
24. 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, 2004.
25. W. B. Langdon, R. da Silva Camargo, and A. P. Harrison. Spatial defects in 5896 HG-U133A GeneChips. In Joaquin Dopazo, Ana Conesa, Fatima Al Shahrour, and David Montener, editors, *Critical Assesment of Microarray Data*, Valencia, 13-14 December 2007. Presented at EMERALD Workshop.
26. W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, 2008.
27. W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
28. 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*, 2009. In press.
29. William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In Michael O'Neill *et al.*, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, *LNCS 4971*, pages 73–85, Naples, 26-28 March 2008. Springer.
30. Fredrik Lindblad, Peter Nordin, and Krister Wolff. Evolving 3D model interpretation of images using graphics hardware. In David B. Fogel *et al.*, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 225–230. IEEE Press.
31. Weiguo Liu, Bertil Schmidt, Geritt Voss, Andre Schroder, and Wolfgang Muller-Wittig. Biosequence database scanning on a GPU. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 25-29 April 2006. IEEE Press.
32. Youquan Liu and De Suvranu. CUDA-based real time surgery simulation. *Studies in Health Technology and Informatics*, 132:260–262, 2008.
33. Jörn Loviscach and Jennis Meyer-Spradow. Genetic programming of vertex shaders. In M. Chover, H. Hagen, and D. Tost, editors, *Proceedings of EuroMedia 2003*, pages 29–31, 2003.
34. Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks, IJCNN '05*, number 1, pages 622–626, 31 July-4 Aug 2005.
35. 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.
36. Jennis Meyer-Spradow and Jörn Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, editors, *Eurographics 2003 Short Paper Proceedings*, pages 301–306, 2003.
37. Lance D. Miller, Johanna Smeds, Joshy George, Vinsensius B. Vega, Liza Vergara, Alexander Ploner, Yudi Pawitan, Per Hall, Sigrid Klaar, Edison T. Liu, and Jonas Bergh. An expression

signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. *Proceedings of the National Academy of Sciences*, 102(38):13550–5, Sep 20 2005.

38. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.
39. NVIDIA GeForce 8800 GPU architecture overview. Technical Brief TB-02787-001_v0.9, Nvidia Corporation, November 2006.
40. NVIDIA CUDA compute unified device architecture, programming guide. Technical Report version 0.8, NVIDIA, 12 Feb 2007.
41. John Owens. Experiences with GPU computing, 2007. Presentation slides.
42. 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.
43. 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.
44. Yudi Pawitan, Judith Bjohle, Lukas Amler, Anna-Lena Borg, Suzanne Egyhazi, Per Hall, Xia Han, Lars Holmberg, Fei Huang, Sigrid Klaar, Edison T Liu, Lance Miller, Hans Nordgren, Alexander Ploner, Kerstin Sandelin, Peter M Shaw, Johanna Smeds, Lambert Skoog, Sara Wedren, and Jonas Bergh. Gene expression profiling spares early breast cancer patients from adjuvant therapy: derived and validated in two population-based cohorts. *Breast Cancer Research*, 7:R953–R964, 3 Oct 2005.
45. 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).
46. George R. Price. Selection and covariance. *Nature*, 227, August 1:520–521, 1970.
47. J. Reggia, M. Tagamets, J. Contreras-Vidal, D. Jacobs, S. Weems, W. Naqvi, R. Winder, T. Chabuk, J. Jung, and C. Yang. 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.
48. Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Population parallel GP on the G80 GPU. In Michael O'Neill *et al.*, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, *LNCS 4971* pages 98–109, Naples, 26-28 March 2008. Springer.
49. Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474. 2007.
50. A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.
51. Graham J. G. Upton and Ian Cook. *Introducing Statistics*. Oxford University Press, 2nd edition, 2001.
52. Garnett Wilson and Wolfgang Banzhaf. Linear genetic programming GPGPU on Microsoft's Xbox 360. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Press.
53. Garnett Wilson and Simon Harding. WCCI 2008 special session: Computational intelligence on consumer games and graphics hardware (CIGPU-2008). *SIGEvolution*, 3(1):19–21, 2008.
54. Adrianto Wirawan, Chee Kwoh, Nim Hieu, and Bertil Schmidt. CBESW: sequence alignment on the PlayStation 3. *BMC Bioinformatics*, 9(1):377, 2008.
55. Zhijin Wu, Rafael A. Irizarry, Robert Gentleman, Francisco Martinez-Murillo, and Forrest Spencer. A model-based background adjustment for oligonucleotide expression arrays. *Journal of the American Statistical Association*, 99(468):909–917, 2004.
56. Jianjun Yu, Jindan Yu, Arpit A. Almal, Saravana M. Dhanasekaran, Debashis Ghosh, William P. Worzel, and Arul M. Chinnaiyan. Feature selection and molecular classification of cancer using genetic programming. *Neoplasia*, 9(4):292–303, April 2007.
57. George Kingsley Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press Inc., 1949.