Soft Computing ***Online first*** Prepublication
DOI 10.1007/s00500-008-0296-x

GP on SPMD parallel Graphics Hardware for mega Bioinformatics Data Mining

W. B. Langdon, A. P. Harrison

Mathematical and Biological Sciences, University of Essex, Wivenhoe Park, Colchester, CO4 3SQ, UK

Abstract We demonstrate a SIMD C++ genetic programming system on a single 128 node parallel nVidia GeForce 8800 GTX GPU under RapidMind's GPGPU Linux software 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 5 million programs winnow useful variables from the chaff at more than 500 million GPops per second. Sources available via FTP.

1 Introduction

Due to its speed, price and availability, there is increasing interest in using mass market graphics hardware (GPUs) for scientific applications. So far there are a few reported successful applications of GPUs to Bioinformatics (Section 3). In Section 4 we will describe one where soft computing [Langdon and Buxton, 2004] is used to data mine a small number of indicative mRNA gene transcript signals from breast cancer biopsys (tissue samples) each with more than a million variables. In Sections 5 and 6 GP [Koza, 1992; Banzhaf *et al.*, 1998] [Langdon and Poli, 2002] is used on a powerful GPU [Langdon and Banzhaf, 2008] to find a simple non-linear combination of three mRNA measurements which predicts long term outcomes at least as well as DLDA, SVM and KNN using seven hundred measurements [Miller *et al.*, 2005].

2 Using Games Hardware GPUs for Science

[Owens *et al.*, 2007] gives a recent survey of running scientific or indeed general purpose computation on mass market graphics cards (GPGPU). Whilst there is increasing interest, to date both Bioinformatics and soft computing 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. [Fernando, 2004, page 4]). 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 [Moore, 1965][1] 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 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 gamming 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

[1] Although forty years ago Intel's Gordon Moore wrote about number of components per chip, "Moore's Law" has popularly taken on a wider meaning, which includes doubling of speed.

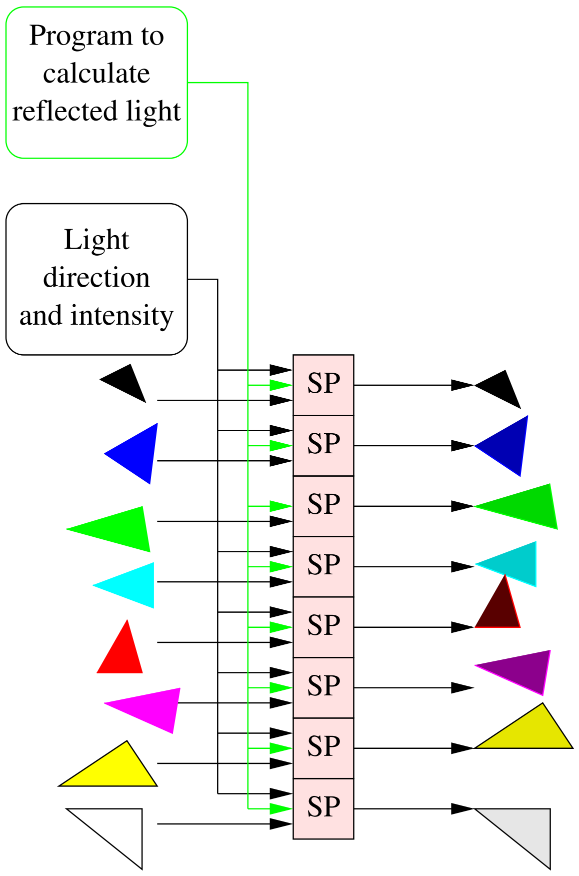
Program to calculate reflected light
Light direction and intensity
SP
SP
SP
SP
SP
SP
SP
SP

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.

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 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} FLOP/Second
- 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 16) 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 [NVIDIA, 2006, page 36], 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.

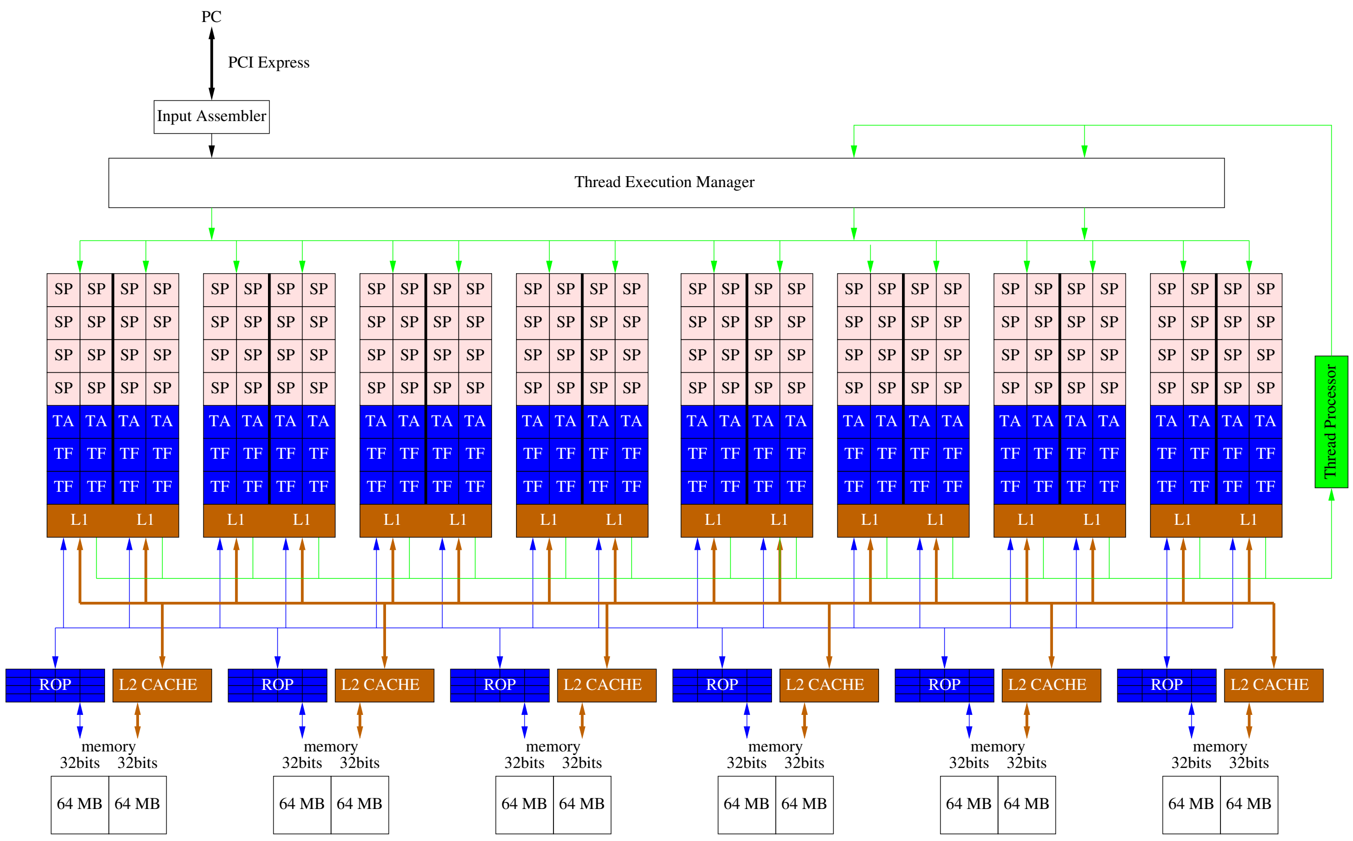

Fig. 2 nVidia 8800 Block diagram. The 128 1360 MHz Stream Processors are arranged in 16 blocks of 8. Blocks share 16 KB memory (not shown), an 8/1 KB L1 cache, 4 Texture Address units and 8 Texture Filters. The bus (brown) links off chip RAM at 900 (1800) MHz [NVIDIA, 2006; NVIDIA, 2007]. There are 6 Raster Operation Partitions.

3 GPUs in Bioinformatics and Soft Computing

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. However early results have been mixed.

[Charalambous *et al.*, 2005] successfully used a relatively low powered GPU to demonstrate inference of evolutionary inheritance trees (by porting RAxML onto an nVidia). However a more conventional MPI cluster was subsequently used [Stamatakis, 2006]. Sequence comparison is the life blood of Bioinformatics. [Liu *et al.*, 2006] ran the key Smith-Waterman algorithm on a high end GPU. They demonstrated a reduction by a factor of up to 16 in the look up times for most proteins. Using CUDA [Schatz *et al.*,] ported a different sequence searching tool (MUMmer) to a more modern G80 GPU and obtained speed ups of 3–10 when matching short DNA strands against much longer sequences. By breaking queries into GPU sized fragements, they were able to run short sequences (e.g. 50 bases) against a complete human chromosome. [Gobron *et al.*, 2007] 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.

Soft computing applications of GPUs have included artificial neural networks (e.g. multi layer perceptrons and self organising networks [Luo *et al.*, 2005]), genetic algorithms [Fok *et al.*, 2007] and a few genetic programming experiments [Lindblad *et al.*, 2002], [Loviscach and Meyer-Spradow, 2003; Ebner *et al.*, 2005; Reggia *et al.*, 2006; Harding and Banzhaf, 2007a] [Harding and Banzhaf, 2007b; Harding *et al.*, 2007] [Chitty, 2007].

Most GPGPU applications have only required a single graphics card, however [Fan *et al.*, 2004] shows large GPU clusters are also feasible. As [Owens *et al.*, 2007] makes clear games hardware is now breaking out of the bedroom into scientific and engineering computing. So far only certain niches have been explored.

4 Gene Expression in Breast Cancer

[Miller *et al.*, 2005] 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. [Miller *et al.*, 2005]'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. Obviously such tiny measuring devices are very subject to noise are so between 11 and 20 reading 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 11 or more pairs of readings into a single value to represent the activity of a gene. [Miller *et al.*, 2005] used Affymetrix' MAS5. MAS5 uses outlier detection etc. to take a robust average of the 22 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 [Wu *et al.*, 2004], ignore the control member of each pair.

[Miller *et al.*, 2005] 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. 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 GeneChips using a new technique [Langdon *et al.*, ; Langdon *et al.*, 2007a]. While [Miller *et al.*, 2005] 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.

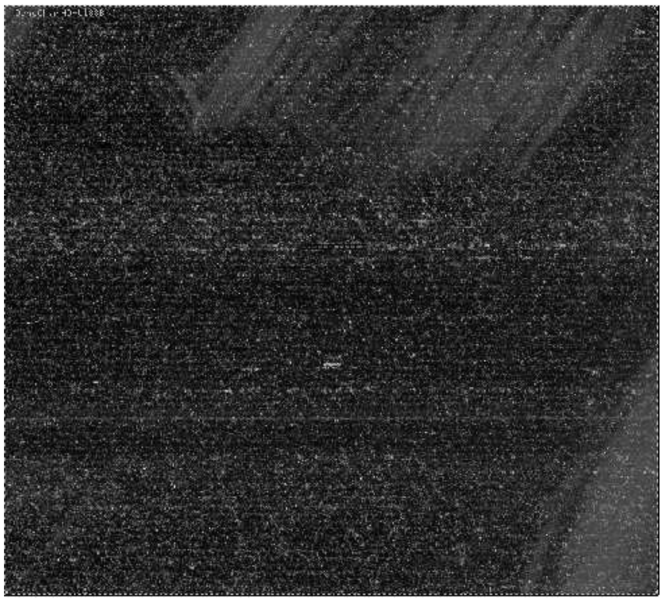
Fig. 3 First HG-U133B Breast Cancer GeneChip. Data have been quantile normalised (effectively log transformed). Large spatial flaws can be seen at the top and lower right hand corners.

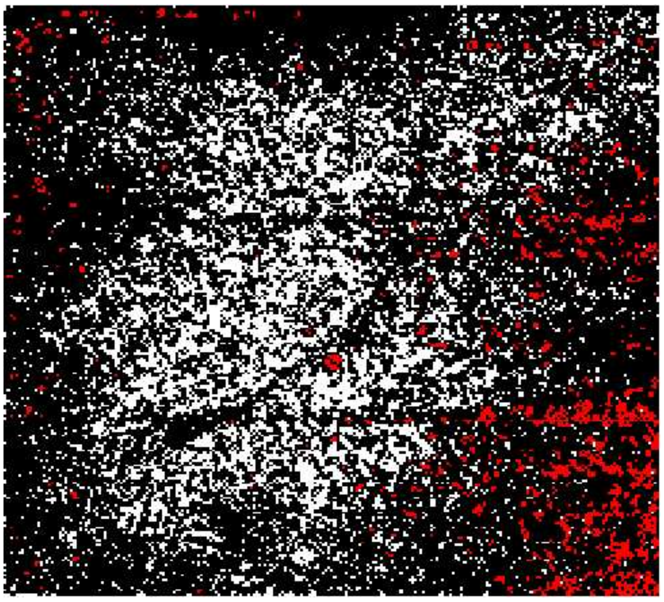
Fig. 4 Density of spatial flaws in 98 HG-U133A Breast Cancer GeneChips. Red more than 20 of 98 GeneChips are flawed (Black at least one).

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 (like Lisp S-expressions) or as networks (Cartesian GP) [Harding and Banzhaf, 2007a] and used the GPU for fitness evaluation. Harding compiled his graphs 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. [Langdon, 2007a] explains how it is possible for SIMD GPU to interpret multiple programs simultaneously. Next we shall briefly recap how to interpret mul-

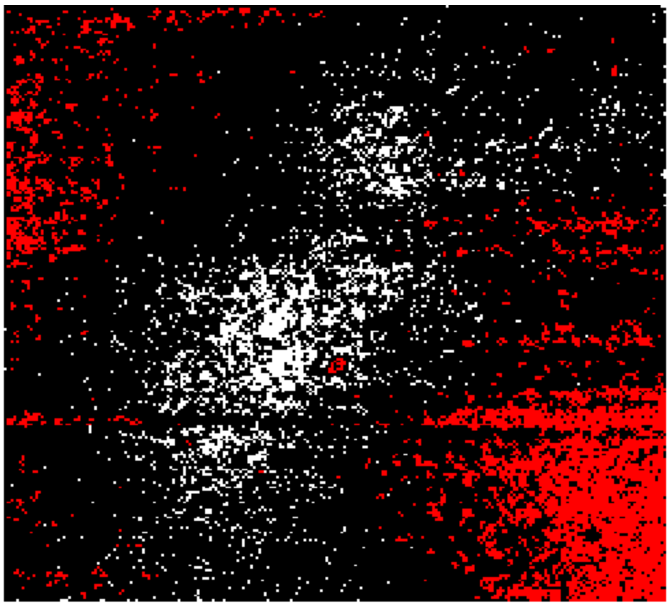

Fig. 5 Density of spatial flaws in 98 HG-U133B Breast Cancer GeneChips showing HG-U133B have more spatial errors than HG-U133A, c.f. Fig. 4.

tiple programs simultaneously on a SIMD computer 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.

A `cond` statement has three arguments. The first is the control. It decides which of the other two arguments which actually be 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. Otherwise to do nothing. (See Figures 6 and 7.) In traditional computing, this would be regarded as wasteful since each instruction in the program must be tested against every legal opcode. 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 shall see in Section 7.1, the no-ops and indeed the

```
#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. 6 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.

```
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. 7 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 8 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 8 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)`.

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 modified, they are stored in the GPU at the start of the run. Each generation, only

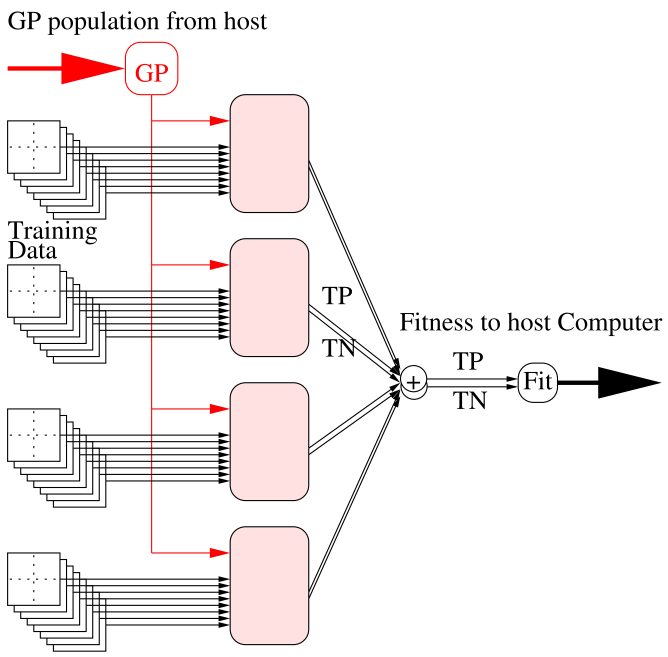

Fig. 8 GPU software architecture needed to overcome 2^{22} and $\leq$ 16 arrays GPU limits in order to access 512MB of training data and a population of 5 million GP programs. The population is split into 20 256k parts by the host CPU. 256k GP programs are passed to GPU (red). There are four parameterised instances of the SIMD interpreter (pink). Each uses 1+8+2 arrays (plus others for control, not shown, and debug, total 12 or more). Each instance is limited to $\leq$ 16 arrays. We pack four sets of patient data ($4 \times 1\,013\,888$) per array. 4 groups of 8 arrays allows 512M of training data.

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 8.

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 16 inputs. I.e. the interpreter cannot access all 32 training data arrays simultaneously. Since it 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 8 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 RapdiMind is shown in Figure 9.

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

//NP is Number of programs in Population
const int NP = 2560*2048;

//Maximum GP individual length, allow stop code
const int LEN =15+1;

//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. 9 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 given by `write_data()`. RapidMind variables `TP0` to `TN` are used to calculate fitness, cf. Figure 8. 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.

As described in [Langdon and Banzhaf, 2008] 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 [Langdon and Banzhaf, 2008] instead at least 20 bits are needed. In fact we us a full word per opcode. This means a population of 5 million 15 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 16 words (15, 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 [Langdon and Banzhaf, 2008] that, to get the best from the GPU, its work should be fed into the GPU in units of between 1 and 10 seconds.

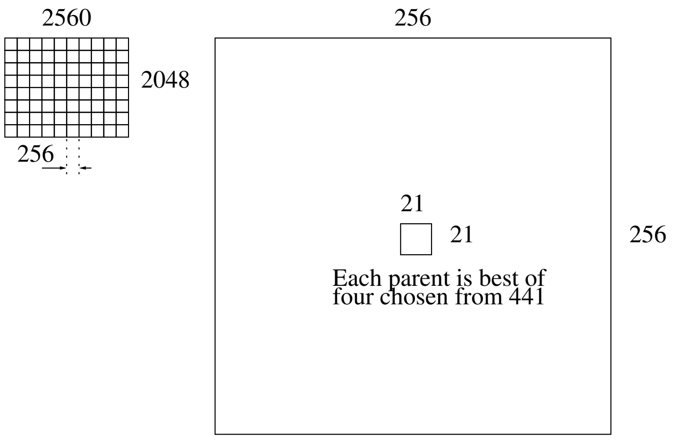

Fig. 10 Left: The GP population is arranged on a 2560×2048 grid, which does not wrap around at the edges. At the end of the run the best in each 256×256 tile is recorded. Right: (note different scale) parents are drawn by 4-tournament selection from within a 21×21 region centred on their offspring.

5.1 GP for large scale data mining

We have previously described using genetic programming to data mine GeneChip data [Langdon and Buxton, 2004]. 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 15 nodes. (Whilst comparatively small, [Yu *et al.*, 2007] successfully evolved classifiers limited to only 8.)

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 [Langdon and Buxton, 2004]. 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 [Price, 1970]. 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 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 [Feller, 1957, p284]. On average $n(\log(n)+c)$ random trials are needed to collect all of n coupons. The exact value of c depends upon n, but for large n, c is in the region of e^{-1}. 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 $14\,10^6$. The spread in the distribution of answers to the coupon collector problem is of the order of square root of n. Therefore if we overshoot by a few thousands, we are sure to get all the leafs into the initial population. Since a program of 15 nodes has 8 leafs and half of these are constants we need at least $\frac{1}{4}14\,10^6 = 3.6\,10^6$ random trees. An initial population of 5 million ensures this.

In [Langdon and Banzhaf, 2008] 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 4 inputs). We do not want to run our GP 25 000 times. The compromise was to use overlapping fine grained demes to delay convergence of the population [Langdon, 1998]. The GP population is laid out on a rectangular 2560×2048 grid (cf. Figure 10). This was divided into 80 256×256 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 [Langdon and Barrett, 2004] $\left(\frac{1}{2}\frac{\text{TP}}{\text{No. pos}} + \frac{1}{2}\frac{\text{TN}}{\text{No. neg}}\right)$ less 1.0 if number of true positive cases (TP=0) or number of true negative cases (TN=0).
Selection:	tournament size 4 in overlapping fine grained 21×21 demes [Langdon, 1998], 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

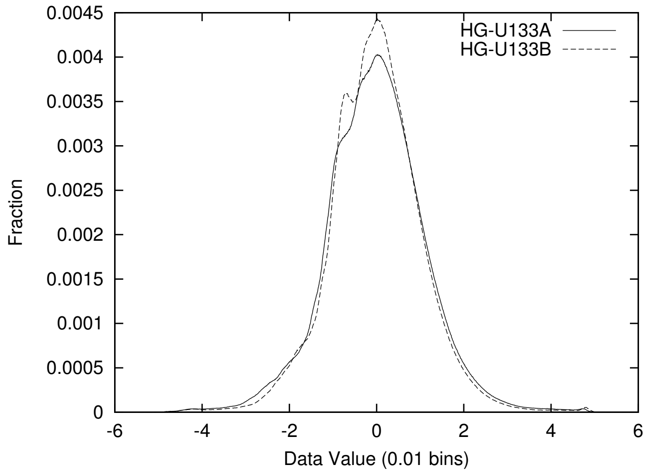

Fig. 11 Uppsala breast cancer distribution of log deviation from average value.

5.2 Data Sets

As part of our large survey of GeneChip flaws we had already down loaded all the HG-U133A and HG-U133B data sets in GEO (6685 and 1815 respectively) and calculated a robust average for each probe [Langdon *et al.*, ; Barrett *et al.*, 2007]. These averages across all these human tissues were used to separately quantile normalise the 251 pairs of HG-U133A and HG-U133B GeneChips and flag locations of spatial flaws. (Cf. Figures 3–5.) 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 11.

The GeneChip data created by [Miller *et al.*, 2005] 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 [Miller *et al.*, 2005] 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 80 256 × 256 squares was recorded. The distribution of inputs used by these 100 × 80 programs is given in Figure 12. 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).

The GP found several good matches to the 91 training examples. Ever mindful of overfitting. As a solution we chose one with the fewest inputs (3). GP found a nonlinear combination of two PM probes and one MM probe from near the middle of HG-U133A, cf. Figure 13 and Table 2. The evolved predictor is the sum of two nonlinear combination of two genes (decorin/C17orf81 and C17orf81(2.94 + 1/S-adenosylhomocysteine hydrolase), cf. Figure 14). Both sub expressions have some predictive ability. The three probes chosen by GP are each highly correlated 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.

Table 2 Top 20 Affymetrix probes used most in 8000 best of generation 10 second pass GP programs. Cf. Figure 12.

	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

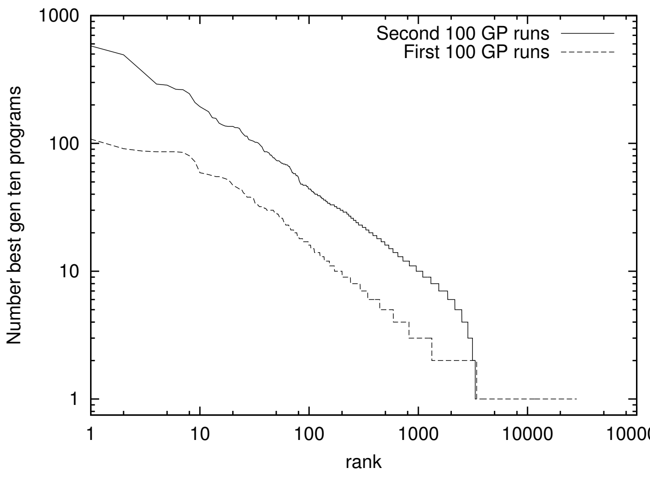

Fig. 12 Distribution of usage of Affymetrix probe in 8000 best of generation 10 GP programs. Both distributions are almost a straight lines (note log scales). Cf. [Zipf, 1949].

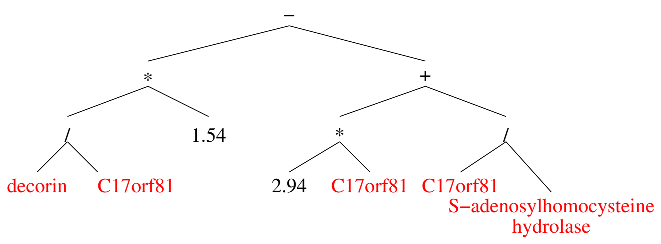

Fig. 13 GP evolved three input classifier. (Using Affymetrix probe names) survival is predicted if $1.54\frac{201893\_x\_at.2pm}{219260\_s\_at.7pm} - 2.94\,219260\_s\_at.7pm - \frac{219260\_s\_at.7pm}{200903\_s\_at.8mm} < 0$

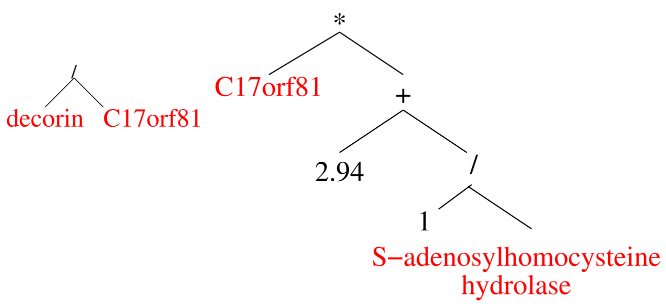

Fig. 14 The GP classifier (Figure 13) is the weighted addition of two two input classifiers (left and right).

The evolved classifier gets 70% of the verification set correct. If we use the 3 input predictor on the whole Uppsala dataset (excluding the 15 case where the outcome is not known), it gets right 184 out of 236 (78%).

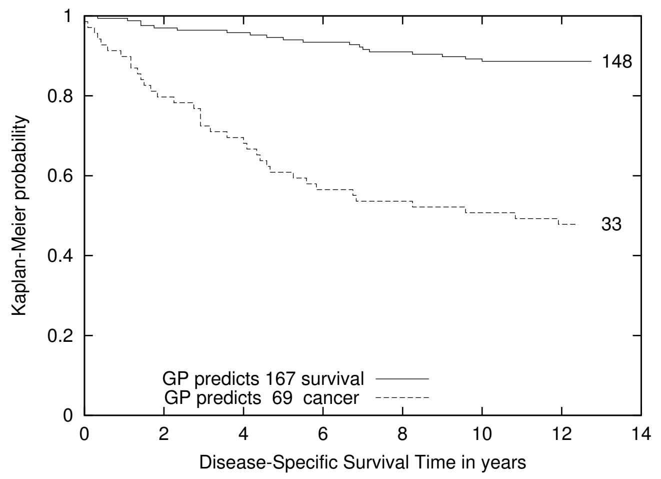

Fig. 15 Kaplan-Meier survival. Three input GP classifier predicts 167 survivors and 69 breast cancer fatalities. The survival times are plotted for each group separately.

Figure 15 shows this non-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 [Miller *et al.*, 2005, Fig. 3 B].

We tried applying our evolved classifier to a different Breast tumour dataset [Pawitan *et al.*, 2005]. 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 8 years [Langdon *et al.*, 2007a, Figure 6] is less well separated than in Figure 15 for 12 years.

7 Discussion/Practicalities

7.1 Limits to speed of GPU Computing

Measurements of data transfer into and out of the GPU show the CPU to GPU data transfer rate is $613\,10^6$ byte/S for this application. This is near the bandwidth of the host computer's PCI bandwidth (800 MByte/S). As expected, reading data back from the GPU is slower, at $181\,10^6$ byte/S. (nVidia supports PCI Express, which in theory should provide 4 GB/S both to and from the GPU. NB. this requires special support in the PC's mother board to actually obtain data transfer at 4 GB/S.) Once the GP interpreter and training data has been loaded, only the new GP individuals (16 floats each) have to be transfered to the GPU and their fitness values (1 float each) read back. Since the GPU can cope with arrays of 2^{22} we break the population into 20 units, each of 256k. (NB. $256\text{k}\times 16 = 2^8 \times 2^{10} \times 2^4 = 2^{22}$.) The program upload is calculated to take 27mS and the fitness read back to take 6mS.

To calculate the fitness of an individual requires the GPU to schedule 7 programs (cf. Figure 8) per individual. It appears the GPU's thread scheduler (Figure 2, green) works in units of 16 programs. So each 256k unit will require 114 688 scheduling operations. We estimate this and other book keeping operations will take 15mS.

Figure 2 shows the dominant role of the GPU bus linking the GPU chip to the GPU's 6 pairs of memory chips (arranged as 6 pairs of 16M 32 bit words [Samsung, 2007]). Details of how contentions for the bus when multiple stream processing units, etc. need access to the same memory bank, how address and data signals are multiplexed by the bus and how RapidMind arrays are allocated to memory banks are not clear. However if we make some reasonable assumptions we find that the GPU's performance is limited by the off chip RAM.

For simplicity and since they are relatively small, we will ignore data transfers inside the GPU needed for the programs themselves, their fitness and the class labels. Instead we concentrate upon the inputs to the GP programs during fitness training. We also assume the data caches are able to hold all local variables, the active programs and avoid multiple data reads during the execution of one program. (There are two caches: L1 8KB and L2 128KB [NVIDIA, 2007; Rys, 2006].) We assume the GPU splits the 256k programs into 2048 128 units and runs each on its own processor. The 128 units are themselves split into 8 groups of 16. Each 16 share access to memory, cf. Figure 2. We assume the GPU will wait until all 16 are finished before starting the next group. Therefore runtime will be dominated by the slowest programs. I.e. those needing the most data. The maximum possible is eight inputs per GP individual. Whilst each of the GP programs may have different inputs, they will all come from the same training example, which will be stored in the same RapidMind array. We assume each array is stored in one of the six memory banks. Therefore that memory bank will receive up to 128×8 requests for single floats almost simultaneously.

The caches will probably not help, since each GP program will want different data. Given the random locations requested, reading ahead and transferring 64 bits rather than just the requested 32 will also not help.

Thus to process 256k programs on 128 training case will entail transferring up to $256\text{k}\times 128\times 8 = 256\text{M}$ floats. Since each memory chip can deliver data at 900M 32 bit word/S [Samsung, 2007], even ignoring bus contention, this will take at least $256 \times 2^{20}/900\,10^6 = 298$mS. Perhaps a more realistic estimate would be about 650mS (i.e. ignoring time taken to issue the memory request and assuming contention between the L1 caches can be resolved in about 1 nS). We did try spreading the simultaneous memory load more evenly but it appears that the L1/bus hardware cannot take advantage of overlapping memory reads to different chips. Since timing will be dominated by the slow threads each stream processor is effectively blocked when it needs data which is not in the L1 cache. That is, until the GP data value arrives from one of GPU's 12 memory chips.

Without knowing the details of the machine code generated by RapidMind, it is difficult to estimate how many instructions the GPU is actually performing. However it seems reasonable to assume that the stack join operations needed by the interpreter are fairly complex and so we guess about ten GPU instructions for each of the five GP opcodes (+ - $\times$ / and leaf). Allowing 3 to control the innermost loop gives 53 floating point operations per GP primitive. Again the SIMD parallel operations will move at the speed of the slowest program. Therefore we can reasonably estimate GPU time spent in computation at $256\text{k} \times 128 \times 15 \times 53 = 267\,10^6$. (128 training examples and each program being 15 instructions long.) This will take about $267\,10^6/(128 \times 1.350\,10^9) = 154\text{mS}$. This is smaller than the lower bound for time taken by memory transfers and confirms that, for this Bioinformatics application, only a little improvements can be extracted by improving the GPU interpreter itself.

One of the standard objections to interpreted languages, is that they tend to be slow in comparison to compiled programs. We would expected a decent compiler to be able to reduce the number of GPU operations by at least a factor of ten compared to our SIMD interpreter. Supposing it did. This might reduced the time from 154mS to 15mS. Assuming we still do not get the advantage of 64 bit data transfers and multiple GPU programs can be loaded and scheduled efficiently, this would and reduce the execution time of 256k programs from 883mS to 775mS, a speed up of less than 19%. This highlights a problem with using FLOPS: the compiler approach uses only a tenth as many floating point operations, so its FLOP rating will be about a tenth that of the interpreter, despite being slightly faster.

To a first approximation, any soft computing supervised learning technique, which used this training data in the same way will take about a second or more to test 256k random classifiers; be they rules, artificial neural networks or programs.

7.2 Over fitting and Cross Validation

The evolved program has been tested on data that was never used in any part of the training process. Therefore its performance on this independent verification dataset can be reasonably taken as a true indication of its performance on new data from the same population. I.e. women of the same age distribution etc. as those in Uppsala, whose breast tumours have been biopsied and who undergo the same follow up treatment. However cancer treatments have changed since 1987. This emphasises the problem of working with historical data and the difficulties of predicting the future!

n-fold cross validation is another popular over fitting technique. In cross validation the available training data is split evenly into n "folds". The machine learning approach uses $n-1$ folds as training data and produces an output (e.g. a artificial neural network, a decision tree). The performance of the predictor is measured on the remaining data. Notice that where $n \gg 1$, this has the potential advantage that the predictor is trained on almost all the available data. These train and test operations are repeated n times, each time leaving out a different fold. This produces n predictors. The performance of the learning technique is taken as the average of the performances across the n folds that were left out.

In general such a n-fold scheme might work well on a GPU. Firstly we have to run the machine learning n times, so having a fast GPU implementation would be attractive. Secondly, in principle, all of the n folds of training data could be pre-loaded into the GPU and be shared by the n training runs. Of course this is not feasible in our example, since we have filled the GPU with about half the training data. Nevertheless most machine learning tasks are far less demanding.

Unfortunately n-fold cross validation, including the case where n =the total number of training examples (known as leave-one-out cross validation) usually assumes the machine learning technique is deterministic. I.e. that if run on the same training examples, it will always produce the same ANN (or decision tree etc.). Genetic programming is a stochastic algorithm. Even if run on the same training inputs, the evolved answer may be different when the GP is run again. In principle, the average GP performance on a given fold could be estimated by running the GP several times. Say A times. This could be repeated n times, i.e. once per fold, to give an estimate of GP performance on average. Thus we have $A \times n$ GP runs on $(n-1)/n$ of the whole of the available data. However even this does not address the problem of which of evolved programs produced by this multiplicity of runs to actually use for new data. Any selection based on how well they were measured to do, is liable to be suspect of over fitting.

Sometimes cross validation is used to tune the learning parameters (e.g. c in SVMs). The worth of each parameter setting is assessed by using that particular parameter setting in n cross validation runs. Only when an optimal parameter setting has been found, is the machine learner run for real. Where there are multiple parameters to set, one can readily see a combinatorial explosion in the number of runs required. Since in most machine learning examples the whole of training data could be loaded into the GPU, running this multiplicity of training operations on the GPU, could be highly attractive.

7.3 FLOP ratings and Speed Up

Measurements show each 256k unit takes on average 883mS. Adding I/O time (27mS+6mS) and scheduling overhead (15mS) to estimated computation time (154mS) leaves 680mS, which, if we include contention time, comes close to our estimate of time taken to read the 500MB of training data, cf. Section 7.1.

Running $267\,10^6$ floating point instructions in 885mS gives the GPU an estimated 30 GFLOPS rating for this application in practise. In parallel computing, it is common for performance to rise linearly with number of processing elements initially but for the increase to fall off as more processing elements are added. It appears the nVidia card is unable to keep all 128 stream processors busy all the time.

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 [Langdon and Banzhaf, 2008] 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, 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 [Langdon and Banzhaf, 2008]. The GPU interpreter's performance on a number of problems has been in the region $\frac{1}{2}$ to 1 giga GPops. In contrast the performance of compiled GPs on GPUs has varied widely. E.g. with number of training examples and program size.

7.4 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, 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 16.

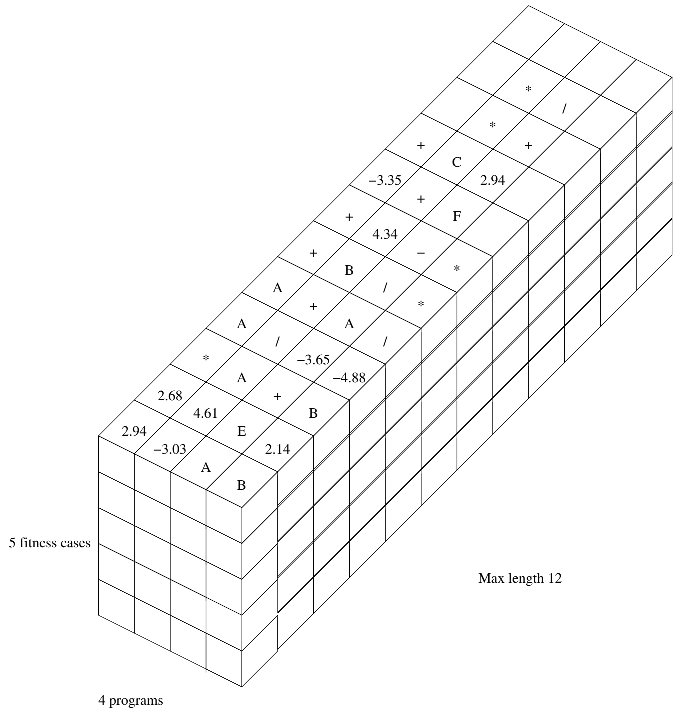

Fig. 16 Evaluating a GP population of four individuals each on the same five fitness cases. There are upto $4 \times 5 \times 12$ GP operations to be performed by, in principle, 240 GPU threads. Each cube needs the opcode to be interpreted, the fitness test case (program inputs) and the previous state of the program (i.e. the stack).

In our implementation (Section 5) the computational cube is sliced vertically (Figure 16) 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 loops along the length of the program and process all the fitness cases for that program. We belive 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 program 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 and 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 (i.e. is read only).

It appears that efficient use of current GPUs requires thousands of active threads. 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.

7.5 GPU Power Requirements

Although obvious after-the-fact, it came as surprised just how much power the GPU would take and the consequent replacement of both the power supply and enclosure which were needed to re-equip a modern PC with an nVidia GPU.

nVidia's figure of 145 watts per card appears accurate. A figure of 200W per PC chassis is a reasonable estimate for electricity bills and air conditioning. Nevertheless you must ensure your PC's power supply is able to meet the peak demand of your GPU and all the other PC components. For the GeForce 8800 GTX, nVidia suggest the PC's power supply must be at least 450W.

7.6 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 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.

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 back 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. However to some extend these might be address by Google's PeakStream, which has some simularities with RapidMind, but "is the first platform to provide profiling and debugging support" [Owens *et al.*, 2008].

For GPU manufactures GPGPU remains an add-on to their principle 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 [Owens *et al.*, 2008]. 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"[2].

7.7 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.

7.8 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 [Langdon and Banzhaf, 2008]. Whereas `random-numbers/gpu_park-miller.tar.gz` is for generating random numbers [Langdon, 2007b].

8 Conclusions

We have taken a large GeneChip breast cancer biopsy dataset with more than a million inputs to demonstrate a successful soft computing 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 soft computing 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.*, 2005] tried support vector machines and k nearest neighbour but say diagonal linear discriminant analysis worked better for them. [Miller *et al.*, 2005] used DLDA to construct a non-linear model with more than 704 data items per patient. The non-linear model

[2] Experiences with gpu Computing, 2007

evolved by genetic programming uses only three. It has been demonstrated on a separated verification dataset. As Figure 15 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 Derek Foster, John Owens and Lance Miller.

References

[Banzhaf *et al.*, 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[Barrett *et al.*, 2007] 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), January 2007.

[Charalambous *et al.*, 2005] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Advances in Informatics, 10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece, November 11-13, 2005, Proceedings*, pages 415–425, 2005.

[Chitty, 2007] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, 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.

[Ebner *et al.*, 2005] 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*, volume 3447 of *Lecture Notes in Computer Science*, pages 261–270, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[Fan *et al.*, 2004] 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.

[Feller, 1957] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley and Sons, New York, 2 edition, 1957.

[Fernando, 2004] Randy Fernando. GPGPU: general general-purpose purpose computation on GPUs. NVIDIA Developer Technology Group, 2004. Slides.

[Fok *et al.*, 2007] 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.

[Gobron *et al.*, 2007] Stephane Gobron, Francois Devillard, and Bernard Heit. Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, 2007. Online First.

[Harding and Banzhaf, 2007a] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101, Valencia, Spain, 11 - 13 April 2007. Springer.

[Harding and Banzhaf, 2007b] 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 Computer Society.

[Harding *et al.*, 2007] Simon L. Harding, Julian F. Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, 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.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Langdon and Banzhaf, 2008] A SIMD interpreter for genetic programming on GPU graphics cards. In *EuroGP, LNCS*, Naples, 26-28 March 2008. Springer. Forthcoming.

[Langdon and Barrett, 2004] 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.

[Langdon and Buxton, 2004] W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257, September 2004.

[Langdon and Poli, 2002] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[Langdon *et al.*, 2007a] 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.

[Langdon *et al.*,] 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. In preparation.

[Langdon, 1998] William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, Boston, 1998.

[Langdon, 2007a] 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.

[Langdon, 2007b] W. B. Langdon. PRNG random numbers on GPU. Technical Report CES-477, Computing and Electronic Systems, University of Essex, Colchester, UK, 29 November 2007.

[Lindblad *et al.*, 2002] Fredrik Lindblad, Peter Nordin, and Krister Wolff. Evolving 3D model interpretation of images using graphics hardware. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 225–230. IEEE Press, 2002.

[Liu *et al.*, 2006] 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*, pages 8–, 25-29 April 2006.

[Loviscach and Meyer-Spradow, 2003] 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.

[Luo *et al.*, 2005] 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.

[Miller *et al.*, 2005] 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.

[Moore, 1965] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[NVIDIA, 2006] NVIDIA GeForce 8800 GPU architecture overview. Technical Brief TB-02787-001_v0.9, Nvidia Corporation, November 2006.

[NVIDIA, 2007] NVIDIA CUDA compute unified device architecture, programming guide. Technical Report version 0.8, NVIDIA, 12 Feb 2007.

[Owens *et al.*, 2007] John D. Owens, David, 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.

[Owens *et al.*, 2008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), May 2008.

[Pawitan *et al.*, 2005] 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.

[Price, 1970] George R. Price. Selection and covariance. *Nature*, 227, August 1:520–521, 1970.

[Reggia *et al.*, 2006] 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.

[Rys, 2006] Rys. NVIDIA G80: Architecture and GPU analysis, 8 Nov 2006. Last updated: 25th Apr 2007.

[Samsung, 2007] Samsung. Graphics memory product guide. General information, Memory Division, Jan 2007.

[Schatz *et al.*,] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474. Published 10 December 2007.

[Stamatakis, 2006] A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, Nov 1 2006.

[Upton and Cook, 2001] Graham J. G. Upton and Ian Cook. *Introducing Statistics*. Oxford University Press, 2nd edition, 2001.

[Wu *et al.*, 2004] 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.

[Yu *et al.*, 2007] 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.

[Zipf, 1949] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press Inc., 1949.