Creating and Debugging Performance CUDA C

W. B. Langdon

Abstract Various practical ways of testing, locating and removing bugs in parallel general-purpose computation on graphics hardware GPGPU applications are described. Some of these are generic whilst other relate directly to stochastic bioinspired techniques, such as genetic programming. We pass on software engineering lessons learnt during CUDA C programming and ways to obtain high performance from nVidia GPU and Tesla cards including examples of both successful and less successful recent applications.

Key words: C programming, GPU, GPGPU, GPPPU, parallel computing, computer game hardware, graphics controller, parallel computing, rcs, randomised search

1 Introduction

The absence of sustained increases in computer clock speed which characterised the second half of the twenty century is starting to force even consumer mass-market applications to consider parallel hardware. The availability of cheap high speed networks makes loosely linked CPUs, in either Beowulf, grid or cloud based clusters attractive. Even more so since they run operating systems and programming development environments which are familiar to most programmers. However their performance and cost advantages lie mostly in spreading overheads (e.g. space and power) across multiple CPUs. In contrast, in theory, a single high end graphics card (GPU) can provide similar computing power and indications are that GPU performance increases will continue to follow Moore's law [24] for some years. The com-

W. B. Langdon
CREST, Computer Science, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK
e-mail: W.Langdon@cs.ucl.ac.uk

Preprint Parallel Architectures and Bioinspired Algorithms. F. Fernandez de Vega, J.I. Hidalgo Perez and J. Lanchares (eds.), *Handbook of Metaheuristics*, Studies in Computational Intelligence 415, Chapter 1, pp 7-50. DOI 10.1007/978-3-642-28789-3_2 Springer 2012

petitive home computer games market has driven and paid for GPU development. For example, nVidia has sold hundreds of millions of CUDA compatible cards [8]. Engineers and scientists have taken advantage of this cheap and accessible computer power to run parallel computing. nVidia is now actively encouraging them by marketing GPUs dedicated to computation rather than graphics. Indeed the field of general purpose computation on graphics hardware GPGPU has been established [26].

The next section will give a brief summary of a few recent successful Bioinspired applications running on GPUs or nVidia's Tesla cards. Also, to illustrate there are pitfalls, we also include one less successful GPGPU application.

I shall assume the reader is already familiar with nVidia's parallel computing architecture, CUDA. Nonetheless Section 3 gives a quick introduction to it. Section 4 gives some ideas on how to produced reasonably fast GPGPU applications. In practice this always requires interaction between implementing "improvements" and measuring your software's performance to see if they really did have the desired effect (speeding up your code). Section 5 describes practical ways to measure performance.

There are many documents and tutorials on programming graphics hardware for general purpose computing. Mostly they are concerned with perfect high performance code. Most software engineering effort is not about writing code but about testing it, debugging it, etc., etc. Development of GPGPU software remains an art, often at the edge of feasibility. Testing and debugging are key to any software development but little has been published about getting non-trivial CUDA applications to work.

Although tools are improving, we concentrate upon how debugging is done for real. Many of the lessons are general. However the examples use nVidia's GPUs with their CUDA C compiler, nvcc, and some examples assume the reader is familiar with the Unix operating system. Section 6 describes coding techniques to aid debugging. Section 7 describes testing CUDA C applications, whilst Section 8 describes some bugs, the techniques used to find them and how they were fixed.

This is not a general tutorial on CUDA, however the last two sections give practical advice for when you get started (Section 9) and some ideas for where to look for help if you hit problems and discuss alternative approaches (Section 10).

2 GPGPU Bioinspired Algorithms

For a long time bioinspired algorithms were limited by the need to be sparing in their use of computer resources. As time has progressed computer power has increased enormously and so more and more realistic models of nature have been applied. Many of the natural phenomena which have inspired computer scientists concern multiple agents, each of which has to be simulated. For example, groups of nerve cells, swarms of insects, populations of plants or animals and diverse antibodies. Typically each agent is more-or-less independent and to some extent can be simulated independently of the others. At present each simulation is still often done one

after another on a single computer. Since such simulations need a lot of computer time, this has tended to limit: the size of neural networks, the size of swarms, the number of simulated antibodies and the number of individuals in simulated populations. However in almost all cases, where parallel computers are available, the simulations can readily be run in parallel (rather than sequentially). The ease of with which this can be done has lead to many bioinspired algorithms being classified as "embarrassingly parallel" [23, p182]. Recently there has been considerable interest in using graphics hardware (GPUs) which readily provide cheap parallel hardware. Even a humble laptop can contain a low cost but powerful GPU.

Artificial neural networks come in a variety of flavours. We shall only discuss two. Perhaps the most realistic and hence the most computationally demanding are known as spiking neural networks. Whilst many flavours of ANN represent nerve cell activity as a continuous valued activation level, spiking networks represent nerve synapse activity as individual spikes. Given the computational complexity of even approximate chemical/electrical models of synapses, it is not surprising that the computational power of GPUs have been harnessed by several research teams. Yudanov *et al.* [36] showed fairly realistic (IZ) models of a few thousand neurons could be run in real-time by using CUDA and an nVidia GTX 260 GPU. A rather different approach is used by self organising maps (SOMs) or Kohonen networks. These can be thought of as unsupervised or clustering techniques which after multiple training periods learn to group similar concepts. Prabhu [28] used Microsoft's Accelerator GPGPU tool to get substantial speed increases from what is now modest hardware (an nVidia GeForce 6150 Go).

Some of the first uses of GPUs in evolutionary algorithms used them for graphics processing. This is closer to the original purpose of graphics hardware, nevertheless Ebner *et al.* [5] show genetic programming could evolve GPU code (vertex and pixel shaders written in Cg [6]) to generate images. However Fok *et al.* [7] were the first to implement a general purpose evolutionary algorithm on a GPU. They showed a complete evolutionary algorithm, including population mutation (but not crossover) and selection, as well as fitness evaluation running on an nVidia GeForce 6800 Ultra and obtained substantial speedups on a number of benchmarks with populations of several thousands. They also used the GPU to visualise their evolving populations. (Some animations of distributed genetic programming populations evolving under crossover and selection [21] can be found via http://www.cs.ucl.ac.uk/staff/W.Langdon/gp_on_gpu.html.) Harding was the first to show general purpose genetic programming running on GPUs [10]. Harding has considered a number of approaches however mostly he has required populations of GP individuals to be compiled [11]. Since the nVidia compiler is designed to optimise the speed of the GPU code it generates, rather than its own run time, it is often faster to interpret GP code rather than compile it [21]. Indeed the fastest single computer GP system uses a parallel GPU interpreter [17].

Bioinformatics contains many computationally demanding problems. Many of these are naturally parallel and so bioinformaticians are increasingly using GPUs. Restricting ourselves to bioinspired algorithms, there are several examples. For example in [19] we used an interpreted GP system built on RapidMind software run-

ning on an nVidia 8800 GTX to datamine human breast cancer biopsys to predict survival following surgery. Using a cascade of populations containing 5 million programs, a small intelligible model was distilled from noisy Affymetrix HG-U133A and HG-U133B GeneChip gene activity measurements. Whilst in [15] we used GP and public datasets to model factors influencing noise in the GeneChip's themselves. (In [18] we made a start at looking at automatic generation of GPU code.) Sinnott-Armstrong *et al.* have twice won the GPU competition at the GECCO conference for innovative uses of GPUs. In 2010 for a GPU based artificial immune system (AIS) [32] and in 2009 for epistasis analysis in human genetics. Their published work includes using three nVidia GeForce 295 (a total of 6 GPUs) to datamine a dataset of 547 people each having more than half a million genetic variations (SNPs). They were looking for gene-gene interactions to help treat sporadic amyotrophic lateral sclerosis (ALS) [9].

Rieffel *et al.* [31] showed an nVidia 9800GT could be used to evolve movement in a soft robot. The target pneumatic robot was simulated using PhysX. Such a soft bodied robot requires even more computational power than simulating a rigid robot. Realism was further enhanced by evolving a spiking neural network controller for the robot. As computer games continue to demand increased realism, dedicated "physics engines" (PPUs) will be used to offload from the CPU simulations of the physics of games, e.g. rock falls, in the same way that dedicated graphics processors (GPUs) are used now to offload graphics processing from the CPU. It is anticipated that PPUs will also contain substantial computing power and that this too will be used for algorithmic computing. Thus GPPPU will become popular in the same way that GPGPU has taken off.

Particle swarm optimisation (PSO) is a successful bioinspired algorithm in which a swarm moves under the influence of a fitness function. Mussi *et al.* [25] used a PSO to locate road signs in video images. With nVidia's CUDA they showed a swarm of particles was able to locate road signs in synthetic road images. A single GeForce 8800 GT GPU was powerful enough to run their PSO system at better than real-time (up to 150 video frames/second).

In ant colony optimisation (ACO) the swarm of flying insects is replaced by a colony of ants which navigate by following chemical trails left by other ants. There are various schemes so that successful ants guide the others but ACO explicitly includes the notion of forgetting as it requires the chemical to disperse over time. This ensures the ants do not get locked into the current best trail forever. The notion of exploiting (i.e. searching near the best solution found so far) versus exploring (searching more widely) comes up repeatedly (in different guises) in search and optimisation. Zhu and Curry [37] again used CUDA this time with a GeForce GTX 280 and show it considerably sped up their ACO on a wide range of continuous optimisation benchmarks.

GPUs have even been used to speed of simulations of artificial chemistries and regulatory networks [35].

While fuzzification is perhaps not normally thought of as "bioinspired", it too has substantial parallel components. Anderson *et al.* [12, 1] were the first to show fuzzy logic running substantially faster by running it in parallel on a GPU.

Although not a bioinspired approach, it is worth considering an unsuccessful approach. It is unclear exactly why [20] failed to achieve a big speed up. It may be that the underlying "close-by-one" FCA algorithm does not have sufficient arithmetic intensity (Section 4.1). Unlike the approaches described above, its inner loop only requires one Boolean logical operation per data item, whereas in the bioinspired approaches each data item may refer to an agent whose complete lifetime many have to be simulated. I.e. typically there is a huge volume of computing per data item. Thus even though the GPU beam search approach succeeded in parallelising the work over millions of threads this did not solve the problem that each data item had to be moved but only acted upon once. This in turn suggests, at least in this application, an arithmetic intensity of 1.0 is too low to make the GPU approach attractive. We now turn to the problems of actually getting code to work and getting the best from your parallel hardware.

3 CUDA – nVidia's Compute Unified Device Architecture

Although the reader will need to be familiar with nVidia's parallel computing architecture, we start with Figure 1 which shows how a CUDA application must make a trade off between the various storage areas, parallel computation threads and how having very many threads ready to run helps keep the many computation stream processors busy and the whole application efficient.

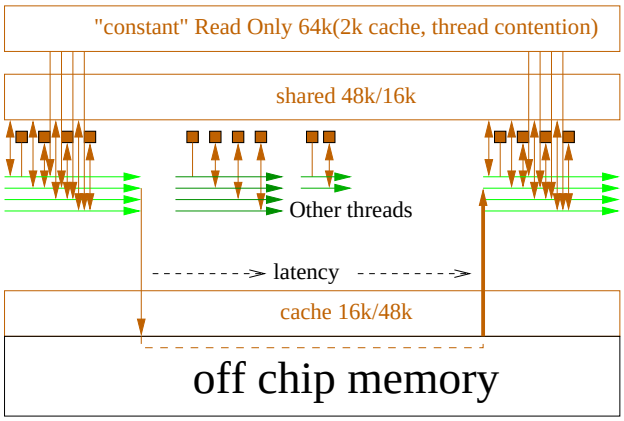

Fig. 1 nVidia CUDA mega threading (Fermi, compute level 2.0 version). Each thread in a warp (32 threads) executes the same instruction. When a program branches, some threads advance and others are held. This is known as thread divergence. Later the other branches are run to catch up. Only the 32 768 registers per block (brown □) can be accessed at full processor speed. If threads in a warp are blocked waiting for off chip memory (i.e. local, global or texture memory) another warp of threads can be started. The examples assumes the requested data are not in a cache. Shared memory and cache can be traded, either 16 Kbytes or 48 Kbytes. Constant memory appears as up to 64 Kbytes via a series of small on chip caches [3], Section 8.4.

Figure 2 emphasises the need to divide the work between many threads. As expected performance rises more or less linearly as more threads are used. However notice that this continues even when the number of threads exceed the number of processing elements. While application and GPU specific, a rule of thumb suggests maximum performance needs at least 10 threads per stream processing core.

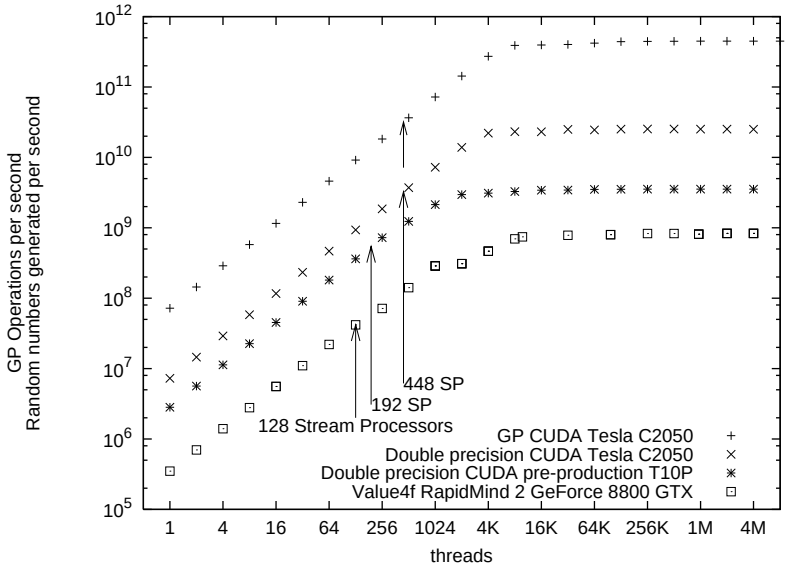

Fig. 2 Speed of genetic programming interpreter [17] and Park-Miller random numbers [16] (excluding host-GPU transfer time) versus number of parallel threads used on a range of nVidia GPUs. Top 3 plots refer to CUDA implementations and lowest one to RapidMind code. Code available via ftp `cs.ucl.ac.uk /genetic/gp-code/`.

4 Performance

As novice programmers we were taught that we should get the code working before we worried about performance. However typically as CUDA developers we approach the code from the other direction. Typically there is a working serial version of the application which may need porting to CUDA. Ideally we should start by planning how the code will be run in parallel. This and the next section are about designing CUDA applications for performance, whilst Sections 4.2–4.4 deal with what happens when you try to run your initial design on your GPU and Section 5 describe some practical ways to locate and fix performance problems when pure design collides with real GPU hardware and software.

A high performance design will need to consider how many threads are to be used and how they are to be grouped into blocks. (A block of threads all execute the same kernel code on the same multiprocessor. They can pass data rapidly between themselves via shared memory, Section 6.6. High end GPUs typically have several multiprocessors, so multiple blocks of threads are needed to keep them all busy.) You will also need to consider where data will be stored, how much memory will they occupy and how and in what way memory will be accessed. In other words we should start by designing for performance. However coding a subroutine which runs on the GPU (known as a kernel) remains difficult and no software plan survives first contact with the GPU hardware. The alternative of developing prototype kernels has its attractions however getting a perfect prototype kernel is not necessarily a lot easier than coding the real kernel. In practice GPGPU software production tends to fall between the two. That is as problems arise, some can be fixed immediately, while others cause more drastic changes to the plan. These problems need not cause the wrong answer to be calculated but may be performance related or because, for a particular new work load, it is realised that some data will not fit into an available memory store. Since faulty kernels tend to give little indication of ultimate performance it becomes necessary to debug each new implementation of each new design. This is time consuming.

4.1 Performance by Design

We have the usual problem that we do want to spend ages debugging a poor design and we do not know for sure how software will perform until we have written it. This section gives some rules of thumb to consider when designing your CUDA application. These might also be illuminating when trying to tune it.

- How much of your application can be run in parallel? If it it less than 90% then stop. Even if you are able to speed up the parallel part infinitely, so that it takes no time at all, you will still only increase the whole application ten fold. This is not worth your effort.
- In Bioinspired applications the resource consuming part is the fitness evaluation. Usually the fitness of each member of the population can be run independently in parallel and so fitness evaluation is an ideal candidate for parallel computation. This has been repeatedly recognised [30, 33, 4]. Indeed the comparative ease of parallelising population based algorithms has lead to them being called "embarrassingly parallel" [23, p182].
  Recall from Figure 2, CUDA applications typically need thousands of threads to get the best of GPUs. If your network or population does not contain thousands of cells or individuals, perhaps there are aspects of each individual fitness evaluation or learning which could be run in parallel? Obviously this is application specific.
- Estimate how much computation your application will need. Express this as a fraction of your GPU's performance. Is the fraction low enough to make the GPU a viable approach? Remember nVidia's performance figures are the best

that the GPU can do and so are typically much more than your GPU kernel will get in practice.

- It is worth considering how much computation is needed per data item. I.e. the "arithmetic intensity". Often in Bioinspired algorithms we are concerned with computationally intensive tasks that most be done for every for every member of a network, swarm or population but only a few bytes are needed to represent the individual. Thus arithmetic intensity is usually high. However if only a few instructions are needed per word, arithmetic intensity should be considered carefully at the design stage. Effectively arithmetic intensity is another way of looking at the problem of communications bandwidth bottle necks.

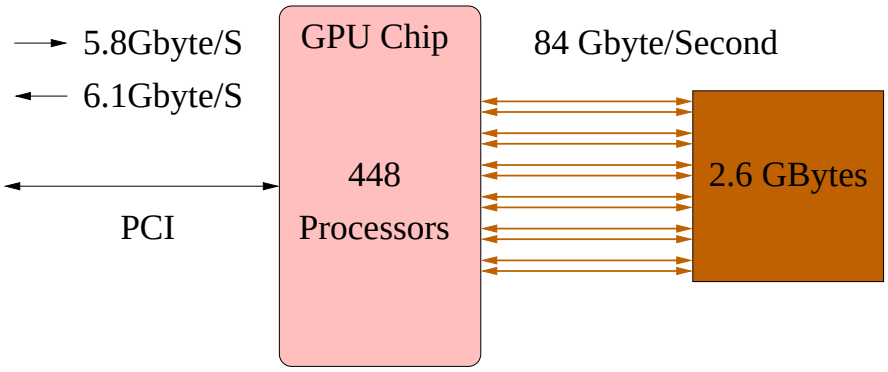

Fig. 3 Links from GPU chip to host computer via PCIe bus and to memory on the GPU board. Fermi C2050.

- From your block level design locate its bottle neck. See Figure 3. We can try and find the limiting part of your design in advance of coding by estimating:

  1. The number of bytes of data uploaded into your GPU.
  2. The number of bytes from your GPU back to your PC.
  3. How many times the PC interacts with the GPU (either to transfer data or to start kernels).
  4. Do the same for global data flows from global memory into your kernel and from it back to global memory. Assume you are going to code your kernel so it uses registers rather than local memory.
  5. In principle we could consider other bottle necks but already we are getting into detail and relying on assumptions which may turn out to be wrong.

  For GPUs connected to a traditional PC via a PCIe bus we can get a good estimate of the time taken to transfer data across the PCIe by dividing the size of the data to be passed by the advertised speed of the bus. Take the lower estimate of your bus's speed and your GPU's PCI interface speed. Remember the speed into the GPU can be different from the speed back from it. If you already have the hardware, nVidia's bandwidthTest program will report the actual speeds. (band-

widthTest will also give you the maximum speed of transfers between global memory inside your GPU.)

For PCIe transfers, with good coding, the estimates can be accurate enough. With internal transfers so much will depend upon the details: how well the threads overlap computation with fetching data, how effective are the various caches.

- Normally the ratio of the volume of PCIe data size to the size of PCIe data buffers will give the number of times the operating system has to wake up your PC code so that it can transfer data. Typically there are a few data transfers before and after each time your GPU kernel software is launched. Usually the system overheads of rescheduling your process and CUDA starting your kernel are both well under a millisecond. Nonetheless if your design requires more than a thousand PCIe I/O operations or kernel launches per second it is probably worth considering the initiation overhead.
- This should have given you an idea of where the bottle neck is in your design and if your design is feasible.

  If the bottle neck is the GPU's computational speed, then it probably makes sense to proceed. It probably means your application is sufficiently compute intensive that it needs to be run in parallel. If it still not going to be fast enough then a redesign could consider a GPU upgrade, multiple GPUs and/or traditional code optimisation.

  If the bottle neck is bandwidth, which bus is limiting? Concentrate upon the most constricting part of the design. There are two things to consider: passing less through the bottle neck and making the bottle neck wider.
- In the case of the PCIe bus, only hardware upgrades can widen the bottle neck.

  Can you compress your data in some way? Often a huge fraction of computer data is zero. Do you need to pass so many zero's? Can you pack data more tightly? Can you use char rather than int? (Will the cost of compress/decompress be excessive?)

  Does your application need so much data to be passed? Could you pass some of it to the GPU once, when the application starts, and leave it on the the GPU to be reused, rather than being passed to the GPU each time the kernel is used?

  The host–GPU bottle neck can be critical to the whole GPU approach. The above calculations have the advantage of often being feasible to estimate in advance and typically applications really do get the host–GPU advertised bandwidth. So you can get good estimates of its impact on your application at the design stage. However the PCIe bus is inflexible. Unlike internal GPU buses, there is no coding to increase its bandwidth. If your design requires 110% of the PCI's bandwidth it is not going to get more than 100%. At this point many GPU designs fail and alternatives must be considered.
- As already mentioned with internal GPU transfers design stage calculations are much trickier. Perhaps consider algorithm or design level changes, e.g. splitting kernels, spreading the work differently across different kernels. Again can the bottle neck be made wider? E.g. by larger data transfers and/or coalesced transfers. Remember advertised figures and data reported by bandwidthTest have already taken into account such optimisations.

With the much lower bandwidth of PCIe it might make sense to reduce data transfer size by compression, e.g. using 8-bit bytes rather than 32 bits. This is probably not true within the GPU. Although the full range of C types are supported by the CUDA C/C++ compiler nvcc, the hardware works on multiples of 32-bits.
It is usually better to read data once, process it (without re-reading), then write the processed data once. Although nVidia's recent Fermi architecture caches local and global data and most GPUs cache textures such caches are quickly overwhelmed by the sheer volume of data to be processed. It is better to "cache at the design stage" rather than hope the data will still be in a cache if it is needed a second time. This is unlike traditional CPU coding, where it appears to cost nothing to read and write to program variables. On the CPU it is often better to calculate intermediate results, save them, then read them back and use them again. Whereas in a GPU it might be better to recalculate rather than save–re-read.

4.2 Performance By Hacking

The previous section has talked about designing high performance GPGPU applications. Essentially the same basic idea applies whilst writing the GPGPU program code: Is performance good enough? Stop. Can performance be made good enough? If not then also stop. Identify and remove the bottle neck (e.g. by using the techniques to be described in Section 5). Before Section 5, the next section reminds us that it is not always necessary to implement everything the existing serial version does, whilst Section 4.4 considers how to include multiple GPUs into your design.

4.3 Performance By Omission

Fundamentally the best way to improve performance is not by doing things better but by doing less.

The following need not be the best example but it is real. It turned out that about 30% of the time used by a kernel was spent looking for just one case in hundreds of thousands. It was not even a particularly interesting case and it was guaranteed to be found eventually. So a 30% speed up could be made by ignoring it. Further, once it was treated as impossible other parts of the kernel could be simplified giving a further speed up. By leaving out something unimportant to the users, the code went about twice as fast.

4.4 Multiple GPUs

The processing power and capacity of single GPU cards continues to grow as new hardware is announced. However CUDA supports multiple GPUs per host PC and it may be attractive to use multiple cards. There are many twin GPU systems but three and even four card systems are also in use. (Be sure your host PC has sufficient power to support the additional hardware.)

To take advantage of multiple GPUs, parts of the host application must be run in parallel. That is the host programmer must explicitly organise the parallel operation of the PC's GPUs. CUDA does not (yet) allow you to launch a kernel across multiple GPUs or retrieve its results from multiple GPUs. Instead the programmer has to explicitly launch the kernel on each GPU. This is done in the same way as for one GPU but it does force explicit parallel multi-threaded code on the PC. Although CUDA provides some support for multi-threading of your PC code, it may be better to use your operating system's multi-threading support (e.g. the p-threads library). The standard advice is that your PC should have one CPU core per GPU card plugged into it. However the host multi-threading support should ensure 1) this is not absolutely necessary 2) your application will be able to take advance of dual or quad core CPUs without coding changes.

To avoid the surprisingly high CUDA initialisation overhead it is a good idea to start one host thread per GPU and repeatedly use it to pass data between the host and the thread's GPU and to launch kernels on its GPU. (I.e. the host threads live as long as your application itself.) Dual cards like the 295 GTX are programmed as two CUDA devices and so should have two threads (one each) in your host code. It is a good idea to record which devices your application is using.

```
cudaDeviceProp deviceProp;
cutilSafeCall( cudaSetDevice( dev ));
cutilSafeCall( cudaGetDeviceProperties(&deviceProp,0));
printf("Using CUDA device %d: \"%s\"\n",
       dev, deviceProp.name);
```

5 Measuring Performance

The main tool for measuring performance is the CUDA profiler (next section) but timing operations on the host (Section 5.2) yourself can also be useful. These give kernel level statistics but Section 5.3 will describe some ways to estimate the performance impact of program statements within your kernel. Obviously consider if there is a need for tuning and higher level aspects of tuning before getting sucked into the details (as described in Section 5.3).

5.1 CUDA Profiler

nVidia's CUDA profiling tools can be downloaded from their web pages. As with other parts of CUDA, nVidia also freely provides downloadable documentation.

There are two parts to the CUDA performance profiler. The part on the GPU which records when certain operation took place. It logs the time of host-GPU data transfers and when kernel start and when they finish. It also counts other GPU operations. E.g. it can count the number of local or global memory cache hits and misses. Finally it transfers the logged data to the host PC. The second part runs on the PC. It can control the GPU based profile logging and also display both this data and previously logged data. Unfortunately certain Linux versions of this part (known was the CUDA visual profiler) are not stable.

As may be imagined the GPU part of the profiler is limited. Its job is to monitor performance not to interfere with it. Top end GPU contain several multiprocessors, since they are identical it is assumed their workloads and hence performance will be similar, therefore only one of them is monitored. Also the GPU profiler can gather a range of statistics but not all of them simultaneously. One of the main jobs of the visual profiler is to allow you to easily specify which data should be collected. (Different GPUs support different counters. Sometimes counters are not supported on a particular GPU because the counter was introduced to monitor a particular performance bottle neck which has been removed from the new GPU.)

If you specify more counters than the GPU can manage in one go, the visual profiler automatically runs your application multiple times collecting different profile data each time and then integrating them for you. Again the number of simultaneous counters depends on which type of GPU you are using. The visual profiler has the great advantage that it knows which GPUs support which counters and which can be simultaneously active. It also provides a wide range of plots and tables. A few of the interactive menus are a bit difficult to navigate and the documentation and menu layout may be slightly out of step.

When testing stochastic algorithms, such as Monte Carlo sampling or evolutionary computation, it is much easier if your code does exactly the same thing when run again. E.g. a genetic programming system should be coded so that its use of pseudo random numbers (PRNGs) can be controlled via the command line (see Section 7.1). By telling the visual profiler to pass the same PRNG initial seed to your GP when it runs it multiple times in order to collect a number of performance indicators, you should be able to ensure that these indications are consistent with those gathered by it on other runs.

Under the Linux operating system you can also control the GPU profiler directly by using environment variables, see Table 1.

The CUDA profiler gives some performance information which could be very useful but which would be either difficult or impossible to get elsewhere (e.g. cache hits). It also gives ready access to some critical information about the code that the compiler, nvcc, generated for your kernel. E.g. the number of registers the kernel needs.

Table 1 Unix environment variable controlling CUDA profiling

Name:	Example	
CUDA_PROFILE	1	Switch on profiling
CUDA_PROFILE_CSV	0	Produce "comma separated values" suitable for importing into a spreadsheet or the CUDA visual profiler. With the value 0 a simple text file is produced.
CUDA_PROFILE_CONFIG	profile_r266a.txt	The name of a file containing instructions for the GPU profiler including which counters to enable. I suggest you start by copying CUDA_Profiler_3.0.txt from nVidia's web pages and then modifying it.
CUDA_PROFILE_LOG	profile_r266a.csv	The name of the profiler's output file. NB. the file will be overwritten if it already exists.

If using CUDA_PROFILE_LOG directly, some counters become very large and difficult to comprehend. It would probably be worth using a spread sheet or simple script to rescale counters by the "instruction" count. (E.g. divide warp_serialize count by total number of instructions.) This helps make clear which data are important. Even if a counter has a five or six digit value, after it has been normalised by dividing by the instruction count it is clear which ratios are near zero and can be ignored.

Another useful measure is to calculate the number of "instructions" your kernel is executing per microsecond. The profiler is the only convenient route to these data. On a GTX 295, the profiler says a totally compute bound kernel will run in the region of 370 instructions per microsecond. Depending upon their "compute level" and because of the arcane way in which the profiler reports "instructions" other GPUs will each have their own value. (It is a useful exercise to construct your own compute bound kernel and see what figure your GPU gives.) Your application kernels will not reach the GPU's peak rate. If they are getting more than half the peak rate congratulate yourself and stop. I have had GTX 295 kernels as disastrously low as 5 instructions per microsecond.

5.2 CUDA timing functions

CUDA's timing functions can be used to time operations. They have the advantage of using the GPU's own high resolution clock but, as the following example shows, they tend to end up with voluminous code.

```
cutilCheckError(cutCreateTimer(&hTimer));

                          ⋮

cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError( cutResetTimer(hTimer) );
```

```
cutilCheckError( cutStartTimer(hTimer) );

cutilSafeCall(
  cudaMemcpy(d_1D_in,In,In_size*sizeof(int),
             cudaMemcpyHostToDevice));

cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError(cutStopTimer(hTimer));
const double gpuTimeUp = cutGetTimerValue(hTimer);
gpuTotal += gpuTimeUp;
```

As well as the reassurance of knowing what your code is doing, using the CUDA timing routines allows easy integration of timing information with the other data about your use of the GPU. However very similar timing information is available from the CUDA profiler without coding (Section 5.1). It is often convenient to create a CUDA timing data structure (`hTimer` in the above example) at the same time as you create your CUDA buffers (Section 6.1.3).

Notice some CUDA calls are asynchronous. Typically, this means, on the host they start a GPU operation and then return and allow the PC code to continue operation even though the GPU operation has only been started and will finish some time later. This allows 1) host PC and GPU operations to be overlapped and 2) the use of multiple GPUs on a single PC. However it does mean care is needed when timing operations on the PC, hence the heavy use of `cudaThreadSynchronize()` in the timing code. A common error is to omit calling `cudaThreadSynchronize()`. If it is not used `hTimer` typically gives the time taken to start an operation, e.g. the time taken to launch your kernel, rather than the time your kernel takes to run.

Except where multiple GPUs are to be used and assuming the GPU is doing the heavy computation, there is little advantage in allowing GPU and PC to operate asynchronously. This sort of parallelism is radically different from that provided by the CUDA and the GPU, it is just as error prone and hard to debug and typically offers only a modest performance advantage.

In production code you can use conditional compilation switches to disable `hTimer`. However, in practice (even when removing many `cudaThreadSynchronize()` calls) typically this will only make a marginal difference.

5.3 GPU Kernel Code Timing

Although the GPU has on chip clocks, a useful approach is to add code to your kernel and see how much longer the kernel takes. This can be quite informative but needs to be done with care. Usually it is best to ensure the new code does not change subsequent operations in any way since their timing effects could totally cancel the timing effect of your new code.

Timing operation of the kernel from the PC is subject to noise from other activities on the PC. Random noise can be averaged out but it is better to ensure the timing effect that is being measured is much bigger than the noise. (E.g. perhaps do the additional operation a thousand times rather than just once.) When adding code you must remember that nvcc is an optimising compiler. In particular this means it will try to remove code that makes no difference to the kernel's outputs. To prevent nvcc optimising away the timing code we have just added, what is often done is to make the new code calculate a result and then use an "if" to ensure the result is discarded. Perhaps the if can depend upon one of the kernel's inputs, so that nvcc cannot easily reason about it, but we ensure that the if is always false. E.g. since `in_length` should never be negative, the code "`if(in_length<0) d_out=timing_info;`" will never be executed but nvcc does not know this and so cannot remove it and so it cannot remove the calculation of `timing_info` either.

This can be a useful way of confirming which parts of your kernel are expensive. However benefits can be disappointing. Kernels that are working well usually overlap reading from global memory with computation. So even large reductions in computation time can have little reduction in total time because the I/O time is unchanged. In the worse case, the more efficient coding simply increases the idle time waiting for data held in global memory to arrive.

Of course there is also always the dilution effect of Amdahl's law. In one example a function was made thirty times faster. However even the inefficient version of the function was responsible for only a small proportion of the total time. So vastly speeding it up made only an 11% change to the speed of the whole application.

6 GPU Debugging Techniques

6.1 Defensive Programming

6.1.1 GPU Kernel Infinite Loops

The hardest problem to debug is probably when the kernel fails. Since CUDA GPUs do not have timeouts, this can mean the kernel never returns. It may lock the whole GPU. If you are using the same GPU to drive your computer's monitor, it will appear as if the whole computer has failed. It may require the computer to be restarted to reset the GPU. (Section 9.1 has some suggestions for reducing the impact of this.)

Notice not only is the result painful but you may get no indication of what has gone wrong or where. Further it is quite likely that it will happen again.

Given this is one of the worse bugs it is probably worth some defensive programming. A useful approach, particularly during development is to write a description of every kernel launch *before* it is started to a log file. (It may also be necessary to flush the log file before asking CUDA to start the kernel.) Conditional compilation

switches could be used to remove it from production code. When a kernel fails, or is interrupted, the last thing in the log should give you an indication of where the error lies. I tend to write not just the kernel's name but also the thread grid dimensions, block size, number of bytes of shared memory requested and parameters to the kernel. In the case of arrays, I write the volume of the data in the array, rather than all it's values. This is probably unnecessary for most bugs but it is easier to be consistent and it is impossible to be sure in advance which information in the log will be useful.

```
printf("kernel_name<<<%d,%d,%d>>>(%d,%d,%d,<%d>,<%d>,<%d>:",
       grid_size, block_size, shared_size,
       height,width,len,
       len*sizeof(int),
       len*width*sizeof(unsigned int),
       len*sizeof(int));
printf("<%d>,<%d><%d>)\n", //outputs
       len*width*sizeof(unsigned int),
       len*width*sizeof(unsigned int),
       3*sizeof(int));

kernel_name<<<grid_size, block_size, shared_size>>>
(height,width,len,d_in,d_a,d_y,d_out1,d_out2,d_status);
cutilCheckMsg("kernel_name execution failed.\n");
```

Typically kernels have a main thread loop which allows you to change the block and/or grid size without recoding or recompiling but still ensures it steps through all of the input array. (See second example in this section.) Given the CUDA parallel processing architecture, it is seldom necessary to have other loops in kernel code. Similarly recursion is seldom used (in fact it is has only recently become possible). Thus it should not be too difficult to track all (potential) loops in your code and make absolutely sure that they terminate. A recent bug will show how this was used and proved very helpful.

```
int id   = -1;  //found it
int free = -1;  //free slot
int i    =  hash(value,Nvalue); //start search at i
int loop =  0;  //prevent looping forever
do {
  if(s_value[i]==value) {id  =i; break;}
  if(s_value[i]==    0) {free=i; break;}
  i++; if(i>=Nvalue) i=0; //Goto beginning of s_value
} while(loop++ < Nvalue);
if(id == -1 && free <0 ) Error(0x99960000,Nvalue);
}
```

The `do while` loop searches `s_value` for `value`. On successful exit `id` will indicate where it is. If it has not already been stored, `free` will say where it can be

stored. hash() is only used to speed the search. If things were working as expected, the `while` loop could have been coded as `while(1)` However we know in advance the maximum number of times the loop should go round. (It is Nvalue, the size of the array `s_value`.) Therefore we can use `while(loop++ < Nvalue)` to force the loop to terminate, knowing it will catch indefinite loop errors but not abort the loop too soon. In fact the two `break` statements are the only legitimate ways of exiting the loop. An older programmer may have used `goto`, which might have simplified the last line.

The last line, checks the loop terminated as expected and if not reports an error. If, in some unexpected future run, we have more examples of `value` than we have space in `s_value` the error could arise legitimately. If we had not provided a check on `loop++`, this would cause the kernel to lock up the GPU and hence the monitor would freeze.

In an actual bug, hash() returned a very negative value. The search loop terminated and the problem was reported by via Error() on the last line.
The second example bug arose in the following loop structure which is based on CUDA's SDK examples:

```
int tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
int threadN = MUL(blockDim.x, gridDim.x);
for(int i = tid; i < length; i += threadN) {
  ...
}
```

CUDA should provide legal values for blockDim.x, gridDim.x, threadIdx.x and so the loop *should* always terminate and so is commonly not protected. However in one kernel it was desired to dedicate different blocks of threads within the grid to different parts of the calculation and a bug was introduced when the second `MUL` was changed. This lead to `threadN` being set to zero and the kernel running until manually aborted. Of course, after the fact, it is also possible to add code to detect indefinite loop errors in this construct too. However, as errors here are not expected, it it seldom done.

6.1.2 CUDA Kernel Launch Failure

When launching a kernel always follow `kernel_name<<<...>>>` with `cutilCheckMsg("kernel_name execution failed.\n");` This will ensure you know which was the first kernel to fail. Normally the string you supply to `cutilCheckMsg()` is fixed. However it need not be. If, for example, you start your kernel in a loop, you could use sprintf() to make the string you pass to `cutilCheckMsg()` include the loop index.

Since there is seldom a good reason for allowing the code to continue passed an error, you should wrap all host calls to CUDA routines with cutilSafeCall() or cutilCheckError(). See the examples in CUDA's SDK routines.

Sometimes the error message supplied by CUDA can be helpful but often it is very general. E.g. `cutilCheckMsg cudaThreadSynchronize error: kernel_name execution failed in file <kernel.cu>, line 1455 : unspecified launch failure`. This error message says there is an error somewhere. It is probably related to a particular kernel launch and the message tells you where in your source code to start looking. Sometimes starting your program via `cuda-memcheck --continue` can give additional information perhaps confirming the bug is an addressing error within the kernel.

The information you have written to the log file can sometimes be very helpful. For example did you tell CUDA to launch a kernel with zero threads per block? Was the grid size more than 65535? Or did you tell it to use more shared memory than the GPU has? Sometimes index array out of bounds errors inside the kernel can be reported as `unspecified launch failure`.

6.1.3 GPU Device Buffers

High end GPUs typically have a lot of high speed "graphics" memory. PCs with their lower performance typically have lower speed memory. Since it is cheaper, host computers typically have more memory than GPUs.

A good CUDA coding convention is to allocate a buffer in the PC's memory for each buffer in the GPU's global memory. The host and device buffers are of the same type and same size.

Given the high initial overhead on both starting kernels and transferring buffers, GPGPU applications tend to have a few large buffers. Even a complex application is unlikely to have more than a dozen PC/GPU buffer pairs.

It turns out that allocating CUDA device buffers has a very high overhead, so typically they and their shadow PC buffer are allocated once when the application starts and reused many times. I suggest you adopt a naming convention which makes it is obvious which buffers are on the CPU and which on the GPU and which shadows which.

Using `cudaMalloc` to create GPU global memory buffers:

```
cutilSafeCall( cudaMalloc(
    (void**)&d_buffer, buff_size*sizeof(int) ));
```

As with other C code, when debugging it is a very good idea to set all variables into a defined state before using them. In the case of GPU buffers this can be done with `cudaMemset()`:

```
cutilSafeCall( cudaMemset(
    d_buffer, 0, buff_size*sizeof(int) ));
```

`cudaMemset()` is fine for use whilst debugging. Often applications can be written which do not require large buffers to be cleared. However if yours does, it may be slightly more efficient to use GPU kernel code to initialise a large array in global memory, rather than to use `cudaMemset()`.

6.1.4 Host PC Buffers – Non-Paged "Pinned Memory"

The host buffers on the PC can be created in the usual C or C++ ways however it is more efficient to ensure that they are locked into the PC's memory rather than being pageable. (This avoids the GPU driver copying the data twice.) Normally this effectively doubles the transfer speed to and from the GPU. However in one case, switching to non-paged memory gave a 27 fold speed up. `cudaMallocHost` provides a convenient way of allocating "pinned memory":

```
printf("Allocating non-paged host memory\n");
cutilSafeCall( cudaMallocHost(
    (void**)&Buffer, buff_size*sizeof(int) ));
```

Even though "pinned memory" is in host RAM, some versions of the GNU GDB debugger cannot access it. Instead it produces error messages confusingly similar to those it produces if you try and access the GPU's memory via GDB.

6.1.5 Debugging GPU Device Buffers

GPU device buffers are often huge, typically containing thousands or millions of data. Too many to check all individually. It is not always easy to construct small test examples which highlight particular bugs. Indeed the bug may only manifest itself with larger data sets.

Sometimes the GNU GDB debugger can deal with whole arrays. However the ability to interactively display arrays, even in an intelligible screen format, rapidly becomes less useful as the arrays get bigger. The CUDA programming style tends to mean pointers to buffers are passed around the code and (even without the problem of "pinned memory" mentioned in the previous section) GDB rapidly loses the sense of data as being an array and human access is only via pointers and offsets. Interactive access via pointers and offsets is tedious and hence error prone.

What has proved useful is creating a suite of host functions, one per data type, which simply dump an entire GPU buffer into a disk file in human readable format. (Depending upon your application, you may also want functions to load data from disk.) The files mean the whole of a buffer can be rapidly inspected by eye. They can also be subjected to semi-automatic sanity checks. Such checks might be informal or only true in particular circumstances. E.g. you might want to double check that there are exactly 273 non-zero elements in the buffer. It can be easier to apply such variable checks outside your application code.

Notice the following debug code does not use the host/GPU shadow but creates it's own dedicated buffer and reads from the GPU. The idea is to avoid cross talk between the debug code and the code being debugged. Also we avoid making assumptions about what we thought we had put into the GPU and instead read what is actually there.

```
if(debugging) {
  my_type* in = new my_type[size];
  cutilSafeCall(
    cudaMemcpy(in, d_in, size*sizeof(my_type),
               cudaMemcpyDeviceToHost) );
  print_my_type("In.txt",in,size);
  delete[] in;
}
```

Each of the print routines sends each datum to an output file one per line. The human readable format of each data item is as simple and as clear as possible.

```
void print_my_type(const char* fname,
               const my_type buff[],
               const int length) {
  FILE* ifd = open_(fname);
  for(int i=0;i<length;i++) {
    fprintf(ifd, "%8d %g\n",
            buff[i].timestamp,buff[i].pressure);
  }
  fclose(ifd);
}
```

The idea is to have a file for each GPU buffer. It may be that during a particular debug/test cycle not all of them will be needed.

When you have a working version of your application these files become valuable in their own right. The assumption is, since you know your application is working, then the contents of the GPU buffers and hence these files is also correct. Therefore when we produce a new version of the code (e.g. to tune it's performance or port it to different hardware) we can readily re-run the new code on the old input and use these files to confirm that the contents of the GPU buffers are the same as they were before.

The idea of `open_()` is to automatically give each file a name which depends on the version of the kernel we are running. `open_()` uses a `Version` macro containing the source file kernel.cu's version number: `#define Version "Revision: 1.266a "`. Thus GPU buffer `d_in` will be automatically saved in file `In.266a`

```
FILE* open_(const char* fin) {
  FILE* ifd;
  //replace fin type by Version
  char fname[80];
  char* p = strrchr(fin,'.');
  const int len = p-fin+1;   assert(len >0 && len <80);
  strncpy(fname,fin,len);
  char buf[80], buf2[80];
  strncpy(buf,Version,79);
```

```
    char* p2 = strrchr(buf,'.')+1;
    {const int len2 = p2-buf;  assert(len2>0 && len2<80);}
    strcpy(buf2,p2);
    char* p3 = strrchr(buf2,' ');
    {const int len3 = p3-buf2; assert(len3>0 && len3<80);}
    *p3 = '\0';
    strcpy(fname+len,buf2);

    ifd = fopen(fname, "w");
    if(ifd==NULL) {
      printf("Failed to fopen %s w\n",fname);
      exit(1);
    }
    printf("Printing to %s\n",fname);
    return ifd;
}
```

When either debugging or conducting regression tests there are at least two reasons why simple comparisons between two versions of a file might fail. 1) Your code has changed and the effect of the change on the GPU is being entirely correctly reflected by differences in the files. 2) The code is not deterministic but the details of it's output (even when correct) depends upon the exact order in which parallel operations appear in the files. Thus running the program twice need not produce identical files (see Section 8.5). This makes the whole of testing and debugging much more complicated and so nondeterminism should be avoided. The increased possibility of creating a successful application may mean it is better to have deterministic code, even if it is slower.

If nondeterminism or potential future code changes mean that the order of data inside the GPU might change it is better to avoid saving line numbers, indexes, time stamps, etc., in the file. If blocks of data can legitimately move in the buffer, utilities like diff can often report this as a simple move of data about the file. Another approach is to sort the two files and then compare the sorted files. If data have simply been rearranged, the two sorted files will be identical.

It is now possible to generate your own debug text from inside your kernel using `printf()`, however my preference is still to use the "dump whole GPU buffer to disk file" approach. It is less intrusive to the code you are trying to debug and requires no change to the kernel code at all. Although, with very large files, it can have an impact on performance, the impact is readily isolated when inspecting either your own timing log or the CUDA profiler output. As mentioned above, it gives easy access to the whole of large data structures and typically integrates well with regression testing. With modest kernels, studying their source code, inputs and outputs is often sufficient to quickly locate problems. Perhaps as kernels grow in complexity, `printf()`'s ability to report kernel internals will be more important.

6.2 Debugging GPU Bioinspired Algorithms

Whilst some aspects of getting Bioinspired algorithms to work on GPUs will be specific to the algorithm many of the ideas I have described will also apply. However a particular class are the stochastic search algorithms, such as evolutionary computation. I have already mentioned the need to control their use of randomness (see Section 5.1, and also Section 7.1). Some algorithm specific techniques developed for serial versions can also be very useful when run on GPUs. For example in genetic programming [27, chpt. 13] it is recommended to test your implementation by seeding the population with one or more individuals of known fitness. E.g. create a GP seed individual which passes no tests or a "perfect" individual which passes all tests. Then verify that the fitness of these individuals, when calculated by your code on the GPU, is 0% and 100% respectively. (This is similar to the idea, described in Section 7.1, of testing by ensuring the GPU implementation gives the same results as a serial version of the algorithm.)

Another way of verifying your algorithm is to try it on a published "benchmark", e.g. 6-Mux [14] and ones max [29]. Your GPU code should give the same answer. With randomised algorithms like genetic algorithms (GAs) it will be necessary to do many runs and compare the mean number of fitness evaluations or other statistic to be sure that minor differences can really be put down to random chance fluctuations.

However this raises the awkward question of what to do if your GPU really does generate different answers. Unfortunately some benchmarks are not well described. This suggests you use well established simple benchmarks with published results from a range of authors. Also consider problems which you have worked on and have (debugged) serial code implementations. Can you compare your new GPU code against them? Of course for such comparisons to make sense, your GPU algorithm must be doing the same as the serial algorithm.

One oft repeated discovery is that GAs with distributed populations tend to do better than those with a single monolithic (panmictic) population. This is due the populations searching in different ways. For example, while it may make perfect sense for your GPU GA to contain several isolated populations each stored in isolated shared memory, we would not expect it to behave the same as a GA with the same combined population size, which every generation allowed complete mixing of all members of the population. These are different algorithms. One may work better than the other but it probably does not make sense to compare them when looking for bugs.

6.3 Your First GPU Kernel

The following way of debugging GPU kernels was suggested by Gernot Ziegler of nVidia. The idea is not to have the kernel do anything but simply prove to yourself that it can read it's inputs and send output to the right place.

When you come to debug more complex kernels, these steps may still be important. 1) Does the input data reach the kernel? This may be particularly important if the data were created by another kernel. 2) Does output leave the kernel? 3) Do the various threads put the data in the correct places? Are their values correct?

Lets start with a simple CUDA kernel which checks "does the GPU sends data to the right places?"

```
int tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
int threadN = MUL(blockDim.x, gridDim.x);
for(unsigned int t = tid; t < LEN; t += threadN) {
  d_1D_out[t] = threadIdx.x;
}
```

You will need fair amount of code on the PC to support even this simple kernel. See the examples in the CUDA SDK sources directories. These directories include compilation command scripts. Remember to include code to check the kernel really is working. Once satisfied with your first kernel, inject a fault into it [22]. Did it fail in the way you expected? Did your error checking code catch the error, and handle it in an appropriate way? Did your revision control system (Section 7.3) allow you to recover your working version reliably and correctly?

Ok so now try both input and output. E.g. replace the contents of the loop with:

```
d_1D_out[t] = 1 + d_1D_in[t];
```

What values did you put in `d_1D_in`? Did you get the expected values in `d_1D_out`? Did you get the expected values in `d_1D_in`? How fast is it? How does the speed vary: if the arrays are bigger or smaller? if the arrays are types other than integer? what happens with different numbers of threads per block? (Remember Figure 1.) what happens if the grid size and dimensions are changed? what happens if adding one is replaced by a more demanding calculation? (Remember to check the answers the GPU gives.) What do you expect to happen if you run your kernel on a different GPU?

6.4 GPU Coding Style

Often GPGPU applications contain only one or two kernels. Less than half a dozen is very typical. It is common to design them to be small (e.g. between 10 and 100 lines).

Earlier nVidia GPU's had quite limited numbers of registers, however current Fermi designs include 32 768 registers per multi-processor. The registers are used by every thread active on the multi-processor. Thus a kernel which used 100 registers could use at most 327 threads per block. With small purpose written kernels, the number of registers is no longer as big a factor as it used to be on earlier GPUs with fewer registers. However if large serial functions are converted into large kernels the number of registers could be an issue. The nvcc compiler has various options to

allow fewer registers to be used and/or to "spill over" registers into local memory. However remember local memory is actually stored off-chip and even with caches it has the same performance impact as using global memory.

6.5 GPU __device__ functions

The CUDA C compiler, nvcc, efficiently supports functions on the GPU. Since nvcc inlines function calls, there is no overhead in calling them but the GPU code is not reduced by being able to use common subroutines to implement functionality needed in multiple places. Nonetheless nvcc implements them as full C functions and so one gets the normal development advantages of data scoping and variable arguments. Indeed there is no parameter passing overhead. Nonetheless one must always remember that the functions are to be run by many threads in parallel.

A coding problem, unique to parallel computing, is that the programmer must keep track of which threads are really going to execute the function. The following shows how this (and programmer error) created a bug.

Suppose a GPU function assembles an answer in GPU shared memory, it then wishes to send the answer to the host PC. It must first write it to global memory. In a kernel the following loop might be used.

```
for(int i=threadIdx.x; i<Nvalue; i+=blockDim.x) {
  d_out[i] = s_value[i]; //Bug
}
```

Notice how it spreads the work evenly amongst all the threads and allows the GPUs I/O hardware to efficiently bunch together large numbers of simultaneous writes into large low overhead blocks. Even access to the shared memory `s_value` avoids the overhead of bank conflicts. Unfortunately the code may be wrong.

Worse the error lies not in the code itself but in how it is used. Even worse the error may be very subtle, with almost all data correct and only incorrect every so often, depending on exactly what data the kernel is processing. Indeed if, e.g. for performance reasons, `d_out` is not reset between kernel invocations, it's last contents may be close to the values expected.

If instead of using a function, we had placed the above code inside the kernel itself we might have spotted the error immediately.

```
for(unsigned int t = tid; t < LEN; t += threadN){
  for(int i=threadIdx.x; i<Nvalue; i+=blockDim.x) {
    d_out[i] = s_value[i]; //Missing threads bug
  }
}
```

It starts to become clear that there is a relationship between the threads in the outer loop and those in the inner loop. The inner loop assumes all `blockDim.x` threads will run it. So does the outer one. However the problem arises, because once `t`

reaches LEN the outer loop assumes it is done and effectively stops any remaining threads. Thus these threads are not available to the inner loop. This only happens in the last iteration of the outer loop, it only effects the highest numbered threads. At least it is deterministic but without studying the code and knowing the details of the parameters used to launch the kernel and the value LEN we do not know which threads will be affected. At the start of the kernel, both loops work well, however at the end some parts of `d_out` may not be updated and which ones depends on too many details.

Notice, to detect this error, it would be better to check the end of the buffer, rather than it's start. In fact the bug was picked up by noticing a regular pattern of zeros towards the end of the output file (generated using the debugging technique described in Sections 6.1.3–6.1.5).

This bug arises from parallel computing. In serial computing, once we have coded a subroutine and debugged it, we are now confident in it and only limited further checks are made. This is the case here. The code has been checked and when it is run it works. The problem arises because we think certain threads are going to run it but they do not. The code would have worked but it was never run.

To avoid the detailed consideration needed to ensure this bug does not happen, you should try to code `__device__` functions so as to avoid operations which need interaction between threads. This also has the advantage that `__syncthreads()` should not normally be needed in `__device__` functions.

6.6 nVidia GPU Shared Memory

GPU shared memory is rapid access read write on-chip memory available to blocks of kernel threads, see Figure 1. It gives CUDA it's only modifiable rapid access arrays. (Individual CUDA threads can have modifiable "local" arrays but until Fermi all local data was off chip and consequently slow. Fermi provides a cache which potentially makes read/write access to local arrays competitive with shared memory.) Shared memory can also be used as a very rapid way of passing data between computation threads in the same block. It cannot link threads from different blocks.

Shared memory is required for parallel computing "reduction" techniques (see SDK's `reduction_kernel.cu`). Whereby each thread calculates part of an answer but the whole answer is created by reducing these partial answers hierarchically into one (usually thread 0). It takes $\log_2 n$ steps to combine the answers of n threads.

There is only a small amount of shared memory and it may be quickly be exhausted. CUDA's SDK has examples (e.g. `histogram`) where data are first stored in shared memory and then results from different blocks of threads are combined. SDK's matrix manipulation examples also make heavy use of shared memory.

In kernels where data are not processed independently shared memory can be a good place to store intermediate results. E.g. When scanning and removing dupli-

cates from large arrays, multiple threads are needed to read the array rapidly but each thread needs to know which duplicates the others have found [20].

As with global, local and constant memory, there is a "best" way to arrange your threads when they access shared memory (which will of course be simultaneously). However unlike the other three types of memory the penalty for not using the best is slight and shared memory "bank conflicts" are seldom worth worrying about before the kernel is debugged. However as I have got a better understanding of how GPUs work, my kernels have used shared memory less.

It is often suggested that shared memory be used as a cache for your kernel. This can be a bit misleading. It is not worth using shared memory to buffer either input or output data whilst it is being read from or written to off chip memory. If you use `__syncthreads()` to ensure all data has arrived before you try and use them the GPU loses a large part of it's ability to overlap I/O with computing and performance falls horribly. Each thread has a number of registers. Global data can be read/written directly to/from a thread register very efficiently without using shared memory.

6.7 Error Reporting

The function Error(), mentioned in Section 6.1.1 (and in Section 8.9.2) was introduced into a kernel which was proving very hard going. It is designed to report the first error detected to the host PC, where code retrieves it and reports it to the log file. Given a parallel multithreaded environment, it need not always be clear which is the first error. The implementation of Error() does not try overly hard and the debugger must always be aware that events may be reported in an unexpected order. Even so Error() is probably more sophisticated than necessary for most kernels.

```
__device__ void Error(const int error,
                      const short int aux) {
  if(s_error==0) s_error = error | (aux & 0xffff);
}
```

In the main loop of the kernel we also have

```
if(s_error) {d_status[2] = s_error; break; }
```

Notice `d_status[2]` is shared between all the blocks of threads and so will suffer from "races". We do not take special precautions about this since: it is code that should not normally be in use, the simpler it is the easier it will be to understand and the less likely it too will have bugs in. (Debugging debug code is especially annoying[1].) However `d_status` is an array of 32 bit values, so each 32-bit word will be self consistent. Often several blocks of threads will encounter errors and `d_status[2]` will contain the first error reported by the last block of threads. When using it to assist your debugging you may need to be aware that it was not

[1] When you are in the swamp killing alligators, the thing to remember is that you are not supposed to be killing alligators; you are supposed to be draining the swamp.

necessarily the only error reported by your kernel. You will also need some code to transfer `d_status[2]` to the host PC and check it's value:

```
cutilSafeCall(cudaMemset(d_status,0,3*sizeof(int) ));
                              ⋮
cutilSafeCall(
  cudaMemcpy(Status, d_status, 3*sizeof(int),
             cudaMemcpyDeviceToHost)
);
if(Status[2]) {
  printf("ERROR reported by kernel 0x%x\n",Status[2]);
  exit(99);
}
```

In the production code it is tempting to remove Error() or similar sanity checking code (such as assert in the host code). I suggest you do not remove it from the source code. In code that is in use, there will always be another bug and what you have already developed might help you or the next programmer find it. Again conditional compilation might be a good way to disable it. However in one complex kernel, commenting out "unneeded" sanity checking code saved only 6% of it's overall execution time.

7 Testing Parallel Software

Assume new or modified code is wrong. This is particularly important with stochastic bioinspired techniques. Guided by a fitness function, there are many occasions where evolution has worked around horrendous implementation bugs. From an application point of view, this is of course a strength. If the genetic algorithm came up with a good solution, we do not care the implementation was poor. Indeed it might be argued buggy GAs are considerably cheaper to implement than perfect ones. From a scientific point of view this is less satisfactory.

If we are researching an improved stochastic search operator (e.g. a new GA crossover operator) for a particular application domain, we want to be sure that any differences are really due to the crossover operator and not due to bugs in either our GA or in the GA we are comparing against. The fact that a good solution was found, does not mean the GA code we used did what we thought it did.

7.1 Comparison with a "Gold Standard"

Many of nVidia's SDK examples, not only show how to code an example in CUDA but also include comparing the GPU's results with a traditional implementation of

the example. Can you do the same? Do you have a convenient solution to your problem (which you are confident is correct)? Can you knock up a simple (even inefficient) conventional version? This need not even be written in C, perhaps python, gawk or spreadsheet, as long as it produces correct (but non-trivial) answers. (Nonetheless remember to use your revision control system, Section 7.3.)

It is much easier to compare results if your CUDA code produces identical results to your gold standard. Insist on it. Once you get into heavy coding it is easy to assume small differences are unimportant and as data volumes ramp up larger differences can be overlooked in a mass of minor ones.

With stochastic methods use (at least during testing) deterministic sources of random numbers (PRNGs), e.g. [16]. Keep a record of the seeds used in the log file. Perhaps use the same seeds with the GPU and your gold standard code. (Do not use these seeds during production runs).

With floating point numbers the GPU will produce different answers. Decide in advance how big a difference you expect. When comparing PC and GPU results, use an automated method which will only show you unexpected differences. Consider if you should include -0, NaN, etc., as different.

7.2 Regression Testing

Be sparing in your inclusion and careful in the placement of: version numbers, date stamps and elapse times in output files. Even in correct code, these will be reported as different and you can quickly be swamped by uninteresting differences, which may (particularly if mixed with other data) conceal important differences.

7.3 Software Version Control

You will create multiple version of your source code. At some point you will insert a fault into it and want to revert to an earlier version. You will want to be able to compare different versions. You should start using a convenient version control system when you start coding.

Having said that the best way to use it will depend on you. It is easy to delay saving a version whilst coding/debugging is going well and then find at the end of the day (usually when tired) that an error has been made and you do not want to throw away all the nice code written since the last time you checked kernel.cu into your revision control system (rcs) before the error was made. However you did not spot the error as it was made and either your editor will not allow you to undo the changes or you need to undo so many individual character changes that that it itself becomes tedious and error prone. On the other hand it is possible to check-in source code too often so the rcs history log becomes a sequence of meaningless messages

of the type “changed function xxx: still not working”. My preference is for too often. After all saving a revision will take less time than compiling it.

8 GPU Bugs

The following sections described a few examples of real GPGPU bugs, how they were found and how they were fixed.

8.1 Not all threads available

Another manifestation of the problem described in Section 6.5 occurred when a function was called inside conditional code within the main kernel.

```
if(data) {
  ... lookup data ...
  if(missing) save_data(data,...);
}
```

It is obvious from this code that only certain threads (those for which `data` is both non-zero and has not already been saved) will call save_data(). However this is not so clear when studying, as one is trained to do, save_data() in isolation.

Initially there were other problems with save_data() and this bug mearly added to the confusion. For performance reasons, save_data() was redesigned several times and eventually detailed knowledge of how it handled threads in different warps was used to implement it efficiently.

Large volumes of test data were passed through the kernel both to soak test it and to give reasonable estimates of how it will perform for real. The soak test gives some reassurance that the heavily inspected code really can cope with all combinations of simultaneous arrival of identical and non-identical data.

8.2 nVidia GPU Shared Memory Bug

The optional third parameter in nvcc’s `<<< >>>` CUDA kernel launch syntax allows you to specify the number of bytes of shared memory available to each block of threads in the kernel. The nVidia CUDA C programming guide says how to write your kernel. Unfortunately it is complicated and, as we shall see, error prone.

`kernel<<<grid_size,block_size,shared_size>>>(...)` effectively gives the kernel an anonymous array[2] which the kernel (with the compiler’s

[2] Anyone else old enough to remember Fortran unnamed common blocks? They were also a bug waiting to happen.

help) has to convert into usable C variables. I have evolved the following (which is based on the CUDA C programming guide).

There is one shared array. (It appears that if you try and declare two, they will actually be placed on top of each other.) It is declared in your .cu file using `extern __shared__ unsigned int shared_array[];` Every shared variable is explicitly defined as an offset from the start of it. You should provide host based checks (cf. `shared_size`) that these do not run off the top of shared memory. CUDA will check at run time you have not asked for more shared memory than your GPU has.

For every kernel that uses shared memory, we define macros like set_shared. Each such macro is used in the scope of it's kernel and/or the kernel's `__device__` functions.

```
#define set_shared                                        \
  volatile int* xs_error =  (int*) &shared_array[0]; \
  volatile int* xs_ndata =  (int*) &shared_array[1]; \
  volatile unsigned int* s_data =  &shared_array[2]; \
  volatile int* s_ptr =     (int*) &s_data[Nvalue]

#define shared_size ((3+2*Nvalue)*sizeof(int))

#define s_error  xs_error[0]
#define s_ndata  xs_ndata[0]
                          ⋮

__device__ void Error(...) {
  set_shared;
  if(s_error==0) s_error = ...
}
```

The additional macros s_error and s_ndata allow the kernel code to treat them as scalars rather than arrays. Notice array s_ptr should lie after s_data and none of the data should overlap. The particular bug arose as a cut and paste error whereby instead of using starting s_ptr at the last plus one element of s_data (i.e. s_data[Nvalue]) another value was used. Nvalue is a const int set to 800. The wrongly used variable was set to 600. Hence a quarter of the two arrays overlapped. This meant the code worked on some small examples but failed horribly on others. The device buffers described in Sections 6.1.3–6.1.5 and regression testing were used whilst finding and fixing this bug. However knowing which parts of the source code had been recently changed lead quickly to the location of the problem.

8.3 nvcc C++ compiler volatile keyword

I tend to avoid exotic parts of programming languages and so had overlooked nvcc's use of volatile when declaring shared memory variables. volatile essentially turns

off nvcc's optimisations whereby it uses registers rather than direct access to shared memory. Normally I would simply let the compiler get on with generating code but here was a bug in the making. Shared memory was deliberately used by multiple threads. When multiple threads of the same warp write to the same shared data, the hardware ensures one of them succeeds and the data from the others is discarded.

When nvcc optimises code which does not use volatile it may replace an access to shared memory by using a thread register. This lead my C code to think all the threads had succeeded in writing. Now that I realise what can happen, I use volatile on all shared memory declarations. The performance penalty of accessing shared memory rather than a register is small and I have not yet found an example where I am sure it is safe to allow the compiler to prevent inter-thread communication. After all I am mostly using shared memory to communicate between threads.

8.4 nVidia GPU Constant Memory

I going to call this a bug because even though the correct answers were calculated: in supercomputing we don't just want the correct answers but we want them fast, and this wasn't.

At first sight constant memory (Figure 1) appears attractive. Often applications have important data that we know is not going to change. Sometimes it looks small enough that it will fit into 64Kbytes. Or perhaps it is sparse and we can compress it into 64K. Essentially it can be much faster than global memory but it is not really 64Kbytes but a 64K window onto a much smaller caching system [34]. One view is to use textures instead since these are cached. Another possibility might be to take advantage of Fermi's cache and assume it will have the kernel's (read only) data in it most of the time that it is needed.

Here is my view of how constant memory works. Each kernel has a 64Kbytes window onto the same patch of regular global memory. Only the host PC is allowed to update that window but it can do it multiple times. Each time a thread tries to read from constant memory, the read request works it's way up through a hierarchy of caches. I am sure the details will vary between GPU architectures but Wong *et al.* [34] suggests the closest and hence fastest cache has only space for 512 integers or floats. (They say the largest useful cache has space for 2048.) Hence, we might think, if each thread block uses somewhat less than 8 KB (ideally less than 2 KB) there is a reasonable chance constant memory will help. Now it might be that we manage our kernel so a different thread block reads a different 8 KB, so it may be we can actually efficiently use all 64 KB if we are lucky (or skillful) with the details of how we write our kernel's reading of `__constant__` data. (Have I put you off yet? It gets worse.)

The hardware restrictions mean only one word can be read at a time from the `__constant__` cache. So if you code your kernel so that all threads in a warp read the same datum at the same time all is well. If they read two data, even if both are in the `__constant__` cache, the hardware will stall some of the threads and

the whole read will take twice as long. In the worse case, where each thread accesses it's own datum, the read takes 32 times as long. So while we have an advertised 64 KB, this is actually something like 512 words of real fast memory and then we can efficiently only read one of them!

The CUDA profiler turned out to be very useful. It can display the compute level 1.x GPU counter warp_serialise. In one case warp_serialise was huge, about 23 times the instruction count. This required the whole application to be redesigned. Essentially random access was replaced by a system where each block of threads uses only a limited part of the 64K and usually threads in the same warp read the same elements of the array at the same time. warp_serialise fell to an average of less then 1% across the kernel and the kernel at last started to run at a reasonable speed.

The following two code snippets declare and set constant memory. They are in the same .cu file and so are compiled in one go by nvcc.

```
__constant__
unsigned int Constant[15*1024]; //Leave 1kw free
```

The host PC code uses `cudaMemcpyToSymbol()` to initialise `Constant[]`. `cudaMemcpyToSymbol()` can also be used to change `Constant[]` between kernel executions. Placing it in a host function allows the GPU `Constant` array to be changed anywhere in the host PC code.

```
assert(0<matrix_size &&
       matrix_size<=15*1024*sizeof(unsigned int));
cutilSafeCall(
  cudaMemcpyToSymbol((const char*)Constant,matrixw,
    matrix_size, 0, cudaMemcpyHostToDevice) );
```

In principle the compiler can use the C `const` quantifier to recognise read only inputs to your kernel and access them via constant memory. In practise I have not seen it do this. At present it appears that only GPU data you explicitly denote with `__constant__` is accessed via constant memory.

nvcc uses `.const` in the ptx assembler it generates to indicate `__constant__` memory. (See the nvcc `-keep` command line option.) Note although the compiler generates human readable assembler, inspecting it is very rarely helpful.

8.5 Non-Reproducible Parallel Bugs

In industry it can be standard practise to ignore non reproducible bugs. They are hard to find and hard to fix. And besides there are plenty of well behaved bugs to fix. In parallel code the fact that it behaves differently in different circumstances can give you a clue that it suffers from some race condition. It may be that an asynchronous update problem has been in your code sometime but is only exposed by a change in the way it used. For example running on a different GPU or a change in load within

the kernel or a change in the way it uses threads, particularly increasing the number of threads above 32.

8.6 Impossible Bugs

Sometimes is just impossible to see why something does not work. It may be this is an opportunity to re-read the relevant CUDA documentation, find examples which do work, or consult the various online discussion groups, e.g. the nVidia CUDA Programming and Development forum. However perhaps you should use your revision control system (Section 7.3) to rewind your source code back to some earlier stable version.

Is it absolutely essential you implement the feature in the buggy kernel code? If so, is the bug related to the parallel threads? Perhaps it would be sufficient to have a serial version?

The following example shows using thread zero to force what should have been done in parallel to be done in series. (Remember the warning in Section 6.5 that thread zero must actually execute your serial code.)

```
//ugly hack
if(threadIdx.x==0) {
  s_ndata = 0; //Number of non-zero elements in s_data
  for(int i=0; i<Nvalue; i++) {
    if(s_data[i]) s_ndata++;
  }
}
__syncthreads();
```

8.7 Difficult Code

Perhaps if you suspect something is going to be hard you should consider writing a prototype first. The idea is the prototype should the opposite of CUDA. It need not be fast, it should not be run in parallel and it should be easy to implement. I tend to use gawk scripts because they handle reading input files much better than C. But it needs to be something you are comfortable with programming. Hack about your prototype until you have worked out the transformation you want the kernel code to do and the algorithm whereby it should do it. Kernels do not take kindly to being hacked. It should be much easier to work through your ideas in simple serial host PC code.

The GPU buffer files described in Section 6.1.3 might be quite a useful source of test data for your prototype. Ensure at least the "final" version of your prototype and any scripts/command lines needed to run it are saved in your revision control system (Section 7.3) before you go back to coding your kernel.

8.8 CUDA Bugs

Very rarely you will come across bugs in nvcc. Old versions of nvcc are not going to be fixed. If you have the very newest nvcc, you can report the problem. In all cases you will have to work around the problem.

Some advocate the C++ Standard Template Library (STL), but C++ templates have caused compiler bugs in the past.

8.9 C Coding Bugs

8.9.1 loop++

This was a logic error and not particularly related to CUDA or parallel computing. I had provided hash() to speed up searches. The monitoring code suggested a huge problem with many more hash clashes than searches. Inspecting the kernel code suggested the problem lay here:

```
int loop =  0;
do {
  if found .... break;
  else ... continue to search ...
} while(loop++ < large limit) //avoid infinite loop
if(loop) report long search
```

If hashing was working well, in almost all cases the `loop` should be exited before the `while` statement but in many cases `loop` was not zero and a hash clash was being reported. The wrong fix was applied. "Obviously" `loop++` had incremented loop from 0 to 1, so the last line should have been checking `if(loop>1)` not `if(loop)`. This was wrong (and did not resolve the problem). It turned out the hashing algorithm was flawed, resulting in a hash clash in many cases causing the `while` loop to be reached and `loop` to be correctly incremented and a hash clash to be correctly reported. Eventually hash() was improved and the number of hash clashes reported fell dramatically.

Part of the reason for the misdiagnosis was the delay between when I had first (correctly) written the loop and the availability of hash() and so the ability to test the loop. In the intervening period I had forgotten the logic of how `while(loop++` was expected to work. Better comments in the source code might have helped.

8.9.2 C Shift Operations and `unsigned int`

Given the dire warnings about the computational expense of division on GPUs and for other "efficiency" reasons the use of left shift << and right shift >> is common place. It is easy to overlook the warning in [13, p49] which says >>

on an `int` can either fill with copies of the sign bit ("arithmetic shift") or with zeros ("logical shift") depending on the hardware. This gave rise to the bug mentioned in Section 6.1.1. For example when kernel local variable `int v` has the value 0x80000000 and it is right shifted 24 instead of getting 0x00000080 (128), `v >> 24` gives 0xffffff80 (-127). Once found, this is readily fixed by declaring `v` as `unsigned int`.

It is claimed that the CUDA optimising compiler, nvcc, will spot division by integer powers of two and replace them by the correct shift operation. So it is common to use `/32` rather than `>>5` and rely on nvcc to create efficient code.

Although `Error(0x99960000,Nvalue)` quickly trapped the error, it was actually localised by remembering that the nearby hash() function had been recently changed and then asking the rhetorical question "how could hash() generate unexpected values".

hash() is expected to return a value between 0 and `Nvalue-1`, so conditional code was added to report if hash() returned something outside this range. E.g. `if(i<0 || i>=Nvalue) Error(0x999a0000,i);` Once this confirmed hash() was misbehaving (probably producing negative values) `if.. Error` could be used to further localise the bug but fundamentally hash() is short enough for the unexpected source of negative integers to be traced to my wrong assumptions about `v >> 24` and the declaration of `v` to be corrected.

8.9.3 Defensive Coding and Conditional Compilation

Again this is a bug which should not have happened, nevertheless it gives an example of where defensive coding was helpful. I had changed my GP CUDA kernel so that it ran all possible test cases rather than just a sample. Thinking I would only want to use this in special cases I intended wrapping it in conditional compilation marks `#ifdef ALL20` unfortunately I placed the corresponding `#endif //ALL20` after the new code. Thus leaving the original code to be compiled regardless of whether `ALL20` was defined or not. This meant when `ALL20` was defined each individuals fitness was calculated using both all the tests and the original sample. It was thus quite possible to score more than 100%. Here the defensive coding came in.

When the GP had been ported to the GPU all values calculated by the GPU were regarded with suspicion. In particular there was an `assert` which checked for both negative fitness values and values above 100%. After the GP had been debugged "obviously" the check was no longer needed, however fortunately I had not removed it. When the "improved" kernel was run, the check was quickly triggered and the error reported. The bug was easily located using the revision control system (Section 7.3) to highlight code that I had recently changed.

8.9.4 Graphics Card Hardware Monitoring

When debugging is hard it is always tempting to look for hardware problems. The nvidia-smi program can be used. E.g. `nvidia-smi -a` will tell you which GPUs you have and their temperature. However this is seldom useful.

Sometimes, e.g. when not using X-11, the nVidia GPU driver can unload software when the GPU is not in use. It can take several seconds, particularly if you have multiple GPUs, to reload it. This can delay the start of some GPU tools. A hack to avoid the driver thinking the GPUs are not in use, is to run nvidia-smi continuously in the background. E.g: `nvidia-smi -l -i 10 > /dev/null &` keeps nvidia-smi looping every ten seconds but discards its output.

`lspci` can be useful during installation for confirming you have the GPUs you expected plugged into the computer you expected them to be in.

9 GPGPU Development Environment

Section 9.1 suggest ways of setting up your system to ease development. Sections 9.2 and 9.3 described compiling your code whilst Section 9.4 discusses common configuration problems.

9.1 Hardware Environment

As mentioned in Section 6.1.1, the most disruptive problem to debug is probably when the kernel locks up your computer. There are a range of ways to set up your GPU programming system to ameliorate this:

- Test kernels on a dedicated computer.
- Have the test computer and GPU physically adjacent to your desk.
- Have multiple GPUs in the computer. E.g. a small cheap one that only drives the monitor and one or more GPU that are used for kernel development. It may be there is already a GPU on your PC's motherboard which was disabled when the development GPU was plugged into it. Perhaps it can be renabled?
  Make sure your CUDA application uses the GPU you want it to. It is probably sufficient to be able to specify which CUDA device your application will use via the command line.

```
if(argc>1 && argv[1][0]) {
  const int dev = atoi(argv[1]);
  cutilSafeCall( cudaSetDevice( dev ));
}
else cudaSetDevice( cutGetMaxGflopsDeviceId() );
```

- If the PC you use to develop CUDA applications is on the network, arrange that another networked computer is nearby so that you can log in via the network (e.g. using ssh). While this may allow you to gain reassurance that it really is a GPU problem rather than anything else, in the event of a GPU lock up it may be that there is little you can do, other than reboot. However you should have the option of telling the operating system to shutdown in a more controlled fashion. Perhaps informing other users/applications before their resources are removed.
- Some computer rooms have facilities to allow remote reboot. This may be under software control or you may have to ring up the operator and ask them to do it for you. Make sure you tell them the right computer!
- Make sure all of your CUDA system restarts automatically on reboot. Remember to include all the "little" tweaks to the operating system and X-11 windows that were done when CUDA was installed. This is especially important if CUDA was installed by someone else or if any of the "tweaks" need the root system password to reapply them.
- With its default setting, X-11 times out your screen if it fails to respond in about 10 seconds. E.g. suppose your kernel sometimes takes 12 seconds. Every so often it will cause the GPU on which it is running not to respond to X windows fast enough. For someone who is using the screen, this appears the same as if the GPU had failed, even though the GPU may be ok. Since this only effects X-11, you may be able to recover without rebooting Linux. For example, use one of the methods mentioned above to log into the host PC and restart X. It is also possible to disable the X-11 timeout or change its default setting.
  If the GPU can be reserved for calculations only, it might make sense to configure X-11 to ignore the monitor connected to the compute only GPU. However this might effect non-GPU uses of X-11, e.g. ssh -Y.

9.2 Compiling CUDA C/C++ Programs

You will need to compile your kernel with nVidia's CUDA compiler, nvcc. nvcc is also able to compile regular C and C++ code. nvcc host and GPU code can be linked with PC code compiled in the normal way. nvcc recognises many of the command line switches used by the GNU gcc compiler, such as setting conditional compilation switches (e.g. `-DUNIX`) and the debug flag `-g`). You will probably also need the GPU specific switch which tell the compiler to produce code for a particular nVidia GPU compute level (e.g. `-arch sm_20` for Fermi compute level 2.0). Check with the nvcc compiler documentation.

CUDA supports both 32 bit and 64 bit host PCs. You may need to double check you are linking the right libraries when you ask the linker to create your executable program.

9.3 CUDA SDK Makefile common.mk

The CUDA SDK examples include compilation scripts, known as Makefile. Most of their complexity is common to all SDK examples and is kept in a common make file (known as `common.mk`). One approach is to organise your application so that it follows the same directory structure and file naming conventions as CUDA's SDK. This will allow you to use `common.mk`. However it is also possible to adapt one of the SDK Makefile for your own project.

A disadvantage of using `common.mk` is that it assumes particular locations for your object and executable files. By default, the GNU GDB debugger run within emacs, is not compatible with this and refuses to show your host sources inside an emacs window as you use step through (the host part) of your application. If so, it may be easier to compile and link in your usual fashion. (`cuda-gdb` and commercial debuggers, e.g. Parallel Nsight and Allinea DDT, are increasingly available and increasingly capable.)

9.4 CUDA Compilation and Linking Problems

We next describe some errors that are common when you first use CUDA or after upgrading it and suggest potential solutions.

If using Unix and SDK's `common.mk` a helpful option is to run `make` in verbose mode so that it tells you the commands it is running. This is enabled in Unix by setting the environment variable `verbose`. E.g. `setenv verbose 1`.

On some older CUDA systems the additional line, `"NVCCFLAGS += -include=vararg-fix.h"` in `common.mk` may be required to get your kernel to compile.

Error `mkdir: cannot create directory '/opt/cuda/sdk': Read-only file system` suggests a problem with `ROOTDIR` or some inconsistency between your Makefile and `common.mk`. Perhaps you need to try overriding `ROOTDIR`, e.g. `ROOTDIR := /my_directory/cuda/sdk`, where `/my_directory...` refers to the directory tree you are using for your application.

nvcc compilation error `error: cutil_inline.h: No such file or directory` suggests a problem with `COMMONDIR` or some inconsistency between `common.mk` and your Makefile. Perhaps try overriding `ROOTDIR2`, e.g. `ROOTDIR2 := /usr/local/cuda/NVIDIA_GPU_Computing_SDK/C/tools`. Of course the actual setting for `ROOTDIR2` will depend on where exactly the files were placed when CUDA was installed.

nvcc compilation error `error: cuda_runtime.h: No such file or directory`. Again perhaps a problem with `ROOTDIR2`, however also check your system does really have a copy of cuda_runtime.h installed somewhere. It might also be a problem with `CUDA_INSTALL_PATH`. If so, you could try overriding it with something like `CUDA_INSTALL_PATH := /usr/local/cuda-3.0`

The Unix linker error `/usr/bin/ld: cannot find -lcutil` suggests a problem with `LIBDIR` or inconsistency between make files. This can occur when there are multiple versions of CUDA installed. Perhaps try overriding `LIBDIR`, e.g. by adding something like `LIBDIR := /my_directory/cuda_3.1/cuda/NVIDIA_CUDA_SDK/lib`. However eventually it may be better to resolve the problem of multiple version of CUDA and/or create your own make file or compilation script or process.

The Unix linker error `ld: skipping incompatible /usr/local/cuda-3.0/lib/libcudart.so when searching for -lcudart` might be a 32 bit v 64 bit problem. The Unix `file` utility will tell you if `libcudart.so` contains 32 or 64 bit code. Perhaps you need to change `LIBDIR` with something like `LIBDIR := /usr/local/cuda/lib64`

If you get `error while loading shared libraries: libcudart.so.2: cannot open shared object file: No such file or directory` this suggests your `LD_LIBRARY_PATH` environment variable is incorrectly defined. `LD_LIBRARY_PATH` allows the Unix program starter to search for `libcudart.so.2` in multiple directories. These are separated by a ":". Assuming you have an existing `LD_LIBRARY_PATH` environment variable then an option is to append the directory holding `libcudart.so.2` E.g. `setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":/usr/opt/cuda/lib`.

10 Other Sources of Help with Parallel Software Development

10.1 nVidia

nVidia has made available a host of documentation for CUDA and each of its components. Typically these are freely downloadable in PDF format.

A typical CUDA installation comes in three parts: GPU operating system drivers, CUDA toolkit and CUDA SDK. It is well worth installing the SDK directory tree when you install the first two. It contains more than 70 CUDA programming examples and GPGPU utilities, including their source code and in some case detailed documentation.

The SDK examples often both explain and give examples of tricky but highly efficient parallel computing approaches and are of course written for a GPU like yours. Examples include fast matrix multiply and calculating histograms in parallel. These examples show how to efficiently use shared memory in CUDA C.

10.2 nVidia Forums

nVidia hosts an impressive array of discussion fora at forums.nvidia.com. There are perhaps too many for an individual and it is better to stick to the one closest to your interest. For GPGPU the CUDA Programming and Development forum has proved useful.

10.3 Other Venues

There are many other Internet web pages in addition to those hosted by nVidia. For example, Simon Harding runs gpgpgpu.com specifically for combining genetic programming and GPUs whilst gpgpu.org is more generic. Whereas gpgpu.org does not deal specifically with bioinspired algorithms, there are a number of workshops and special events which do. For example, Computational Intelligence on Consumer Games and Graphics Hardware CIGPU, has run annually since 2008. Similarly the Workshop on Parallel Architectures and Bioinspired Algorithms WPABA has also run each year. With a winder remit than just GPUs, EvoPAR is set to become a track within the european evolutionary computing EvoApplications conference.

10.4 Alternative Approaches

We have talked about CUDA C. Is CUDA C the right language to choose? C is notoriously difficult and other languages are being added (e.g. Fortran, Matlab, Mathematica and Python). Nevertheless we can be reasonably confident that in the near term C/C++ will remain both the most efficient high level language for GPU computing and the most advanced and best supported CUDA programming language. CUDA is and is expected to remain nVidia's best way into the GPGPU world. However you might want your application to run on other manufacturer's GPUs or even non-GPU parallel hardware. OpenCL has been proposed by a small group of companies (including nVidia, AMD, Intel, Apple and IBM) as a way of implementing parallel applications. In theory it offers the possibility of running code on both GPUs from different manufactures and other parallel architectures. Currently support is patchy in practice.

In 2007 Harding gave a nice summary GPGPU tools [10]. It is notable that many have already fallen out of use. The software side of GPU computing has proved less stable than the underlying GPU architectures.

11 Conclusions

Some physical devices, e.g. some types of disk drive, give some indication of being used (e.g. audible hum or clicks, change in appearance or shaking, or getting hot). However, as with most solid state electronic devices, GPUs give little physical indication of how much they are being used or how close to they are to their maximum performance. To get the best of GPGPU you must use software techniques to predict, design and monitor performance. Sections 4 and 5 described how high GPGPU performance can be obtained and measured in practise.

Although tools continue to improve debugging CUDA C remains hard. Section 6 and 7 gave practical ideas for debugging and testing, whilst Section 8 describes how they were used with real bugs. The last two sections give practical advice on setting up your CUDA development system, other sources of help and alternatives.

Computation is cheap. Data is expensive

Perhaps slightly too strong but I have put it strongly to make the point. Wasting computing power does not come naturally. It is the opposite of what we were told as students. Nevertheless when on a GPU it can be more efficient to waste computation than move data.

It may be better to recalculate intermediate results than to store them. E.g. in some large matrix calculations. This is especially true if the intermediate results have to be saved on the host computer. On a GPU it often takes longer to move data than it does to calculate with it once it has arrived.

Since the GPU multiprocessors can only execute one instruction at a time, closely linked parallel threads which need to run different code have to run sequentially, not in parallel. Divergence represents idle compute resources. Effectively divergences is throwing away computation. But computation is cheap! It may be better to discard it than be unable to use a GPU at all.

The trend is for the cost of computation to fall faster than the cost of moving data. Thus the balance will continue to move in favour of more intensive calculations.

Debugging is the most expensive thing you can do

Avoid writing new code. Do you really need new code? Can you reuse nVidia's examples? Can you use an existing library?

Does it makes sense to treat your application as a matrix manipulation problem.

Is there an existing solution written in a matrix manipulation language (e.g. Matlab) which will run on your GPU?

Is there an existing GPGPU solution? Perhaps it is available on the Internet via FTP?

The Future of Bioinspired Applications

Life is parallel. Nature runs in parallel. Chemical molecules react when they meet. Antibodies neutralise antigens. Nerve cells fire at the same time. Ants follow trials. Bees swarm. Birds flock. Fish school. Populations mate and rear their young simultaneously. In many cases bioinspired algorithms are naturally parallel. Even embarrassingly parallel. Typically there is a good fit to parallel computing. This is especially true of low cost GPGPU computing.

Although a main stream break though in parallel computing has been forecast for at least 30 years [2] the 3 GHz ceiling has forced the hardware manufactures to generate affordable massively parallel computers and provide software support for them. Already there are many parallel bioinspired applications (Section 2) and with improving parallel development tools and cheap hardware, GPGPU (perhaps soon GPPPU) based applications have a great future.

Acknowledgements

I am grateful for the assistance of Gernot Ziegler of nVidia, Steve Worley, Sarnath Kannan and Erick Cantu-Paz. The T10P early engineering sample and C2050 Teslas were given by nVidia. Funded by EPSRC grant EP/G060525/2.

References

1. Derek T. Anderson, Robert H. Luke III, and James M. Keller. Speedup of fuzzy clustering through stream processing on graphics processing units. *IEEE Transactions on Fuzzy Systems*, 16(4):1101–1106, Aug 2008.
2. Hamid R. Arabnia and Martin A. Oliver. A transputer network for the arbitrary rotation of digitised images. *The Computer Journal*, 30(5):425–432, 1987.
3. Ali Bakhoda, George L. Yuan, Wilson W.L. Fung, Henry Wong, and Tor M. Aamondt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, Boston, MA, USA, 26-28 April 2009.
4. Erick Cantu-Paz and David E. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2-4):221–238, 2000.
5. 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.
6. Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, nVidia, 2003.
7. 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.
8. Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

9. Casey S. Greene, Nicholas A. Sinnott-Armstrong, Daniel S. Himmelstein, Paul J. Park, Jason H. Moore, and Brent T. Harris. Multifactor dimensionality reduction for graphics processing units enables genome-wide testing of epistasis in sporadic ALS. *Bioinformatics*, 26(5):694–695, 1 March 2010.
10. 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.
11. Simon L. Harding and Wolfgang Banzhaf. Distributed genetic programming on GPUs using CUDA. In Ignacio Hidalgo, Francisco Fernandez, and Juan Lanchares, editors, *Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 1–10, Raleigh, NC, USA, 13 September 2009. Universidad Complutense de Madrid.
12. Nicholas Harvey, Robert Luke, James M. Keller, and Derek Anderson. Speedup of fuzzy logic through stream processing on graphics processing units. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 3809–3815, Hong Kong, 1-6 June 2008.
13. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.
14. John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT press, 1992.
15. 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.
16. W. B. Langdon. A fast high quality pseudo random number generator for nVidia CUDA. In Garnett Wilson, editor, *CIGPU workshop at GECCO*, pages 2511–2513, Montreal, 8 July 2009. ACM.
17. W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 April 2010. Springer.
18. W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010.
19. W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, October 2008. Special Issue on Distributed Bioinspired Algorithms.
20. W. B. Langdon, Shin Yoo, and M. Harman. Formal concept analysis on graphics hardware. In Amedeo Napoli and Vilem Vychodil, editors, *The Eighth International Conference on Concept Lattices and Their Applications*, pages 413–416, Nancy, France, 17-21 October 2011. INRIA Nancy and LORIA.
21. William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 March 2008. Springer.
22. William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, December 2010.
23. Cleve Moler. Matrix computation on distributed memory multiprocessors. In Michael T. Heath, editor, *Proceedings of the First Conference on Hypercube Multiprocessors*, pages 181–195, Knoxville, Tennessee, USA, 24-27 August 1986. Society for Industrial and Applied Mathematics.
24. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

25. Luca Mussi, Stefano Cagnoni, and Fabio Daolio. GPU-based road sign detection using particle swarm optimization. In *Ninth International Conference on Intelligent Systems Design and Applications, ISDA 2009*, pages 152–157, Pisa, Italy, 30 November–2 December 2009. IEEE.
26. 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.
27. 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).
28. Raghavendra D. Prabhu. SOMGPU: an unsupervised pattern classifier on graphical processing unit. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 1011–1018, Hong Kong, 1-6 June 2008.
29. Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms–Principles and Perspectives: A Guide to GA Theory*. Kluwer Academic Publishers, 2003.
30. Jose L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic-algorithm programming environments. *Computer*, 27(6):28, June 1994.
31. John Rieffel, Frank Saunders, Shilpa Nadimpalli, Harvey Zhou, Soha Hassoun, Jason Rife, and Barry Trimmer. Evolving soft robotic locomotion in PhysX. In *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, pages 2499–2504, Montreal, Québec, Canada, 8-12 July 2009. ACM.
32. Nicholas A. Sinnott-Armstrong, Delaney Granizo-Mackenzie, and Jason H. Moore. High performance parallel disease detection: an artificial immune system for graphics processing units. GECCO 2010 GPUs for Genetic and Evolutionary Computation, 2011. Winning Entry.
33. Joachim Stender, editor. *Parallel Genetic Algorithms: Theory and Applications*. IOS press, 1993.
34. Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2010)*, pages 235–246, White Plains, NY, USA, 28-30 March 2010.
35. Lidia Yamamoto, Wolfgang Banzhaf, and Pierre Collet. Evolving reaction-diffusion systems on GPU. In Luis Antunes and H. Pinto, editors, *15th Portuguese Conference on Artificial Intelligence, EPIA 2011*, volume 7026 of *Lecture Notes in Computer Science*, pages 208–223, Lisbon, Portugal, October 10-13 2011. Springer.
36. Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. GPU-based implementation of real-time system for spiking neural networks. In Pilar Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2143–2150, Barcelona, 18-23 July 2010.
37. Weihang Zhu and James Curry. Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In *IEEE International Conference on Systems, Man and Cybernetics, SMC 2009*, pages 1803–1808, San Antonio, Texas, USA, 11-14 Oct. 2009.