GECCO Companion, CIGPU 2011 workshop, Simon Harding et al. *Eds.*, p423–430,

Performing with CUDA

William B. Langdon
Dept. of Computer Science, University College London
Gower Street, WC1E 6BT, UK
W.Langdon@cs.uc1.ac.uk

ABSTRACT

Recently a GPGPU application had to be redesigned to overcome performance problems. A number of software engineering lessons were learnt from this and other projects. We describe those about obtaining high performance from nVidia GPUs and practical aspects of CUDA C software development.

Categories and Subject Descriptors

D.4.8 [**Software Engineering**]: Performance—*Measurements, Modelling and prediction, Monitors*

General Terms

Performance

Keywords

C programming, GPU, GPGPU, parallel computing

1. INTRODUCTION

For about forty years it was a given that if you wanted your software to run faster, all you had to do was wait eighteen months and a faster computer would come along. Those days are gone. The software industry is still coming to terms with having to live with 3GHz processors. However the original Moore's law did not promise ever faster CPUs but talked of doubling the size of electronic circuits [1]. The consumer computer games market continues to reap the benefits of Moore's law by diverting the extra transistors it continues to deliver into ever more capable parallel computing.

Engineers and scientists noted that the graphics card in their computers had become more powerful than its CPU and started devising ways to speed up their applications by running parallel versions of them on their computer's GPU. Thus the field of general purpose computation on graphics hardware GPGPU was born [2].

GECCO'11, July 12–16, 2011, Dublin, Ireland.

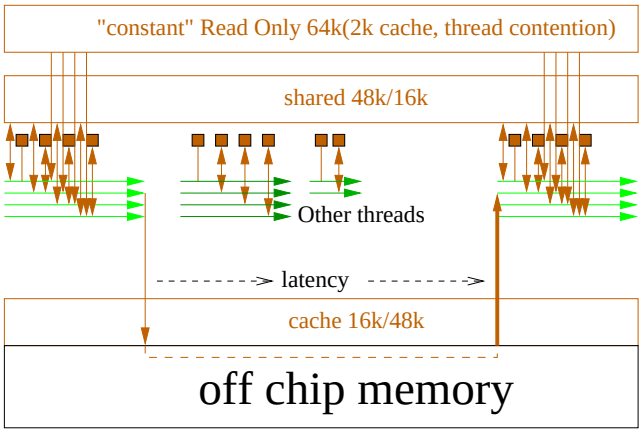

Figure 1: nVidia CUDA Mega Threading (Fermi). Each thread executes the same instruction. When program branches, some threads advance and others are held. Later the other branches are run to catch up (known as thread divergence). If a thread is blocked waiting for off chip memory another set of threads can be started. In Fermi (compute level 2.0) 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]. Threads of the same warp can read the same value but reads to different data are queued (known as warp serialisation).

There are many documents and tutorials on how to program graphics hardware for general purpose computing. Mostly they are concerned with perfect code written by experts. After a quick introduction to CUDA, Section 3 gives some practical 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 4 describes real ways to measure performance. This is not a general tutorial on CUDA, however the last two sections give practical advice for when you get started (Section 5) and some ideas for where to look for help if you hit problems and discuss alternative approaches (Section 6).

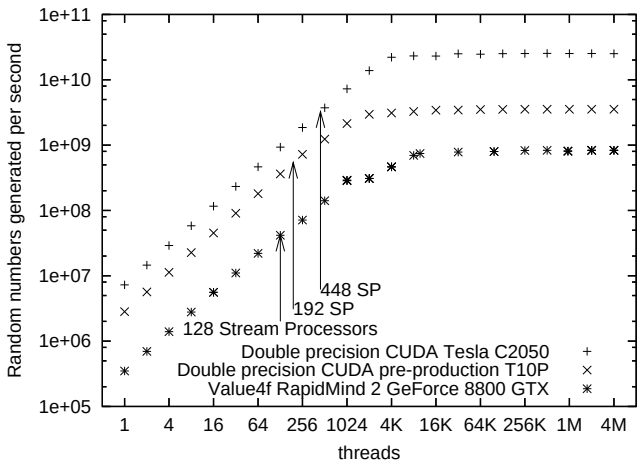

Figure 2: Park-Miller random numbers created per second (excluding host-GPU transfer time) on three nVidia GPUs. Top 2 plots refer to the same CUDA implementation and lower ones to RapidMind code. ftp code `cs.ucl.ac.uk /genetic/gp-code/random-numbers/ cuda_park-miller.tar.gz`. Plots from [4] extended to include Fermi C2050.

2. CUDA

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. (Figure 1 assumes that the requested data are not in Fermi's cache.)

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 with at least 10 threads per stream processing core.

3. PERFORMANCE

As novice programmers we were taught that we should get the code working before we worried about performance. CUDA programmers are often far from being novices. A common assumption is that a serial version of the application exists and it is "only" a matter of porting it to CUDA. Ideally then we should start by planning how the code will be run in parallel, how many threads and how they are to be grouped into blocks. Where data will be stored, how much memory will they occupy and how and in what way will it be accessed. In other words we should start by designing for performance. However coding kernels 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 easier than coding any other perfect software. In practice, GPGPU software production tends to fall between the two. That is as problems arise, some can be fixed immediately, 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 [5].

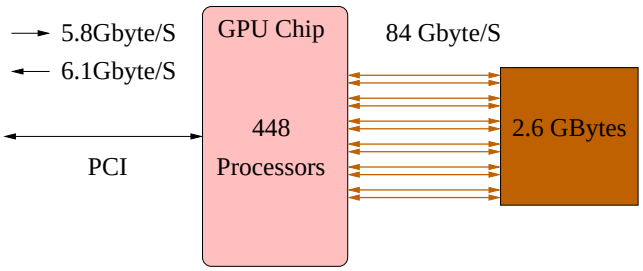

Figure 3: Links from GPU chip to host computer via PCIe bus and to memory on the GPU board. Fermi C2050.

3.1 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 time.
- In evolutionary computing applications the resource consuming part is the fitness evaluation of the population. 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 [6]. Indeed the comparative ease of parallelising population based algorithms has lead to them being called "embarrassingly parallel".

  Recall from Figure 2, CUDA applications typically need thousands of threads to get the best of GPUs. If your population does not contain thousands of individuals, perhaps there are aspects of each individual fitness evaluation 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. Remember nVidia's performance figures are the best that the GPU can do and so are typically much more than your kernel will get in practice. Is the fraction low enough to make the GPU a viable approach?
- From your block level design locate its bottle neck. See GPU block diagram in 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 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 does not use 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. (bandwidthTest 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 kernel 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.

  Whilst Fermi caches local and global data and earlier GPUs cache textures, it is usually better to "cache at the design stage". I.e. read data once, process it (without re-reading), then write the processed data once. This is unlike traditional coding, where it appears to cost nothing to read and write to program variables and 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.

3.2 Performance By Hacking

Once implemented the same basic idea applies to performance. Is performance good enough? Stop. Can performance be made good enough? If not then also stop. Identify and remove the bottle neck.

3.3 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.

3.4 Multiple GPUs

To take advantage of multiple GPUs, parts of the host application must be run in parallel. 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.

CUDA requires cudaSetDevice() to be used to initialise each thread. To avoid the surprisingly high CUDA initialisation overhead it is a good idea to start one thread per GPU. This is associated it with its GPU when your application starts. Normally the threads live as long as your application itself. I.e. each thread is repeatedly used to pass data between the host and its GPU and to launch kernels on its GPU. 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);
```

4. MEASURING PERFORMANCE

4.1 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 in PDF format.

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 others GPU operations. E.g. it can count the number of each type of global memory read and write operation. Finally it transfers these 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. The profiler can count a wide range of operations but not 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. It also provides a wide range of plots and tables for showing you this. A few of the interactive menus are a bit difficult to navigate and the documentation and menu layout may be slightly out of step.

Under Linux there is an alternative route, based on environment variables, see Table 1, whereby you can gather the same data and control the GPU end of the profiler.

The CUDA profiler gives some performance information which could be very useful but which would be either difficult or impossible to get elsewhere. 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.

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.) It then becomes clearer which ratio are near zero (even if their counter has five or six digit values) and can be ignored.

Another useful measure is to calculate the number of "instructions" your kernel is executing per microsecond.[1] (Again 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. Because of the arcane way in which the profiler reports "instructions" other GPUs will have fundamentally similar values. (It is a useful exercise to construct your own compute bound kernel and see what figure your GPU gives.) Your kernel will not reach 370 instructions per microsecond. If you are getting more than half of 370, congratulate yourself and stop. I have had kernels as disastrously low as 5.

4.2 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.

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 4.1).

CUDA provides timing routines. It is often convenient to create a CUDA timing data structure at the same time as you create your CUDA buffers.

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

                    ⋮
```

[1]Fermi compute level 2.0 provides different profile information which replaces the "instruction" count. However new counters like "inst_issued" can be used in a similar way.

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. 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 (see also Table 2). 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.

```
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;
```

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.

4.3 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 (perhaps by doing the operation a thousand times) the timing effect that is being measured is much bigger than the noise. 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. `if(in_length<0) d_out=junk_timing_info;`)

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 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.

5. DEVELOPMENT ENVIRONMENT

5.1 Hardware Environment

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 up. 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. This is especially painful where you are remote from the computer housing the GPU or where the computer is shared with other people or applications.

Notice not only is the result painful but you can get no indication of what has gone wrong or where. With the worrying probability that it will happen again after you get the system running once more.

Possible approaches:

- 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.

Table 2: CUDA Profile Counters

timestamp	
dynsmemperblock	The size of shared memory you specified, via `kernel<<<,,shared_mem>>>`, when the kernel was started.
stasmemperblock	This seems to be typically 80 and represents an overhead in which CUDA uses your shared memory. Hence your kernel does not actually have the full 16Kbytes (up to 48Kb on Fermi). Usually CUDA will refuse to start your kernel if you ask for all 16K.
regperthread	Fortunately recent GPUs have many registers and simple kernels do not run out of threads. and so this should not be a worry during debugging and perhaps only if really straining at the end of development to get the best of your GPU. The threads have to be shared between all active copies of your kernel. Suppose your kernel uses 17 registers per thread and you have a block size of 512 (total 8704). On a GPU with 16384 registers this would mean at most you could only have one block per multiprocessor. nvcc can also be told to report the number of registers when your kernel was compiled. nvcc may also be induced to reduce the number of registers it allocates to your kernel (nvcc `--maxrregcount` command line option). This might have the effect of it no longer unrolling loops.
memtransfersize	
memtransferhostmemtype	0 or 1. 0 if you forgot to tell CUDA to use non-paged memory.
streamid	I have not covered overlapping CUDA kernels by using CUDA streams, so streamid will be 0.
local_load	If the kernel is working well local_load should be zero. Zero means that the kernel is not using off chip "local memory", either via local arrays or because it ran out of registered and so they "spilled over" into local memory. Notice that Fermi GPU contain a cache which intercepts access to local memory. The cache is shared but if your kernel's local data is within it accessing local memory need no longer have disastrous consequence for performance.
local_store	If a kernel is working well local_store should be zero. See local_load.
gld_request	Total number of reads from off chip global memory. There are various types of more or less efficient reads. The different types also have their own counters.
gst_request	Total number of write to global memory. See gld_request.
divergent_branch	
branch	The number of conditionals (including ifs, loops and switch statements). A divergent_branch (previous line) is where some threads in a warp go one way at the branch and others go another. This is obviously a performance issue, since the hardware cannot do both simultaneously but must do both in series and then resynchronise itself. Nonetheless divergent_branch need not be critical and typically global memory access time should be considered first. Suggest you create a script to calculate divergent_branch divide by "instructions". Typically the ratio will be less than a few percent. Only if this ratio is much bigger should you consider divergent_branch.
sm_cta_launched	
gld_incoherent	if the kernel is working well both reading and writing to global memory should be coherent. I.e. gld_incoherent should be zero.
gld_coherent	
gld_32b	Number of 32 byte reads from global memory
gld_64b	Number of 64 byte reads from global memory
gld_128b	Number of 128 byte reads. Obviously it is more efficient to read data in larger units. However this is for the perfectionist performance tuner. Do not worry over much whilst debugging. It is more efficient if adjacent threads read (or write) adjacent data. E.g. with a two dimensional array a 41% saving in a kernel's run time was made by reversing the order of the indexes. There are equivalent gst_ counters for writing to global memory.
instructions	The number of "instructions" run by your kernel. This gets tricky because of course your kernel is being run simultaneously many times. Here instructions refers to the particular multiprocessor which is being profiled, not the whole of your GPU. The deviceQuery program will tell you how many multiprocessors your GPU has. Multiplying by this (e.g. multiplying by 14) will give an estimate of the total number of GPU instructions needed by your kernel. But even then we are not done. How many threads can a single "instruction" process and how does the instruction rate relate to GP clock. Eventually it all becomes consistent with nVidia claims for the performance of their hardware but it is easy to get lost in the details.
warp_serialize	Apart from divergence their are an number of other things which can cause threads to stop being synchronised. warp_serialize counts the number of instructions where the hardware ran threads in series rather than in parallel. I suggest you create a script to calculate warp_serialize divide by "instructions". Only if this ratio approaches or exceed 1.0 need you consider warp_serialize further.
cta_launched	
tex_cache_hit	GPU textures or user specified profiling not covered. There are also some Fermi specific counters.
tex_cache_miss	

Make sure your CUDA application uses the GPU you want it to. The ability to specify which CUDA device your application will use via the command line, is probably sufficient.

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

- If your CUDA test PC 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 them need the root system password to reapply them.

- Notice the possible interaction between CUDA and X-11 give similar symptoms (i.e. the system appears to lock up). 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.

5.2 Compiling CUDA

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.

5.3 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.)

5.4 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, `"NVCC FLAGS += -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`

6. OTHER SOURCES OF HELP

6.1 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.

6.2 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.

6.3 Alternative Approaches

We have talked about CUDA C. Is CUDA C the right language to choose? C is notoriously difficult and other languages are slowly 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 [7]. 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.

7. CONCLUSIONS

Computation is cheap. Data is expensive

Perhaps slightly too strong but I have put it strongly to make the point. It can be more efficient to waste computation. It may be better to have divergent code than be unable to use a GPU. 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. In terms of elapse time, in GPGPU, it often costs more to move data than it does to calculate with it once it has arrived.

In the future, the trend is for the cost of computation versus the cost of moving data to continue to move in favour of intensive calculations.

Acknowledgements

I am grateful for the assistance of Gernot Ziegler of nVidia, Steve Worley and Sarnath Kannan and the anonymous reviewers.

The T10P early engineering sample and C2050 Teslas were given by nVidia.

Funded by EPSRC grant EP/G060525/2.

8. REFERENCES

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

[2] 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.

[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, April 26-28 2009. IEEE.

[4] 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.

[5] W. B. Langdon. Debugging CUDA. In Simon Harding, W.B Langdon, Man Leung Wong, Tony Lewis, and Garnett Wilson, editors, *CIGPU 2011*, Dublin, 13 July 2011. ACM.

[6] Joachim Stender, editor. *Parallel Genetic Algorithms: Theory and Applications*. IOS press, 1993.

[7] 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.