GECCO Companion, CIGPU 2011 workshop, Simon Harding et al. *Eds.*, p415–422.

Debugging CUDA

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

ABSTRACT

During six months of intensive nVidia CUDA C programming many bugs were created. We pass on the software engineering lessons learnt, particularly those relevant to parallel general-purpose computation on graphics hardware GPGPU.

Categories and Subject Descriptors:
D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*
General Terms Verification
keywords computer game hardware, graphics controller, GPU, parallel computing, RCS

1. INTRODUCTION

The absence of sustained increases in processor clock speed, which characterised the second half of the twenty century, is starting to force even 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 overhead (e.g. space, power) across multiple CPUs. In contrast, in theory, a single high end graphics card (GPU) can provide similar performance and indications are that GPU performance increases will continue to follow Moore's law [1] for some years. The competitive home computer games market has driven and paid for GPU development, with nVidia selling hundreds of millions of CUDA compatible cards [2]. Engineers and scientists have taken advantage of this cheap, powerful 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 [3].

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, maintaining it, and even trying to understand it. Despite a number of GPGPU conferences and workshops, such as CIGPU, development of GPGPU software remains a black art, often at the edge of feasibility. Debugging is key to any software development but little has been published about getting non-trivial CUDA application to work.

Although tools are improving, we concentrate upon how debugging is done for real. We shall assume the reader is already familiar with nVidia's parallel computing architecture, CUDA. Although many lessons are general, the examples use nVidia's GPUs with the CUDA C compiler, nvcc, directly rather than via Microsoft's visual studio. One potential debugging technique would be to dump intermediate results directly to the GPU's DVI or HDMI output and thence to a screen or speaker, however we are concerned with general purpose computing algorithms and so we don't assume the reader is an expert graphics programmer. The next section describes coding techniques to aid debugging. Section 3 described testing CUDA C applications, whilst Section 4 describes some bugs, the techniques used to find them and how they were fixed.

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

2. DEBUGGING TECHNIQUES

2.1 Defensive Programming

2.1.1 Kernel 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. Notice not only is the result painful but you may get no indication of what has gone wrong or where. Further its 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 log a description of every kernel launch, *before* it is started. Conditional compilation switches could be used to remove it from production code. (It may also be necessary to flush the log before asking CUDA to start the kernel.) 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<<<%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<<<grid_size, block_size, shared_size>>>
(height,width,len,d_in,d_a,d_y,d_out1,d_out2,d_status);
cutilCheckMsg("kernel() execution failed.\n");
```

Given the CUDA parallel processing architecture, it is seldom necessary, apart from the main thread loop, to have 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 following loop structure was based on CUDA's SDK examples. Note this loop is not protected.

```
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.

2.1.2 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. In fact you should wrap all calls to CUDA routines with cutilSafeCall() or cutilCheckError(). (There is seldom a good reason for allowing the code to continue passed an error.) See 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 `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 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`.

The string you supply to `cutilCheckMsg()` need not be fixed. If, for example, you start your kernel in a loop you could make the string you pass to `cutilCheckMsg()` include the loop index.

2.1.3 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 pairs of buffers.

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.

Use `cudaMalloc` to create GPU global memory buffers:

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

2.1.4 Host PC Buffers

The host buffers 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. (Effectively this saves a buffer copy). Normally this effectively doubles the transfer speed to and from the GPU. However in once 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.

2.1.5 Debugging 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. These might be informal or only true in particular circumstances. E.g. you might want to double check 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. 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);
}
```

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 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`

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 (cf. Section 4.5). 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.

`printf()` has been available inside CUDA kernels for more than a year 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.

2.2 Your First CUDA Kernel

The following way of debugging CUDA kernel's 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?

We start with does the GPU send data to the right place

```
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 [7]. 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 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? 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?

2.3 CUDA 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 9 and 106 lines).

2.4 GPU functions

Although the CUDA C compiler, nvcc, efficiently supports functions on the GPU, `__device__` functions tend not to be used. 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 function assembles an answer in shared memory (or the answer is passed to it via shared memory) it then wishes to send the answer to the host PC. It must write it to global memory. A kernel might use a loop like this:

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

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];
  }
}
```

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. 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 2.1.3–2.1.5).

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 functions.

2.5 Shared Memory

Shared memory is rapid access read write on-chip memory available to blocks of kernel threads. It gives CUDA it's only modifiable rapid access arrays. (CUDA threads can have modifiable "local" arrays but until Fermi, compute level 2.0, 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.

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 not 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 [4]. 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.

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.

SDK's matrix manipulation examples 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 duplicates from large arrays, multiple threads are needed to read the array rapidly but each thread needs to know which duplicates the others have found.

There is only a small amount of shared memory and it may be quickly be exhausted. CUDA's SDK has examples where data are first stored in shared memory and then results from different blocks of threads are combined.

2.6 Error Reporting

The function Error(), mentioned in Sections 2.1.1 and 4.9.2 was introduced into a kernel which was proving very hard going. It is designed to (eventually) report the first error detected to the host PC, where code retrieves it and reports it to the log. 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 one 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 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.

3. TESTING

Assume new or modified code is wrong. This is particularly important with stochastic AI techniques, such as evolutionary algorithms. 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 crossover for an 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.

3.1 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, shell script, as long as it produces correct (but non-trivial) answers.

[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.

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). Keep a record of the seeds used in the log. 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.

3.2 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.

3.3 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.

4. GPU BUGS

4.1 Not all threads available

Another manifestation of the problem described in Section 2.4 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.

4.2 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 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 past 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

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

overlapped. This meant the code worked on some small examples but failed horribly on others. The device buffers described in Section 2.1.3 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.

4.3 volatile

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 (which is mostly why I am using shared memory).

4.4 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 answer but we want it fast, and this wasn't.

At first sight constant memory 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 [5]. 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 64Kbyte window onto the same patch of regular memory. Only the host PC can 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.* [5] suggests the closest and hence fastest cache has only space for 512 integers or floats (the largest useful cache is 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 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 compiled in one go by nvcc. Placing `cudaMemcpyToSymbol()` in a host function allows the GPU `Constant` array to be changed anywhere in the host PC code.

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

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

4.5 Non-Reproducible 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 CUDA the fact that your code 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 load within the kernel or a change in the way it uses threads, particularly increasing the number threads above 32.

4.6 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 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 2.4 that thread zero must actually execute your serial code.)

```
if(threadIdx.x==0) { //ugly hack
  s_ndata = 0;
  for(int i=0; i<Nvalue; i++) {
    if(s_data[i]) s_ndata++;
}}
__syncthreads();
```

4.7 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 2.1.3 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 before you go back to coding your kernel.

4.8 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.

4.9 C Coding Bugs

4.9.1 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 we should have been checking `if(loop>1)` not `if(loop)`. This was the wrong fix 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 code might have helped.

4.9.2 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 [6, 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 2.1.1. For example when 0x80000000 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.

5. CONCLUSION

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?

6. REFERENCES

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

[2] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun ACM*, 53(11):58–66.

[3] J. D. Owens, *et al.* GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. Invited paper.

[4] W. B. Langdon. Performing with CUDA. In S. Harding *et al*, eds., *CIGPU 2011*, Dublin, 13 July 2011. ACM.

[5] Henry Wong *et al.* Demystifying GPU microarchitecture through microbenchmarking. *ISPASS 2010*. IEEE.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. second edition, 1988.

[7] W. B. Langdon, M. Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *J Syst Software*, 83(12):2416–30