Introduction

GPUs represent a new holy grail for HPC users – the opportunities of lower power, higher FLOP rates, and smaller form factors tick all the right boxes – and a majority of HPC sites are either investigating them, or planning new systems that revolve around the technology.

The architecture of a GPU is unusual for those of us who are settled with the von Neumann model – and we are having to learn to embrace the hierarchical memory and finegrained parallelism that accompany the GPU.

NVIDIA CUDA is a popular route for developing applications that exploit the power of GPUs – and many applications are being ported to CUDA. Core computational kernels are rewritten for CUDA and then mixed with the existing code.

Naturally, where there is development, there will also be bugs. This white paper introduces the new CUDA enabled version of Allinea DDT – and shows some of the powerful features that are in the product to help track down CUDA bugs – showing how to use memory debugging to track memory access errors, or how to perform tasks like stepping through a CUDA kernel and examining variables and memory.

The Allinea DDT parallel debugger from Allinea Software has been setting the standard for usability for many years and is tearing up scalability records. It is used on the world’s largest systems – debugging over 220,000 processes in some cases – so all those GPU threads should be within reach of Allinea DDT! CUDA support is available as an option to Allinea DDT, and has licensing choices covering single workstations through to parallel development with mixed MPI and CUDA code.

Preliminaries

To start using CUDA with Allinea DDT, you first will need a Linux system with an NVIDIA graphics card – any recent NVIDIA graphics card will be usable.

You will need to download and install the SDK and latest drivers from NVIDIA too, at time of writing this is SDK 3.0. You cannot currently debug the same GPU that is also displaying your X server – so consider logging in remotely to a machine, using X forwarding or VNC.

The CUDA C language is an extension of C that exposes the architecture of the GPU to a level that allows the developer to achieve the full potential of the platform. The developer creates “kernels”, which are executed on the GPU – by CUDA threads. CUDA threads are
organized into a one or two dimensional grid of blocks of threads.

Each thread within a block can be indexed by one, two or three dimensional coordinates – depending on how the kernel is configured.

There are many examples in the NVIDIA SDK of existing CUDA C code, and you will also find documentation about how to program CUDA systems at http://developer.nvidia.com

Introducing Allinea DDT

Allinea DDT is a graphical parallel debugger – used by many scientific computing centres, universities and corporations to help in the everyday task of finding and fixing bugs, from single process workstations through to the very largest supercomputers. It

Getting started

CUDA enabled Allinea DDT can be downloaded from the Allinea website (www.allinea.com/cuda) – where you will also be able to obtain an evaluation licence. Installation is straightforward. In Allinea DDT’s examples subdirectory you will find an example code for CUDA – prefix.cu. This example takes an array, and computes its “prefix sum” – that is to say, in the output array, the ith element is the sum of the input array between the 0th and ith elements inclusive. This is the sort of computation that is complicated to do on a GPU – because every value depends on the previous values in a chain. It is straightforward for an ordinary CPU – and that makes it an interesting challenge for a GPU developer – and a debugger!

The prefix code contains examples of multiple kernels, data transfer, and synchronization with the syncthreads operation.

Stepping through the example

Firstly, we must compile prefix with support for debugging – this means we must use the “g” and “G” flags.

```bash
% cd {ddt-install-dir}/examples
% nvcc -g -G prefix.cu -o prefix
```

Next, we start DDT.

```bash
% {ddt-install-dir}/bin/ddt {ddtinstall-dir}/examples/prefix
```
If this is the first time you have ever used Allinea DDT, it will take you through the configuration wizard. This won’t take long – do not worry if Allinea DDT cannot configure some of the things it needs for MPI jobs (like attaching and remote access for example) – they won’t be needed today.

Once that’s done – you should see a welcome dialog – select the “Run and Debug a Program” option. Allinea DDT will then show you some options about the program you are going to debug.

Ensure you have chosen the prefix binary correctly and CUDA support is enabled (ie. “run without CUDA support” is not ticked). Press the “Run” button and let Allinea DDT start the program. It will return at the start of the code – in the code being run by your ordinary x86_64 (or x86) processor. Press the Step Over button (or press F8) a couple of times and you will see the highlighted source line move – this shows the process stepping over (executing) these lines of code.

If you are unfamiliar with using a debugger – or just want to feel your way around Allinea DDT to get started, you might like to look over the various components in Allinea DDT’s interface now.

Amongst the components you’ll see – to the right of the source code – are the “Locals” and “Current Line” panels. These are important as they show variables in the current function, and variables on the current line respectively.

Let’s make the process move a little further:

move the mouse to line 193, right click and choose “Run to here”. The process has now been through “devicesDump()” – which printed some detail about the device we’ll be debugging – click on the Input/Output panel in the bottom left to read it.

Next we’ll step into “cudasummer” this is where the GPU work starts by clicking “Step Into” (F5).

In cudasummer the first real calls into the CUDA API are made – setting up some timer events, allocating memory on the device, and copying the input array (data) from the host CPU over to the GPU (devln). We can Step Over these until we reach the “prefixsum” function on line 143. You will see in the Local Variables panel that the devIn and devOut pointer values have changed – although this is GPU memory so you won’t be able to read the contents at those pointers until we are inside a GPU kernel.

Debugging GPU kernels

Scroll up the source code to zarro on line 88 – you’ll see it’s a trivial kernel, it zeroes the memory in the “out” array.
Allinea DDT can control your execution through this kernel: Setting a breakpoint in zarro – by scrolling to line 90 and double-clicking on that line – causes the kernel to stop when threads reach this line.

Click the green Play/Continue icon (F9). You’re now in the zarro kernel! A thread selection panel will have appeared.

**CUDA Thread Selection Panel**

**Examining CUDA thread variables**

In the locals or current line panels, you will now see variables within the current CUDA thread – \(<<(0,0),(0,0,0)>>\). In the “Current Line” panel – it shows “blockIdx” and “threadIdx” – the CUDA variables that are used for thread identification, as well as “x” which is used on this line.

It is worth noting that – depending on the capabilities of your GPU, only some of the GPU threads will exist at this point – others will be created and selectable when they are scheduled.

Allinea DDT has a really easy way to keep track of where GPU threads are: Switch the bottom panel – which is probably still showing the Input Output tab – so that the “Stacks” tab is showing. You should see a count of the extant threads: 1 CPU thread, and, at the bottom of the stack, some GPU threads. This number will vary according to which GPU card you have.

**Stepping CUDA threads**

Change to another process by changing the values in the Thread Selection Panel to \(<<(0,0),(3,0,0)>>\) and pressing “Go”. Line 90 currently has not been executed, we are at the start of the line – the variable “x” is still zero.

Operations such as stepping for a CUDA thread cause a thread and its warp to move – that’s 32 threads in total. The other threads will remain paused. Play/continue causes all threads to play.

Step over line 90 and “x” will change – to the value 3, which is correct for this CUDA thread.

Two other things also changed in the GUI during that step. Firstly the source code display has changed – hover the mouse over the two highlighted lines of code to see which threads are on each line. Secondly, the Stacks display is now showing 32 GPU threads at line 92, and the remainder at 90 – as illustrated earlier.
A more complicated kernel

We now move on from that simple kernel – unset the breakpoint, by double clicking on the breakpointed source line again (90).

Set a breakpoint in prefixsumblock by double clicking at line 51 and press Play to go to this line.

The prefixsum algorithm works by splitting the list of numbers into contiguous blocks – of size BLOCK_SIZE. Each of these blocks has its (local) prefix sum computed by a CUDA block. That makes one GPU thread per element.

Once these individual blocks are computed – the end points of each block are the sums of each block – and we then need to “correct” the prefix sums within all of the blocks to add the appropriate endpoints of other blocks. In this implementation, those endpoints must also have a prefix sum calculated, so this algorithm is recursive and can invoke more kernels for larger array sizes.

In the first pass of the prefixsum, depending on the size of your GPU, the number of blocks required may not fit into your GPU in a single phase: you’ll be able to see this now – look at the number of threads in the parallel stack view – if it is less than the length of the input data (see the “length” variable), then this first breakpoint will be passed more than once as the GPU schedules the threads.

Let’s run as far as the end of the loop – right click on line 67 and choose “Run to here”.

Examining Device Arrays

Now we want to look at the data that has been calculated so far – we have completed one iteration of the for loop which sums locally within each block.

DDT has a feature which is ideal for this – the MultiDimensional Array viewer (the “MDA”). We will take a look at the progress – how the output data has already changed. Right click on “out” in the “locals” tab, and select “View Array” change the expression to read “out[i]” – then set bounds for $i$ of from 0 to 499, and click “Evaluate”.

Examining device array data

It’s often easier to see this kind of data in a graphical form – you can do this by clicking the “Visualize in 3D” button.

The “out” array after one iteration

You can spin and zoom this image to take a look at it from different angles. This kind of feature can help you to see how your kernel code is behaving and which parts of a kernel have completed. Depending on the capabilities of your GPU, all or an initial subset of the array will have changed.

That’s quite enough of the core kernel now, let’s move to another kernel. Double click on line 75 – this sets a breakpoint where the end points are about to be corrected. In the same MultiDimensional Array Viewer, click Evaluate, and then Visualize in 3D to update the view. Select thread $<<(1,0),(2,0,0)>>$ – and then press “Step Over” this will have
caused the threads responsible for elements 64 through to 95 inclusive to be updated. Update the visualization as before and take a look at how the data has now changed. You can change to other threads, and continue to look at progress in the visualization window.

Step Out of this kernel (F6) – and step out once more to go back up to cudasummer, where a cudaMemcpy from the device to the host takes place. Step Over another line of code – and you can now look at the “data” variable in the MultiDimensional Array Viewer and see the completed array – a smooth, quadratic, curve.

CUDA Memory debugging in DDT

So we have a working prefix sum program – or do we?

There is a subtle bug – and we can now turn on the CUDA memory checking feature to check for this – it will tell us when we read/write beyond an array. This kind of bug can really hurt – it can cause a kernel to overwrite data, or cause it to abort without any kind of diagnosis.

Let’s start again, click “Session” on the menu bar, and “New Session”. The Run Dialog will be shown. This time, before pressing “Run” click the Advanced button and enable “CUDA Memory Debugging”. Now click “Run” – and when it has started, click “Play”.

The CUDA memory debugging mode will tell you there is a segmentation violation – a memory access problem – and it even tells you which line of code and which GPU thread it applies to.

Enabling CUDA Memory Debugging

Let’s see if it’s right. Look at “x” which is calculated from threadIdx and blockIdx. Now go to the Stacks window – and click on the prefixsum frame, one up from the bottom of the tree in the parallel stack view. “devEnds” is an array we have allocated a few lines earlier – of 8 integers.

Sure enough we’re reading elements beyond the end of the array (“x” is 32 on our GPU, you may see a different value). We can fix this problem by passing an extra parameter – the length of the devEnds block – and adding an “if” statement to check for this.

This may have been a trivial example, but these kinds of problems happen very easily – when programming in CUDA we choose a block and a grid size – but there is no guarantee that our input data length or size is a multiple of the block sizes or grid sizes – which means there will often be some checking required to ensure the edges are not stepped over.

Compile the code again, and run through this time it should not crash!
Summary

We have seen how Allinea DDT can be used to debug CUDA applications – and some of the most useful features to help users to track down problems, in particular:

- Controlling the kernel execution by setting breakpoints to pause the kernel at a line
- Stepping individual warps of threads
- Examining variables in CUDA – including register based variables, and device memory
- Visualizing arrays that are on the device
- Showing where all threads are inside a kernel with the parallel stack view
- Detecting errors in array or block boundaries with CUDA memory debugging.

The prefix sample code has scope for you to improve its performance – adding shared memory to replace some of the costly global device memory usage for example. It already prints out the time it takes to run – see if you can speed it up – and if you introduce any new bugs, then let DDT help you fix them.