UNIVERSITY OF CAMBRIDGE

GPU Clusters in a Production Environment

Dell – Cambridge HPC Solution Centre
University of Cambridge

Dr Stuart Rankin, Dr Paul Calleja

A. Abstract

In this whitepaper, we describe the installation, configuration and commissioning of a GPU cluster, integrated as a computational unit within a larger HPC production environment. Various issues are addressed covering scheduling, health monitoring, affinity and security.

B. Background

In 2009, Cambridge received CUDA Centre of Excellence status from NVIDIA and the funding to build a large Tesla-based GPU cluster. The new GPU extension was integrated within the existing services, complete with connections to the 10Gbit/s ethernet and QDR Infiniband networks, and access to the Lustre storage under the control of the scheduling system: Moab/Torque. The new GPU computational unit provides a theoretical peak performance of 120 Tflop/s in single precision (1 Tflop/s double precision) from the GPUs alone, in addition to the 2.7 Tflop/s peak provided by the Nehalem CPUs within the servers.

While the hardware that has resulted in this paper is not current generation (described in full in the appendix), the guidance and experience offered in this paper still applies to later 2050/2070 NVidia units. These current generation systems, based on the Dell PowerEdge C6100 host system and the PowerEdge C410x GPU unit, are now under evaluation in Cambridge and are to be reviewed in a supplementary technical bulletin later this year. The GPU computational unit used in this document is a subsystem of a much larger "Darwin" HPC system for researchers at the University of Cambridge and is summarised below:

System	Quantity	Description
Hosting Servers	32	2x Intel X5550, 24GB RAM, Mellanox MT26428 IB, 1Gbe, 2x PCIe Gen2 x16 interfaces
NVidia Tesla S1070	32	2x PCIe Gen2 x16 channels to server 2x Tesla C1060 GPUs per channel, each with • 4GB global memory (not ECC) • 240 cores • 933 Gflop/s (single precision) • 78 Gflop/s (double precision)

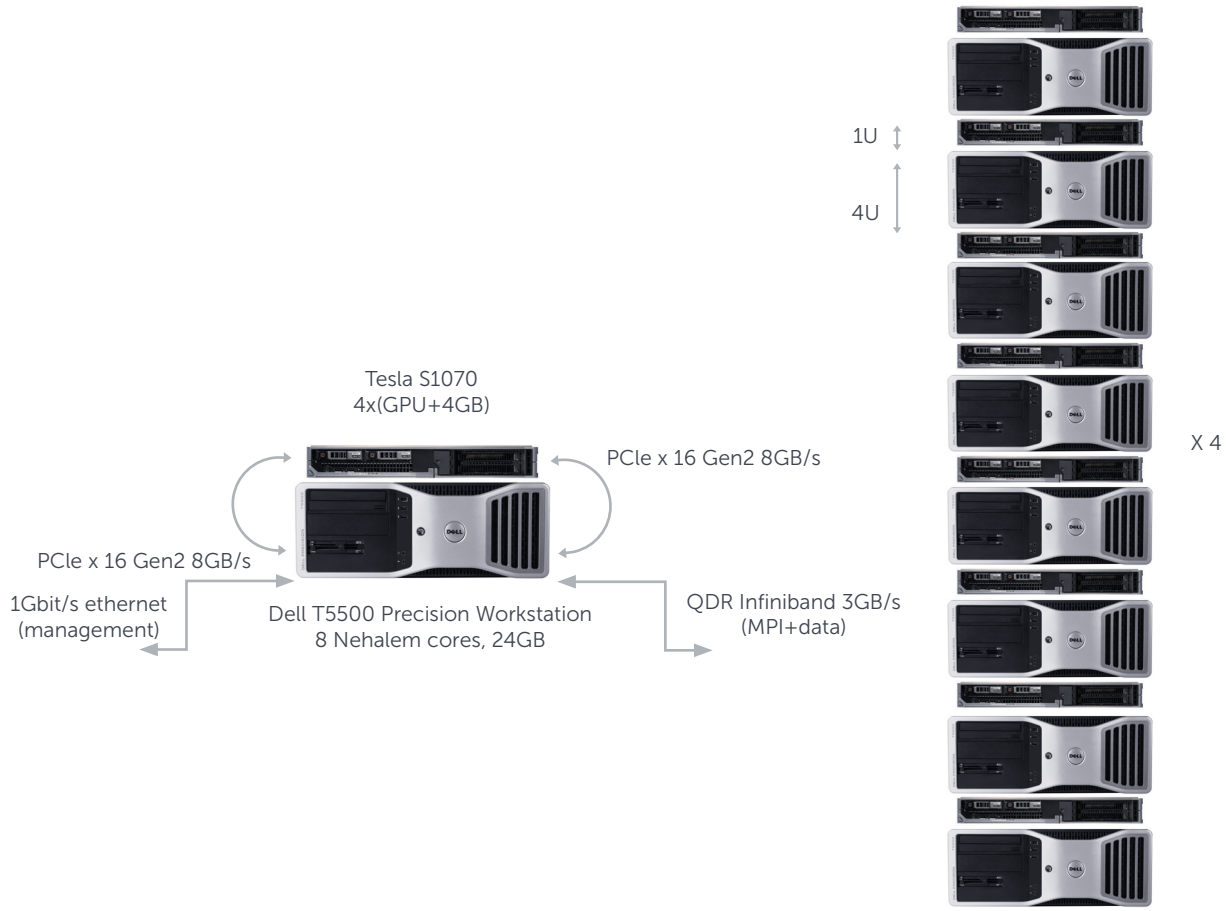

Figure 1. Darwin GPU computational unit (256 Nehalem cores, 128 GPUs)

C. Integration into the HPC Production Environment

C.1. Scheduler configuration

In general the allocation of jobs to a multicore compute node is complicated by the need to ensure that jobs do not stray into resources allocated to other jobs. Failure to do this can easily result in unpredictable performance and even failure for one or both jobs. Furthermore for NUMA nodes (such as those with Intel Nehalem/Westmere CPUs), correct process pinning and memory locality is often important for proper performance – handling this for multiple jobs sharing the same node is a non-trivial task, as it is for large NUMA systems. The addition of GPUs to such a node complicates matters further, extending the problems of segregation and affinity to the GPUs as well, and in this case the tools available for control are somewhat less developed (if they even exist) than the analogous utilities for CPUs (cpusets, numactl, taskset).

For the production system at Cambridge, this problem is greatly simplified by a general policy that only one job may run on a node at any one time. This reduces the issue to the proper placement of a single job across all available CPU cores and GPUs. The typical job uses MPI (either Intel MPI or MVAPICH2) for communication between up to eight CPU tasks per node, each of which may also dispatch CUDA kernels to one of the four GPUs in the S1070. Whereas CPU pinning and host memory locality may be assisted by features in the MPI implementation, there is no standard mechanism for arranging proper

affinity between CPU core and GPU, although the **cuda_wrapper** library from NCSA offers a possible way to do this by overriding CUDA calls. Bandwidth measurements between workstation host and Tesla device suggest that arranging proper affinity may be beneficial (see below). So far the Darwin GPU cluster hasn't fine-tuned for this but in principle the scheduling system prologue script described below could be extended to use cuda_wrapper.

A further policy decision requires that only jobs using GPUs may run on the GPU unit. Thus the selection of GPU-enabled nodes can be reduced to a simple choice of queue or runtime environment configured in the scheduling system. In our case, using Moab/Torque the Moab scheduler then restricts visibility of this queue (or *class* in Moab terminology) to nodes in the GPU computational unit (Tesla01 – Tesla32) with a directive such as:

```
CLASSCFG[tesla] HOSTLIST=r:tesla[01-32]REQUIREDQOSLIST=QOS1,QOS2,QOS3,SUPPORT
```

The REQUIREDQOSLIST clause lists the allowed qualities of service (QOSs) in the Tesla queue (in this case, all of them). In order to allow interactive jobs on the GPU nodes, which are granted higher priority and intended for development work, there is also the Tesla-int queue/class, controlled in Moab with:

```
CLASSCFG[tesla-int] HOSTLIST=r:tesla[01-32] REQUIREDQOSLIST=INTR
```

(the INTR QOS in this case being allowed only).

For a time we wished to allow access to the Tesla nodes from the default Torque queue. It was then necessary to allow jobs to specify a requirement for GPU – this was done by assigning each Tesla node a "GPU" consumable resource, with a node definition within Moab like the following:

```
NODECFG[tesla01] RACK=25 SLOT=1 GRES=GPU
```

Jobs requiring GPU would then indicate this at submission time in the resource string, e.g.

```
$ qsub -l nodes=2:ppn=8,mem=24000mb,gres=GPU jobscript
```

C.2. Node scripts

As with any compute node in a cluster, two types of script are required (at least) in order to monitor the health and usability of a GPU node. The presence of GPU hardware necessitates the introduction of new tests in addition to those needed by a pure CPU/IB node.

C.2.1 Health check script

As with other schedulers, such as LSF or Grid Engine, the Torque pbs_mom daemon, which runs on each node and is responsible managing jobs sent to the node, can be configured to run a health check script periodically. On Darwin this is done by adding two lines to the /var/spool/PBS/mom_priv/config file on each node:

```
$node_check_script /var/spool/PBS/mom_priv/compute_node_check.sh
$node_check_interval 20          # i.e. 15 minutes in 45 sec mom intervals
```

pbs_mom then runs compute_node_check.sh every 15 minutes. The script is executed as root, and if it blocks, so does pbs_mom. If the script detects an issue with the node, it is required to send a string to standard output of the form ERROR: error message. Moab's default response to this is to mark the node down.

The aim of a health check script is to promptly detect a problem with a node that would either need to be corrected before jobs are sent to it, or that would require a running job to be requeued and rerun elsewhere for successful completion. The script needs to run at regular intervals, whether or not a job is already present, however that places limitations on what it can safely be allowed to test. For example, probes to the hardware may well be unsuccessful if a job holds the devices open, and transient timeout issues with Lustre or NFS mounts, from which jobs should safely recover, should not lead to a requeue of the job. In short, health checks should be sufficiently non-aggressive that they can coexist with running jobs.

Another possible role for a health check script is to purge left over processes belonging to users whose jobs have ended, so that these don't continue to consume resources.

The GPU node health check script performs the following checks. A failed check causes the whole script to signal ERROR unless indicated:

Basic checks

1. Is the local disk writable?
2. Is NIS running? If not, quietly restart ypbind but don't signal failure.
3. Check for unavailable Lustre OSTs. If any are found, quietly down up the data network interfaces but don't signal failure. This test is only performed during periodic health checks.
4. Purge all user processes and block SSH logins if there are no active jobs (no failures signalled).

This test is only performed during periodic health checks.

Infiniband checks

1. Verify that the Infiniband devices are present.
2. Verify the link status and reported data rate for the expected devices.

GPU-specific checks

1. Run the **/usr/bin/nvidia-smi** utility (included in the driver package) and verify:
   - the reported number of GPUs is 4
   - air intake and per GPU temperature readings are less than 35 and 75 deg C respectively
   - fan, PSU and LED status are all OK.
2. Run the **deviceQuery** SDK example program and verify that for each GPU:
   - there is 4GB of global memory
   - the kernel run time limit is set to 'No'
   - the test result is 'PASSED'.

C.2.2 Job prologue script

All popular schedulers provide the option of running a prologue script, which executes immediately prior to the execution of the job script on the execution host. The pbs_mom daemon will execute a custom prologue script as root just before a job sent to the node is executed. If the node is the master node in the set of nodes allocated to the job, the script */var/spool/PBS/mom_priv/prologue* is run; the remaining nodes instead run */var/spool/PBS/mom_priv/prologue.parallel*. For Darwin nodes, these two files are symbolic links to the same script.

The aim of a prologue script is to verify the node's readiness to run the job, and to perform any necessary preparation. Since in our configuration, only one job at any time runs on a given node, these checks can be relatively aggressive, can assume exclusive access to devices, and can take whatever steps are required to restore a proper state without worrying about disturbing current user processes.

Another function of the prologue script is to set up interactive access to the node via SSH for the owner of the job.

The GPU node prologue script performs the same checks as the health check script, plus an additional set. A failed check causes global failure unless indicated, in which case the pbs_mom daemon is shut down (by the script) after a short delay. This in turn causes Moab to record node failure and to requeue the job. If the prologue does not return after a configurable delay of 5 minutes, the node is also marked down (by Torque). The additional checks performed are described below:

Basic checks

1. Does a NIS lookup succeed? If not, restart ypbind, retry and signal failure if there is still no success.
2. Check all network filesystems are listed as mounted.
3. Check all network filesystems report usage figures with df.
4. Purge all user processes belonging to users other than the owner of the job and allow the owner interactive access via SSH (no failures signalled).
5. Delete all files from local node storage that do not belong to the owner of the job (no failures signalled).

GPU-specific checks

Note that to minimise variation these are performed via numactl so as to lock the tests onto a particular CPU core with only local memory allocation, i.e. via

```
numactl --physcpubind=3 --membind=0 check_script
```

1. Use **/usr/bin/nvidia-smi** to ensure that the mode of each GPU is correct:
   - either all Exclusive (load balancing between GPUs, but only one context each)
   - or all Normal (no load balancing, but contexts can share a GPU).

At the moment, all GPUs are placed into the Exclusive mode, as this works best for our current mix of applications. By extending the idea of consumable GPU resource we will probably soon allow jobs to select the best mode through the resource requirement string.

2. Run the **bandwidthTest** SDK example program and verify that for each GPU:
   - host to device bandwidth is as expected (4860MB/sec with 1% tolerance)
   - the test result is 'PASSED'.
3. Run the **matrixMul** SDK example program and verify that for each GPU:
   - computational performance is as expected (86.6Gflop/s with 1% tolerance)
   - the test result is 'PASSED'.

BandwidthTest measures the speed at which data can be copied from host memory to GPU global memory. Performance should be around 4000MB/sec, however initially we found it necessary to allow a large tolerance (20%) as results could easily vary in this range. This situation has improved markedly with the CUDA 3 compatible drivers and we have been able to tighten this to 4860MB/sec, or less by no more than 1%, but it is important to pass the *–memory=pinned* option to *bandwidthTest* for consistency. The *matrixMul* (linear algebra) sample program was initially even more variable, and for consistent results it was necessary to arrange for the script to perform the test three times and take the maximum value, in addition to allowing a tolerance of 20%. Also we found at first that as the GPUs were used, matrixMul performance consistently dropped until the prologue test always failed – a reboot was required in these cases to restore normal behaviour. The situation here is again much improved with the CUDA 3 driver software, although the occasional outlying result from this test means that we have retained the triple testing in the prologue script. It is possible that the gradually declining performance observed initially was due to increasing numbers of remote memory accesses as local memory slowly became unavailable over time, and that the wrapping of the tests inside numactl would have been sufficient to solve the problem (this was an improvement made later). The current criterion for matrixMul success is at least 86.6Gflop/s, or less by no more than 1%.

C.2.3 Job epilogue script

The pbs_mom daemon in torque, as with the execution-host daemons associated with other scheduling systems, will also execute a custom epilogue script as root, just after a job exits. As in the case of the prologue, there are two forms of the epilogue script: if the node is the master node in the set of nodes allocated to the job, the script */var/spool/PBS/mom_priv/epilogue* is run, whereas the remaining nodes instead run */var/spool/PBS/mom_priv/epilogue. parallel.* For Darwin nodes, these two files are again symbolic links to the same script.

The main aim of an epilogue script is to perform any extra steps that may be necessary to restore the node to a state in which it is fit to accept the next job. Note that the purging of rogue processes takes place in the health check script and prologue, rather than in the epilogue - this is designed to allow a user accessing the node interactively a window in which they may cleanly log out, or commence a new job, without experiencing a sudden hiatus. Only the GPU nodes in Darwin run a non-trivial epilogue script, specifically to purge the Tesla hardware. Again, since in our configuration only one job at any time is running, the epilogue script need not protect against affecting jobs still in progress.

The epilogue runs two utilities, both freely downloadable from NCSA:

cuda_memscrubber

This is distributed as part of the **cuda_wrapper** package. It allocates all available GPU device memory in order to clear any data put there by the preceding job. This is highly desirable in order to ensure that private data cannot leak to other users. It is invoked with

```
cuda_memscrubber -g gpu.txt
```

where *gpu.txt* contains a device number per line: 0, 1, 2, 3.

cuda_memtest

This utility runs a quick stress test on each GPU and its global memory in order to verify the continued correct operation of both. This will detect, for example, a GPU left in a bad state by the previous job. It will also attempt to verify the correct functioning of each global memory by writing patterns and reading them back. It is invoked with

```
cuda_memtest --stress --num_passes 1 --num_iterations 100
```

D. Production experiences

At the time of writing (December 2010), the Darwin GPU computational unit has been running user code since March 2010. The code run the most (by some margin) has been *turbostream*, a structured multi-block flow solver for flows in turbomachines developed by Tobias Brandvik, Graham Pullan and John Denton at the Whittle Laboratory at the University of Cambridge. This is a multi-node, MPI code using four CPU cores and four GPUs per node, and has to date accumulated around 400.000 (CPU) core hours of run time on Darwin. Several other projects are actively developing CUDA code on a smaller scale.

Initially, the GPU nodes were using CUDA 2.3 compatible drivers and libraries. Stability during this period was a notable issue, and it was common for nodes to reboot between jobs, either because the epilogue script registered a problem, or because the matrixMul test's performance during the prologue had become consistently poor. Earlier driver versions have also been reported to occasionally leave GPUs in an inoperable state, which could be worked around by reloading the kernel driver. These problems are not observed now that the CUDA 3 drivers and libraries are in use.

With regard to hardware reliability, we have seen four faults in a batch of 32 S1070 units after nine months of use: two S1070s no longer detect all four GPUs, and another two generate memory errors during cuda_memtest.

E. Conclusions and future developments

In our introduction of a Tesla-based, GPU computational unit to the Darwin cluster we have found it particularly important to have good health checking and prologue/epilogue scripts to identify nodes in a sub-optimal state. The need for this at present is greater than for a normal compute node with only Infiniband hardware added, although there has been a notable improvement in stability in going from the CUDA 2.3 to CUDA 3 compatible drivers. In addition, there are special concerns such as the need to ensure data security by wiping GPU memory between jobs.

The policies enforced on Darwin, specifically the restriction of only one job at any one time on a node, simplify the issue of efficient resource allocation, which would be critical in a situation in which several jobs, perhaps belonging to different users, would need to be granted access to a subset of CPU cores, main memory and GPUs concurrently. On the host side, the natural approach would be to create disjoint cpusets for each job, perhaps in combination with numactl to optimise the locality of memory and cores. There is some support in recent versions of Torque for allocating cpusets on NUMA systems. The analogous approach with Tesla GPUs would be to make use of the features of NCSA's cuda_ wrapper library, which overrides CUDA calls made by jobs in such a way that only a particular subset of GPUs is visible to the application. Although there is no counterpart of CPU-memory locality on the GPU side (because of the way entire grids of CUDA thread blocks are dispatched, all running the same CUDA kernel, one at a time to a given GPU), there is still an issue of non-uniform memory bandwidth when copying data between different CPU cores and a fixed GPU. This non-uniformity can be measured directly: using bandwidthTest bound to each CPU core in turn (with memory allocated locally) via numactl, the device-to-host and host-to-device bandwidths to the memory in a fixed GPU can be ascertained, e.g. the command-line

```
numactl --physcpubind=0 --localalloc \
./bandwidthTest  --device=0 --dtoh --memory=pinned --noprompt
```

reports bandwidth in MB/s from GPU device 0 to core 0. It is clear from Table 1 that physical socket 1 (as seen from /proc/cpuinfo, this socket corresponds to cores 0-3) has a reduced bandwidth to and from the GPU devices relative to socket 0 (cores 4-7).

Table 1.
CPU/GPU affinity as measured by *bandwidthTest*. Socket 1 (cores 0-3) has reduced bandwidth to the GPUs.

Direction of transfer	Cores 0-3 (MB/sec)	Cores 4-7 (MB/sec)
device to host	1950	3400
host to device	4860	5530

The cuda_wrapper library also provides a mechanism for binding processes to CPU cores with optimal bandwidth to the allocated GPUs. Exploring this as a way of improving code performance, in conjunction with cpusets and numactl, is an item on our agenda.

The current version of CUDA contains limited internal support for resource sharing, through the concept of compute modes (see Table 2).

Table 2. GPU compute modes supported by CUDA

Compute Mode	Description
Normal	Kernels from multiple host threads may run on the GPU device. If no particular device is specified, this can lead to multiple threads contending for the default device (device 0).
Exclusive	Only one host thread may run kernels on the GPU device at any given time, however if no particular device is specified and the default device is unavailable , another device is tried. Thus this provides simple load balancing, but no oversubscription of GPUs is possible.
Prohibited	No CUDA kernels may run on the GPU device.

Some codes, e.g. turbostream, require the use of *exclusive* mode, and this is the mode currently set on the Darwin GPU nodes. Other codes perform better if allowed to oversubscribe GPUs, which would require *normal* mode; a minor modification to our Moab/Torque configuration and prologue scripts would be needed to incorporate this possibility. Ideally, there would be an additional mode in which different processes were allowed to share GPUs if necessary, but those not specifying a particular GPU device would be sent automatically to the least busy device; for situations involving a single job per node, this would probably be sufficient.

Finally, we hope to be in a position to test some Tesla 20-series ("Fermi") units soon – these have ECC memory (unlike the 10-series units described here) and significantly improved double-precision floating point performance (515 Gflop/s compared to 78Gflop/s).

A. Appendix A – Hardware and Software configurations

The hardware is described briefly in the introduction section to this paper. The T5500 workstations were chosen because, at the time, these were the only possibilities for Nehalem servers equipped with both twin PCIe Gen2x16 interfaces (to accept the interface cards required by the S1070) and one 8-way PCIe slot as required by the Infiniband card. The workstations were racked horizontally (4U each), with one Tesla S1070 unit each (1U) sitting directly on top of the server and eight 5U workstation/Tesla systems were mounted in a single 42U rack, with the whole solution being housed across four racks. A fifth rack contained the Mellanox QDR switches (creating a non-blocking Infiniband cluster across the 32 nodes), and a 1Gbit/s ethernet switch providing connection to the management/data ethernet network, completed the physical configuration (see Figure 1).

A.1. Bios settings

The BIOS on the T5500 nodes is version A04. The following settings were chosen:

1. Boot order: NIC first
2. NUMA (no memory interleaving)
3. Logical processors disabled
4. Turbomode enabled
5. Cstates enabled

A.2. Software configuration

All nodes use Scientific Linux 5 as their primary operating system, provisioned using a modified ClusterVisionOS stack (image-based). The new features of the GPU nodes required that the images include the appropriate Mellanox-modified OpenFabrics software stack plus proprietary low-level mlx4_ib driver for the ConnectX2 IB card, and of course the proprietary NVIDIA CUDA-enabled graphics drivers for the Tesla GPUs (as downloaded from the respective vendor web sites).

The current OpenFabrics stack is OFED-1.5.1-mlnx9 from Mellanox, on top of which GPU jobs requiring MPI use either Intel MPI 4.0.0.028 or MVAPICH2 1.5.1p1. The version of Lustre used is 1.8.4, initially connecting to the shared storage via the tcp driver over gigabit ethernet, and later via o2ib over QDR Infiniband.

At the time of installation, the current NVIDIA drivers were version 190.53, supporting CUDA 2.3. At the time of writing, we are using version 260.19.12 (supporting CUDA 3.2).

A.3. NVIDIA software installation

Three packages are required (freely downloadable from the NVIDIA web site):

- Developer drivers for 64-bit Linux
  - packaged as devdriver_3.2_linux_64_260.19.12.run
  - contains hardware driver and runtime libraries.
- CUDA toolkit for 64-bit RedHat Enterprise Linux 5.5
  - packaged as cudatoolkit_3.2.9_linux_64_rhel5.5.run
  - contains CUDA development tools and libraries.
- GPU computing SDK code samples
  - packaged as gpucomputingsdk_3.2_linux.run
  - contains sample codes (graphical and non-graphical).

A.3.1. Developer drivers

The package replaces some libraries installed by the distribution's native mechanism, thus it is important to cleanly remove the NVIDIA package before applying distribution updates that might affect these libraries, e.g. any update RPMS of the form xorg-x11-*. Furthermore, at least the kernel module must be rebuilt whenever a new kernel is installed.

The drivers should be installed on each GPU node – depending on the cluster management software employed this might mean performing the procedure explicitly on each node, or (as in the case of Darwin) performing it once on the appropriate image.

In the following we assume that a clean install of the NVIDIA driver package is required, either because it is being updated, or because we wish to apply distribution updates that might interact with it. Firstly, remove any existing driver package by running the uninstall utility, if it exists:

```
# nvidia-uninstall
```

This should return the system to a state consistent with the expectations of the distribution package management software (yum and rpm in the case of RedHat Enterprise/Scientific Linux/CentOS). At this point, any distribution updates or new kernels should be installed in the usual way. Next, run the embedded driver installation script. If (as in our case) you are running this inside an image via chroot, the script may try to build a kernel module against the incorrect kernel version – the correct version can be forced by specifying it as follows (here our GPU node kernel is 2.6.18-194.17.1.el5.darwin):

```
# sh devdriver_3.2_linux_64_260.19.12.run --kernel-name=2.6.18-194.17.1.el5.darwin
```

A series of curses screens then appear, displaying the license, warnings and options. If running inside an image, you may receive a warning about there being no supported card present – ignore this. Respond 'Yes' to installing the 32-bit libraries. The support libraries and X11 components are then installed, and a kernel module built against the specified kernel.

A.3.2. CUDA Toolkit and SDK

The remaining packages (the toolkit and the SDK code samples) can be installed on a shared network filesystem. On Darwin an appropriate location was */usr/local/Cluster-Apps/cuda/3.2* (/usr/local being a NFS mount). This installation was not performed as root:

```
$ sh cudatoolkit_3.2.9_linux_64_rhel5.5.run
Enter install path (default /usr/local/cuda, '/cuda' will be appended):
/usr/local/Cluster-Apps/cuda/3.2
```

Respond negatively to the question about uninstalling previous versions if, like us, you prefer to have several versions available accessible via modules.

Having installed the toolkit, one needs to set up the appropriate environment – the module file used on Darwin is appended below, however the same environment variables (PATH, MANPATH, CPATH, FPATH, LIBRARY_PATH and LD_LIBRARY_PATH) can be set directly in the user's shell initialization scripts:

```
#%Module -*- tcl -*-
##
## dot modulefile
##
proc ModulesHelp { } {
  puts stderr "\tAdds NVIDIA CUDA 3.2 (and gcc 4.3.3) to your environment.\n"
  puts stderr "\tThe SDK may be copied from:"
  puts stderr "\t/usr/local/Cluster-Apps/cuda/3.2/NVIDIA_GPU_Computing_SDK\n"
}
module-whatis "adds NVIDIA CUDA 3.2 (and gcc 4.3.3) to your environment"
module add gcc/4.3.3
set             root            /usr/local/Cluster-Apps/cuda/3.2/cuda
setenv          CUDA_INSTALL_PATH $root
prepend-path    PATH            $root/bin:$root/computeprof/bin
prepend-path    MANPATH         $root/man
prepend-path    CPATH           $root/include
prepend-path    FPATH           $root/include
prepend-path    LIBRARY_PATH    $root/lib64:$root/lib
prepend-path    LD_LIBRARY_PATH $root/lib64:$root/lib:$root/computeprof/bin
```

Having initialized the user environment (either manually or by loading a module such as the one above) the SDK can be installed:

```
$ mkdir /usr/local/Cluster-Apps/cuda/3.2/NVIDIA_GPU_Computing_SDK
$ sh gpucomputingsdk_3.2_linux.run
Enter install path (default ~/NVIDIA_GPU_Computing_SDK):
/usr/local/Cluster-Apps/cuda/3.2/NVIDIA_GPU_Computing_SDK
```

The final step is to actually build the code samples – note that this needs to be done on a GPU node on which the driver package has been installed:

```
cd /usr/local/Cluster-Apps/cuda/3.2/NVIDIA_GPU_Computing_SDK/C
make x86_64=1
```

The compiled code samples are installed into

```
/usr/local/Cluster-Apps/cuda/3.2/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
```

Some of these (e.g. fluidsGL, smokeParticles) produce graphical output – sadly it appears to be no longer possible to configure a node with a Tesla attached to perform off-screen, hardware-accelerated OpenGL, so there is little point trying to run these on a GPU node. Other purely command-line utilities such as deviceQuery, bandwidthTest and matrixMul are useful for node health checking, as described above.