An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications using CUDA*

Allen D. Malony Scott Biersdorff
Wyatt Spear
Department Computer and Information Science
University of Oregon
Eugene, OR 97403
{malony,scottb,wspear}@cs.uoregon.edu

Shangkar Mayanglambam
Qualcomm Corporation
3765 Tamarack Ln, Santa Clara CA 95051-0804
shangkar.meitei@gmail.com

ABSTRACT

Heterogeneous parallel systems using GPU devices for application acceleration have garnered significant attention in the supercomputing community. However, to realize the full potential of GPU computing, application developers will require tools to measure and analyze accelerator performance with respect to the parallel execution as a whole. A performance measurement technology for the NVIDIA CUDA platform has been developed and integrated with the TAU parallel performance system. The design of the TAUcuda package is based on an experimental NVIDIA CUDA driver and associated runtime and device libraries. In any environment where the CUDA experimental driver is installed, TAUcuda can provide detailed performance information regarding the execution of GPU kernels and the interactions with the parallel program without any modification to the program source or executable code. The paper describes the TAUcuda technology and how it is integrated with the TAU measurement framework to provide integrated performance views. Various examples of TAUcuda use are presented, including CUDA SDK examples, a GPU version of the Linpack benchmark, and a scalable molecular dynamics application, NAMD.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

Performance tools, GPGPU, profiling, tracing

*

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

1. INTRODUCTION

There is growing interest in heterogeneous computing platforms based on multi-core accelerators to improve the performance of parallel applications. However, achieving the full potential of these machines will be challenging due to complexity of the multi-core hardware, the few heterogeneous programming tools available, and the multiple factors involved in optimizing performance overall. Significant promise has been shown in the use of general purpose GPUs (GPGPUs) for application acceleration. The CUDA system was created to support the programming of GPU-based accelerators from NVIDIA and is now being touted as a key component for a heterogeneous architecture. However, few tools exist to help the parallel application developer measure and understand accelerator performance. Performance analysis tools for GPGPU developers to date have been largely oriented towards aiding developers on individual workstation-class machines with limited parallelism present within the GPU host. However, when used in large-scale parallel environments, it is important to understand the performance of accelerators, such as CUDA-programmed GPUs, in the context of the whole parallel program's execution. This will require the integration of GPU accelerator measurements with existing scalable performance tool infrastructures.

This paper describes an experimental approach to CUDA performance measurement and its integration in a parallel performance toolkit. The approach is experimental because it is based on a non-production version of the NVIDIA CUDA driver. This particular driver implements a callback API specifically to support tool use. We have developed a package called *TAUcuda* that builds on the NVIDIA driver library and callback interface to enable the capture of CUDA performance events in profile and trace forms. TAUcuda is merged with the TAU Performance System® [16] to allow events from the CUDA system and GPU operation to be measured and analyzed in the context of the parallel application as a whole.

Several challenges had to be overcome in TAUcuda development, mainly due to the concurrent, asynchronous execution model supported by the CUDA system. The detailed design and implementation of TAUcuda is presented in Section 2. How TAUcuda is integrated with the TAU performance measurement and analysis environment is discussed in Section 3. Several experiments were conducted to evaluate TAUcuda's capabilities, from simple CUDA SDK examples to scientific applications running on GPU clusters.

Results from these experiments are shown in Section 4. Related work is covered in Section 5. We conclude the paper with a discussion of future directions.

2. TAUCUDA DESIGN AND IMPLEMENTATION

The CUDA system architecture supported by NVIDIA consists of a CUDA *driver* library and a *device* library. The driver library implements the CUDA execution model and provides an API for application software to interface with the CUDA system. The driver library routines are named with a "cu" prefix and enable the launching of GPU kernels, copying of memory between the host and device, and other driver operations. The device library is responsible for low-level interfacing with the GPU device and is mainly called within the CUDA system.

In addition to the CUDA driver and device libaries, NVIDIA also provides the CUDA runtime library that functions as a higher-level programming API. The runtime library routines are prefixed with "cuda" and support thread, stream, device, control, memory, and error operations, as well as interoperability with graphics interfaces. A C-style and C++-style interface is available.

The fundamental challenge for the TAUcuda design is to be able to observe GPU memory and kernel operations and to associate the GPU performance information with the application running on the CPUs. This is complicated by the fact that GPU operation occurs asynchronously from CPU activity. Furthermore mechanisms different from CPU performance measurement must be used for the GPU. Fundamentally, the CUDA system has to provide some support to observe GPU operation. The role of TAUcuda then is to use this support and integrate GPU performance measurement with the TAU measurement infrastructure running as part of the application on the CPU. Figure 1 shows the architecture of the TAUcuda design and its integration with the CUDA system and TAU.

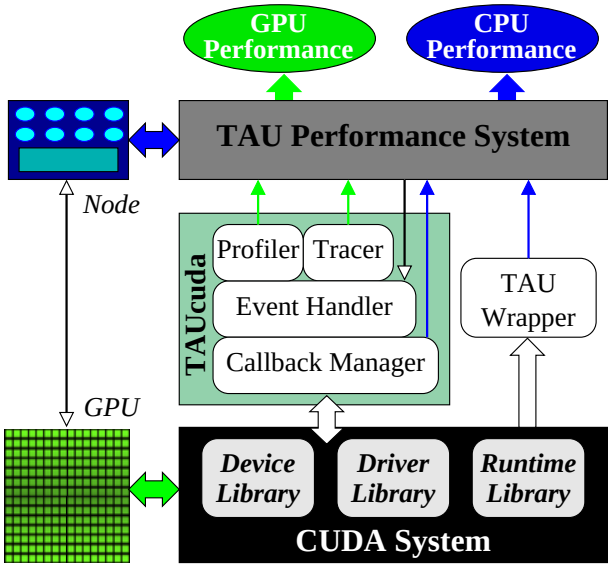

Figure 1: TAUcuda architecture design.

2.1 Using the experimental NVIDIA CUDA driver

NVIDIA created a version of the Linux CUDA driver library (R190.36) that supports an experimental interface for tools. The interface implements a callback mechanism that allows all driver routines to be monitored. The interface also exposes functions that enable GPU measurements to be retrieved from the CUDA system. With this support, together with driver header files provided by NVIDIA, TAUcuda is implemented as follows:

The CUDA driver library contains a routine (*cuDriverGetExportTable()*) which returns handles to interface tables based on a global unique identifier (GUID). These interface tables are used for different purposes. The CUDA version supporting a callback interface for tools defines a table which can be seen by supplying GUID *cuToolsApi_ETID_Core*. The core table returned exposes all the interfaces necessary to control the callback setup. At startup, TAUcuda first calls a construct member function of the core table to initialize it. After the construct call, TAUCuda then registers the callback handler by calling the *SubscribeCallbacks()* function and passing its callback handler function.

After callback setup, CUDA driver routines are routed through the callback handler. TAUCuda then intercepts only the routines of interest using GUIDs filtering, such as:

cuToolsApi_CBID_EnterGeneric
cuToolsApi_CBID_ExitGeneric
cuToolsApi_CBID_ProfileLaunch
cuToolsApi_CBID_ProfileMemcpy

The callback handler dispatched with:

cuToolsApi_CBID_EnterGeneric
cuToolsApi_CBID_ExitGeneric

enables TAUCuda to intercept the driver API routines for TAU instrumentation at the entry/exit points of the call. Section §2.2 discusses this in more detail. The *Profile* callback interfaces relate to the recording of GPU measurements and are delivered at CUDA context synchronization or when the GPU measurement buffer is full. (The term "profile" is used in the CUDA system to refer to GPU measurements it makes internally.) In this way, TAUcuda gains access to the GPU measurements. Section §2.3.1 describes how TAUcuda processes this data.

It is important to understand that callbacks execute in the application thread calling the driver library. For each thread receiving a CUDA driver *EnterGeneric* callback for the first time, the *TAUcuda Event Handler* will initialize data structures for managing CUDA performance measurements with respect to that application thread.

2.2 TAU instrumentation of CUDA driver and runtime libraries

Through the use of the CUDA driver *ToolsApi* inteface, TAUcuda is able to translate *EnterGeneric* and *ExitGeneric* callbacks into TAU entry/exit performance events. This is accomplished by calling the TAU event creation and measurement routines (start/stop) for the particular CUDA driver routine (*cuXXX()*) identified in the callback. While it is possible to create a TAU event for every *cuXXX()* routine, only a subset may be of interest. The routines instrumented in our experimental version of TAUcuda are listed in Table 1.

cuLaunch();	*cuLaunchGrid();*
cuLaunchGridAsync();	*cuMemcpyHtoD();*
cuMemcpyHtoDAsync();	*cuMemcpy2D();*
cuMemcpy2DUnaligned();	*cuMemcpy2DAsync();*
cuMemcpy3D();	*cuMemcpy3DAsync();*
cuMemcpyAtoA();	*cuMemcpyAtoD();*
cuMemcpyAtoH();	*cuMemcpyAtoHAsync();*
cuMemcpyDtoA();	*cuMemcpyDtoD();*
cuMemcpyDtoH();	*cuMemcpyDtoHAsync();*
cuMemcpyHtoA();	*cuMemcpyHtoAAsync()*

Table 1: CUDA driver routines instrumented by TAUcuda to produce TAU events.

In addition to instrumenting CUDA driver routines, we wanted to generate TAU events for the CUDA runtime library (*cudaYYY()*). However, NVIDIA does not implement callback support for these routines. Instead, TAU's library wrapping utility, *tau_wrap*, was used to generate a TAU-instrumented CUDA runtime wrapper that could be interposed with the actual library. Because NVIDIA provides only the CUDA runtime library header files, the source code can not be instrumented directly. Instead, *tau_wrap* read the library's interface by parsing the header files, and automatically generating a library that redefines the routines specified. Internally, each routine in the wrapper library searches for the corresponding call from the CUDA runtime library and calls the routine with the appropriate arguments. The wrappers are instrumented to make calls to the TAU library before and after calling the wrapped runtme library routines. In this way, when a CUDA runtime routine is called during execution, the TAU instrumentation is invoked at entry and exit to generate performance measurements for the associated routine. (Note, this mechanism to instrument the CUDA runtime library and generate TAU events during execution requires the wrapper library to gain control at startup.) TAU's event selection methods can be used to filter which of the CUDA runtime library routines are actually instrumented.

For the TAU events created for both CUDA driver and runtime routines, the TAU measurement system is responsible for generating performance profile or trace data. TAU can collect the number of calls, CPU timing information, and CPU performance counters for each event on every CPU thread where they occur.

2.3 Instrumentation and measurement of CUDA GPU operations

While the wrapping of the CUDA runtime and the callback support in the Linux CUDA (R190.36) driver makes it possible for entry and exit of CUDA library routines to be seen by the TAU system, the challenge for TAUcuda is to capture performance events and measurements for the GPU kernal execution and memory transfer operations. These GPU operations take place asynchronously to execution of the application code on the CPU. TAUcuda works with the CUDA system to address this complexity.

The *cuToolsApi_CBID_EnterGeneric* callback occurs for *cuXXX()* driver routines that invoke GPU kernel launch and memory transfer operations. The CUDA system manages these operations and uses the device layer to coordinate their execution on the GPU. It also collects performance measurements for each operation. The TAUcuda *Event Handler* will create an *call record* for every operation occurrence in which it will record the *event name*, *call ID*, *operation type*, *API routine name*, *TAU context*, *CUDA context*, *GPU device*, and *GPU stream*. Most of the call record fields are retrieved from the CUDA driver routine reflected in the callback. The event name is the *cuMem()* API name for memory transfers or the GPU kernel name in the *cuLaunch()* routine. The call ID is a monotonically increasing value incremented for each *cuXXX()* routine invocation. The GPU device is extracted by using the *CtxGetDevice* interface of the context table and the CUDA context.

To associate the GPU operation to the application code that invokes the CUDA driver routine, the *TAUcuda Event Handler* call into the TAU system to retrieve the current TAU event stack during the *EnterGeneric* callback. The TAU event stack essentially represents the nested application events that TAU is currently measuring. This is the *TAU context* and is stored in the call record as a string of concatenated TAU event names.

There can be several concurrently active GPU operations for different CUDA contexts. Thus, TAUcuda can be handling multiple outstanding call records at any time. When any GPU operation completes, the CUDA system will capture performance data for the operation. However, it is not until the CUDA system initiates a *Profile* callback that TAUcuda will have the opportunity to see this data and create a TAU performance measurement. *Profile* callbacks are dispatched some time after the actual *cuLaunch()* or *cuMem()* APIs are called. The callback delivers performance counters associated with a particular call ID and CUDA context. The list of call records is then scanned by TAUcuda to match the unique (call ID, CUDA context) pair.

2.3.1 TAUcuda profiling

If TAUcuda is in profiling mode, it invokes the *TAUcuda Profiler* to create a profile event for any unique tuple combination defined by (event name, TAU context, GPU device, stream ID). Every encounter of the same tuple reuses the profile event for that tuple. The performance values returned by the *Profile* callback are used to calculate profile statistics by the TAUcuda Profiler. Once this is done, the Event Handler is informed to delete the call record.

The TAUcuda Profiler calculates multiple metrics for profile events. The *elapsed time* metric is valid for both kernel launch and memory transfer activities of the GPU. The CUDA system returns the GPU time and the elapsed time which is a sum of GPU time for all instances of the same profile event. For memory transfers, a *memory transfer size* metric is computed as the accumulated amount of bytes transferred. There are different GPU counters associated to every kernel execution event. TAUCuda currently supports only the default counters, including *shared memory usage*, *registers per thread*, and *core occupancy*. These metrics are generated for kernel events only. TAUcuda writes out the profiles in TAU profile format when the CUDA application exits. A separate profile is created with the different profile metrics for every GPU device and stream on which GPU operation occurred. The proile writer routine for the elapsed time metric automatically adds a top level event by computing the sum of all the time elapsed in the entire GPU computation.

2.3.2 TAUcuda tracing

Implementing tracing in TAUCuda is more complex. For each GPU operation, two trace events need to be generated: one for when the operation begins and one for when the operation ends. Both events are generated when the *TAUcuda Tracer* handles the *Profiler* callback. The same TAUcuda data structures described above are used for keeping track of the GPU operations. The TAUcuda traces are generated for memory transfer and kernel operations. Information from the call record about the operation is placed in the trace event records and is used during trace post-processing to match TAU trace events from the application.

One difficulty is in generating the timestamp to put in the trace records. The timestamps generated for the TAUCuda trace events must be from the TAU time reference, which comes from the CPU. However, the time information from CUDA is from the GPU. Thus, the CPU and GPU clocks must be synchronized. To do this, TAUcuda performs two experiments on startup. First, it captures start and end time values for all active GPU devices for a known interval. The *DeviceGetTimestamp* interface exposed by the device table is used to extract the device timestamp. Second, it repeats the experiment for the TAU clock on the CPU device 0 (the reference GPU). With a table containing TAU start/end time, reference GPU start/end time, and other GPU start/end time, TAUcuda can create a TAU-synchronized timestamp for each GPU operation on any GPU device during trace event generation.

TAUCuda tracing also produces special trace events for GPU memory transfers. It is desirable to see the memory transfer actions in both the CPU and GPU traces. Thus, memory transfers from CPU to GPU are represented by a pair of events related by the unique tuple (CUDA context, call ID). The CPU-side event would represent when the memory transfer started and the GPU-side event when the transfer ended. The CPU-side event must be placed in the TAU application trace, but it is impossible to do so at runtime. Why? This is because TAUcuda does not get the necessary information about the GPU operation performance until after the operation completes! Although TAUcuda has all the data to generate the trace event, it can not be properly put in the TAU trace until trace post-processing. Similarly, memory transfer from the GPU to the CPU is also represented by a pair of trace events, one in each CPU and GPU trace stream. In this case, TAUcuda must produce an event representing when the memory transfer completed in the TAU CPU trace. Presently, the timestamp used for this event is assigned the timestamp when context synchronization occurs. Our primary goal in doing so was to represent the data receive from the perspective of the earliest point of synchronization in the application after which the memory might be used. Further improvements to reflect the actual timeline of memory transfer can be made.

TAUCuda event traces are written out in the TAU trace format with the help of the TAU trace writing API. TAU trace writer keeps a buffer for the trace events and writes out to the file only when the buffer is full or the trace writer handle is closed. A separate trace file is created for each device and stream. In addition to the events described, other user defined events corresponding to grid dimension, block dimension, shared memory size and registers are created and triggered at runtime.

3. ENVIRONMENT INTEGRATION

The TAUcuda design and development, as well as how it is used in practice, is determined by the experimental CUDA callback API for tools implemented in the NVIDIA R190.36 CUDA driver. One practical consequence is that TAUcuda will not work on a GPU machine unless this particular driver library is installed. The callback architecture and lack of access to the CUDA runtime source code also constrains how TAUcuda is applied. In general, TAUcuda must be integrated in a performance measurement and analysis environment that includes the CUDA platform, TAU system, and other tools.

3.1 Measurement activation

For a performance measurement with TAUcuda to take place, several libraries are required. TAUcuda is built as a dynamic library that will be preloaded at application startup, linking to CUDA driver and device library interfaces and registering to receive driver callbacks. TAUcuda's use of the TAU measurement library requires its loading, which also takes place dynamically. Furthermore, the TAU wrapper library must be loaded to generate events for the CUDA runtime routines, if desired.

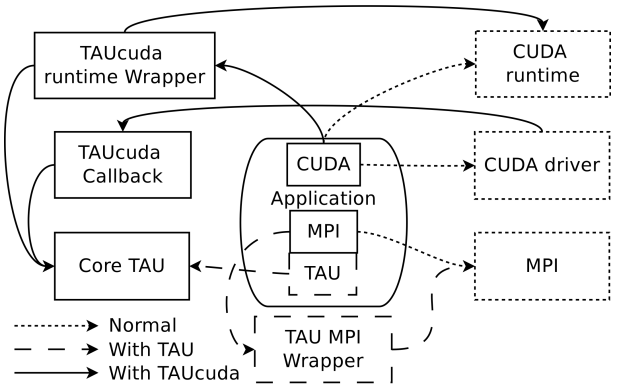

Figure 2: Dependencies between libraries under different measurement scenarios.

Figure2 shows how library dependences are resolved at runtime under different measurement scenarios. If neither TAU nor TAUcuda is enabled, the CUDA application links with the driver, device, and runtime libaries, plus any necessary application libraries, by default (as shown with the dotted lines). If the application is instrumented with TAU, either through source, compiler, or MPI library wrapping, there are dependencies to the TAU core library. In this case, the CUDA application will execute and produce only TAU performance data for the application events that are instrumented. A third case is when TAUcuda is preloaded and additional dependencies (solid lines) involving all libraries arise. Calls made to the CUDA runtime library are intercepted by the TAUcuda runtime wrapper library and the TAUcuda callback library is registered by the CUDA driver and called when GPU operations take place. (Note, it is possible to wrap the CUDA runtime library to get TAU events without linking in TAUcuda. In this case, the entry and exit of the CUDA runtime routines will be seen as TAU events in the profile or trace.)

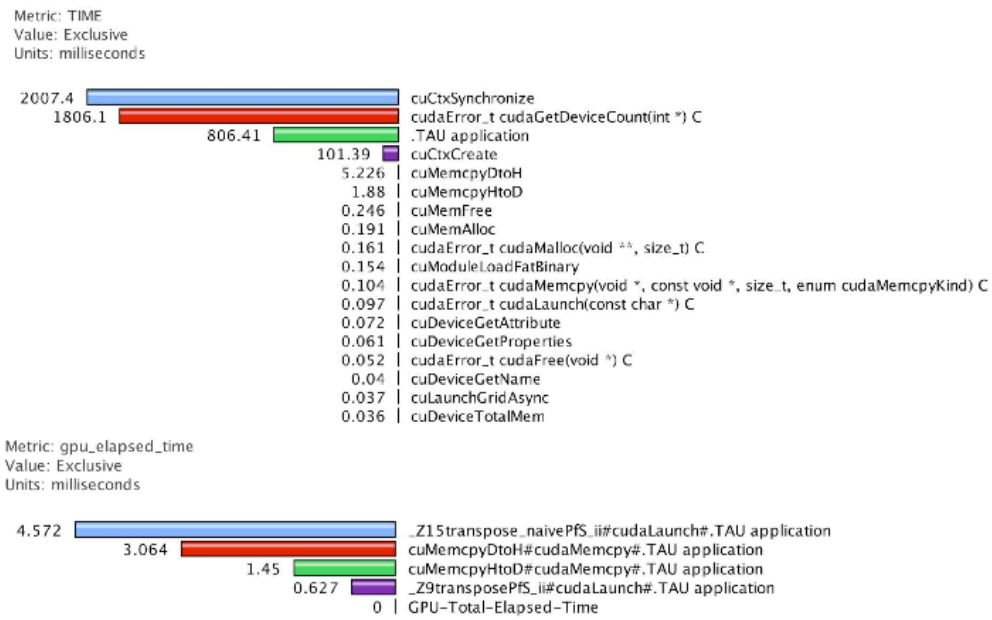

Figure 3: CUDA SDK transpose CPU (top) and GPU (bottom) profile.

3.2 Running with TAUcuda

To run an CUDA application with TAUcuda, all of the necessary libraries must be dynamically linked. The advantage is that TAUcuda can be used on unmodified CUDA application binaries. To simplify the different TAUcuda measurement options, we developed scripts to hide the linking complexities for different scenarios:

Profiling: *taucuda_profiler.sh* / *taucuda_mpirun.sh*
Tracing: *taucuda_tracer.sh* / *taucuda_mpirun_tracer.sh*

TAUcuda produces profiles or traces in the current working directory in sub-folders to distinguish them from TAU performance output. TAUCuda profiles are populated in different metric sub-folders, such as:

gpu_elapsed_time
gpu_memory_transfer
gpu_shared_memory

3.3 Trace post-processing and translation

Trace output from TAUcuda is generally similar to standard TAU trace output. The exceptions lie in the node/thread ID numbering and the representation of communication events. To address both of these issues it is necessary to perform post-processing on the TAUcuda trace output. Once the traces are processed and merged, the final output can be translated to the standard trace formats supported by TAU, allowing trace analysis and visualization tools such as *Jumpshot* [18] (slog2 format) and *Vampir* [4] (OTF [10] format) to be used.

A TAU trace for any given thread consists of two files. A *.trc* file is a binary file containing the time stamped events (e.g., entry, exit, send, receive). A *.edf* file is a text file mapping the actual event names to the numeric identifiers used in the *.trc* file. Each thread of execution produces a *.trc* file, but there is only one *.edf* file per node. It is assumed that the same event IDs will be used for every thread on the same node. However, this assumption does not apply to TAUcuda trace output.

The file formats for TAUcuda traces are the same as standard TAU traces, however *.trc* and *.edf* files are produced for each context and stream pair. Because each CUDA device is distinct in its initialization it is impossible to assign the same event ID's to different CUDA devices accessed from the same CPU node. In this respect each CUDA device acts as a single self contained process. However, they may also be analogous to threads in the case that a CUDA device is invoked by and subordinate to a single CPU process. To represent this relationship in a merged TAUcuda trace, the CUDA traces are treated as their own processes, but the trace IDs are reordered so that any given node's trace output is followed by its thread trace output and then by its CUDA device trace output. This makes it necessary to remap the default assigned node IDs to account for the inserted CUDA nodes, by incrementing the index of each node after a CUDA trace insertion. For example, given trace output for an application with two MPI processes, each accessing one CUDA device, the CPU trace output would initially be assigned node ID's 0 and 1 respectively. The trace post-processor would insert each associated TAUcuda trace after its CPU process, so the new ordering would be CPU trace 0 (Node ID 0), CUDA trace 0 (Node ID 1), CPU trace 1 (Node ID 2), CUDA trace 1 (Node ID 3).

Part of the problem here is in getting the traces in a form that can be translated to formats that trace analysis and visualization tools can read. This can be seen in the special consideration required for TAUcuda communication (i.e., message transfer) events. Messaging events in trace formats use node IDs to designate source (sender) and destination (receiver). At the time of measurement, TAUcuda has no information on the node ID of its host process. Message transfer events between the device and the CPU are instead identified by values unique to the device, stream, and source call site. It is necessary to map the unique TAUcuda

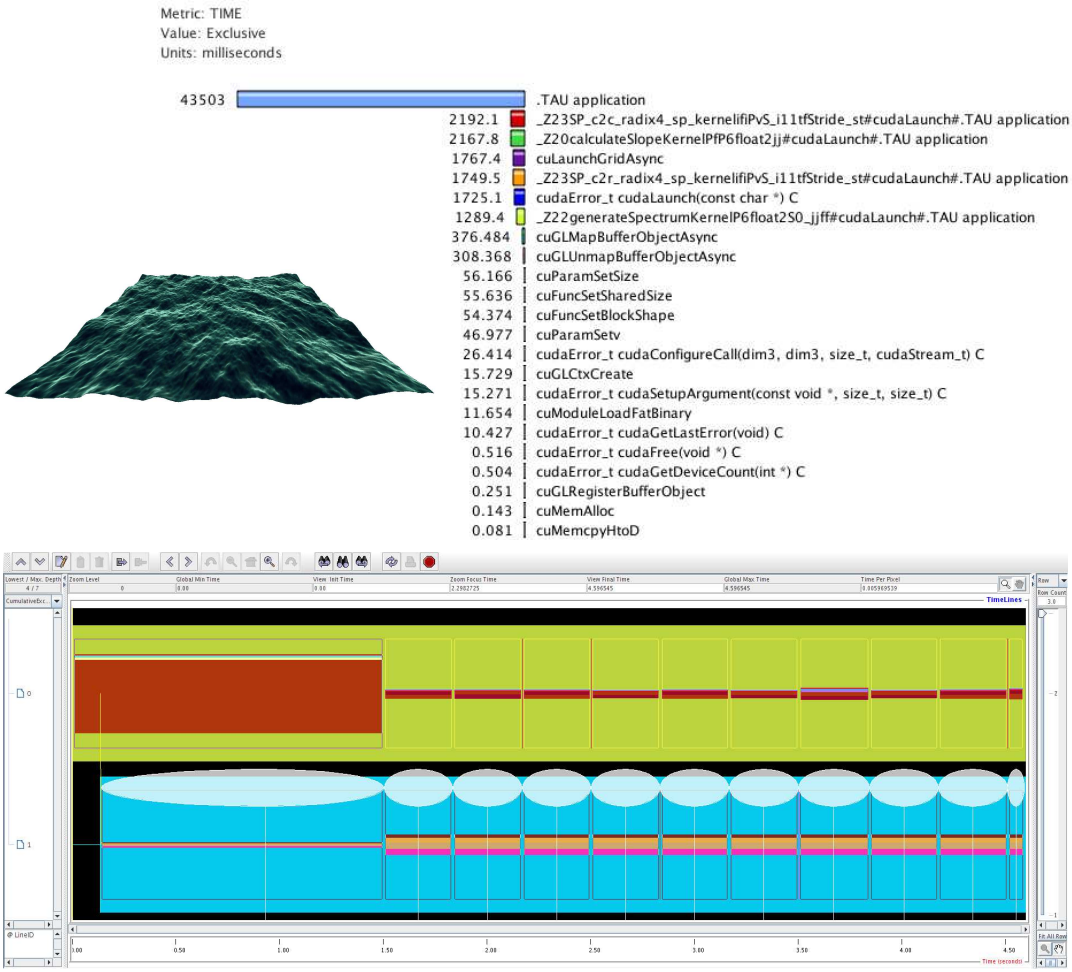

Figure 4: Performance profile and trace of the CUDA SDK ocean simulation.

message IDs to their respective nodes in the post-processing step. Once this mapping is complete, the local and remote node IDs are available for each CUDA communication event as it is encountered in the final pass of the trace translator.

4. TAUCUDA EXPERIMENTS

Using the TAUcuda environment described above for CUDA performance measurements on Linux systems with NVIDIA GPUs, we conducted several experiments to assess the TAUcuda profiling and tracing capabilities. The only system requirement was that the NVIDIA R190.36 Linux CUDA driver library be installed. While this limited the platforms available to us due to administrative control, we were able to test TAUcuda on three systems:

- Linux workstation
  - dual quadcore Intel Xeon
  - GTX 280 GPU
- University of Oregon's GPU cluster (*Mist*)
  - 4 dual quadcore Intel Xeon server nodes
  - 2 S1070 Tesla servers (4 Tesla GPUs per S1070)
- Argonne National Lab GPU cluster (*Eureka*)
  - 100 dual quadcore NVIDIA Quadro Plex S4
  - 200 Quadro FX5600 GPUs per S4

The experiments ranged from example programs distributed with the NVIDIA CUDA SDK to scalable scientific applications run on multi-GPU clusters. The results reported below are mainly from Mist and Eureka.

4.1 CUDA SDK examples

TAUcuda can be used to generate CUDA performance measurements with any CUDA application binary. Thus, we were able to easily test TAUcuda on several of the CUDA SDK example codes, including:

matrixmul, *transpose*, *oceanFFT*, *gaussian*

Figure 3 show the CPU and GPU performance results from the transpose example run on a 256 x 4096 matrix. Two profiles are shown from TAU's ParaProf [2] tool: one for the CPU and one for the GPU. In the CPU profile we see all CUDA driver callback events (`cuXXX`) and runtime library events (`cudaYYY`) invoked during the execution. In the GPU profile we see the two GPU kernels and the memory transfer, host-to-device (HtoD) and device-to-host (DtoH).

The profile and trace of the ocean simulation (oceanFFT) example is shown in Figures 4. The profile display merges CPU and GPU performance data. Notice the `cuGL` routines used for the interactive graphics. The Jumpshot [18] tool was used to generate the trace display with the CPU trace

above and GPU trace below. It shows a memory transfer early on, followed by a series of kernels as the simulation repeats.

4.2 Linpack

Our primary interest in developing TAUcuda was to measure the performance of scalable heterogeneous parallel applications using GPUs for acceleration. To demonstrate this capability, we obtained an GPU-accelerated Linpack benchmark whose performance was reported by Fatica [11] and ran it on the Mist cluster[1]. This provided us with an excellent opportunity to evaluate TAUcuda against a well-known cluster benchmark. The ParaProf profile for a four-process Linpack run on Mist is shown in Figure 5. This shows mean execution time values across the processes for all events. The corresponding Jumpshot trace is shown in Figure 6[2]. The profile shows TAU source instrumentation of the Linpack routines and MPI communication events, in addition to the CUDA driver, CUDA runtime, and GPU and memory transfer events.

The trace display emphasizes the temporal performance behavior of the Linpack execution. The yellow lines show the MPI message communication occuring between the CPU nodes. The white lines highlight the CUDA message transfers taking place. What is interesting to see here is the effect of our algorithm to determine the timestamp of the DtoH message receive event in the CPU trace. This point is set to the time of the next context synchronization based on the rationale that this is the earliest time when the results can be used, from the perspective of the application program.

4.3 NAMD

To demonstrate TAUcuda with a realistic parallel application that is utilizing GPGPU acceleration, we considered the NAMD [7] application. NAMD is a parallel molecular dynamics simulation designed for high-performance simulation of large bimolecular systems. NAMD is developed with the Charm++ [9] framework. Recently, the TAU measurement system has been integrated with Charm++ to enable profiling of Charm++ events [3] and NAMD was used as an evalution testcase. The addition of GPGPU acceleration using CUDA in NAMD makes it also an interesting testcase for TAUcuda.

The computationally intensive part of NAMD involves computing interactions between atoms. Two GPU kernels, *dev_nonbonded* and *dev_sum_forces*, are applied to these computations. These two events map to two different CUDA kernels and both kernels are launched from the same CPU function context *WorkDistrib::enqueCUDA*.

We ran NAMD on a medium sized benchmark on four processes with four Tesla GPUs on Mist. The parallel profile obtained with TAU/TAUcuda is shown in Figure 7. The performance of the two kernels are shown in the GPU device profiles, along with the memory copy driver routines and other distributed work routines in the CPU profiles. The magenta hashed lines are navigational aids in the ParaProf 3D display.

We moved to the Eureka cluster to run NAMD with a larger number of GPUS. Performance experiments were conducted with 4 to 64 GPUs to see the effects of strong scaling. Figure 8 presents the results of the scaling study, showing the relative efficiency of the kernel operations and memory transfers. We chose to plot the performance of four GPUs used in the NAMD execution to indicate how the their efficiencies vary for each kernel. The nonbounded calculations have reasonably good scaling for this model benchmark. The sumforces kernel less so. The memory transfer line is based on the mean value and reflects increasing overheads for smaller data sizes transferred.

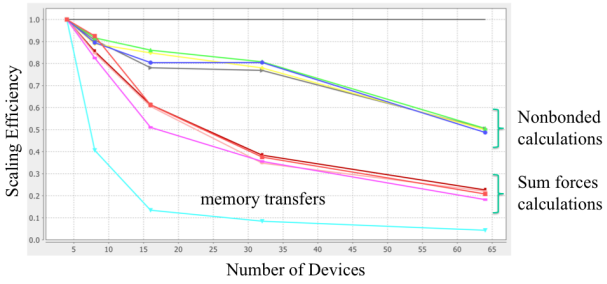

Figure 8: Scaling of GPU kernels and memory transfers in NAMD.

4.4 SHOC Benchmarks

Scalable heterogeneous computing systems are composed of a mix of general multicore processors, application accelerators, and possibly special-purpose devices. The integration of TAUcuda in TAU targets performance measurement for heterogeneous systems with CUDA acceleration. We have already seen this demonstrated with the Linpack experiments. Another opportunity recently presented itself in the form of Oak Ridge National Laboratory's *Scalable HeterOgeneous Computing benchmark suite (SHOC)* [6]. SHOC is a spectrum of programs that test the performance and stability of scalable heterogeneous platforms. Low-level microbenchmarks are used to assess architectural features, and SHOC uses application kernels at higher levels to determine system-wide performance such as intranode and internode communication among devices. Benchmark implementations are in both OpenCL and CUDA in order to compare programming models.

We selected the SHOC Stencil2D benchmark to try with TAUcuda since it contained an interesting combination of MPI, memory transfer, kernel execution, and other events to measure. Stencil2D is a standard two-dimensional nine-point stencil calculation. Shown in 9 is a trace of the application showing the parallel execution of four CPUs along with four GPUs.

5. RELATED WORK

The features in TAUcuda and its integration in a parallel performance system are unique in the heterogeneous computing space where CUDA and GPUs are involved. Earlier research work has pursued similar objectives in computing environments where the IBM Cell Broadband Engine is applied. Trace measurement and analysis of the Cell BE was implemented in the Vampir tracing system and shown to be effective in understanding the runtime behavior of Cell applications [8]. The Paraver [1] performance systems can

[1]It was not possible to run the Linpack program on Eureka because it did not support DGEMM.

[2]We can also generate TAUcuda traces for visualization using Vampir [4].

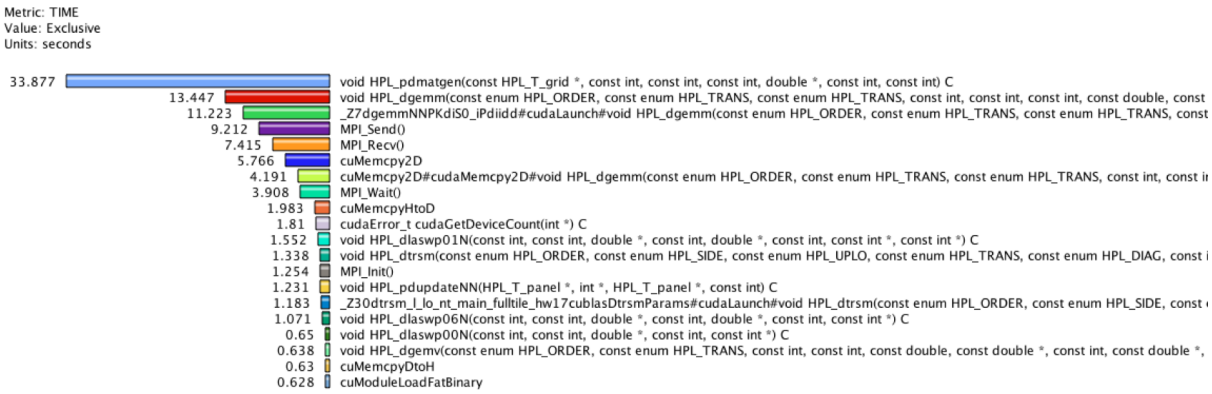

Figure 5: Mean profile of a four-process execution of Linpack with GPU acceleration.

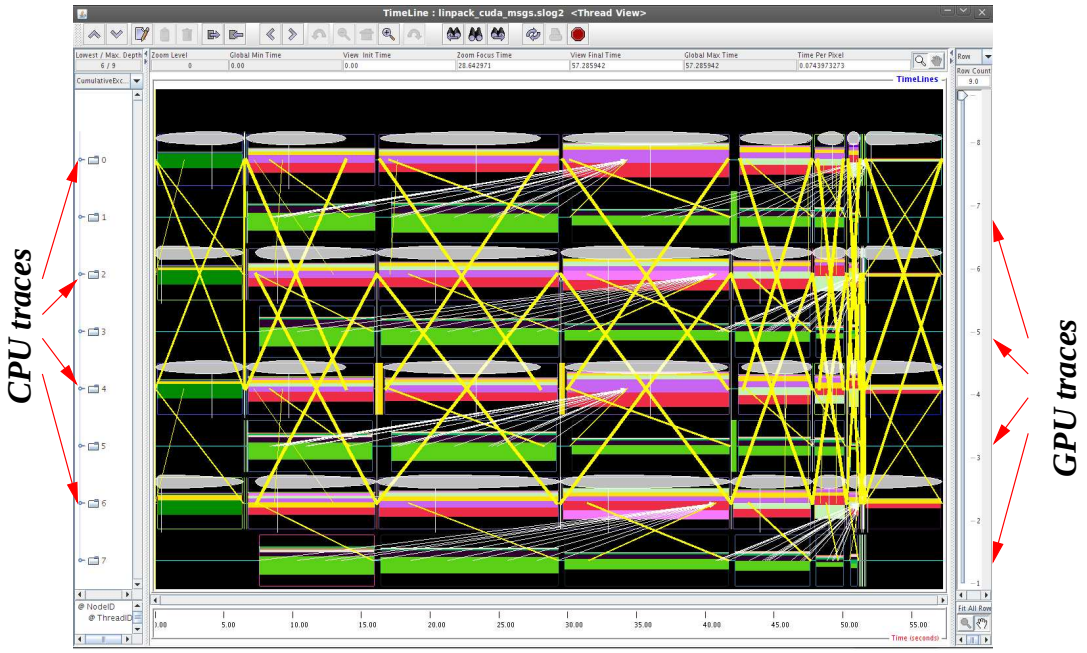

Figure 6: Trace of a four-process execution of Linpack with GPU acceleration.

also measure Cell BE performance as part of a sophisticated parallel programming environment.

The differences with TAUcuda mainly come from working with GPUs and CUDA. We initially considered the use of NVIDIA profiling tools to address the performance measurement problems. NVIDIA has a rich performance SDK known as *PerfKit* [13] for profiling the GPU driver interface. It provides access to low-level performance counters inside the driver and hardware counters inside the GPU itself. However, PerfKit is limited for use with the CUDA programming environment. Also, we need different measurement semantics to capture the CUDA program performance and integrate the data with parallel application performance. NVIDIA also provides the *CUDA Profiler* [14] which includes performance measurement in the CUDA runtime system and a visual profile analysis tool. While the CUDA Profiler provides extensive stream-level measurements, it collects the data in a trace and does not provide access until after the program terminates. We want to be able to produce profiles that show the distribution of accelerator performance with respect to application events. This performance view is difficult to produce with the CUDA Profiler trace data.

Recently, NVIDIA launched Nexus [15], a tool to debug, profile, and analyze GPU code using the standard workflow and tools of Visual Studio 2008. Nexus is a very powerful tool that utilizes the same underlying capabilities in the CUDA driver that TAUcuda does. However, it is available only on Windows. Also, Nexus is not intended for use with heterogeneous parallel applications written using MPI.

An earlier version of TAUcuda was developed based on the CUDA *Event* interface [12]. Defined as part of the CUDA specification and implemented in the runtime library, the Event interface can be used to obtain performance data for a particular GPU stream's execution, including each kernel's precise timing. The performance data is measured by the CUDA runtime system and returned through the CUDA performance measurement library we developed. The main

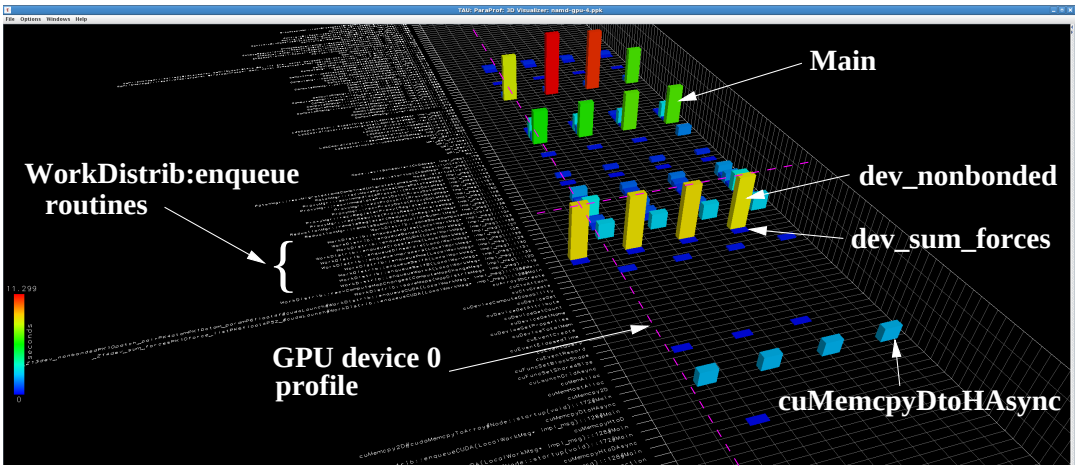

Figure 7: Four process NAMD integrated profile showing GPU kernel and driver events.

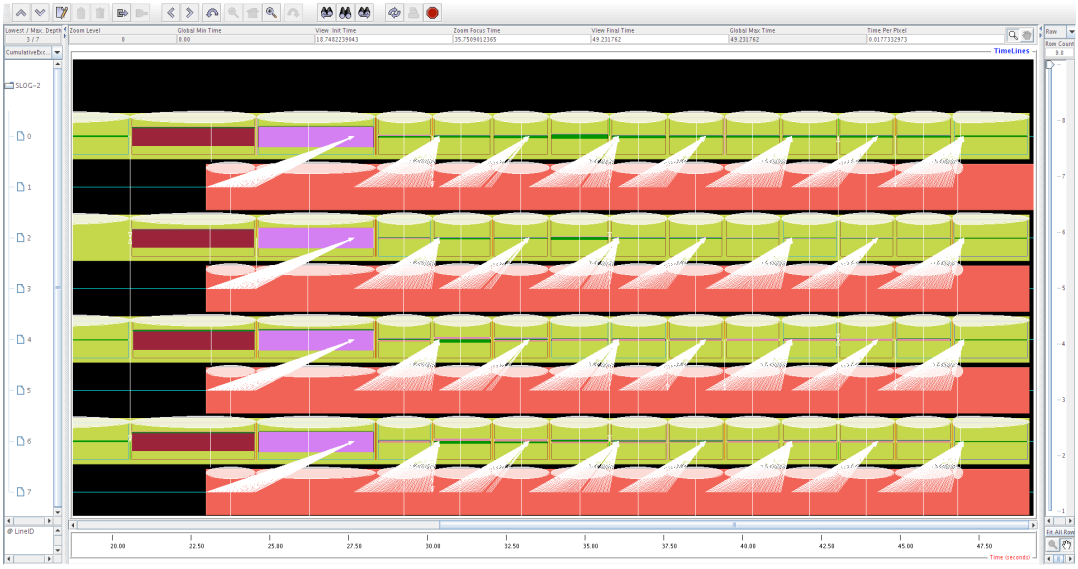

Figure 9: SHOC Stencil2D benchmark, 512 iteration on four cpus and four Tesla GPUs. The pink bars represent GPU execution and the lines show memory transfers from the Device to Host.

problem with this approach was that it required the CUDA application to be instrumented with calls to the Event interface. In addition, it only provided a runtime library view of performance. Although detailed information would be returned about kernel execution, it was in a high level application context. No access to driver-level operation was available and it was not possible to observe memory transfers as a result.

6. CONCLUSION

The TAUcuda design and implementation demonstrates the power of a callback interface to make visible internal CUDA routines and provide a means to build an infrastructure for GPU performance measurement. Its integration in the TAU performance system further leverages the technology for use in heterogeneous parallel systems. A variety of examples of TAUcuda application are given in the paper to highlight the level of detail about GPU performance it can provide and the incorporation of this information in whole application performance views.

That said, it is clear the TAUcuda system described in this paper would not have been possible without access to the experimental Linux CUDA driver from NVIDIA. Furthermore, the prospects forward for practical use of TAUcuda depend significantly on NVIDIA's plans for tools support in a future Linux production version. We are working closely with NVIDIA to help them in this endeavor through the reporting of our experiences in the TAUcuda development and testing[3]. For instance, there are several areas where TAUcuda can be made more efficient with additional functionality in the callback interfaces and internal support in the CUDA system. Our hope is that this feedback will eventually translate to support that any performance tool which

[3]It is hoped that a production version of the Linux NVIDIA CUDA driver will be available by the time of the ICS conference.

wants to observe GPU performance can benefit from in the future.

Additionally we would like to have TAUcuda fully support other GPGPU programing paradigms. A good example is the PGI Accelerator compilers[17] and HMPP compilers[5] both of which provide an alternative method for GPU based acceleration on CUDA compatible hardware. They work on the principle of automated code generation by first performing analysis of the unaccelerated code followed by optimization and remapping of loops to available hardware accelerators. Fine tuning may be accomplished via pragma statements, but CUDA level coding is not necessary.

Currently the TAUcuda system inter-operates transparently with applications compiled with PGI's or HMPP's GPU acceleration[4]. It provides the same GPU-level performance data and uses the same operating procedure as with typical CUDA applications. As TAUcuda evolves, a tighter integration between TAUcuda and these compilers is imagined.

Acknowledgment

The authors would like to thank Will Ramey and Greg Smith of NVIDIA Corporation for providing access to the experimental CUDA driver (R190.36) that made the TAUcuda research possible. Their technical assistance during the work was invaluable in moving the project forward. The research was supported by a NVIDIA Professor Partnership award and grants from the U.S. Department of Energy, Office of Science, under contracts DE-PS02-08ER08-19 and DE-FG02-07ER25826. The research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

7. REFERENCES

[1] Barcelona Supercomputing Center. *Paraver*. http://www.bsc.es/paraver/.

[2] R. Bell, A. Malony, and S. Shende. A portable, extensible, and scalable tool for parallel performance profile analysis. In *European Conference on Parallel Computing (EuroPar 2003)*, 2003.

[3] S. Biersdorff, C. Lee, A. Malony, , and L. Kale. Integrated performance views in charm++: Projections meets tau. In *International Conference on Parallel Processing (ICPP)*, Sept. 2009.

[4] H. Brunst, D. Kranzlmüller, and W. E. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *Distributed and Parallel Systems, Cluster and Grid Computing*, 777, 2004.

[5] CAPS Entreprise. *HMPP Workbench*. http://www.caps-entreprise.com/hmpp/.

[6] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[4]As of this writing TAUcuda does not support PGI accelerated MPI applications. MPI compatibility for PGI acceleration will be included in an upcoming release of TAUcuda.

[7] J. P. el al. Scalable molecular dynamics with namd. In *Journal of Computational Chemistry*, pages 1781 – 1802, Oct. 2005.

[8] D. Hackenberg, H. Brunst, and W. Nagel. Tracing and visualization for cell broadband engine systems. In *European Conference on ParallelProcessing (EuroPar 2008)*, volume LCNS 5168, pages 172–181. Springer, 2008.

[9] L. Kale, E. Bohm, C. Mendes, T. Wilmarth, and G. Zheng. Programming petascale applications with charm++ and ampi. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[10] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In *International Conference on Computational Science (ICCS 2006)*, volume 3992 of *Springer Lecture Notes in Computer Science*, pages 526–533, May 2006.

[11] F. Massimilian. Accelerating linpack with cuda on heterogeneous clusters. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 46–51, Mar. 2009.

[12] S. Mayanglambam, A. Malony, and M. Sottile. Performance measurement of applications with gpu acceleration using cuda. In *International Conference on Parallel Computing (ParCo)*, Sept. 2009.

[13] NVIDIA Corporation. *NVIDIA Performance Toolkit*, da-01800-001v03 edition, May 2006.

[14] NVIDIA Corporation. *NVIDIA CUDA Visual Profiler*, 1.1 edition, 2007.

[15] NVIDIA Corporation. *NVIDIA Nexus*, 2009. http://developer.nvidia.com/nexus/.

[16] S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[17] STMicroelectronics. *PGI Accelerator Compilers*. http://www.pgroup.com/resources/accel/.

[18] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *High Performance Networking and Computing (SC00)*, Nov. 2000.