Performance Measurement of Applications with GPU Acceleration using CUDA

Shangkar MAYANGLAMBAM [1] and Allen D. MALONY and Matthew J. SOTTILE
Dept. of Computer and Information Science, University of Oregon
Eugene, OR 97403
{smeitei,malony,matt}@cs.uoregon.edu

Abstract. Multi-core accelerators offer significant potential to improve the performance of parallel applications. However, tools to help the parallel application developer understand accelerator performance and its impact are scarce. An approach is presented to measure the performance of GPU computations programmed using CUDA and integrate this information with application performance data captured with the TAU Performance System. Test examples are shown to validate the measurement methods. Results for a case study of the GPU-accelerated NAMD molecular dynamics application application are given.

1. Introduction

There is growing interest in the use of multi-core accelerators to improve the performance of parallel applications, with GPU computing devices gaining the most traction. Achieving the performance potential of accelerators is challenging due to complexity of the multi-core hardware and their operational/programming interface. In the case of general purpose GPUs (GPGPUs), CUDA was created to support program development targeting GPU-based accelerators. 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. When used in large-scale parallel environments, it is important to understand the performance of accelerators (such as with a CUDA measurement library when GPUs are used) in the context of whole parallel program's execution. This will require the integration of accelerator measurements in scalable parallel performance tools.

This paper describes our approach to performance measurement of GPGPU execution using CUDA in the context of a larger parallel performance measurement environment. We consider the problem from the point of view of a parallel application where host (CPU-side) performance measurement already has robust support, in our case from the TAU Performance System [3]. The goal is to measure the performance of GPU com-

[1] Corresponding E-Mail: smeitei@cs.uoregon.edu

putational kernels, wherever they are invoked in the application, and integrate the measurements with the TAU parallel performance data. Methods developed for CUDA performance measurement are presented and the TAU CUDA measurement interface is described. Test examples are shown to validate the measurement model. Results for a case study of a GPU-accelerated molecular dynamics application are given.

2. CUDA Performance Model and TAUcuda Approach

The CUDA programming environment enables easy development of applications with GPU acceleration of certain components. Computationally intensive parts of applications can be launched as tasks into the GPU device. Measuring performance of CUDA application would appear straightforward for simple usages. However, the concurrent and asynchronous model of CUDA programs relative to the GPU host makes it problematic to create an accelerator performance view in general with respect to the performance of the parallel application as a whole. Programmers can use multiple concurrent CUDA streams to queue independently executable sequences of GPU tasks. Furthermore, different strategies can be used to overlap CPU and GPU execution, as well as CPU-GPU data transfers. There can also be multiple GPU devices accelerating different parts of the application for different CPU threads. To understand and optimize GPU-accelerated parallel applications using CUDA, all these scenarios are of interest and important performance factors should be measured, such as GPU utilization and CPU waiting time, but standard parallel performance tools can not be applied directly.

There are two general approaches to GPU performance measurement. First, we could consider making the measurement on the CPU (host) side. If the GPGPU is used in a *synchronous* manner (the CPU immediately waits for GPGPU execution to finish), we could just place measurement points before and after launching the GPU kernel to determine performance. If the GPGPU is used in an *asynchronous* (the CPU does not immediately wait), the measurements could still be done in this way, but it would be difficult to determine exactly when the GPGPU completed execution. In CUDA, a GPU *kernel* is launched on a *stream* and multiple kernels on the same stream run sequentially. However, multiple streams can be concurrently active and multiple GPU devices can be used. In such cases, the performance measurement becomes even more complex. For instance, consider the four simple scenarios shown in Figure 1. The top row shows the synchronous cases for one and two streams. The bottom row shows the asynchronous cases. Notice in the synchronous case for two streams, it will be difficult to extract the performance for each stream independently.

We initially considered the use of NVIDIA profiling tools to address the performance measurement problems. NVIDIA has a rich performance SDK known as *PerfKit* [4] 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. 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* [5] 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.

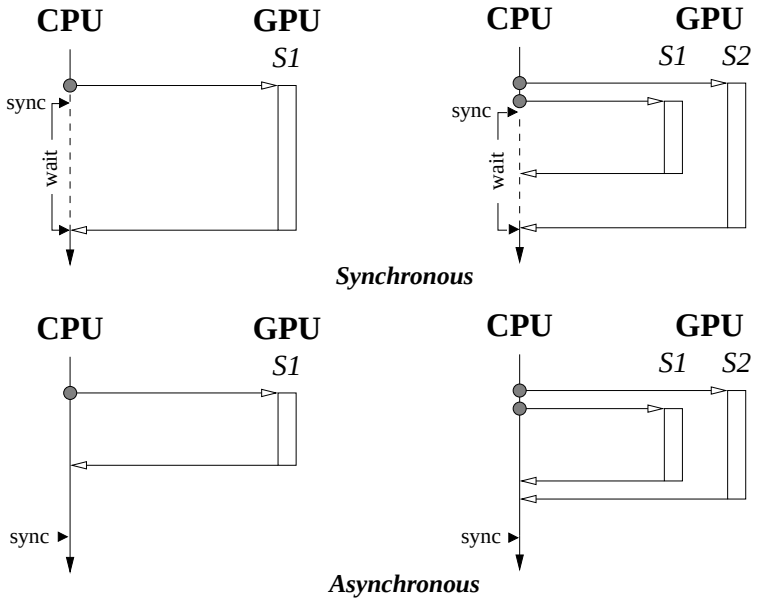

Figure 1. Scenarios of Host-GPGPU use.

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.

On the other hand, CUDA provides an *event* interface that can be used to obtain performance data for a particular stream's execution, including each kernel's precise termination. The performance data is measured by the CUDA runtime system. We developed a CUDA performance measurement library based the stream event interface called *TAUcuda*. The TAUcuda approach is described below and the library API is presented in section §3.

Consider the execution of a single GPU kernel execution on a stream. We can use a CUDA *begin* event and *end* event placed around the CUDA statement to measure the begin time and end time of the kernel's execution in the GPU. When the begin and end events are retrieved from the CUDA runtime system, TAUcuda can calculate the kernel's elapsed execution time. In addition, TAUcuda can calculate the *waiting* time from when it requested the CUDA events to when they were delivered and the *finalize* time from the beginning. The integration with the TAU measurement system occurs with the begin event when the *TAU context* (most recent TAU performance event) is sampled and stored.

Of course, many GPU kernels can be executed during an application on multiple streams and GPU devices. TAUcuda events are kept per stream and per GPU device. In addition, TAUcuda events can be nested. At the end of execution, the TAUcuda measurement library creates performance profiles for each event showing the *event_name*, *tau_context*, *device_id*, *stream_id*, *#_calls*, *inclusive_time*, *exclusive_time*, *wait_time*, and *finalize_time*.

3. TAU CUDA Measurement Interface

The TAUcuda measurement library implements a versatile interface for the application programmer to measure performance for GPGPU computations. A TAUcuda object is created for each block of CUDA code to be measured. At the core of the library, CUDA event objects play a vital role in tracking GPGPU computation time. Internally the TAUcuda objects map to two CUDA event objects which record *begin* and *end* execution times for the code block, as measured by the GPU clock. The CUDA event objects are scheduled in the GPGPU by calling the *cudaEventRecord* interface and specifying the corresponding stream of execution. However, it is imperative that the two events are scheduled immediately before and after the CUDA code block.

CUDA provides both blocking and non-blocking interfaces to check the event status. Correspondingly, TAUcuda also exposes both blocking and non-blocking interfaces to gather the event data and compute performance profiles. The TAUcuda data structures are managed for every CPU thread independently in thread local memory. This avoids the overhead of ensuring thread safety and also works the around limitations of CUDA event objects. CUDA event objects are not reliably accessible outside the scope of their originating thread lifetime. Hence, the profile data needs to be processed and written out before each thread exits.

The TAUcuda library interface shown in Table 1. The initialization interface *tau_cuda_init* should be called at the start of the application. It initializes data structures and sets up the initialization time for both CPU and GPGPU. To process and write TAUcuda profiles, each thread must call *tau_cuda_exit* before exiting. TAUcuda enables programmers to choose the granularity of CUDA code block observation. The *tau_cuda_stream_begin* and *tau_cuda_stream_end* interfaces are used to mark the begin and end TAUcuda events in application source code. The *event* name token passed to the begin call identifies the TAUcuda object associated with the GPGPU computations enclosed. The library also provides an interface, *tau_cuda_update*, which returns a vector of completed event statistics for a single stream or all streams at any time. The call to this interface returns without blocking. On the other hand, *tau_cuda_finalize* performs the similar action except that it waits for outstanding events to complete. Both interfaces free up the CUDA event objects after processing them. Frequent use of these interfaces is recommended in experiments with large number of profile events.

4. Examples

4.1. Computational Scenarios

We evaluated the TAUcuda measurement library with scenarios of computation varying the interactions between CPU, CUDA streams, and GPGPU devices. All experiments were performed on a NVIDIA Tesla S1070 GPU server. The single stream experiment in Table 2 illustrates how a TAUcuda profile can detect CPU cycles wasted in waiting for GPGPU computations to complete. We observe decreases in wait time with more utilization of CPU in parallel with the GPGPU computation. Similar results are observed for experiments in Table 3 with two CUDA streams executing on a single GPGPU device. We can also see the proportional inclusive computation time for variations of the computation loads in the streams.

void tau_cuda_init(int argc, char **argv) ○ To be called when the application starts ○ Initializes data structures and checks GPU status
void tau_cuda_exit() ○ To be called before any thread exits at end of application ○ All CUDA profile data is output for each thread of execution
void* tau_cuda_stream_begin(char *event, cudaStream_t stream) ○ Called before CUDA statements to be measured ○ Returns handle which should be used in the end call ○ If event is new or the TAU context is new for the event, a new CUDA event profile object is created
void tau_cuda_stream_end(void * handle) ○ Called immediately after CUDA statements to be measured ○ Handle identifies the stream ○ Inserts a CUDA event into the stream
vector<Event> tau_cuda_update() ○ Checks for completed CUDA events on all streams ○ Non-blocking and returns # completed on each stream
int tau_cuda_update(cudaStream_t stream) ○ Same as tau_cuda_update() except for a particular stream ○ Non-blocking and returns # completed on the stream
vector<Event> tau_cuda_finalize() ○ Waits for all CUDA events to complete on all streams ○ Blocking and returns # completed on each stream
int tau_cuda_finalize(cudaStream_t stream) ○ Same as tau_cuda_finalize() except for a particular stream ○ Blocking and returns # completed on the stream

Table 1. TAUcuda measurement interfaces.

CPU Load	GPU Load	Event	Inclusive Time	Wait Time
0	X	Interpolate (C-[main|]#D-0#S-0)	75222.4922	75134.7656
0	2X	Interpolate (C-[main|]#D-0#S-0)	150097.7031	149995.6094
0	3X	Interpolate (C-[main|]#D-0#S-0)	225034.2031	224915.5312
Y	X	Interpolate (C-[main|]#D-0#S-0)	74985.6953	64097.1680
2Y	X	Interpolate (C-[main|]#D-0#S-0)	75058.5234	42563.9648
10Y	X	Interpolate (C-[main|]#D-0#S-0)	75032.9609	0.0000

Table 2. TAUcuda profiles for a single stream (time measured in milliseconds).

In multi-GPU experiments, individual CPU threads launch computations on corresponding devices. Table 4 shows experiments with two GPGPU devices. *D-0* and *D-1* are device identifiers and are included in the TAUcuda event names to identify the corresponding CPU threads using the device. We observe meaningful results for inclusive time as well as the wait time appropriate to the computational size.

More sophisticated profile results are shown in Table 5, demonstrating how TAUcuda can capture the CPU context (from the concurrent TAU measurement layer) in which the GPGPU computation is launched. The information is again encoded in the

	GPU Load		Time Measured (in milliseconds)		
CPU Load	S-1	S-2	Event	Inclusive Time	Wait Time
0	2X	X	Interpolate (C-[main|]#D-0#S-1)	149982.8750	149858.8906
0	2X	X	Interpolate (C-[main|]#D-0#S-2)	74929.6953	74909.6719
0	X	2X	Interpolate (C-[main|]#D-0#S-1)	74993.2188	74869.6250
0	X	2X	Interpolate (C-[main|]#D-0#S-2)	150055.8750	150019.0469
Y	X	X	Interpolate (C-[main|]#D-0#S-1)	75054.0156	53687.0117
Y	X	X	Interpolate (C-[main|]#D-0#S-2)	74989.4688	53708.9844
2Y	X	X	Interpolate (C-[main|]#D-0#S-1)	74899.1406	32293.9453
2Y	X	X	Interpolate (C-[main|]#D-0#S-2)	74948.7344	32429.6875
5Y	X	X	Interpolate (C-[main|]#D-0#S-1)	75007.4219	0.0000
5Y	X	X	Interpolate (C-[main|]#D-0#S-2)	75008.5469	0.0000

Table 3. TAUcuda profiles for two streams.

CPU Load		GPU Load		Time Measured (in milliseconds)		
D-0	D-1	D-0	D-1	Event	Inclusive Time	Wait Time
0	0	X	2X	Interpolate (C-[main|]#D-0#S-0)	75068.2500	74855.4688
0	0	X	2X	Interpolate (C-[main|]#D-1#S-0)	149795.0156	149698.7344
0	0	2X	X	Interpolate (C-[main|]#D-0#S-0)	150171.8750	150054.6875
0	0	2X	X	Interpolate (C-[main|]#D-1#S-0)	74969.5625	74892.5781
2Y	Y	X	X	Interpolate (C-[main|]#D-0#S-0)	75121.7266	53530.7617
2Y	Y	X	X	Interpolate (C-[main|]#D-1#S-0)	75864.0938	18769.0430
Y	2Y	X	X	Interpolate (C-[main|]#D-0#S-0)	75119.8750	53557.1289
Y	2Y	X	X	Interpolate (C-[main|]#D-1#S-0)	75123.8984	18204.1016

Table 4. TAUcuda profiles for two devices.

Event	Calls	Inclusive Time	Exclusive Time
All-Interpolate (C-[FirstWrapper|]#D-0#S-0)	1	300019.9375	65.3992
InterpolateA (C-[FirstWrapper|]#D-0#S-0)	10	150013.6250	150013.6250
InterpolateB (C-[FirstWrapper|]#D-0#S-0)	10	149940.8750	149940.8750
All-Interpolate (C-[SecondWrapper|]#D-0#S-0)	1	300111.6250	65.0635
InterpolateA (C-[SecondWrapper|]#D-0#S-0)	10	150018.1719	150018.1719
InterpolateB (C-[SecondWrapper|]#D-0#S-0)	10	150028.3750	150028.3750

Table 5. TAUcuda profiles with two tau contexts and nested events.

TAUcuda event name by *[FirstWrapper]* and *[SecondWrapper]*, representing two different CPU function contexts. We also see here the calls field which accounts for the repetitive access of the TAUcuda object for the stream, device, and context. Again, CUDA computation can be profiled at the level of the programmer's choice of granularity and nesting of TAUcuda profile events. The exclusive time measure is included only in this table as it is meaningful with nested events.

To verify the TAUcuda performance values, we turned on the *CUDA Runtime Profiler* [5] functionality to dump elapsed time measures for all the kernels and other memory related GPGPU tasks. The profiling feature is integrated in CUDA runtime system

Measurement Scenarios		Inclusive Time (in milliseconds)	
Event	GPU Load	TAUcuda	CUDA Profiler
Interpolate (C-[main|]#D-0#S-0)	X	75065.9844	75045
Interpolate (C-[main|]#D-0#S-0)	2X	150012.6094	150067
Interpolate (C-[main|]#D-0#S-0)	3X	225058.2500	224950
Interpolate (C-[main|]#D-0#S-0)	4X	300173.8438	299928
Interpolate (C-[main|]#D-0#S-0)	5X	374917.5625	374887

Table 6. TAUcuda versus CUDA Runtime Profiler.

and a visual profile analysis tool is provided. Table 6 shows CUDA Profiler values together with the TAUcuda data. The results are seen to be very close.

4.2. Profiling Inside CUDA Kernels

We have also prototyped measurement interfaces for use inside a CUDA kernel. However, the performance data managed by these interfaces is not yet fully integrated to the TAUcuda system. Collecting profile data from device address space requires a good approach to limit the profiler memory usage. Our approach transmits out chunks of performance data from device to the host CPU. The kernel profile data is managed with a data structure called the *TAU Data Unit* (TDU) frame. The TDU frame has a header segment which contains fields to communicate with the device. The host CPU can inform the device about the frame structure and the device can inform the status of profiling to the host. Due to the high cost of writing out profile records in the GPGPU global memory, we manage a cache buffer of shared memory for manipulating the profile records. Further details of kernel measurement will be produced after fully integrating it into the TAUcuda system.

4.3. Application Case Study

To demonstrate TAUcuda with a realistic parallel application that utilizes GPGPU acceleration, we considered the NAMD [2] application. NAMD is a parallel molecular dynamics simulation built with the Charm++ framework. The TAU measurement system has recently been integrated with Charm++ to enable profiling of Charm++ events [1]. NAMD has been programmed for GPGPU acceleration using CUDA. We use the *tau_cuda_stream_begin* and *tau_cuda_stream_end* interfaces to capture TAUcuda profiles for certain CUDA code in NAMD, namely *dev_nonbonded* and *dev_sum_forces* GPU kernels. We ran NAMD on four MPI processes each using a Tesla GPU on our S1070 server.

The TAUcuda profiles generated are displayed shown in Figure 2 together with the TAU profile for the four MPI processes. We can see that the GPGPU computation time for each event is almost uniform across all 4 processes. However, *dev_nonbonded* inclusive time is much higher than that of *dev_sum_forces*. These two events maps to two different CUDA kernels and both kernels are launched from the same CPU function context *WorkDistrib::enqueCUDA*. In this experiment, the performance is improved by about 4 times compared to the computation without GPGPU acceleration.

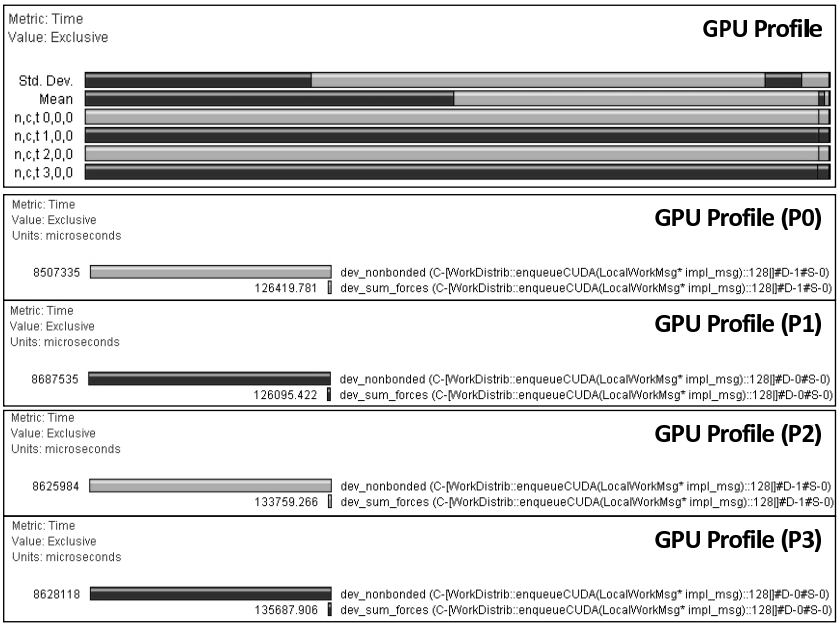

Figure 2. TAUcuda profiles for NAMD.

5. Conclusion

We have developed a profiling system for measuring and integrating performance data on both the host CPU and the GPU kernel components of a CUDA application. This work describes and demonstrates performance measurement techniques for parallel programs using acceleration technologies such as GPUs. With the increased presence of accelerator technologies in conventional parallel computers such as clusters, integration of performance measurement on acceleration devices within the overall parallel application is critical for maintaining a complete picture of large scale parallel program performance. Our initial work described in this paper forms the basis for accelerator performance measurement being integrated into current and future versions of the TAU performance analysis framework.

References

[1] S. Biersdorff, C.W. Lee, A. Malony, and L. Kale. Integrated Performance Views in Charm++: Projections Meets TAU. In *International Conference on Parallel Processing*, September 2009. To appear.

[2] J. Phillips el al. Scalable molecular dynamics with namd. In *Journal of Computational Chemistry*, pages 1781 – 1802, October 2005.

[3] A. Malony, S. Shende, A. Morris, S. Biersdorff, W. Spear, K. Huck, and Aroon Nataraj. Evolution of a Parallel Performance System. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *2nd International Workshop on Tools for High Performance Computing*, pages 169–190. Springer-Verlag, July 2008.

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

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