Identifying Optimization Opportunities Within Kernel Execution in GPU Codes

Robert Lim(✉), Allen Malony, Boyana Norris, and Nick Chaimov

Performance Research Laboratory, High-Performance Computing Laboratory, University of Oregon, Eugene, OR, USA
{roblim1,malony,norris,nchaimov}@cs.uoregon.edu
http://tau.uoregon.edu

Abstract. Tuning codes for GPGPU architectures is challenging because few performance tools can pinpoint the exact causes of execution bottlenecks. While profiling applications can reveal execution behavior with a particular architecture, the abundance of collected information can also overwhelm the user. Moreover, performance counters provide cumulative values but does not attribute events to code regions, which makes identifying performance hot spots difficult. This research focuses on characterizing the behavior of GPU application kernels and its performance at the node level by providing a visualization and metrics display that indicates the behavior of the application with respect to the underlying architecture. We demonstrate the effectiveness of our techniques with LAMMPS and LULESH application case studies on a variety of GPU architectures. By sampling instruction mixes for kernel execution runs, we reveal a variety of intrinsic program characteristics relating to computation, memory and control flow.

1 Introduction

Scientific computing has been accelerated in part due to heterogeneous architectures, such as GPUs and integrated manycore devices. Parallelizing applications for heterogeneous architectures can lead to potential speedups, based on dense processor cores, large memories and improved power efficiency. The increasing use of such GPU-accelerated systems has motivated researchers to develop new techniques to analyze the performance of these systems. Characterizing the behavior of kernels executed on the GPU hardware can provide feedback for further code enhancements and support informed decisions for compiler optimizations.

Tuning a workload for a particular architecture requires in-depth knowledge of the characteristics of the application [19]. Workload characterization for general-purpose architectures usually entails profiling benchmarks with hardware performance counters and deriving performance metrics such as instructions per cycle, cache miss rates, and branch misprediction rates. This approach is limited because hardware constraints such as memory sizes and multiprocessor cores are not accounted for and can strongly impact the workload characterization. Moreover, the current profiling methods provide an overview of the

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 185–196, 2015.
DOI: 10.1007/978-3-319-27308-2_16

behaviors of the application in a summarized manner without exposing sufficient low-level details.

Performance tools that monitor GPU kernel execution are complicated by the limited hardware support of fine-grained kernel measurement and the asynchronous concurrency that exists between the CPU and GPU. With so many GPUs available, identifying which applications will run best on which architectures is not straightforward. Applications that run on GPU accelerators are treated like a black box, where measurements can only be read at the start and stop points of kernel launches. Moreover, the difficulty of tracking and distinguishing which tasks are associated with the CPU versus the GPU makes debugging heterogeneous parallel applications a very complicated task. Thus, analyzing static and dynamic instruction mixes can help identify potential performance bottlenecks in heterogeneous architectures.

1.1 Motivation

Heterogeneous computing presents many challenges in managing the diverse architectures, high-speed networks, interfaces, operating systems, communication protocols and programming environments. For GPUs, more computational units exist over memory, and PCI bus transfers are limited in latency and capacity (fixed GB/sec). Thus, applications that provide parallelism opportunities will benefit most on GPUs. Algorithms with efficient partitioning or mapping strategies are needed to exploit heterogeneity, while syntax directives such as OpenMP and OpenACC facilitate in program productivity. Tools need to be able to measure the individual heterogeneous components to assess the application's performance behavior.

Performance measurements for GPUs are typically collected using the event queue method [14], where an event is injected into the stream immediately before and after the computation kernel. Performance frameworks such as TAU, PAPI, Intel VTune and NVIDIA nvprof provide this capability [1–3,17], where regions of code are annotated with start/stop calls surrounding kernel execution.

Hardware performance counters are often used to monitor application performance, where measurements can be collected through either instrumentation or sampling. Drawbacks of using hardware performance counters include overcounts of results, lack of support across architecture vendors, incompatibilities of events and counters, limited number of hardware counters, and inability to pinpoint transient behavior in program runs [13,18].

In Fig. 1, we show a time series of hardware counters sampled in the GPU, a capability we've added in TAU, and kernels that were executed for the LULESH application. The plot reveals spikes in the hardware samples for the application. However, one cannot correlate those spikes to the dense regions of activities in source code. If timestamps were used to merge GPU events with CPU events for purposes of performance tracing, the times will need to be synchronized between host and device [5], as the GPU device has a different internal clock frequency than the host. Using timestamps to merge profiles may not be sufficient, or even correct. Thus, optimizing and tuning the code would require a best guess effort

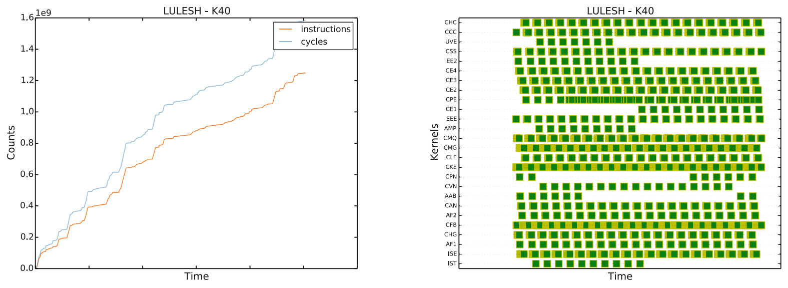


Fig. 1. Sampled hardware counters of instructions executed and active cycles (left) and individual kernel executions (right), both for LULESH (Color figure online).

of where to begin. This motivates our exploration of the use of instruction type mixes in aiding the analysis of potential performance bottlenecks.

1.2 Contributions

In our work, we perform static analysis on CUDA binaries to map source text regions and generate instruction mixes based on the CUDA binaries. We define instruction mix as the types of operation codes in GPU programming [12]. This feature is integrated with TAU to sample region runs on the GPU. We also provide visualization and analysis to identify GPU hotspots and optimization opportunities. This helps the user better understand the application's runtime behavior. In addition, we repeatedly sample instructions as the application executes. To the knowledge of the authors, this work is the first attempt at gaining insight on the behavior of kernel applications on GPUs *in real time*. With our methodology, we can also identify whether an application is compute-bound, memory-bound, or relatively balanced.

2 Background

The TAU Parallel Performance Framework [17] provides scalable profile and trace measurement and analysis for high-performance parallel applications. TAU provides tools for source instrumentation, compiler instrumentation, and library wrapping that allows CPU events to be observed. TAU also offers parallel profiling for GPU-based heterogeneous programs, by providing library wrappings of the CUDA runtime/driver API and preloading of the wrapped library prior to execution. Each call made to a runtime or driver routine is intercepted by TAU for measurement before and after calling the actual CUDA routine.

TAU CUPTI Measurements. TAU collects performance events for CUDA GPU codes asynchronously by tracing an application's CPU and GPU activity [14]. An activity record is created, which logs CPU and GPU activities. Each event kind (e.g. CUpti_ActivityMemcpy) represents a particular activity.

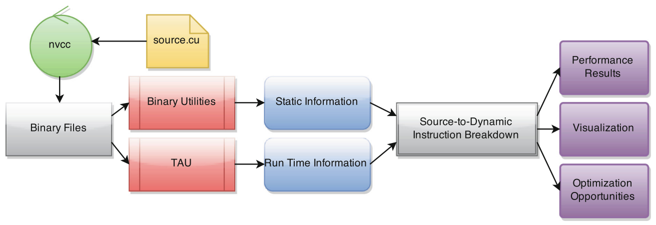


Fig. 2. Overview of our proposed methodology.

CUDA Performance Tool Interface (CUPTI) provides two APIs, the Callback API and the Event API, which enables the creation of profiling and tracing tools that target CUDA applications. The CUPTI Callback API registers a callback in TAU and is invoked whenever an application being profiled calls a CUDA runtime or driver function, or when certain events occur in the CUDA driver. CUPTI fills activity buffers with activity records as corresponding activities occur on the CPU and GPU. The CUPTI Event API allows the tool to query, configure, start, stop, and read the event counters on a CUDA enabled device.

3 Methodology

Our approach to enabling new types of insight into the performance characteristics of GPU kernels includes both static and dynamic measurement and analysis.

3.1 Static Analysis

Each CUDA code is compiled with CUDA 7.0 v.7.0.17, and the "`-g -lineinfo`" flags, which enables tracking of source code location activity within TAU. Each of the generated code from `nvcc` is fed into cuobjdump and nvdisasm to statically analyze the code for instruction mixes and source line information. The generated code is then monitored with TAU, which collects performance measurements and dynamically analyzes the code variants.

Binary Utilities. CUDA binaries are disassembled with the binary utilities provided by the NVIDIA SDK. A CUDA binary (cubin) file is an ELF-formatted file, or executable and linkable format, which is a common standard file format for representing executables, object code, shared libraries and core dumps. By default, the CUDA compiler driver nvcc embeds cubin files into the host executable file.

Instruction Breakdown. We start the analysis by categorizing the executed instructions from the disassembled binary output. Figure 3 displays the instruction breakdown for individual kernels in LULESH and LAMMPS applications

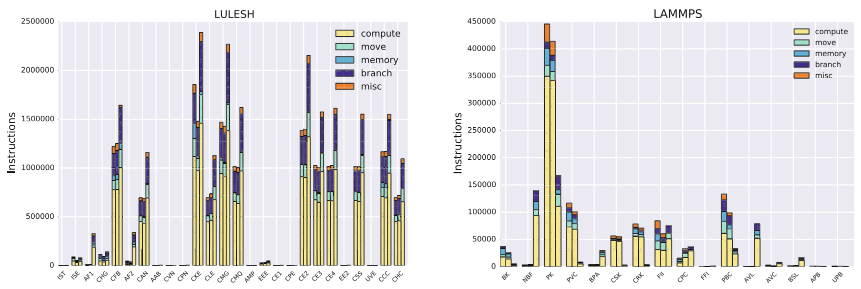


Fig. 3. Instruction breakdown for M2090, K80, and M6000 for individual kernels in LULESH and LAMMPS applications (Color figure online).

for M2090, K80 and M6000 architectures (one per generation). For LAMMPS, the PK kernel shows more computational operations, whereas FII and PBC shows more move operations. For the LULESH kernels CKE, CMG, and CE2 we observe more compute-intensive operations, as well as branches, and moves. One thing to note is that the Maxwell architectures (M6000) in general shows more compute operations for all kernels, when compared with Tesla.

3.2 Dynamic Analysis

The TAU Parallel Performance System monitors various CUDA activities, such as memory transfers and concurrent kernels executed. TAU also tracks source code locator activities, as described below. Hardware counter sampling for CUPTI is also implemented in TAU and is enabled by passing the "ebs" flag to the tau_exec command line [15]. In addition, the environment variable `TAU_METRICS` is set with events to sample. TAU lists CUPTI events available for a particular GPU with the `tau_cupti_avail` command. For our experiments, we monitored instructions executed and active cycles, since those events are available across all GPUs.

Source Code Locator Activity. Source code locator information is an activity within the CUPTI runtime environment that makes possible logging of CUPTI activity. Instructions are sampled at a fixed rate of 20 ms. Within each sample, the following events are collected: threads executed, instructions executed, source line information, kernels launched, timestamps, and program counter offsets. Our research utilizes the information collected from the source code locator and instruction execution activities. The activity records are collected as profiles and written out to disk for further analysis.

Runtime Mapping of Instruction Mixes to Source Code Location. Using the source locator activity discussed in Sect. 3.2, we statically collect instruction mixes and source code locations from generated code and map the instruction mixes to the source locator activity as the program is being run. The static analysis of CUDA binaries produce an objdump file, which provides assembly information, including instruction operations, program counter offsets, and line information. We attribute the static analysis from the objdump file to the profiles collected from the source code activity to provide runtime characterization of the GPU as it is being executed on the architecture. This mapping of static and dynamic profiles provides a rich understanding of the behavior of the kernel application with respect to the underlying architecture.

3.3 Instruction Operation Metrics

We define several instruction operation metrics derived from our methodology as follows. These are examples of metrics that can be used to relate the instruction mix of a kernel with a potential performance bottleneck. Let op_j represent the different types of operations as listed in [12], $time_{exec}$ equal the time duration for one kernel execution (ms), and $calls_n$ represent the number of unique kernel launches for that particular kernel.

Efficiency metric describes flops per second, or how well the floating point units are effectively utilized:

$$efficiency = \frac{op_{fp} + op_{int} + op_{simd} + op_{conv}}{time_{exec}} \cdot calls_n \tag{1}$$

Impact metric describes the performance contribution (operations executed) for a particular kernel j with respect to the overall application:

$$impact = \frac{\sum_{j \in J} op_j}{\sum_{i \in I} \sum_{j \in J} op_{i,j}} \cdot calls_n \tag{2}$$

Individual metrics for computational intensity, memory intensity and control intensity can be calculated as follows:

$$FLOPS = \frac{op_{fp} + op_{int} + op_{simd} + op_{conv}}{\sum_{j \in J} op_j} \cdot calls_n \tag{3}$$

$$MemOPS = \frac{op_{ldst} + op_{tex} + op_{surf}}{\sum_{j \in J} op_j} \cdot calls_n \tag{4}$$

$$CtrlOPS = \frac{op_{ctrl} + op_{move} + op_{pred}}{\sum_{j \in J} op_j} \cdot calls_n \tag{5}$$

4 Analysis

4.1 Applications

LAMMPS. The Large-scale Atomic/Molecular Massively Parallel Simulator [16] is a molecular dynamics application that integrates Newton's equations of motion for collections of atoms, molecules, and macroscopic particles.

Developed by Sandia National Laboratories, LAMMPS simulates short- or long-range forces with a variety of initial and/or boundary conditions. On parallel machines, LAMMPS uses spatial-decomposition techniques to partition the simulation domain into small 3D sub-domains, where each sub-domain is assigned to a processor. LAMMPS-CUDA offloads neighbor and force computations to GPUs while performing time integration on CPUs. In this work, we focus on the Lennard-Jones (LJ) benchmark, which approximates the interatomic potential between a pair of neutral atoms or molecules.

LULESH. The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [8] is a highly simplified application that solves a Sedov blast problem, which represents numerical algorithms, data motion, and programming styles typical in scientific applications. Developed by Lawrence Livermore National Laboratory as part of DARPA's Ubiquitous High-Performance Computing Program, LULESH approximates the hydrodynamics equation discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. LULESH is built on the concept of an unstructured hex mesh, where a node represents a point where mesh lines intersect. In this paper, we study the LULESH-GPU implementation with TAU.

4.2 Methodology

We profile LULESH and LAMMPS applications on seven different GPUs (listed in [12]) by using the TAU Parallel Performance System. Next, we calculate the performance of the kernel for one pass. Then, we apply the metrics from Sect. 3.3 to identify potentially poorly performing kernels that can be optimized. Note that $calls_n$, which represents the number of times a particular routine is called, can easily be collected with TAU profiling. The overhead associated with running the static analysis of our tool is equivalent to compiling the code and running the objdump results through a parser.

4.3 Results

Figure 5 shows statically analyzed heatmap representations for LAMMPS and LULESH on various architectures. The x-axis represents the kernel name (listed in Appendix of [12]), while the y-axis lists the type of instruction mix. For LAMMPS, overall similarities exist within each architecture generation (Tesla vs. Maxwell), where Maxwell makes greater use of the control and floating-point operations, while Tesla utilizes conversion operations. The GTX980 makes use of predicate instructions, as indicated on the top row of the bottom-middle plot. For LULESH, more use of predicate and conversion operations show up in Fermi and Tesla architectures, versus Maxwell which utilizes SIMD instructions for both AF1 and AF2 kernels. Load/store instructions are particularly heavy in M2090 and the GTX480 for the CKE kernel.

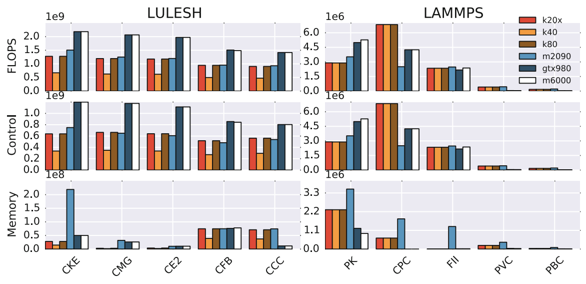


Fig. 4. Metrics for individual kernel execution in LULESH and LAMMPS applications (Color figure online).

Figure 4 displays normalized metrics for FLOPS, control operations and memory operations for the top five poor performing kernels, determined by the *impact* metric (Eq. 2, Fig. 6b). Generally speaking, ideal kernel performance occurs in balanced FLOPS and memory operations, and low branch operations. FLOPS and branches were higher in general for LULESH on the Maxwell architectures, when compared to Tesla. The M2090 architecture showed higher memory operations for the CKE kernel and for all LAMMPS kernels. The M2090 has a smaller global memory compared to Tesla (5 GB vs 11.5 GB), and a smaller L2 cache compared to Maxwell (0.8 MB vs. 3.1 MB), which explains its poor memory performance.

Figure 6a compares divergent branches over total instructions in GPU codes using hardware counters and instruction mix sampling for the top twelve kernels in LULESH, calculated with the *CtrlOPS* metric. The kernels that are closest to the y-axis represent divergent paths that weren't detected with hardware counters (about 33 %), which further affirms the counter's inconsistencies in providing accurate measurements. Our methodology was able to precisely detect divergent branches for kernels that exhibited that behavior.

Figure 7 shows the correlation of computation intensity with memory intensity (normalized) for all seven architectures for the LAMMPS application. For static input size independent analysis (left), differences in code generated are displayed for different architectures. However, the figure in the right shows the instruction mixes for runtime data and reflects that there isn't much of a difference in terms of performance across architectures. The differences between dynamic and static results are primarily due to the lack of control flow information in the static analysis, which will be added in future. While this addition will likely improve the match between the static and dynamic instruction counts, there will always be some discrepancies because not all dynamic behavior can be inferred from the static code for most codes. Nevertheless, by using our static analysis tool, we were able to identify four of the top five time-consuming kernels based only on instruction mix data. The static instruction mixes provide qualitatively comparable information, which can be used to guide optimizations.

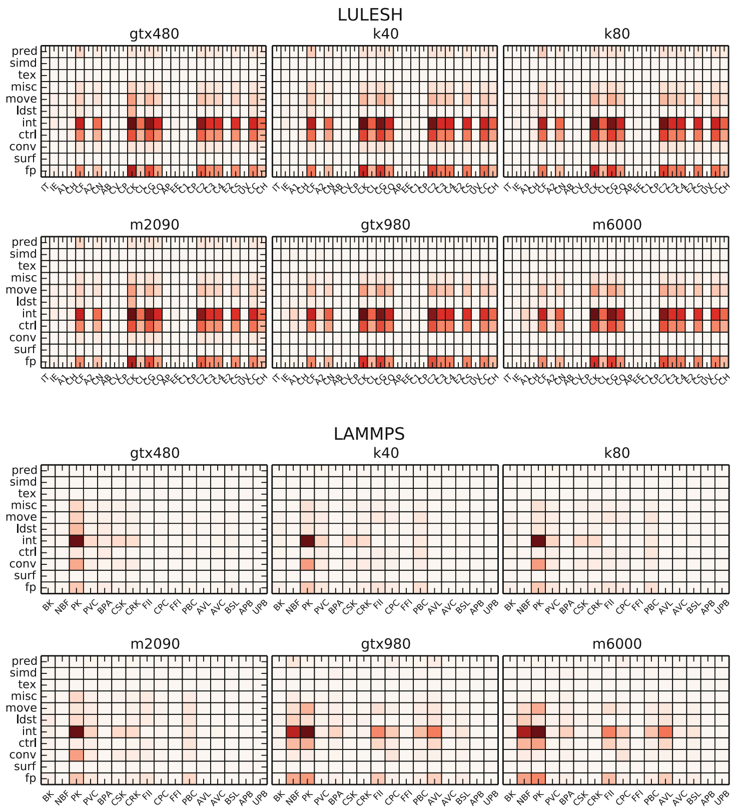
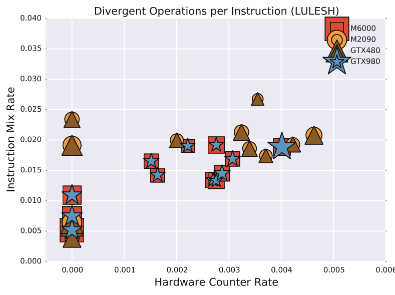


Fig. 5. Heatmap for micro operations for LULESH and LAMMPS benchmarks on various GPU architectures (Color figure online).



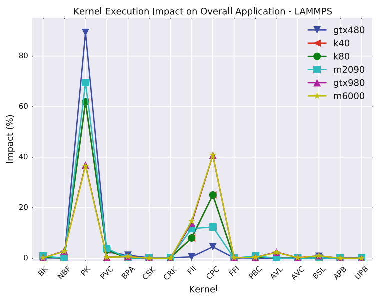


Fig. 6. Two approaches to measuring divergent branches in LULESH: instruction mix sampling, and hardware counters. Kernel impact on overall application in LAMMPS.

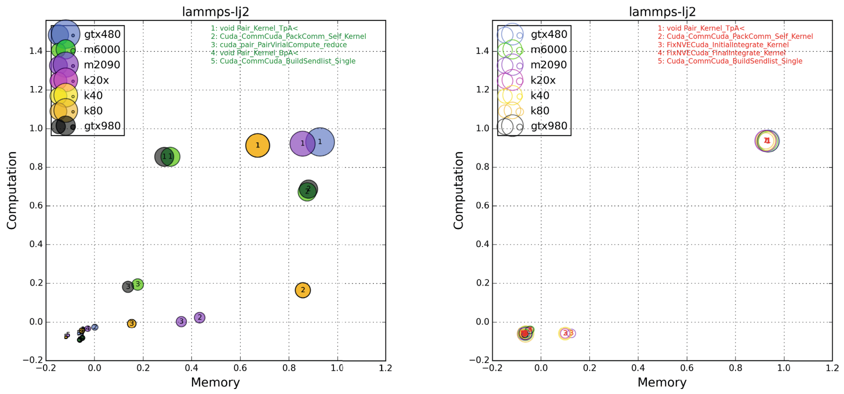


Fig. 7. Static (left) and dynamic (right) analyses for various architectures showing performance of individual kernels in LAMMPS (Color figure online).

5 Related Work

There have been attempts to assess the kernel-level performance of GPUs. However, not much has been done to provide an in-depth analysis of activities that occur inside the GPU.

Distributed with CUDA SDK releases, NVIDIA's Visual Profiler (NVP) [2] has a suite of performance monitoring tools that focuses on CUDA codes. NVP traces the execution of each GPU task, recording method name, start and end times, launch parameters, and GPU hardware counter values, among other information. NVP also makes use of the source code locator activity by displaying source code alongside PTX assembly code. However, NVP doesn't quantify the use of instruction mixes which differs from our work.

G-HPCToolkit [4] characterizes kernel behavior by looking at idleness analysis via blame-shifting and stall analysis for performance degradation. In this work, the authors quantify CPU code regions that execute when a GPU is idle, or GPU tasks that execute when a CPU thread is idle, and accumulate blame to the executing task proportional to the idling task. Vampir [11] also does performance measurements for GPUs. They look at the trace execution at the start and stop times and provide a detailed execution of timing of kernel execution, but do not provide activities that behave inside the kernel. The authors [9] have characterized PTX kernels by creating an internal representation of a program and running it on an emulator, which determines the memory, control flow and parallelism of the application. This work closely resembles ours, but differs in that we perform workload characterization on actual hardware during execution.

Other attempts at modeling performance execution on GPUs can be seen in [7] and [10]. These analytical models provide a tractable solution to calculate GPU performance when given input sizes and hardware constraints. Our work is complementary to those efforts, in that we identify performance execution of kernels using instruction mixes.

6 Conclusion and Future Work

Monitoring performance on accelerators is difficult because of the lack of visibility in GPU execution and the asynchronous behavior between the CPU and GPU. Sampling instruction mixes in real time can help characterize the application behavior with respect to the underlying architecture, as well as identify the best tuning parameters for kernel execution.

In this research, we provide insight on activities that occur as the kernel executes on the GPU. In particular, we characterize the performance of execution at the kernel level based on sampled instruction mixes. In future work, we want to address the divergent branch problem, a known performance bottleneck on accelerators, by building control flow graphs that model execution behavior. In addition, we plan to use the sampled instruction mixes to predict performance parameters and execution time for the Orio code generation framework [6]. The goal is to substantially reduce the number of empirical tests for kernels, which will result in rapid identification of best performance tuning configurations.

Acknowledgements. We would like to thank Duncan Poole and J-C Vasnier of NVIDIA for providing early access to CUDA 7.0 and to the PSG Clusters. We also want to thank the anonymous reviewers in providing constructive feedback. This work is supported by the Department of Energy (Award #DE-SC0005360) for the project "Vancouver 2: Improving Programmability of Contemporary Heterogeneous Architectures."

References

1. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe
2. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**, 189–204 (2000)
4. Chabbi, M., Murthy, K., Fagan, M., Mellor-Crummey, J.: Effective sampling-driven performance tools for GPU-accelerated supercomputers. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE (2013)
5. Dietrich, R., Ilsche, T., Juckeland, G.: Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures. In: First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI). IEEE Computer Society (2010)
6. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio. In: International Symposium on Parallel & Distributed Processing (IPDPS). IEEE (2009)
7. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: ACM SIGARCH Computer Architecture News. ACM (2009)
8. Karlin, I., Bhatele, A., Chamberlain, B., Cohen, J., Devito, Z., Gokhale, M., et al., R.H.: Lulesh programming model and performance ports overview. In: Technical report, Lawrence Livermore National Laboratory (LLNL) Technical Report (2012)

9. Kerr, A., Andrew, G., Yalamanchili, S.: A characterization and analysis of PTX kernels. In: International Symposium on Workload Characterization (IISWC). IEEE (2009)
10. Kim, H., Vuduc, R., Baghsorkhi, S., Choi, J., Hwu, W.: Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). Morgan & Claypool Publishers (2012)
11. Knpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Mller, M., Nagel, W.: The VAMPIR performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing. Springer, Heidelberg (2008)
12. Lim, R.: Identifying optimization opportunities within kernel launches in GPU architectures, Technical report. University of Oregon, CIS Department (2015)
13. Lim, R., Carrillo-Cisneros, D., Alkowaileet, W., Scherson, I.: Computationally efficient multiplexing of events on hardware counters. In: Linux Symposium (2014)
14. Malony, A., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., Lamb, C.: Parallel performance measurement of heterogeneous parallel systems with GPUs. In: IEEE International Conference on Parallel Processing (ICPP) (2011)
15. Morris, A., Malony, A., Shende, S., Huck, K.: Design and implementation of a hybrid parallel performance measurement system. In: International Conference on Parallel Processing (ICPP) (2010)
16. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. J. Comput. Phys. **117**(1), 1–19 (1995)
17. Shende, S., Malony, A.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
18. Weaver, V., Terpstra, D., Moore, S.: Non-determinism and overcount on modern hardware performance counter implementations. In: International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE (2013)
19. Shao, Y., Brooks, D.: ISA-independent workload characterization and its implications for specialized architectures. In: IEEE International Symposium Performance Analysis of Systems and Software (ISPASS) (2013)