Identifying Optimization Opportunities within Kernel Execution in GPU Architectures

Directed Research Project

Robert Lim
University of Oregon
Eugene, OR 97403-1202
roblim1@cs.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 [11]. 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 behaviors of the application in a summarized manner without exposing sufficient low-level details.

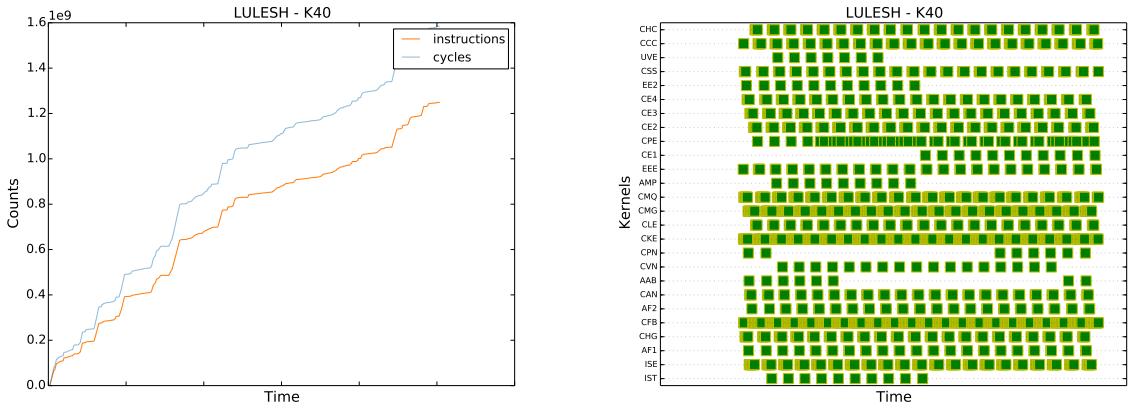

Figure 1: Sampled hardware counters of instructions executed and active cycles (left) and individual kernel executions (right), both for LULESH.

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 activities are on the CPU versus the GPU makes debugging applications a very complicated task. Thus, analyzing static and dynamic instruction mixes can help identify potential performance bottlenecks in heterogeneous architectures.

In Figure 1, we show a time series of hardware counters sampled in the GPU, a capability we've added in TAU (Section 3.2.2), 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 of where to begin. This motivates our exploration of the use of instruction type mixes in aiding the analysis of potential performance bottlenecks.

1.1 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. 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.

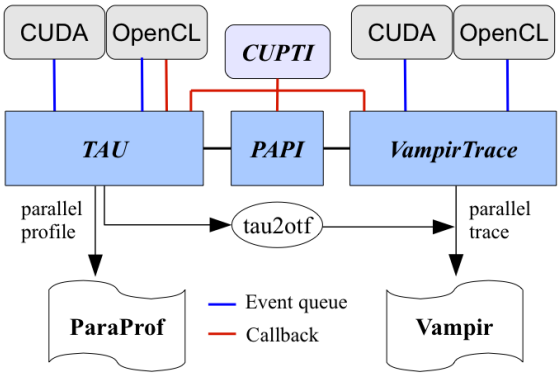

Figure 2: TAU CUPTI tools framework.

2 Background

This section provides an overview of CUDA-related semantics and events with respect to TAU.

2.1 TAU Performance System

The TAU Parallel Performance Framework [1] 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 (Figure 2). Each call made to a runtime or driver routine is intercepted by TAU for measurement before and after calling the actual CUDA routine.

2.1.1 TAU CUPTI Measurements

TAU collects performance events for CUDA GPU codes asynchronously by tracing an application's CPU and GPU activity [10]. An activity record is created, which logs CPU and GPU activities. Each event kind (e.g. `CUpti_ActivityMemcpy`) represents a particular activity.

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.

CUPTI registers callbacks in the following steps. `Tau_Cupti_Subscribe()` is invoked, which calls `cuptiActivityRegisterCallbacks()`. The callback registration creates an activity buffer, if one has not been created, and registers synchronous events for listening. Within the synchronous event handler, CUPTI activities are declared and parameters are set accordingly. For instance, monitoring a memory copy event would require setting the type of memory copy (host-to-device, device-to-host, device-to-device, etc.), as well its allocation type (pinned, paged, device or host).

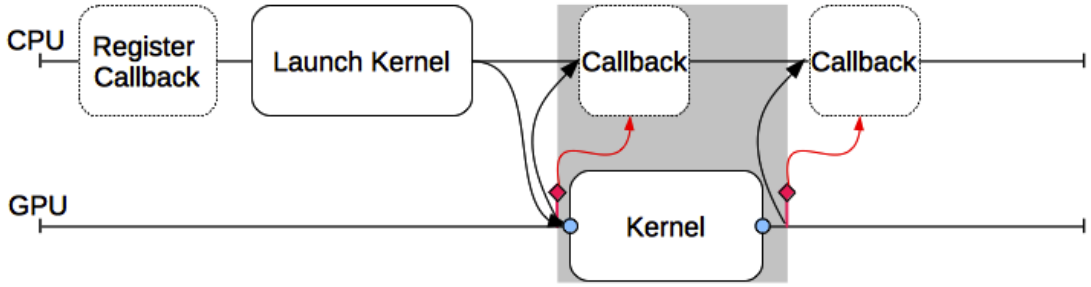

Figure 3: CUPTI callback.

The synchronous event handlers are invoked at synchronous points in the program, where activity buffers are read and written. For each activity task that executes, CUPTI logs activity records at synchronization points that contains the task start and end times. For the memory copy example, the amount of bytes transferred, which kernel transferred those bytes, and start and end timestamps are provided in its activity record.

Callback Method and Asynchronous Buffering 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 (Figure 3). The callback method is a mechanism in the device layer that triggers callbacks on the host for registered actions, such as the beginning and ending of a kernel execution. The callback domain is grouped into four domains (runtime functions, driver functions, resource tracking, synchronization notification) to make it easier to associate callback functions with groups of related CUDA functions or events. The subscriber associates each callback function with one or more CUDA API functions and at most one subscriber `cuptiSubscribe()` at a time, which requires finalizing with `cuptiUnsubscribe()` before initializing another subscriber.

Asynchronous buffering registers two callbacks, where one is invoked whenever CUPTI needs an empty activity buffer, and the other is called to deliver a buffer containing one or more activity records to TAU. CUPTI fills activity buffers with activity records as corresponding activities that occur on the CPU and GPU. The CUPTI client within TAU provides empty buffers to ensure that no records are dropped. Enabling a CUPTI activity forces initialization of the activity API, whereas flushing a CUPTI activity forces CUPTI to deliver activity buffers with completed activity records. Reading and writing attributes are handled via `cuptiActivityGetAttribute` and `cuptiActivitySetAttribute` calls, respectively, which controls how buffering API behaves. `activity_trace_async` uses activity buffer API to collect traces of CPU/GPU activity.

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.

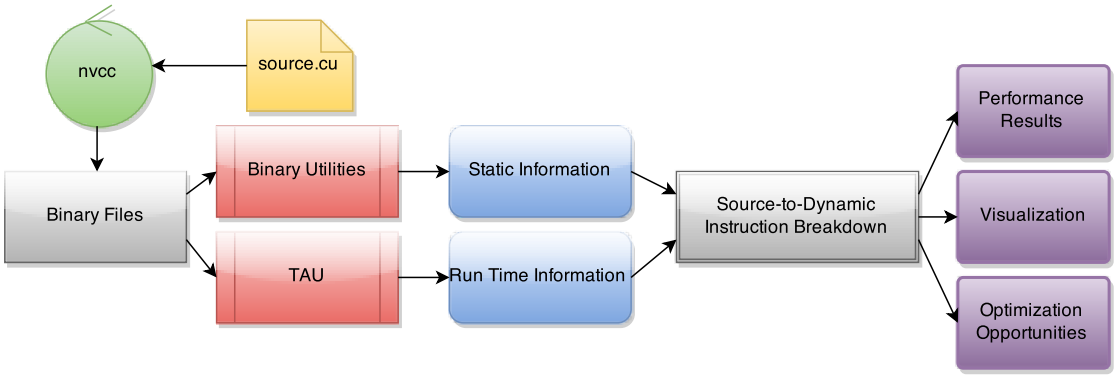

Figure 4: Overview of our proposed methodology.

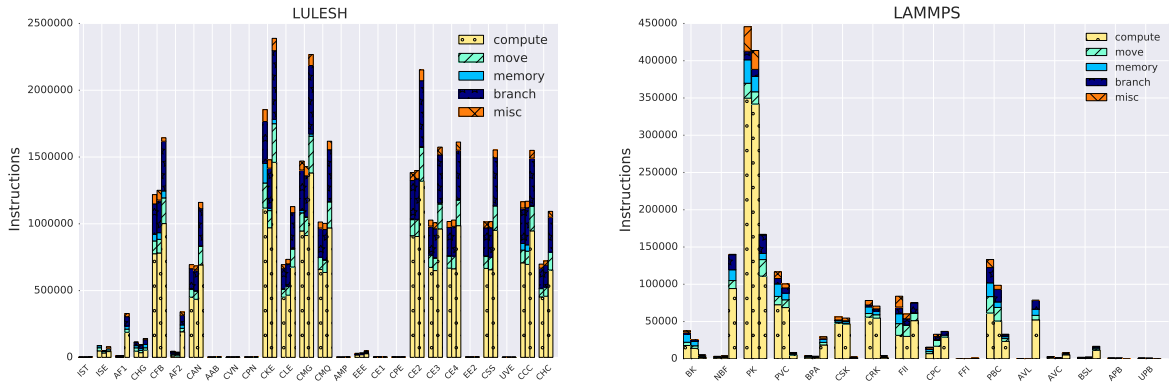

Figure 5: Instruction breakdown for M2090, K80, and M6000 for individual kernels in LULESH and LAMMPS applications.

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.

3.1.1 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. The objdump command is typically used to obtain disassembled binary machine code from an executable file, compiled object, or shared library.

cuobjdump extracts information from CUDA binary files (both standalone and those embedded in host binaries). The output of cuobjdump includes CUDA assembly code for each kernel, CUDA ELF section headers, string tables, relocators and other CUDA specific sections. It also extracts embedded PTX text from host binaries. The cubin file is then

Category	Opcodes
Floating Point	FADD, FCHK, FCMP, FFMM, FMNMX, FMUL, FSET, FSETP, FSWZADD, MUFU, RRO, DADD, DFMA, DMNMX, DMUL, DSET, DSETP
Integer	BFE, BFI, FLO, IADD, IADD3, ICMP, IMAD, IMADSP, IMNMX, IMUL, ISCADD, ISET, ISETP, LEA, LOP, LOP3, POPC, SHF, SHL, SHR, XMAD
Conversion	F2F, F2I, I2F, I2I
Movement	MOV, PRMT, SEL, SHFL
Predicate/CC	CSET, CSETP, PSET, PSETP, P2R, R2P
Texture	TEX, TLD, TLD4, TxQ, TEXS, TLD4S, TLD5
Load/Store	LD, LDC, LDG, LDL, LDS, ST, STG, STL, STS, ATOM, ATOMS, RED, CCTL, CCTLL, MEMBAR, CCTLT
Surface Memory	SUATOM, SULD, SURED, SUST
Control	BRA, BRX, JMP, JMX, SSY, CAL, JCAL, PRET, RET, BRK, PBK, CONT, PCNT, EXIT, BPT
Miscellaneous	NOP, CS2R, S2R, B2R, BAR, R2B, VOTE

Table 1: Instruction mixes broken down according to categories.

fed to nvidsasm, which outputs assembly code for each kernel, listings of ELF data sections, and other CUDA-specific sections. Instruction mixes such as floating-point operations and load/store operations are extracted from the assembly code, along with source line information.

3.1.2 Instruction Breakdown

We start the analysis by categorizing the executed instructions from the disassembled binary output. Table 1 displays the opcodes from nvcc-generated binaries, categorized according to instruction types.

Figure 5 displays the instruction breakdown for individual kernels in LULESH and LAMMPS applications 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 in LULESH, whereas the M2090 makes use of more operations in LAMMPS.

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 [4]. 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.

For each of the sampled regions from the source locator activity, the instruction mixes and line information that was collected from the static analyzer is attributed to those locations. This gives precise information on what instructions are being executed in real time.

3.2.1 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.

3.2.2 Hardware counter sampling

Hardware counter sampling provides a mechanism to periodically read CUPTI counters, where the event API samples event values while kernels are executing. Hardware counter sampling is integrated in TAU and builds upon previous event-based measurement efforts [4]. The event collection mode is set to "continuous" so that event counters run continuously. Two threads are used in event sampling, where one thread schedules the kernels and memory transfers that perform the computation, while another thread wakes up periodically to sample an event counter. When sampling hardware counters, there is no correlation of the event samples with what is happening on the GPU. GPU timestamps provide coarse correlation at the time of the sample and also at other points of interest in the application.

The CUPTI Event API provides a means to query, configure, start, stop, and read the event counters on a CUDA-enabled device. An event is a countable activity, action, or occurrence on a device, which is assigned a unique identifier. A named event represents the same activity, action, or occurrence on all device types. Each event is placed in one of the categories defined by `CUpti_EventCategory`, which describes the general type of activity, action, or occurrence measured by the event. A device exposes one or more event domains, where each event domain represents a group of related events that are managed together and must belong to the same domain. The number and type of events that can be added to an event group are subject to device-specific limits.

3.2.3 Runtime Mapping of Instruction Mixes to Source Code Location

Using the source locator activity discussed in Section 3.2.1, 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 produces 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, $time_{exec}$ 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:

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

Impact metric describes the performance contribution of a particular kernel with respect to the overall application:

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

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

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

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

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

4 Analysis

We use the LAMMPS and LULESH applications to demonstrate the new static and dynamic measurement and analysis capabilities.

4.1 Applications

4.1.1 LAMMPS

The Large-scale Atomic/Molecular Massively Parallel Simulator [12] 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. For computational efficiency LAMMPS uses neighbor lists to keep track of nearby particles, which are optimized for systems with particles that are repulsive at short distances so that the local density of particles never become too large. 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.

4.1.2 LULESH

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [13] 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. Modeling hydrodynamics describes the motion of materials relative to each other when subject to forces. 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 [3]) with the TAU Parallel Performance System. Next, we calculate the performance of the kernel for one pass. Then, we apply the metrics from Section 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 7 shows statically analyzed heatmap representations for LAMMPS and LULESH on various architectures. The x-axis represents the kernel name (listed in Appendix of [3]), 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 makes greater use of the conversion operations. The GTX980 makes use of predicate instructions, as indicated in 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.

Figure 6 displays results for individual metrics for FLOPS, memory operations and control operations for the top five poor performing kernels for each architecture. Poor performing kernels were determined using the *impact* metric. FLOPS and branch instructions 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 8 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 dotted line separates the different approaches, indicating that in all cases the instruction mix method was able to detect divergent paths. The kernels that are closest to the y-axis represent divergent paths that weren't detected with hardware

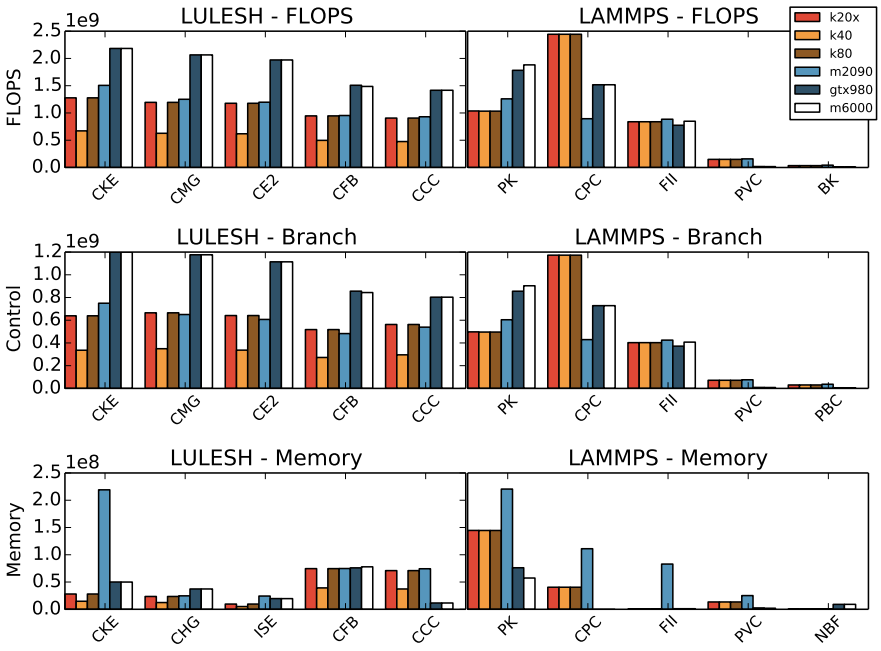

Figure 6: Floating point, control and memory intensity for top five individual kernels in LULESH and LAMMPS applications.

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 9 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. 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.

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) [7] 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 [14] 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

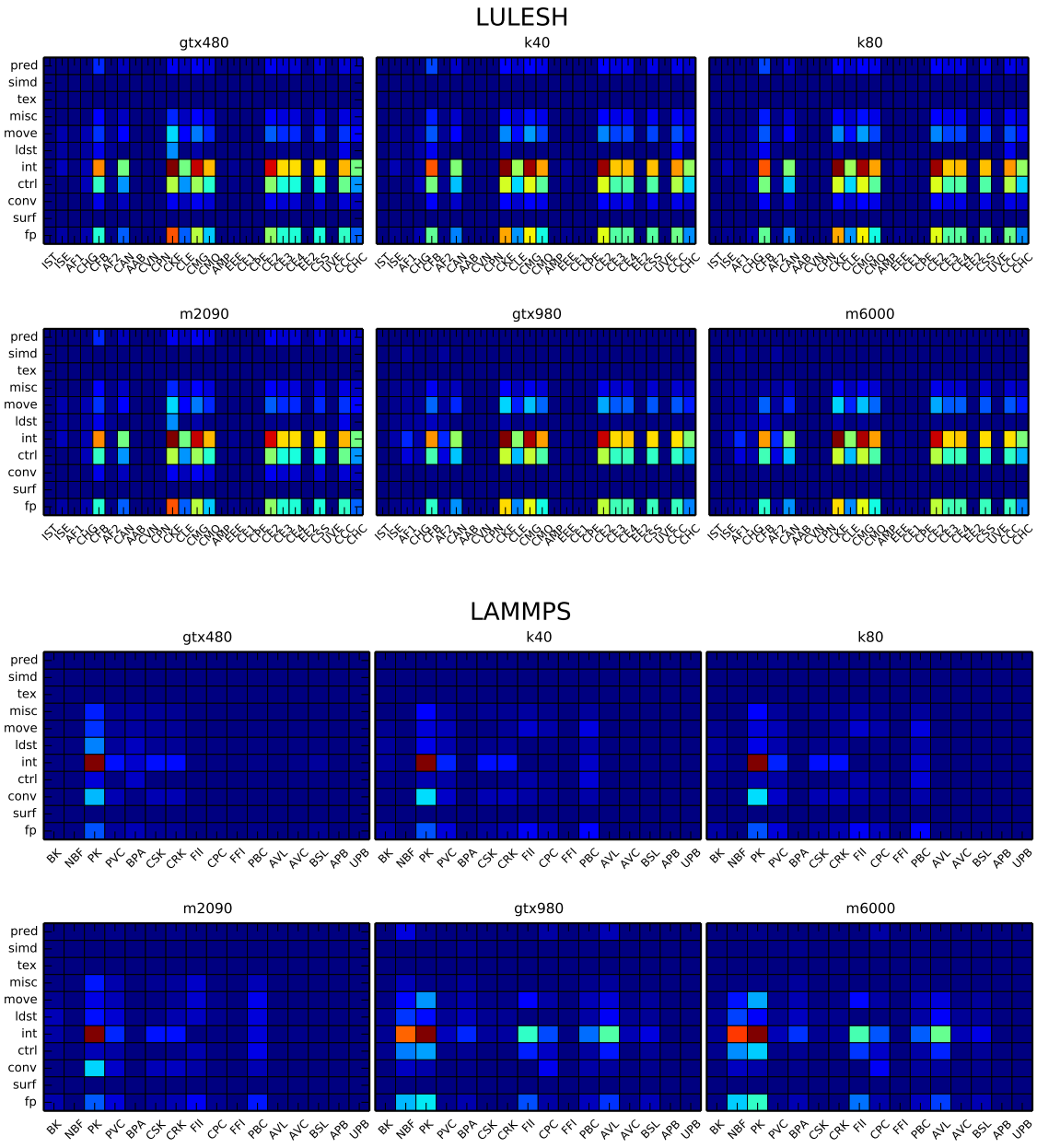

Figure 7: Heatmap for micro operations for LULESH and LAMMPS benchmarks on various GPU architectures.

	GTX480	K20x	K40	K80	M2090	GTX980	M6000
CUDA Capability	2	3.5	3.5	3.7	2	5.2	5.2
Global Memory (MB)	1156	5760	11520	11520	5375	4096	12288
Multiprocessors	15	14	15	13	16	16	24
CUDA Cores per MP	32	192	192	192	32	128	128
CUDA Cores	480	2688	2880	2496	512	2048	3072
GPU Clock Rate (MHz)	1400	732	745	824	1301	1216	1114
Memory Clock Rate (MHz)	1848	2600	3004	2505	1848	3505	3305
L2 Cache Size (MB)	0.786	1.572	1.572	1.572	0.786	2.097	3.146
Constant Memory (bytes)	65536	65536	65536	65536	65536	65536	65536
Shared Memory (bytes)	49152	49152	49152	49152	49152	49152	49152
Registers per Block	32768	65536	65536	65536	32768	65536	65536
Warp Size	32	32	32	32	32	32	32
Max Threads per MP	1536	2048	2048	2048	1536	2048	2048
Max Threads per Block	1024	1024	1024	1024	1024	1024	1024
Architecture Family	Fermi	Tesla	Tesla	Tesla	Tesla	Maxwell	Maxwell

Table 2: Graphic processors used in this experiment.

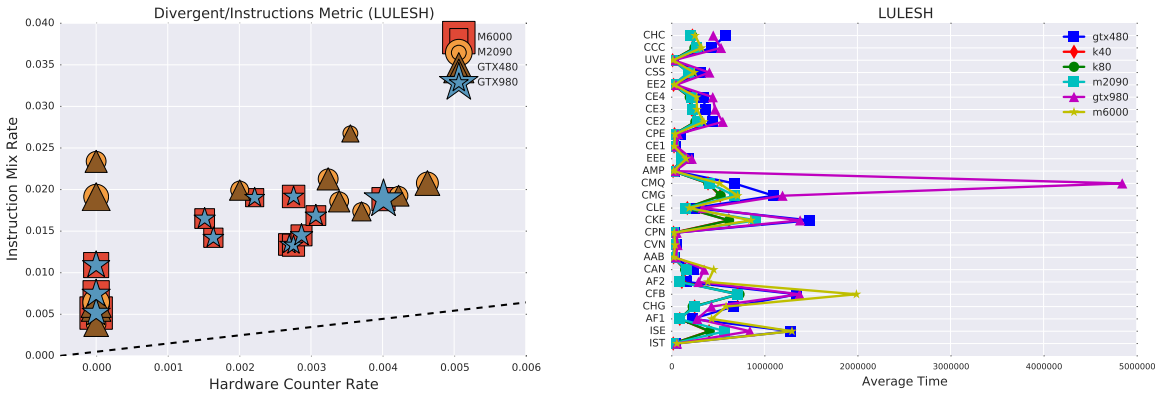

Figure 8: Two approaches to measuring divergent branches: instruction mix sampling, and hardware counters. Average execution times for individual kernels, both for LULESH applications.

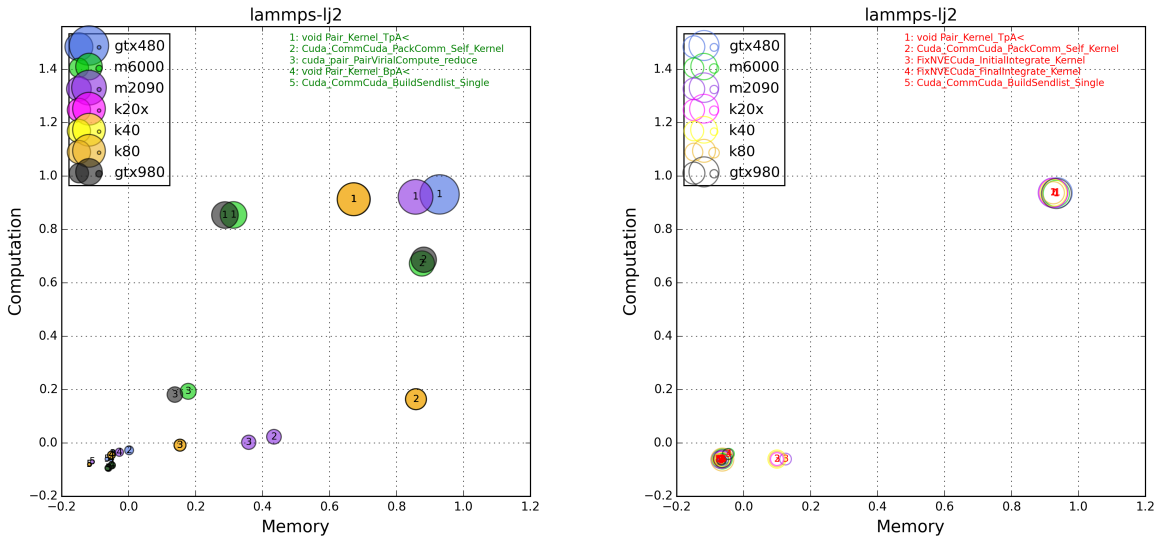

Figure 9: Static (left) and dynamic (right) analyses for various architectures showing performance of individual kernels in LAMMPS.

idling task. Vampir [8] 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 [16] 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 [19] and [20]. 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 inside 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 [9]. The goal is to substantially reduce the number of empirical tests for kernels, which will result in rapid identification of best performance tuning configurations.

7 Acknowledgements

We want to thank Duncan Poole and J-C Vasnier of NVIDIA for providing early access to CUDA 7.0 and to the PSG Clusters. 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] S. Shende, and A. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 2006.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, June 2000.

[3] R. Lim. Identifying Optimization Opportunities within Kernel Launches in GPU Architectures *University of Oregon, CIS Department*, Technical Report.

[4] A. Morris, A. Malony, S. Shende, and K. Huck. Design and implementation of a hybrid parallel performance measurement system. *International Conference on Parallel Processing (ICPP)*, IEEE, 2010.

[5] R. Dietrich, T. Ilsche, and G. Juckeland . Non-intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures *First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, IEEE Computer Society, 2010.

[6] Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe

[7] NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler

[8] A. Knpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Mller, and W. Nagel. "The VAMPIR performance analysis tool-set." In *Tools for High Performance Computing*, pp. 139-155. Springer Berlin Heidelberg, 2008.

[9] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. *International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2009.

[10] A. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel systems with GPUs. *IEEE International Conference on Parallel Processing (ICPP)*, 2011.

[11] Y. Shao, and D. Brooks. "ISA-independent workload characterization and its implications for specialized architectures." *IEEE International Symposium Performance Analysis of Systems and Software (ISPASS)*, 2013.

[12] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics* 117, no. 1 (1995): 1-19.

[13] I. Karlin, A. Bhatele, B. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque et al. Lulesh programming model and performance ports overview *Lawrence Livermore National Laboratory (LLNL)*, Livermore, CA, Tech. Rep (2012).

[14] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for GPU-accelerated supercomputers. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2013.

[15] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2012.

[16] A. Kerr, G. Andrew, and S. Yalamanchili. A characterization and analysis of ptx kernels. *International Symposium on Workload Characterization (IISWC)*, IEEE 2009.

[17] V. Weaver, D. Terpstra and S. Moore Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. *ISPASS Workshop*, April 2013.

[18] R. Lim, D. Carrillo-Cisneros, W. Alkowaileet, and I. Scherson Computationally Efficient Multiplexing of Events on Hardware Counters. *Linux Symposium*, July 2014

[19] S. Hong, and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News, ACM, 2009*.

[20] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W. Hwu Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). *Morgan & Claypool Publishers*, 2012.

8 Appendix

8.1 Applications

Applications		
Benchmark	Kernel	Abb
LULESH	InitStressTermsForElems	IST
	IntegrateStressForElems	ISE
	AddNodeForcesFromElems	AF1
	CalcHourglassControlForElems	CHG
	CalcFBHourglassForceForElems	CFB
	AddNodeForcesFromElems2	AF2
	CalcAccelerationForNodes	CAN
	ApplyAccelerationBoundaryConditionsForNodes	AAB
	CalcVelocityForNodes	CVN
	CalcPositionForNodes	CPN
	CalcKinematicsForElems	CKE
	CalcLagrangeElementsPart2	CLE
	CalcMonotonicQGradientsForElems	CMG
	CalcMonotonicQRegionForElems	CMQ
	ApplyMaterialPropertiesForElems	AMP
	EvalEOSForElems	EEE
	CalcEnergyForElemsPart1	CE1
	CalcPressureForElems	CPE
	CalcEnergyForElemsPart2	CE2
	CalcEnergyForElemsPart3	CE3
	CalcEnergyForElemsPart4	CE4
	EvalEOSForElemsPart2	EE2
	CalcSoundSpeedForElems	CSS
	UpdateVolumesForElems	UVE
	CalcCourantConstraintForElems	CCC
	CalcHydroConstraintForElems	CHC
LAMMPS	Binning_Kernel	BK
	NeighborBuildFillBin	NBF
	Pair_Kernel_TpA	PK
	PairVirialCompute_reduce	PVC
	Pair_Kernel_BpA	BPA
	ComputeTempCuda_Scalar_Kernel	CSK
	ComputeTempCuda_Reduce_Kernel	CRK
	FixNVECuda_InitialIntegrate	FII
	CommCuda_PackComm_self_Kernel	CPC
	FixNVECuda_FinalIntegrate	FFI
	Domain_PBC_Kernel	PBC
	AtomVecCuda_PackExchangeList	AVL
	AtomVecCuda_PackExchange	AVC
	CommCuda_BuildSendlist	BSL
	AtomVecCuda_PackBorder	APB
	CommCuda_UnpackBorder	UPB

Table 3: Applications and kernels with abbreviations.