

Patterns of inefficient performance behavior in GPU applications

Dominic Eschweiler ^{a1}, Daniel Becker ^{b2,c}, and Felix Wolf ^{a,b3,c}

^a*Forschungszentrum Jülich, Jülich Supercomputing Centre, 52428 Jülich, Germany*

¹d.eschweiler@fz-juelich.de

^b*German Research School for Simulation Sciences, Laboratory for Parallel Programming, 52062 Aachen, Germany*

{²d.becker, ³f.wolf}@grs-sim.de

^c*RWTH Aachen University, Department of Computer Science, 52056 Aachen, Germany*

Abstract—Writing efficient software for heterogeneous architectures equipped with modern accelerator devices presents a serious challenge to programmer productivity, creating a need for powerful performance-analysis tools to adequately support the software development process. To guide the design of such tools, we describe typical patterns of inefficient runtime behavior that may adversely affect the performance of applications that use general-purpose processors along with GPU devices through a CUDA compute engine. To evaluate the general impact of these patterns on application performance, we further present a microbenchmark suite that allows the performance penalty of each pattern to be quantified with results obtained on NVIDIA Fermi and Tesla architectures, indeed demonstrating significant delays. Furthermore this suite can be used as a default test scenario to add CUDA support to performance-analysis tools used in high-performance computing.

I. INTRODUCTION

In view of the broadening requirements of today’s graphics applications, graphics processors are shifting toward a more generic architecture with enhanced programmability. Once designed exclusively for computer graphics and difficult to program, today’s *graphics processing units* (GPUs) are extremely flexible parallel processors that are often used in combination with general-purpose processors [1]. Whereas general-purpose processors are optimized for low-latency access to data sets stored in local caches, GPUs are optimized for data-parallel throughput computations using many independent compute cores. Accelerators designed in this way are usually referred to as *many-core architectures* exhibiting hundreds of cores, each linked to various memory spaces, which are accessible either from all compute cores or only from distinct subgroups.

Driven by their growing demand for computational power, developers of scientific applications increasingly take advantage of these more flexible graphics-processor designs to accelerate the general-purpose processors previously used in isolation. In addition, powerful programming models such as CUDA [2] and OpenCL [3] have emerged to harness at different levels of abstraction the enormous processing capabilities offered by these devices. In particular CUDA, primarily developed by NVIDIA [4], is often used by developers because of the wide-spread availability of appropriate accelerator hardware. Given that OpenCL and the frequently used PGI compiler suite already include CUDA support, we focus in this paper on patterns of inefficient performance behavior in GPU applications based on CUDA.

The CUDA programming interface rests on a parallel computing architecture and memory model for GPUs, on top of which a dynamic execution model is specified. Defined as an extension of the C programming language, the programming model provides a software infrastructure that allows the scheduling of lightweight compute kernels executing parallel portions of the application on the graphics device. The device distinguishes different memory spaces: registers and local memory are visible only to a single thread, shared memory to all threads from the same block, and finally global memory to all threads from all blocks.

Accelerating scientific codes through GPUs requires harnessing much higher degrees of parallelism, with programming models such as CUDA shielding the application developers from details of the target architecture through suitable abstractions. Given the complexity of the hardware and software environment, tools that identify performance issues are therefore crucial to ensure programmer productivity. To guide the development of future programming tools for the CUDA compute engine, this paper describes performance-critical patterns of inefficient behavior. To evaluate the potential impact of these patterns on application performance, we further present a microbenchmark suite that allows the performance penalty of each pattern to be quantified with results obtained on NVIDIA Fermi and Tesla architectures.

The remainder of this article is organized as follows: After identifying different patterns of inefficient runtime behavior and classifying them in Section II, we present experimental results to quantify the respective performance penalty of individual patterns in Section III. Then, we review related work in Section IV and, finally, summarize our results and outline on future work in Section V.

II. PATTERN CLASSES

This section describes patterns of inefficient runtime behavior covering memory access and thread management scenarios. While the former refer to bank conflict, replication copy, excessive global memory usage, memory coalescing, and scattered host copy scenarios, the latter refer to thread register imbalance, wait at barrier, and branch-diverging scenarios.

A. Memory access patterns

Given that on GPUs communication is primarily done through memory operations, the memory usage may influence

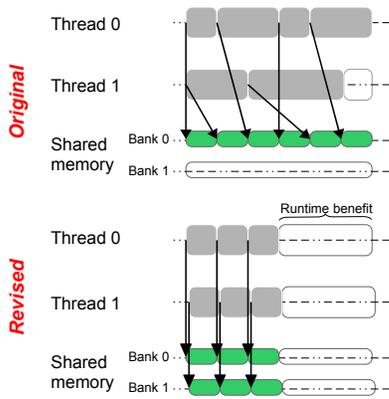


Fig. 1. Bank conflicts.

application performance. As a consequence, memory accesses have to be optimized to raise the application performance.

1) *Bank conflicts*: To decrease memory access times to shared memory, this memory space provides several independent memory banks allowing consecutive addresses to be fetched in parallel [5]. The execution is serialized, when two or more threads concurrently access the same memory bank. Such a scenario is referred to as a *bank conflict* and is further illustrated in Figure 1. While the original case shows two threads accessing the same memory bank resulting in bank conflicts and so serialized memory accesses, the revised version shows two threads accessing two different memory banks reducing the aggregated memory access time and thus resolving the previously observed bank conflicts.

2) *Excessive global memory usage*: One important resource limitation on GPUs is that global memory has limited bandwidth when serving data accesses. As a consequence, application developers are encouraged to load their data from the relatively slow global memory into the relatively fast shared memory before the data is actually needed by their application. This technique is often referred to as *data prefetching* and has the potential to significantly improve the overall program execution time [6]. An *excessive global memory usage* scenario

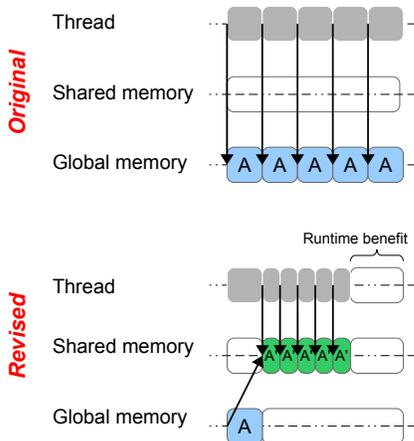


Fig. 2. Excessive global memory usage.

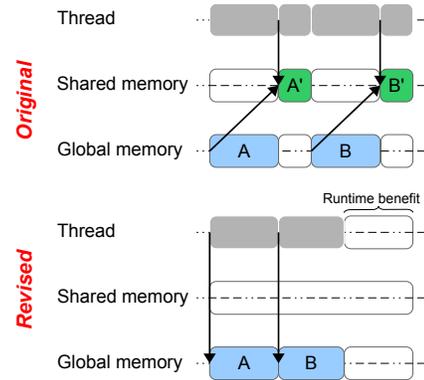


Fig. 3. Replication copy.

(Figure 2) shows up when the global memory is accessed several times. The fast access time to shared memory may reduce the overall runtime if data are accessed several times in shared memory rather than in global memory.

3) *Replication copy*: Data prefetching only guarantees a runtime benefit if the necessary data is accessed several times. Otherwise, if the data is accessed only once, the scenario is referred to as *replication copy* and exemplified in Figure 3. Apparently in such a scenario, the additional data transfer between global and shared memory can be avoided reducing the overall execution time of the revised kernel.

4) *Memory coalescing*: On CUDA compute architectures, memory transfers between global and shared memory are performed in a word size of 128 bit. Given that smaller transfers are padded to that size, copying smaller values between global and shared memory reduces the overall transfer rate. As a consequence, it is more efficient to coalesce narrow memory references into wide ones. Even though compilers occasionally perform memory coalescing automatically, developers are usually responsible for coalesced memory transfers by manually aligning loads and stores. For example, in Figure 4 every thread only copies 64 bit from global memory to shared memory in the original case, but, like in the revised case, it is more efficient if one thread copies two consecutive elements in one operation.

5) *Scattered host copy*: The peak bandwidth between host memory and device memory is much smaller than the peak bandwidth among the different memory spaces of the device. Furthermore if the set of input data is not transferred in a consecutive memory block, several temporarily scattered transfers are necessary (Figure 5). Due to the transfer overhead, such *scattered host copy* scenarios decrease the overall data transfer rate between host and device. To increase the data transfer rate between host and device, it is recommended to gather such data and to invoke a single transfer.

B. Thread management

The coordinated access of threads to memory and compute cores necessitates thread management primitives that may also influence application performance.

1) *Thread register imbalance*: CUDA capable devices are organized in blocks of execution units. Such a block is referred

to as a multiprocessor which provides a limited number of thread slots and thread block slots. Of course, application developers should use the maximum number of thread slots to utilize the multiprocessor efficiently. If all registers are already used by fewer thread blocks than the maximum number of thread blocks, no further thread block can be scheduled on the multiprocessor. In this scenario, which is referred to as *thread register imbalance* and exemplified in Figure 6, thread block slots are left unused, degrading the overall application performance. Apparently, using shared memory instead of registers can increase the performance of the revised kernel, as illustrated in Figure 6.

2) *Wait at barrier*: CUDA allows threads of the same block to coordinate their activities using a barrier. The time a thread spends in a barrier is waiting time, no longer available to perform calculations, which has to be avoided as much as possible. This scenario, shown in Figure 7, is referred to as *wait at barrier*. This scenario can be resolved by balancing the load among threads, reducing the overall execution time.

3) *Branch diverging*: The execution of a thread block is divided into warps with a constant number of threads per warp. When threads in the same warp follow different paths of control flow, these threads diverge in their execution [6] such that their execution is serialized (Figure 8). Such a *branch diverging* scenario can be avoided by aligning the branch granularity to warps.

III. EVALUATION

After different patterns of inefficient runtime behavior have been identified in Section II, this section describes our microbenchmark test suite used to evaluate individual pattern instances and the experimental results for each pattern.

A. Microbenchmark test suite

For the pattern evaluation, we designed a microbenchmark test suite, where the execution time of individual kernels is measured. This environment consists of test kernels for each pattern from the previous section and can be used as a synthetic test case for the development of performance tools. More

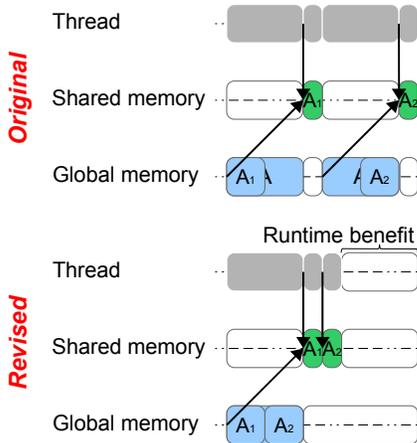


Fig. 4. Memory coalescing.

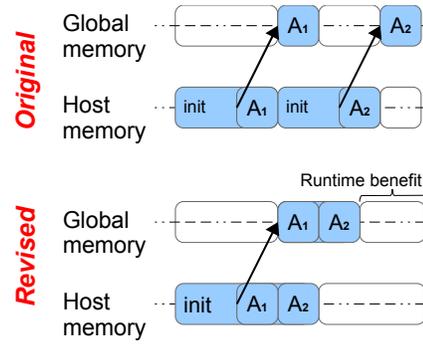


Fig. 5. Scattered host copy

precisely, for each pattern the suite includes an original version that has the described performance problem and a revised version where this problem has been fixed. Measurements are taken on a Fermi architecture with 1.5 GByte of main memory (NVIDIA GeForce GTX 480) and a Tesla architecture with 4.0 GByte of main memory (NVIDIA Tesla T10).

B. Memory access patterns

1) *Bank Conflicts*: The microbenchmark for the bank conflict pattern is derived from a general-purpose Reed-Solomon encoder and stores data arrays in the shared memory, which are used as lookup tables and initialized prior to the computation. In the original kernel, each array is initialized independently by each thread of a block, which causes multiple bank conflicts. In the revised version, the initialization is only performed by one thread of a block although all table entries are still accessible by all threads of a block.

Experimental results can be seen in Figure 9, which shows speedup characteristics between the original and revised versions of individual test kernels. In all cases, measurements taken on the Fermi architecture are shown in the left column, whereas measurements taken on the Tesla architecture are shown in the right column. We took 30 measurements per kernel and averaged the results. According to our previous assumption that bank conflicts increase the overall runtime, the revised version of the bank-conflict kernel outperforms the original version by a factor of 2.50 on Fermi and 18.95 on Tesla (Figure 9).

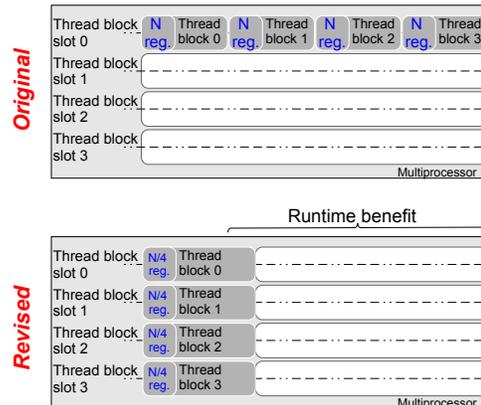


Fig. 6. Thread register imbalance.

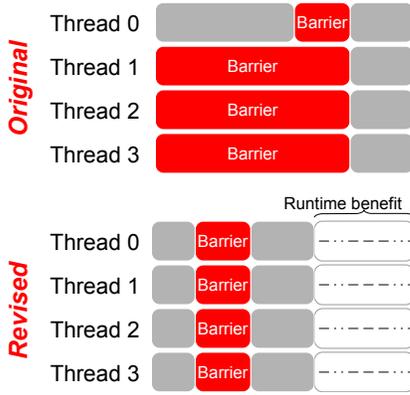


Fig. 7. Wait at barrier.

2) *Excessive global memory usage*: The microbenchmark for the excessive global memory usage pattern calculates $(A_i + B_i) \cdot A_i \cdot B_i / (A_i + B_i)$ for two given arrays A and B , which are stored in global memory. The revised kernel just copies both arrays into shared memory before performing the calculation. The measurement of these kernels shows a speedup of 1.20 on Fermi and 2.53 on Tesla (Figure 9), which may be larger for even more complex functions.

3) *Replication copy*: The microbenchmark for the replication copy pattern is implemented by a simple adder that calculates the element-wise sum of two arrays. The original version uses shared memory to cache the values from the input and output arrays, whereas the revised version just uses global memory directly. As can be seen in Figure 9, the effects on application performance are negligible because the speedup is in the range between 1.03 on Fermi and 1.01 on Tesla.

4) *Memory coalescing*: The original kernel for the memory coalescing pattern periodically copies 32 bit from global to shared memory. In contrast, the revised version directly copies 128 bit and so uses the entire data rate from global to shared memory. As can be seen in Figure 9, this optimized transfer leads to a performance improvement with a speedup of 1.26 on Fermi and 4.18 on Tesla.

5) *Scattered host copy*: To demonstrate the severity of scattered host copy pattern, the original kernel copies 1024 chunks of 1 KByte from the host to the device memory. In contrast, the revised version copies 1 MByte in a single stream from the host to the device memory. As can be clearly seen in Figure 9, this approach leads to an speedup of 17.48 on Fermi and 63.09 on Tesla. Therefore, scattered host copy scenarios can appear as a major performance bottleneck in CUDA applications.

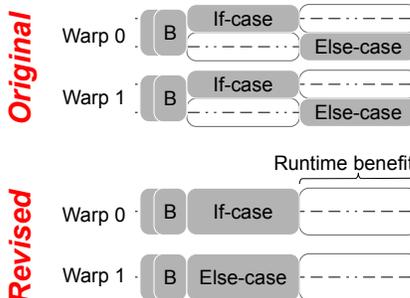


Fig. 8. Branch diverging.

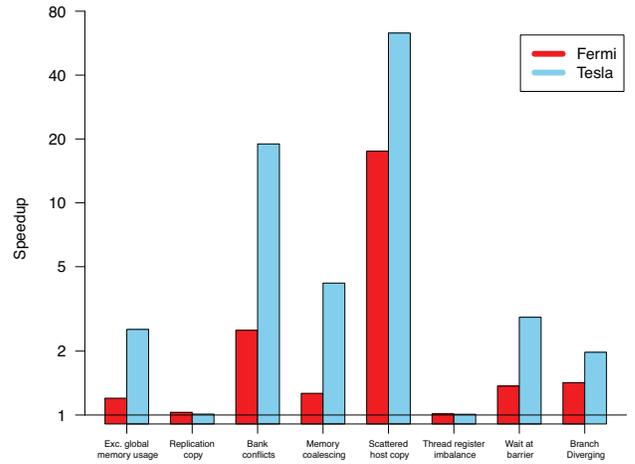


Fig. 9. Experimental results for speedups on Fermi and Tesla

C. Thread management patterns

1) *Thread register imbalance*: On Fermi architectures a single multiprocessor provides 1536 thread slots and 32K registers, whereas on Tesla architectures a single multiprocessor provides 1024 thread slots and 16K registers. To leverage the maximum number of thread slots, a single thread should therefore use at most 21 registers on Fermi and 16 registers on Tesla. The original kernel for the thread register imbalance uses 23 registers to perform its calculations. In contrast, the revised version just uses 16 registers. Given that both kernels are scheduled in blocks of 512 threads, on both architectures 512 thread slots are not used by the original kernel. The revised kernel shows a speedup of 1.01 on Fermi and Tesla (Figure 9), which may be larger if a thread register imbalance pattern coincides with memory access patterns, impeding latency hiding by the thread scheduler.

2) *Wait at barrier*: The microbenchmark for the wait at barrier pattern is derived from the bank conflict kernel. After assigning the array initialization to a single thread of a block, the kernel performs significantly better but now most threads of a block are idling during this initialization phase. In the revised version, the array initialization is evenly distributed among all threads within a block. As can be seen in Figure 9, such load balancing can lead to a speedup of 1.36 on Fermi and 2.89 on Tesla.

3) *Branch diverging*: The microbenchmark for the branch diverging pattern calculates the Euclidean distance between two given vectors. In the original version, threads of a warp perform those calculations in an alternating fashion. Of course, this scheduling results in branch diverging scenarios. In the revised version, this scenario is avoided by aligning the branch granularity to the warp size. As can be seen in Figure 9, experiments show a speedup of roughly 1.42 on Fermi and 1.97 on Tesla, indicating that branch diverging may adversely affect application performance.

IV. RELATED WORK

Implementing a test suite that demonstrates performance properties for a certain programming paradigm is a widespread technique to support the development of performance

tools. Hollingsworth et al. [7] released the Grindstone test suite to demonstrate possible performance properties in PVM programs. Grindstone focuses on communication patterns and also targets computational bottlenecks. Designed by Gerndt et al. [8], the APART test suite demonstrates MPI- and OpenMP-related performance problems, differentiating between different classes of performance properties.

To evaluate the capabilities of emerging accelerator architectures, Che et al. [9] examined a range of computationally demanding applications and showed that those applications could be significantly accelerated on graphics processors using CUDA. In addition, Stratton et al. [10] introduced a framework that allows CUDA programs to be executed effectively on general-purpose multi-core processors. Their experimental evaluation showed that CUDA is an advantageous data-parallel programming model for more than just GPU architectures. To also leverage parallelism of CUDA kernels on reconfigurable devices, Papakonstantinou et al. [11] proposed an FPGA design flow that combines the parallelism of different accelerator types (i.e., FPGAs and GPUs) through CUDA.

Finally, to support programmers in utilizing heterogeneous hardware, some performance tools exist that offer rudimentary support for CUDA. For example, the CUDA Visual Profiler not only provides runtime profiles but also hardware counter information associated with distinct execution phases [12]. Offering similar functionalities as the CUDA Visual Profiler, Parallel Nsight is available as a plug-in for Microsoft Visual Studio that allows programmers to develop for both GPUs and CPUs within the development environment [13]. Vampir is a visual trace browser allowing the fine-grained investigation of an application's runtime behavior [14]. Vampir can also be used to analyze the runtime behavior of CUDA applications since it provides visualization methods to illustrate memory transfer and kernel execution phases [15]. The TAU-CUDA profiler is able to measure CUDA applications using an experimental NVIDIA device driver specifically extended for performance measurements [16]. Boyer et al. [17] presented a proof of concept tool for identifying bank conflicts and find race conditions in CUDA programs at runtime.

V. CONCLUSION

In this paper, we identified performance-critical patterns of inefficient runtime behavior on GPUs, covering memory access and thread management scenarios. We also presented a microbenchmark test suite with kernels reproducing each of the corresponding behaviors. Using our test suite, we quantified the performance penalty of each pattern with results obtained on NVIDIA's Fermi and Tesla architectures. In our experimental evaluation, the most severe performance bottlenecks showed up in bank conflict, memory coalescing, and scattered host copy scenarios, indeed demonstrating significant delays. Nevertheless, measured speedups showed that most of the identified patterns are potential performance bottlenecks on both architectures.

To guide future tool development when using GPU devices through a CUDA compute engine, our microbenchmark test

suite, which we plan to publish online, can be used to evaluate the capabilities of future tools. Given that the amount of runtime performance data tends to become large on accelerator devices and scheduling thousands of threads creates another level of parallelism to be taken into account, future tools will require not only scalable measurement and analysis techniques but also new visualization strategies. Finally, we hope that vendors of heterogeneous systems and tools builders will agree on ways to exploit hardware counters for performance analysis, a feature that so far enjoys only limited support on GPUs.

REFERENCES

- [1] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, 1st ed. Addison-Wesley Professional, 2005.
- [2] T. Halfhill, "Parallel Processing with CUDA," *Microprocessor Report*, vol. 22, no. 1, pp. 1–8, 2008.
- [3] A. Munshi, "The OpenCL Specification," Khronos Group, 2010, v. 1.1 rev. 33. [Online]. Available: <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>
- [4] NVIDIA Corporation. [Online]. Available: www.nvidia.com
- [5] —, "CUDA C Best Practices Guide," 2010, v. 3.1. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf
- [6] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers, 2010.
- [7] J. Hollingsworth and M. Steele, "Grindstone: A Test Suite for Parallel Performance Tools," University of Maryland, Tech. Rep. CS-TR-3703, 1996. [Online]. Available: <http://www.lib.umd.edu/drum/bitstream/1903/852/2/CS-TR-3703.pdf>
- [8] M. Gerndt, B. Mohr, and J. L. Träff, "A Test Suite for Parallel Performance Analysis Tools," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 11, pp. 1465–1480, 2007.
- [9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.
- [10] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs," in *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing*. IEEE, 2008, pp. 16–30.
- [11] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "High-performance CUDA Kernel Execution on FPGAs," in *Proceedings of the 23rd International Conference on Supercomputing*. IEEE, 2009, pp. 515–516.
- [12] NVIDIA Corporation, "NVIDIA Compute Visual Profiler," v. 3.1. [Online]. Available: http://developer.nvidia.com/object/cuda_3_1_downloads.html
- [13] —, "NVIDIA Parallel Nsight." [Online]. Available: <http://developer.nvidia.com/object/nsight.html>
- [14] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool Set," in *Tools for High Performance Computing*. Springer, July 2008, pp. 139–155.
- [15] D. Hackenberg, G. Juckeland, and H. Brunst, "High Resolution Program Flow Visualization of Hardware Accelerated Hybrid Multi-core Applications," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 786–791.
- [16] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using CUDA," in *Proceedings of the 24th ACM International Conference on Supercomputing*. IEEE, 2010, pp. 127–136.
- [17] M. Boyer, K. Skadron, and W. Weimer, "Automated Dynamic Analysis of CUDA Programs," in *Third Workshop on Software Tools for Multi-Core Systems*. IEEE, 2008, pp. 1–6.