Visualizing Complex Dynamics in Many-Core Accelerator Architectures

Aaron Ariel, Wilson W. L. Fung, Andrew E. Turner and Tor M. Aamodt
University of British Columbia,
Vancouver, BC, Canada
aaronariel@hotmail.com {wwlfung,aturner,aamodt}@ece.ubc.ca

***Abstract*—While many-core accelerator architectures, such as today's Graphics Processing Units (GPUs), offer orders of magnitude more raw computing power than contemporary CPUs, their massive parallelism often produces complex dynamic behaviors even with the simplest applications. Using a fixed set of hardware or simulator performance counters to quantify behavior over a large interval of time such as an entire application execution run or program phase may not capture this behavior. Software and/or hardware designers may consequently miss out on opportunities to optimize for better performance. Similarly, significant effort may be expended to find metrics that explain anomalous behavior in architecture design studies. Moreover, the increasing complexity of applications developed for today's GPU has created additional difficulties for software developers when attempting to identify bottlenecks of an application for optimization. This paper presents a novel GPU performance visualization tool, AerialVision, to address these two problems. It interfaces with the GPGPU-Sim simulator to capture and visualize the dynamic behavior of a GPU architecture throughout an application run. Similar to existing performance analysis tools for CPUs, it can annotate individual lines of source code with performance statistics to simplify the bottleneck identification process. To provide further insight, AerialVision introduces a novel methodology to relate pathological dynamic architectural behaviors resulting in performance loss with the part of the source code that is responsible. By rapidly providing insight into complex dynamic behavior, AerialVision enables research on improving many-core accelerator architectures and will help ensure applications written for these architectures reach their full performance potential.**

I. Introduction

The slowing rate of single-thread performance growth for superscalar microprocessors has resulted in widespread interest in using many-core accelerators, such as *Graphics Processing Units* (GPUs) [23], [24], to accelerate non-graphics applications. These many-core accelerators use simple in-order cores and focus on throughput instead of latency to deliver *raw* computing power exceeding the teraFLOP barrier [3], [32] – orders of magnitude higher than what contemporary CPUs can achieve (tens of gigaFLOPS per core). A growing number of applications are being written with the explicit parallelism required to harness such architectures. The challenges of programming these architectures has driven efforts to make GPUs easier to use for non-graphics applications [2], [21], [28], [31].

In particular, the *Compute Unified Device Architecture* (CUDA) programming model [28], [31], an easy-to-learn extension of the ANSI C language introduced by NVIDIA

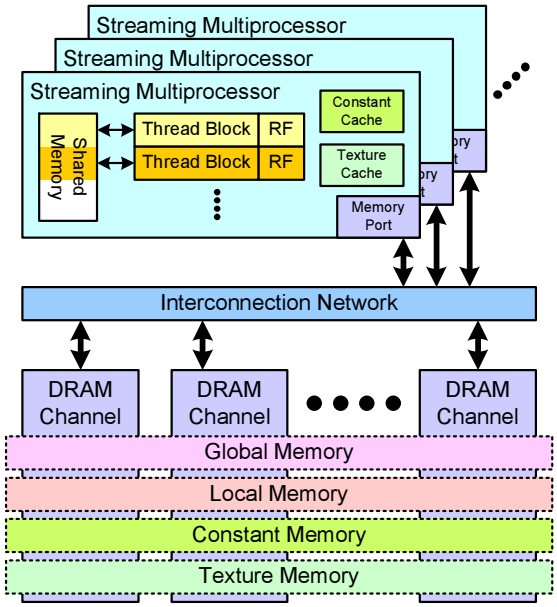


Fig. 1. Example of a many-core accelerator architecture: The CUDA GPU.

Corporation, has been used to accelerate many applications with the GPU. Figure 1 illustrates the underlying GPU many-core accelerator architecture exposed by the CUDA programming model. It consists of multiple processing cores, called streaming multiprocessors (SM), connected to multiple DRAM channels through an interconnection network. The programmer specifies blocks of parallel threads, each of which runs *scalar* code. Each SM is capable of running roughly one thousand of these threads, interleaving their execution in the pipeline to help cover the long latencies of memory operations. Inside the SM, individual threads are grouped into *warps* for synchronous execution using *single instruction, multiple data* (SIMD) hardware. This can be achieved using a hardware stack to support differing control flow among threads in a warp [22]. This execution model has been dubbed a *single instruction, multiple thread* (SIMT) model [23] to distinguish it from the more traditional SIMD model. Parallel threads in each block are launched to a single SM as a single unit of work. They may synchronize with each other and communicate through *shared memory*, a fast, on-chip scratchpad memory located within the SM. Each thread also has access to its own private memory space, called *local memory*, as well as to a public memory space, called *global memory*, that are both located in off-chip DRAM memory. All threads have access to *constant memory*

and *texture memory*, which are read-only memory spaces with on-chip caches. The CUDA Programming Guide [28], [31] contains more details on the distinct usage of each memory space. A GPU with tens of SMs can concurrently execute up to several hundred warps (thousands of threads), and generate hundreds of concurrent memory requests. This throughput-oriented design coupled with highly-parallel software allows the GPU to dedicate more of its silicon area to function units compared to a latency optimized CPU.

While many real world applications benefit tremendously from current GPU architectures, the massively-parallel nature of GPUs may produce complex dynamic behavior resulting in reduced throughput even with simple applications. Warps are interleaved in a fine-grained multithreaded manner and threads in different blocks run asynchronously with each other. Traditional software tuning techniques with performance counters that capture only the overall behavior of a GPU application may miss optimization opportunities that underlie the complex dynamic behaviors possible on such a system. Changes to the GPU architecture or to the application can perturb this complex behavior and can affect the overall performance. Any *a priori* fixed set of (simulator or hardware based) performance counters may not provide insight into these changes. Figure 2 shows the frequency of global memory write operations to each DRAM channel over time (darker color means more data written) for the CUDA application *transpose* (one of the CUDA SDK benchmarks [30]). It illustrates just one example of the above: the effect of intermittent congestion at each DRAM channel that results in an underutilization of available DRAM bandwidth. The lower bits (bits 8 to 10) of the address are used to partition the memory space across different DRAM channels. This scheme distributes memory accesses to successive 256-byte memory chunks across the eight DRAM channels of the simulated system to reduce the likelihood of congestion at a specific DRAM channel [18] (more details on the simulation setup are provided in Section III-A). While each DRAM channel is equally utilized overall, intermittent congestion results in lowered throughput. Metrics measuring the utilization of each channel would not have indicated this. In any given research scenario a detailed measurement metric could be developed to quantify a particular behavior (e.g., a measure of dynamic channel imbalance in this example), and/or a software developer with a detailed understanding of the application may already have significant insight into how their application behaves. However, narrowing down the causes of performance degradation suffered by a new application or architecture can be time consuming.

The need for performance visualization during processor architecture design and software tuning has long been recognized. SimpleScalar [11] has a visualizer for inspecting the pipeline timing of individual instructions [42]. Intel VTune^{TM} is a standard tool for software developers to access hardware performance counters for application tuning [20]. Other major integrated development environments (IDEs) also provide similar functionality [8], [20], [38]. Recently, NVIDIA has released the CUDA Visual Profiler, which accesses hardware

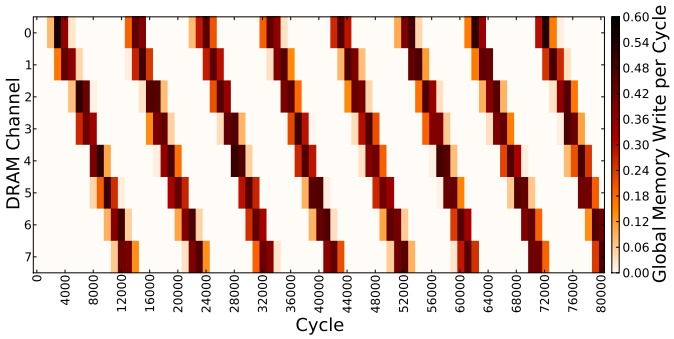


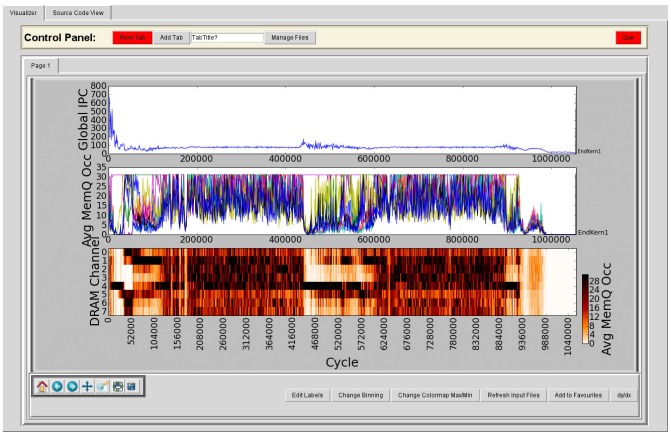
Fig. 2. Intermittent DRAM congestion concealed with traditional performance counter. See Section III-C for more detail.

performance counters in their GPUs and provides coarse-grained performance counter feedback helpful for identifying bottlenecks [29]. The IBM Performance Debugging Tool (PDT) [10] and Cetra [27] are visualization tools for the Cell processor. Tuning and Analysis Utilities (TAU) [37], Open|SpeedStops [1], and HPCToolkit [33] are performance analysis tool suites for large scale parallel applications.

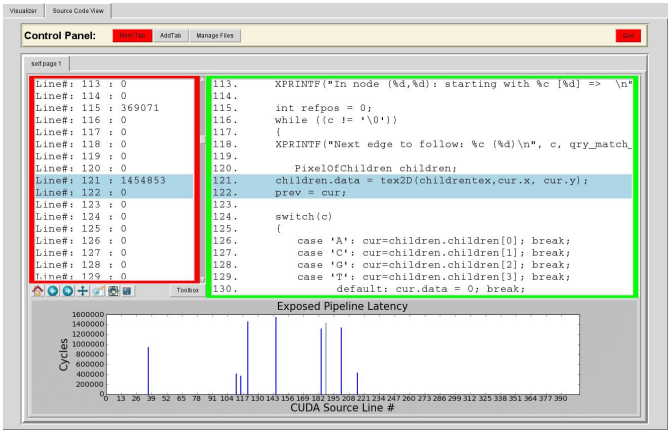
In this paper, we present a novel performance visualization tool, AerialVision. It provides a high-level graphical representation of performance events of CUDA applications running on the GPGPU-Sim simulator [9] at greater detail than existing hardware performance counter-based tools for GPUs. It also provides the ability to attribute microarchitecture events to individual lines in the application source code and a novel mechanism to relate these two levels of detail (a capability not present in current hardware based GPU profilers). In particular, this paper makes the following contributions:

- It presents a novel extensible visualization tool, AerialVision, to interface with GPGPU-Sim [9] for hardware and software design exploration on massively multithreaded, many-core accelerator architectures. Leveraging the capabilities of matplotlib [13], it can create publication quality figures useful for explaining complex behavior in many-core accelerator architectures (all data plots presented in this paper were created by AerialVision).
- It proposes a new methodology for correlating global runtime behavior to the source code that was running concurrently in the program.
- Using several case studies it demonstrates that:
  - Visualizing the dynamic behavior of a CUDA application can help identify non-trivial, intermittent performance bottlenecks.
  - Attributing performance statistics to individual lines of source code can simplify the process of applying existing optimization guidelines to CUDA applications.

AerialVision will enable architecture researchers to explore novel approaches to improve the cost-effectiveness and utility of many-core accelerator architectures more quickly. It will also help CUDA application developers understand performance bottlenecks at a finer granularity than existing tools. The source code for AerialVision along with a detailed usage



(a) Time-lapse view



(b) Source code view

Fig. 3. AerialVision

manual are distributed with GPGPU-Sim[1].

The rest of this paper is organized as follows. Section II introduces AerialVision and describes its key features in detail. Section III describes the implementation and simulation methodology used in this paper and presents several case studies highlighting the key features of AerialVision. Section IV discusses related work on performance visualization and performance tuning. Section V concludes the paper.

II. Visualizing the Dynamic Behavior of Thousands of Threads

AerialVision is a standalone tool, written in the Python scripting language that reads in a trace file produced by GPGPU-Sim version 2.1.1b (or later) along with CUDA program source and PTX assembly. The trace collection overhead is small: simultaneously collecting all 51 metrics currently supported in one simulation results in a 13% slowdown to simulation speed, and each sample interval is only 3.4 kB on average including PC-histogram data (described in Section II-C) or 605 B without the PC-histogram (by default a sample is created every 1000 cycles). AerialVision offers two different views for the user to analyze dynamic behavior: A time-lapse view and source code view. The time-lapse view

[1]`www.gpgpu-sim.org`

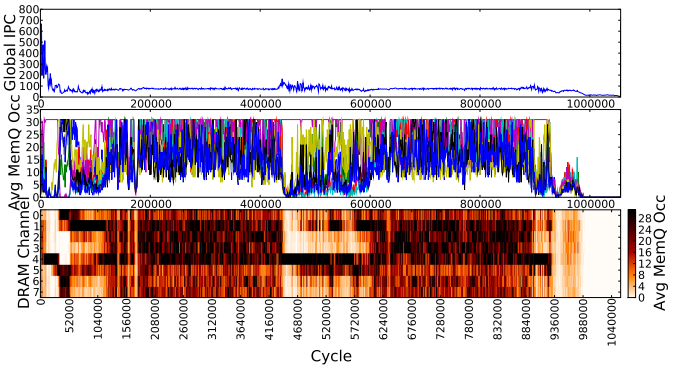


Fig. 4. Example illustrating basic plot formats provided by time-lapse view.

(shown in Figure 3(a)) allows the user to plot several performance metrics versus time so they may be compared. The source code view (shown in Figure 3(b)) displays performance statistics associated with individual lines in the application source code which helps guide the user toward bottlenecks in the source code that may require optimization.

In the rest of this section, we describe the key features of AerialVision. This section gives an overall understanding of the tool. We consider how these capabilities can be used by considering case studies in Section III.

A. Time Lapse View for Global Dynamic Statistics

Figure 4 illustrates three of the plot formats the time-lapse view supports (two other formats will be described subsequently). The time lapse view uses *matplotlib* [13], an open-source Python library, to generate individual plots from the input traces. AerialVision allows the user to place multiple subplots on the same window to simultaneously view different metrics, as well as to perform side-by-side comparisons between runs generated with different hardware and/or software optimizations. The plots can be zoomed and panned to improve legibility. The time-lapse view provides simple line plots for statistics aggregated across the whole GPU, such as the global dynamic IPC shown in the top plot in Figure 4. Statistics that relate to individual hardware units can be visualized using multi-line plots. For example, the average memory queue occupancy[2] for each DRAM channel is shown in the middle plot of Figure 4.

Intensity plots have occasionally been used to display performance-related data for multiple parallel hardware units versus time (e.g., Figure 2 in the work by Meng et al. [26]). To provide a clearer presentation of the statistics of individual units while maintaining a global view across all units, AerialVision can display the statistics of each unit over time using such a *parallel intensity plot*. For example, the bottom plot of Figure 4 also displays the average memory queue occupation for each DRAM channel shown in the middle

[2]In the GPGPU-Sim microarchitecture model, memory requests generated by each SM are sent to an out-of-order memory controller associated with the DRAM channel containing the data. The average memory queue occupancy measures the average number of requests that reside in the request queue inside the memory controller on any cycle during the sampling period.

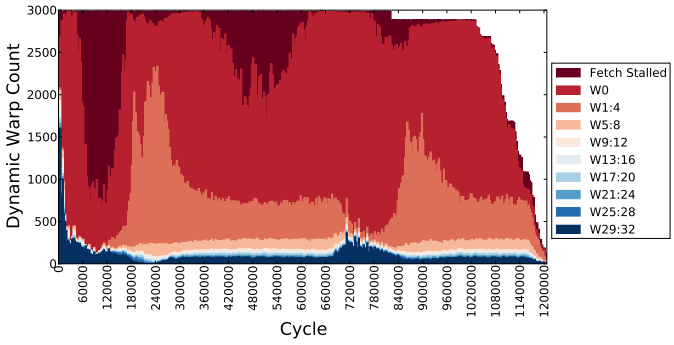


Fig. 5. Runtime warp divergence breakdown.

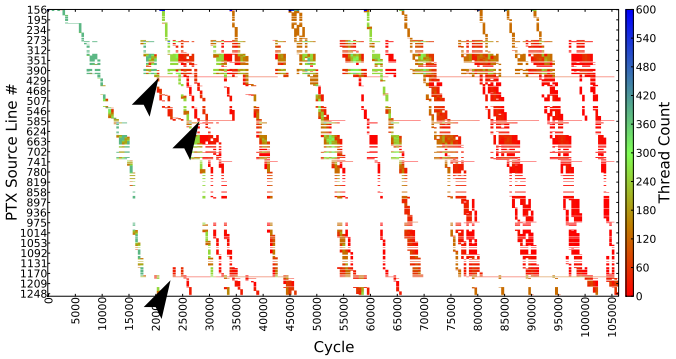


Fig. 6. PC-Histogram of a SM running ray tracing. White = Instruction has not been touched by any thread during the sampling period. Color = One or more threads have touched the instruction during the sampling period.

plot, but using a parallel intensity plot instead. From the parallel intensity plot, one can observe that there are moments in the program execution (e.g., between cycles 416000 and 468000) when the memory queue in DRAM channel #4 is full (represented by darker shading) and memory queues in other DRAM channels are poorly occupied (lighter shading). The ability to discover such intermittent performance bottlenecks in the microarchitecture can provide insights to the programmer or architect that may be useful in optimizing the application or hardware.

B. Runtime Warp Divergence Breakdown

Warp divergence, a phenomenon particular to GPU architectures from NVIDIA and AMD, occurs when threads inside a warp (or wavefront, in AMD terminology) take different control paths after a branch. Since SIMD hardware lacks the ability to process multiple control paths in a single clock cycle, a diverged warp executes instructions from different control paths on different cycles by masking off a subset of the functional units [22]. This serialization can lead to significant underutilization of the SIMD hardware in a many-core accelerator and is typically the performance bottleneck in control-flow intensive applications. To assist application developers wishing to understand the performance impact induced by warp divergence, the time-lapse view in AerialVision also provides the warp divergence breakdown for the simulated CUDA application versus time in addition to the simple line plots and the parallel intensity plots described in Section II-A.

Figure 5 shows the runtime warp divergence breakdown for MUMmerGPU [36]. Each warp issued for execution during the sampling period is classified into different categories according to the number of active threads in the warp. For example, category *W1:4* includes all warps with one to four active threads. Category *W0* denotes the idle cycles in each SM when all the warps in the SM are waiting for data from off-chip memory; whereas category *Fetch Stalled* denotes the cycles when the fetch stage of an SM stalls, preventing any warp from being issued that cycle. This plot is similar to the "active thread count distribution time series" presented by Fung et al. in their analysis of pathological behavior associated with one of their proposed warp scheduling policies [15].

The runtime warp divergence breakdown gives a clear view of the degree of warp divergence in an application, as well as its effect on the overall performance. The runtime breakdown in Figure 5 indicates that MUMmerGPU suffers from a high degree of warp divergence (during much of its runtime warps are in the W1:4 category). In addition, it shows that MUMmerGPU suffers from large memory latencies. This plot suggests that to achieve peak performance MUMmerGPU would need to reduce branch divergence and memory bandwidth bottlenecks. Each of these bottlenecks are separately addressed by mechanisms proposed by Fung et al. [15] and Tarjan et al. [41], but a unified approach has yet to be found.

C. From Runtime Behavior to Source Code Level Analysis

A third type of analysis that the time-lapse view provides is called the PC-Histogram. The PC-Histogram is a time series of histograms, representing the portion of the program that the threads have touched during a given sample period. A thread is considered to have *touched* an instruction after the instruction has been fetched in the pipeline (i.e., scheduled). The instruction continues to be considered touched by the thread until a new instruction is fetched by that thread. This series of histograms are displayed as a 2D color intensity plot, with X-axis being time, Y-axis being the program code layout linearly (in ascending PTX source line number or CUDA source line number), and the color intensity representing the number of threads that have touched the portion of the program in the sample period.

The PC-Histogram provides a collective view of how threads are traversing through the program over time. There are two ways to use this feature. First, if threads are not making forward progress they will show up as a horizontal line on the PC-Histogram plot (indicating that there are threads stuck at a single instruction for an extended period). The PC-Histogram of a SM running ray tracing [25] (shown on Figure 6) shows that some threads are stuck at the PTX instruction at line 419, 585 and 1181 (indicated by the dark arrows in Figure 6). A brief inspection of the PTX source code indicates that these are loop exits where threads in a diverged warp wait for other threads running extra iterations of the loop. The PC-Histogram also shows that even with a simple round-robin warp scheduler, most threads do not execute together.

Second, when put alongside plots of other runtime performance statistics, the PC-Histogram provides a powerful linkage between observed dynamic behavior and the corre-

TABLE I
PERFORMANCE STATISTICS AVAILABLE IN SOURCE CODE VIEW

Execution Count	The number of times this PTX instruction is executed by a thread in the simulator.
Total Pipeline Latency	The total number of SM pipeline cycles experienced by all the threads executing this PTX instruction. The SM pipeline cycles experienced by a thread are the number of cycles it spends in the SM pipeline.
Total Exposed Off-Chip Memory Latency	The total number of cycles where a long latency off-chip memory read operation prevents a warp from being issued and no other warp is available to issue.
DRAM Traffic	The number of memory accesses to DRAM (global/local memory access and texture/constant cache misses) generated by warps executing this PTX instruction.
Non-Coalesced Global Memory Access	The number of non-coalesced global memory accesses generated by warps executing this PTX instruction. When divided by *Non-Coalesced Warp Count*, the result measures the average number of memory requests issued by a warp each time this instruction is executed.
Non-Coalesced Warp Count	The number of dynamic warps executing this PTX instruction that generate non-coalesced memory accesses.
Shared Memory Access Cycle	The total number of SM pipeline cycles spent accessing shared memory by warps executing this PTX instruction. When divided by the next measure, the result measures the average number of bank conflicts experienced by this line of code.
Shared Memory Warp Count	The number of dynamic warps executing this PTX instruction and accessing the shared memory.
Branch Divergence	The number of dynamic warps that diverge after executing this PTX branch instruction.

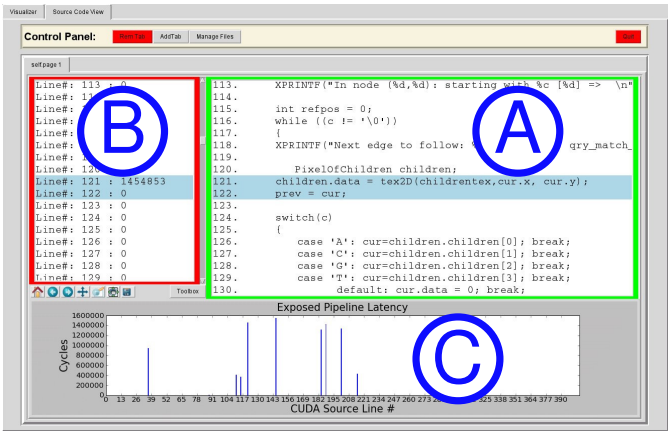


Fig. 7. Source Code View for Source Level Performance Analysis.

sponding source code. We will explore this in more detail with a case study in Section III-E.

D. Annotating Source Code with Performance Statistics

While the ability to visualize performance metrics of an application versus time provides detailed insight into causes of performance degradation, the source code view, shown in Figure 7, augments this by annotating the source code with aggregate performance statistics (similar in style to tools like VTune^{TM} [20]). The source code view consists of three parts. The *code viewer* (labeled **A** in Figure 7) displays the source code that is being analyzed. The user has the choice of viewing statistics associated with lines in PTX assembly code or CUDA source code. The *statistics viewer* (labeled **B** in Figure 7) displays the performance statistics associated with the lines of source code displayed on the code viewer. The type of statistics displayed in this viewer can be specified by the user. The *navigation graph* (labeled **C** in Figure 7) plots the performance statistics against their location in the source code. It provides an overview of the statistics throughout the whole program. Right-clicking on a bar in the navigation graph will bring the code viewer and the statistics viewer to the location of the program corresponding to the horizontal position of the cursor on the plot. This feature allows the user to navigate the program according to the statistics to find the source code of interest.

The performance statistics used by the source code view are collected as the application is simulated by GPGPU-Sim. The simulator tracks statistics for each PTX instruction. AerialVision uses debug information (`.loc` tags) in the PTX assembly code to aggregate the statistics collected for each PTX instruction into its associated statement in the CUDA source code. Table I lists the performance statistics currently available in source code view. AerialVision also provides the ability to visualize the ratios of two different performance statistics. For example, the ratio of *Non-Coalesced DRAM Traffic* and *Non-Coalesced Warp Count* reveals the average number of non-coalesced memory accesses generated by each warp executing a PTX instruction. This feature gives the user the ability to uncover further insights from existing statistics.

The case studies presented in Section III-D will illustrate how the source code view can help simplify performance tuning.

III. CASE STUDIES

In this section, we present several case studies that demonstrate the usage of the features described in Section II. The case studies presented in Section III-B and Section III-C demonstrate how AerialVision can assist hardware architects in design exploration. Section III-D and Section III-E present case studies illustrating the usefulness of AerialVision as an aid for software designers to uncover performance bottlenecks in an application. Note that our focus here is illustrating the potential of the tool rather than generating radical new insights into many-core accelerator architecture challenges or uncovering novel software optimizations.

A. Methodology

The data presented in this section (and throughout the paper) were collected by running CUDA applications on GPGPU-Sim [9]. Table II and III show the configuration we used with GPGPU-Sim for simulating a GPU microarchitecture similar to NVIDIA's Quadro FX 5800 (measured to have a 0.9 correlation coefficient versus real hardware on a set of kernels from the CUDA SDK).

We have extended GPGPU-Sim in version 2.1.1b to collect various runtime performance statistics and output them to a log file at regular intervals. Many of these statistics existed in earlier versions of GPGPU-Sim but were only reported as aggregate statistics at the end of simulation. All statistics are collected in one run of the simulator. In addition, aggregate per-*static* instruction performance statistics are saved into a separate PTX instruction profile written at the end of the

TABLE II
SIMULATED HARDWARE CONFIGURATION

Shader Core Frequency	325 MHz
Interconnect Frequency	650 MHz
DRAM Memory Bus Frequency	1300 MHz
Number of Shader Cores	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Thread Blocks / Core	8
Number of Registers / Core	16384
Shared Memory / Core	16 kB (16 banks, 2 access/cycle/bank)
Constant Cache Size / Core	8 kB (2-way set assoc. 64B lines LRU)
Texture Cache Size / Core	64 kB (2-way set assoc. 64B lines LRU)
Number of Memory Channels	8
GDDR3 Memory Timing	t_{CL}=9, t_{RP}=13, t_{RC}=34 t_{RAS}=21, t_{RCD}=12, t_{RRD}=8
Bandwidth per Memory Module	8 (Bytes/Cycle)
DRAM request queue capacity	32
Memory Controller	out-of-order (FR-FCFS [34])
Branch Divergence Method	Immediate Post-Dominator [14]
Warp Scheduling Policy	Round-Robin among ready warps

TABLE III
INTERCONNECT CONFIGURATION

Topology	Crossbar
Routing Mechanism	Destination Tag
Routing Delay	1
Virtual Channels	1
Virtual Channel Buffers	8
Virtual Channel Allocator	iSLIP
Alloc Iters	1
VC Alloc Delay	0
Input Speedup	1
Flit Size (Bytes)	32

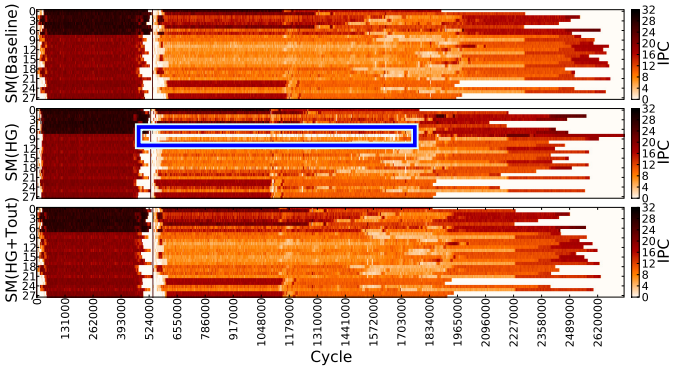


Fig. 8. Dynamic IPC of each SM for *LIBOR* with two different interconnection arbitration schemes. Top: Baseline Parallel Iterative Matching arbitration. Middle: Hold-Grant arbitration proposed by Yuan et al. to preserve row locality of memory access [44]. Bottom: Hold-Grant with time-out mechanism.

simulation. After AerialVision reads in these files, the user can interactively select which statistics to visualize.

B. Identifying Starvation due to Interconnection Arbitration

Yuan et al. proposed a new Hold-Grant arbitration scheme for on-chip interconnection networks to preserve the row locality of memory accesses across the interconnect [44]. This scheme, combined with a banked FIFO in-order DRAM scheduler, achieves up to 91% of the performance obtainable with an out-of-order FR-FCFS (first-ready first-come-first-serve) DRAM scheduler for a crossbar network [34]. When applied to a ring network, however, the Hold-Grant arbitration scheme was not as effective. The worst performing benchmark on a ring network with the Hold-Grant arbitration was LIBOR; most other benchmarks see slight performance improvements over a ring network with the baseline arbitration mechanism, parallel iterative matching (PIM).

LIBOR performs Monte Carlo simulations based on the London Interbank Offered Rate market model [16]. The application accesses the local memory space of each thread frequently and is hence memory-bound, making this an excellent benchmark for evaluating any memory system improvement.

To better understand the performance of LIBOR using Yuan et al.'s proposed mechanism, we started from their modified version of GPGPU-Sim [43], extended it to support AerialVision, and reran several simulations. Figure 8 shows the dynamic IPC of each SM for Hold-Grant (with and without time-out) and the baseline PIM arbitration mechanism. The middle part of the figure shows that, while most of the SMs finish their assigned work faster with Hold-Grant, one of the SMs (specifically, SM #8) suffers from starvation (inside box). This behavior can be removed with a time-out counter (bottom part of figure) which Yuan et al. evaluated while conducting their research [43]. The bottom part of the figure shows that, with the time-out mechanism, execution behavior looks almost the same as the baseline in that SM #8 no longer experiences any starvation. While performance is slightly better with the time-out mechanism (which would involve additional hardware), it is roughly the same as the baseline. From the performance difference alone it may not be obvious whether starvation was occurring without the time out mechanism. In the context of highly multi-threaded many-core architectures such performance effects can especially be masked by thread scheduling effects [9]: Small changes can cause one SM to finish a CTA (group of threads) earlier or later resulting in a different distribution of work to SMs at the end of a kernel. While introducing random perturbations and running each configuration several times is one way to analyze systems susceptible to such scheduling effects [6], AerialVision provides a systematic way to easily visualize intermittent behavior that may be localized to a portion of a many-core accelerator architecture and to a limited part of the execution time. As in this example, it can uncover behavior masked by other effects.

C. Exploring Memory Address Space Mappings to Reduce DRAM Contention

As discussed in Section I, a many-core architecture such as a GPU usually employs multiple DRAM channels to supply the memory bandwidth required to sustain thousands of in-flight threads. Both the local and global memory space are partitioned among multiple DRAM channels according to a memory address mapping. The address of each memory operation determines where the data will be placed in memory. While this design satisfies memory bandwidth requirements in theory, an application can under utilize the available memory bandwidth with a memory access pattern that makes poor use of a given address mapping by only accessing data in a subset of DRAM channels at any given instant of time.

Although sometimes this behavior can be detected with traditional performance counters and fixed accordingly, this form

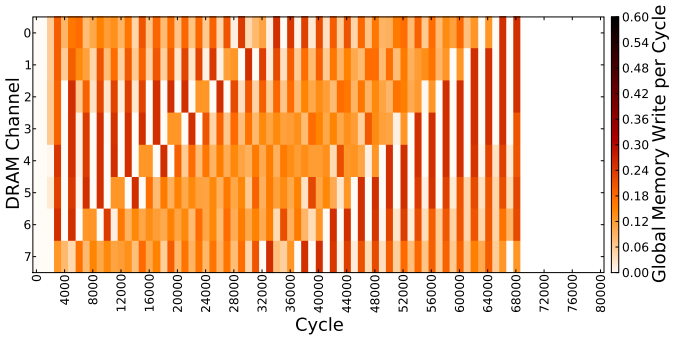


Fig. 9. Intermittent DRAM congestion removed with a different address mapping.

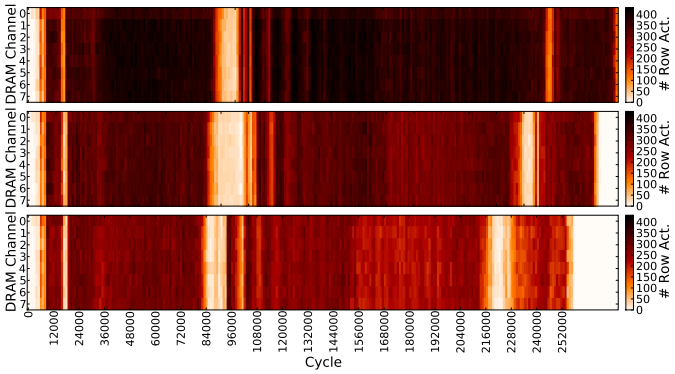


Fig. 10. Dynamic DRAM row activation count of each DRAM channel for *BFS* with three different address mappings. Top: DRAM channel specified by bits 6 to 8. Middle: DRAM channel specified by bits 8 to 10. Bottom: DRAM channel specified by bits 11 to 13, which shows the best performance of all three mappings.

of underutilization can also happen intermittently. Figure 2 shown earlier in Section I shows the number of global memory write accesses of *transpose* with a pathological memory access pattern that leads to intermittent DRAM congestion. Figure 9 shows how this congestion is removed (resulting in a 23% speedup) with a small change in the address mapping. The modified mapping uses bits 6 to 8 to direct the access to one of the eight DRAM channels in the system (instead of bits 8 to 10). There are four banks in each DRAM, and the bits used to map a memory access to a DRAM bank remain unchanged. Both address mappings distribute memory accesses to a relatively small memory footprint evenly among the DRAM channels, so performance metrics counting the overall accesses to each DRAM channel would show an even distribution of work in both cases. Thus, intermittent congestion may go unnoticed if only these metrics are measured.

To further illustrate how AerialVision can be used in architecture research, we highlight the fact that an address mapping that works well for one application may not perform as well for another application. Figure 10 shows the row activation count over time for each DRAM channel for BFS (breadth-first search) [17] with three different address mappings. The middle parallel intensity plot indicates a high frequency row buffer switching which reduces effective DRAM bandwidth leading to sub-optimal performance. Applying the change in address mapping that helped *transpose* spread memory accesses evenly across all DRAM channels reduces row buffer locality (the frequency of consecutive DRAM accesses hitting the same

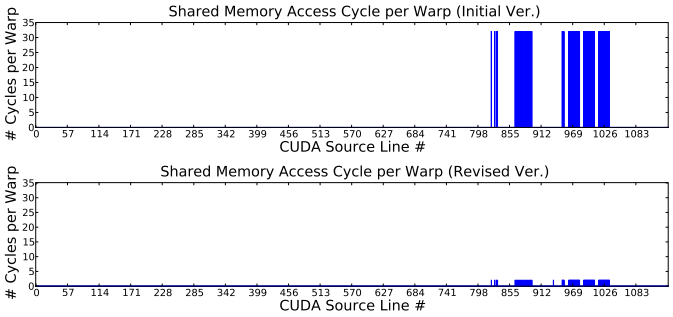


Fig. 11. The average shared memory access cycle of each CUDA source line for two versions of *StoreGPU*. Top: Initial version unaware of shared memory bank conflicts. Bottom: Revised version with the bank conflicts optimized out.

row in a DRAM bank) further as shown in the top plot by the darker shading. This results in lower performance. The bottom plot in Figure 10 shows the row activation count for BFS with a third address mapping that attempts to preserve the row locality while using the higher bits to spread the memory accesses across all DRAM channels. Performance improves by 9% over the best address mapping for *transpose*.

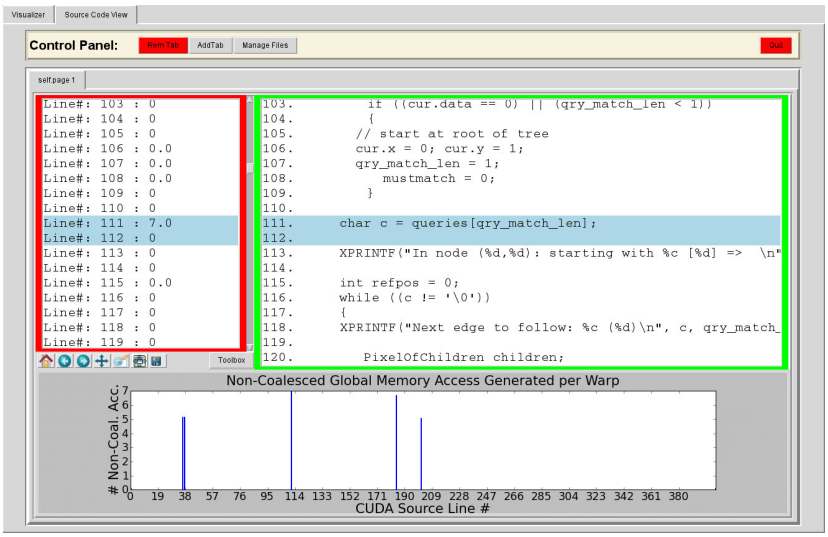
AerialVision enabled us to find this improved mapping quickly by providing a simple and easy to use interface to visualize different statistics.

D. Identifying Performance Bottlenecks at the CUDA Source Level

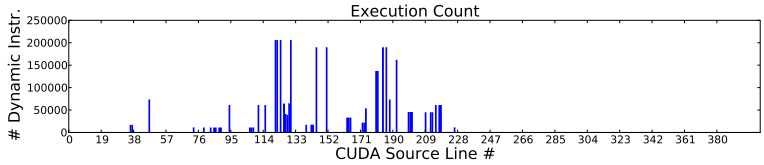
The case studies in the following sections demonstrate that the source code view simplifies the process of applying existing tuning techniques for software developers. The causes of the performance degradations discussed in these case studies are well-documented by NVIDIA [31]. However, current GPU hardware performance counters [29] may only provide coarse-grained (per kernel launch) hints for the specific parts of the code that are responsible for performance degradation, whereas AerialVision can guide software developers to the problematic source line quickly.

1) Shared Memory Bank Conflicts in StoreGPU: To support concurrent shared memory accesses from threads in a warp, the shared memory in each SM is divided into multiple memory banks [31]. A single memory bank can only process one access per cycle. Banks are organized such that successive 32-bit words are assigned to successive banks. Concurrent shared memory accesses from different threads in a warp can be serviced simultaneously if they map to different banks. Conversely, when these concurrent accesses map to the same bank, a bank conflict occurs and the accesses must be serialized, stalling in the SM pipeline and creating a performance bottleneck in the application. In our simulated GPU architecture, there are 16 banks in the shared memory running at twice the frequency of the pipeline. Each warp accesses shared memory in two groups of 16 threads, consuming a total of two shared memory cycles. This case study shows how the source code view in AerialVision can be used to pinpoint the line of source code causing shared memory bank conflicts in StoreGPU, simplifying the optimization process for software developers.

StoreGPU is a CUDA application developed by Al-Kiswany



(a) The number of non-coalesced memory accesses generated by each CUDA source line.



(b) Execution count of each CUDA source line.

Fig. 12. *MUMmerGPU* in source code view.

et al. [5] which uses the GPU to accelerate the MD5 and SHA1 hash calculations for chunks of data to be stored in a distributed network. In the first implementation, each thread fetches data directly from global memory and generates a significant amount of redundant DRAM traffic because each piece of data is accessed more than once. This limited the speedup of StoreGPU to about 4× of its CPU counterpart[3].

To reduce this redundant DRAM traffic, Al-Kiswany et al. evaluated using shared memory in StoreGPU as a temporary storage for the data chunk to be hashed by each thread. While the use of shared memory did cut down redundant DRAM traffic, the initial implementation laid out 64B chunks of data processed by each thread in a contiguous manner in shared memory and thus introduced significant bank conflicts. Nevertheless this implementation manages to achieve a 1.5× speedup over the original CUDA version [4]. It was not until about a month after creating this implementation that Al-Kiswany et al. noticed the significant inefficiencies due to bank-conflicts in shared memory and revised the data layout in shared memory to remove these bank conflicts [4]. The revised implementation achieved another 3.2× speedup (overall speedup of 5× over the original CUDA implementation).

With the source code view in AerialVision it is easy to locate shared memory accesses causing bank conflicts (a common concern for CUDA applications). Figure 11 shows the average shared memory access cycle per warp attributed to each CUDA source line for both implementations of StoreGPU. Many lines of code show 32 accesses per bank per warp in the initial shared memory implementation and the very same lines in the revised implementation only show 2 accesses per bank per warp. In the Quadro FX 5800 we configured GPGPU-Sim to model, shared memory accesses from a warp with 32 threads are split into 2 halves. Each half is serviced in parallel by 16 memory banks, so shared memory accesses from a warp with no bank conflicts will generate 2 accesses per bank. A performance counter measuring the number of bank conflicts will be able to indicate the existence of bank conflicts whereas the source level analysis tool in AerialVision can pinpoint the line the source code causing the bank conflicts, simplifying the optimization process for software developers. While the CUDA Bank Conflict Checker can detect which lines cause shared memory bank conflicts [30] it requires source level modifications whereas AerialVision does not. Furthermore, AerialVision can be extended to measure any other metric of interest in a similar fashion (we currently measure the performance metrics listed in Table I).

2) Non-Coalesced Global Memory Accesses in MUMmerGPU: While the CUDA programming model allows threads within a warp to have random access to global memory, DRAM memory bandwidth can only be fully utilized in the GPU hardware with coalesced memory accesses [31]. To achieve optimal memory performance, the GPU hardware always attempts to coalesce global memory accesses by threads of a half-warp (groups of contiguous 16 threads) in hardware into a single DRAM transaction, known as a coalesced access, to fully utilize the already scarce off-chip DRAM bandwidth. Non-coalesced accesses can potentially become the performance bottleneck of an application by wasting DRAM bandwidth since the amount of data requested

[3]Using the sliding window configuration in StoreGPU [5].

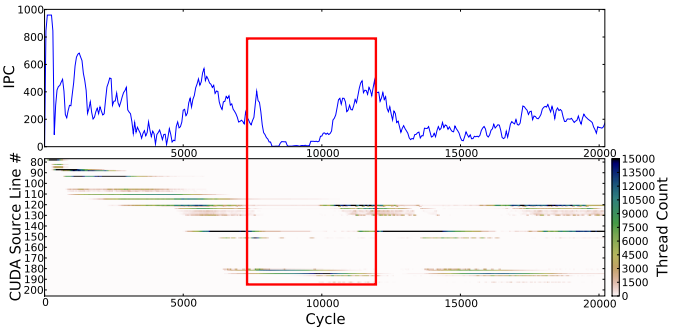


Fig. 13. PC-Histogram example: Performance degradation associated with line 186.

may be much less than the transfer size of the DRAM (16 to 32 bytes per burst for contemporary graphics DRAM). In this case study, we present how AerialVision can help software developers identify the lines in their source code that generate non-coalesced accesses.

MUMmerGPU is a parallel pairwise local sequence alignment program that matches query strings consisting of DNA nucleotides (A,C,T,G) to a reference string for purposes such as genotyping, genome resequencing and metagenomics [36]. In this study we use MUMmerGPU 1.1.

Tarjan et al. recently noted that the input query strings in MUMmerGPU 1.1 are laid out contiguously in memory, leading to performance loss [41]. Since each thread in a warp reads a separate input string from global memory, each query string read operation from a warp generates non-coalesced memory accesses that reduces the efficiency of the application. Tarjan el al. uncovered this inefficiency via simulation and rearranged the input query string in an interleaved layout to remove this inefficiency.

While these non-coalesced accesses were eventually noticed by an experienced GPU architect, the developers of MUMmerGPU could have noticed such an inefficiency using AerialVision. Figure 12(a) shows the number of non-coalesced memory accesses generated per warp for each CUDA source line in MUMmerGPU. The plot indicates that all of the source lines (Line 111, 186 and 205) accessing the query strings generate more than 5 memory accesses on average for each warp (while each would generate only two with fully coalesced accesses). With this plot, we also noticed some non-coalesced accesses generated from the dereferencing of a pointer in set_result() at Line 36 and 37. However, we also checked the execution count (shown in Figure 12(b)) and found that this function is executed infrequently.

E. Identifying Performance Bottlenecks with a PC-Histogram

Figure 13 illustrates a novel feature of AerialVision—the PC-histogram. The PC-histogram shows, for a given sample period, which lines in the source code were executed. The example illustrates how this feature can be used to identify performance issues at the source code level in a more general way than the source code annotation approach described in the previous sections. This figure again shows the execution of MUMmerGPU. The box highlights a region where performance suddenly drops as can be seen in the upper plot (IPC). The bottom plot (PC-histogram) shows that only a couple of lines of code are executed for most of the execution time during this region. These lines correspond to the non-coalesced accesses identified in the earlier section. Note that, whereas the source code annotation feature in the prior section requires some knowledge of the particular performance problem, the PC-histogram can identify lines of source code that are correlated to performance problems even if no metric has yet been identified to measure the specific problem. We have used the PC-histogram feature to identify additional optimizations in MUMmerGPU (not shown). Specifically, we used it to discover that texture memory accesses to the root of the tree-based data structure in MUMmerGPU may cause imbalances in DRAM utilization (eliminating this bottleneck is beyond the scope of this paper).

IV. Related Work

IBM Performance Debugging Tool (PDT) [10] is a user-level tracer developed by IBM that records events specific to the Cell processor during application execution. The tracing mechanism of PDT is implemented by instrumenting the libraries that implement the Cell SDK functionality. At each traced event a PDT hook routine is called to send the trace data to a global trace buffer in the main memory—an overhead that may perturb the behavior of the measured application. With AerialVision, the developers run their applications through GPGPU-Sim. The extra instrumentation code in GPGPU-Sim for collecting data for AerialVision may slow down the simulation by about 13%, but the simulated application behavior is not affected.

Cetra [27] is a collection of tools, complementary to PDT, that provides a mechanism to obtain traces of the interactions between the user application and the OS kernel on Cell. It is designed to provide information on the runtime scheduling decisions of the kernel-level scheduler. The captured traces are post-processed into textual traces visualized by Paraver [12].

Cetra and AerialVision are both used to study global runtime behavior of a parallel processor. Cetra captures OS-level task management events such as contexts and context switching overhead. Instead of capturing individual events that would lead to a trace size explosion on a massively multithreaded GPU architecture, AerialVision focuses on visualizing runtime performance metrics that capture the aggregate effects of these events. This allows AerialVision to serve the purpose of both PDT and Cetra: the user can zoom out to grasp the overall runtime behavior of an application without losing track of the effects from the low-level events.

Paraver [12] is a flexible trace analysis tool that allows the user to visualize global behavior as well as inspecting individual events in textual traces. Paraver trace generation tools can capture hardware performance counters and events from parallel applications written in OpenMP, MPI and Java for visualization. Paraver's graphic view layouts individual runtime events associated with a particular hardware node in a horizontal color stripe. This is similar to the parallel intensity plots in the time-lapse view of AerialVision, except instead

of displaying each individual event, parallel intensity plots display runtime performance metrics that capture the aggregate effects of these events. In addition, Paraver cannot directly expose performance statistics to the source code level for performance tuning (an important feature of AerialVision which allows users to pinpoint the bottlenecks of an application).

Tuning and Analysis Utilities (TAU) [37] is a performance analysis tool framework for parallel Fortran, C++, C, Java, and Python applications that uses source code and static binary instrumentation. Open|SpeedShop (O|SS) [1] is a modular open-source performance analysis tool framework that supports dynamic instrumentation for applications running on Linux clusters. Both tools feature two modes of performance data collection: profiling (or sampling experiment in O|SS) that attributes sampled performance metrics to different parts of an application for a performance overview, and tracing (or tracing experiment in O|SS) that records individual events in detail. Both tools provide flexible graphical user interfaces for visualizing the collected data (ParaProf in TAU and a custom python tool in O|SS). AerialVision's time-lapse view attempts to bridge between these two modes for many-core accelerator architectures, in which browsing through individual events can be overwhelming while aggregate performance metrics alone fail to capture some important performance dynamics.

HPCToolkit [33] is a language-independent tool suite designed to measure the performance of fully-optimized executables generated by vendor compilers. Instead of relying on compiler-generated debug information to map an instruction to its associated source code line (what AerialVision currently does in source code view), HPCToolkit analyzes the application binaries directly to recover the program structure [40]. It then correlates the statistically sampled performance metrics with the source code structure and presents the metrics with the associated source code. HPCToolkit has been applied to petascale parallel systems [39].

The Intel Performance Tuning Utility (Intel PTU) [7] is a performance analyzer enhanced with features to assist performance tuning for parallel applications. Similar to the Intel VTune^{TM} Performance Analyzer [20], Intel PTU interfaces directly with the performance monitoring unit (PMU) on Intel processors to provide low-overhead event sampling. The sampled performance events, such as cache misses and branch mispredictions, are presented to the user together with the source code that causes the events. This allows the user to discover hotspots in the application quickly and helps focus their efforts into optimizing bottlenecks found in these hotspots. Other IDEs also provide similar features [8], [38]. AerialVision is, to our best knowledge, the first tool providing this feature for CUDA applications.

The Graphical Pipeline Viewer (GPV) [42], in conjunction with the SimpleScalar tool set [11], provides a detailed view for each instruction in the pipeline. GPV shows the pipeline stage transitions and various architectural events associated with each instruction. While GPV is useful for analyzing details of an out-of-order superscalar processor with a small window of in-flight instructions, a similar tool for a massively-parallel architecture such as a GPU would require the user to monitor the progresses of tens of thousands of in-flight instructions distributed in tens of cores simultaneously. Even if the tool could provide an interface to do so, the sheer amount of data would overwhelm the user. AerialVision instead focuses on visualizing the global runtime behavior of the whole GPU.

Due to the complex, non-linear nature of GPUs, a methodology for optimizing CUDA applications has been an ongoing research topic. Ryoo et al. [35] have demonstrated the complexity involved in optimizing applications on GPUs and proposed a methodology for pruning the optimization search space to speed up the process. Hong et al. [19] have proposed a simple analytical model estimating the execution time of CUDA applications to provide insights into identifying performance bottlenecks for these applications.

NVIDIA has recently released the CUDA visual profiler [29], a performance profiling tool exposing part of the hardware performance counters in the GPU to CUDA software developers. The CUDA visual profiler reports aggregate performance statistics at the end of each kernel launch. The runtime behavior of the profiled CUDA application is not exposed with these performance statistics. Moreover, due to constraints in the GPU hardware, the profiler can only target one of the SMs in the GPU. To produce an accurate measurement, the CUDA application needs to have reasonably homogeneously-behaving thread blocks and to launch enough threads to keep the GPU busy for an extended period so that the probed SM gets a representative share of the overall workload. CUDA applications with irregular workloads per thread block may not be able to satisfy these requirements. AerialVision, on the other hand, collects the runtime statistics of a CUDA application running on GPGPU-Sim. Statistics from all simulated hardware units contribute to the measurement.

V. Conclusions and Future Work

Due to its complexity, optimization for many-core accelerator architectures has been challenging for hardware/software developers. As we have shown in the paper, subtle design choices in a many-core architecture design can significantly alter overall performance. Some of these design choices can lead to intermittent inefficiencies (such as congestion and starvation) in the system. Discovering these inefficiencies can be non-trivial without a visual inspection of the runtime behavior of the microarchitecture. In this paper, we presented AerialVision, a novel GPU visualization tool interfaced with the GPGPU-Sim simulator to provide hardware/software developers with insights on the performance bottlenecks of various CUDA applications. AerialVision provides a time-lapse view for visualizing the global runtime behavior of various CUDA applications on a many-core architecture. This feature helps hardware and software developers identify sources of dynamic and intermittent inefficiency. In part to help software designers pinpoint bottlenecks in their applications, AerialVision provides a source code view that annotates individual lines of source code with performance statistics. The PC-Histogram provides an orthogonal feature that can pinpoint

intermittent inefficiencies in applications at the source code level. The case studies presented in this paper demonstrated that AerialVision provides a highly effective means for hardware designers and software designers alike to search for performance bottlenecks in applications running on massively-parallel many-core architectures such as GPUs.

ACKNOWLEDGMENTS

We thank Johnny Kuan, Henry Wong, and the anonymous reviewers for their valuable comments on this work. We also thank Ali Bakhoda for his help in implementing part of the interface between GPGPU-Sim and AerialVision and his valuable feedback on this work. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] "Open|SpeedShop." [Online]. Available: http://www.openspeedshop.org/wp/

[2] *ATI CTM Guide*, 1st ed., Advanced Micro Devices, Inc., 2006.

[3] *Press Release: AMD Delivers Enthusiast Performance Leadership with the Introduction of the ATI Radeon HD 3870 X2*, http://www.amd.com, Advanced Micro Devices, Inc., 28 January 2008.

[4] S. Al-Kiswany, "Personal Communication," 2009.

[5] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems," in *Proc. 17th Int'l Symp. on High Performance Distributed Computing*, 2008, pp. 165–174.

[6] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proc. 9th Int'l Symp. on High Performance Computer Architecture*, 2003, pp. 7–18.

[7] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev, "Parallelization Made Easier with Intel Performance-Tuning Utility," *Intel Technology Journal*, vol. 11, no. 4, 2007.

[8] Apple Inc., "Optimizing with Shark." [Online]. Available: http://developer.apple.com/tools/shark_optimize.html

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2009)*, April 2009, pp. 163–174.

[10] M. Biberstein, U. Shvadron, J. Turek, B. Mendelson, and M. Chang, "Trace-based Performance Analysis on Cell BE," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2008)*, April 2008, pp. 213–222.

[11] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," http://www.simplescalar.com, 1997.

[12] CEPBA, "Paraver - Parallel Program Visualization and Analysis tool - REFERENCE MANUAL," 2001.

[13] D. Dale, M. Droettboom, E. Firing, and J. Hunter, "Matplotlib User's Guide," http://matplotlib.sourceforge.net/Matplotlib.pdf.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[15] ——, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, pp. 1–37, 2009.

[16] M. Giles and S. Xiaoke, "Notes on Using the NVIDIA 8800 GTX Graphics Card." [Online]. Available: http://people.maths.ox.ac.uk/~gilesm/hpc/

[17] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *HiPC*, 2007, pp. 197–208.

[18] M. Harris, "UNSW CUDA Tutorial Part 4 – Optimizing CUDA." [Online]. Available: http://www.cse.unsw.edu.au/~pls/cuda-workshop09/slides/04_OptimizingCUDA_full.pdf

[19] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," *Proc. 36th Int'l Symp. on Computer Architecture*, vol. 37, no. 3, pp. 152–163, 2009.

[20] Intel Corp., "Intel VTune^{TM} Performance Analyzer." [Online]. Available: http://software.intel.com/en-us/intel-vtune/

[21] *OpenCL 1.0 Specification*, 1st ed., Khronos Group, 2009.

[22] A. Levinthal and T. Porter, "Chap - a SIMD Graphics Processor," in *Proc. 11th Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, 1984, pp. 77–82.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[24] Marco Chiappetta, "ATI Stream Computing: ATI Radeon^{TM} HD 3800/4800 Series GPU Hardware Overview." [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/pages/Publications.aspx

[25] Maxime, "Ray tracing," http://www.nvidia.com/cuda.

[26] J. Meng, D. Tarjan, and K. Skadron, "Leveraging Memory Level Parallelism Using Dynamic Warp Subdivision," Department of Computer Science, University of Virginia, Tech. Rep. CS-2009-02, 2009.

[27] J. Merino, L. Alvarez, M. Gil, and N. Navarro, "Cetra: A Trace and Analysis Framework for the Evaluation of Cell BE systems," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2009)*, April 2009, pp. 43–52.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar.-Apr. 2008.

[29] *NVIDIA CUDA Visual Profiler*, 1st ed., NVIDIA Corp., 2008. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/cudaprof_2.3_readme.txt

[30] NVIDIA Corporation, "NVIDIA CUDA SDK code samples." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[31] *NVIDIA CUDA Programming Guide*, 1st ed., NVIDIA Corporation, 2007.

[32] *Press Release: NVIDIA Tesla GPU Computing Processor Ushers In the Era of Personal Supercomputing*, http://www.nvidia.com, NVIDIA Corporation, 20 June 2007.

[33] Rice University, "HPCToolkit." [Online]. Available: http://hpctoolkit.org/

[34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proc. 27th Int'l Symp. on Computer Architecture*, 2000, pp. 128–138.

[35] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, J. Stratton, and W. W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," in *Proc. 6th Int'l Symp. on Code Generation and Optimization (CGO)*, April 2008, pp. 195–204.

[36] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-Throughput Sequence Alignment Using Graphics Processing Units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/474

[37] S. S. Shende and A. D. Malony, "The TAU Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[38] Sun Microsystems, "Sun Studio Performance Analyzer." [Online]. Available: http://developers.sun.com/sunstudio/

[39] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing Performance Bottlenecks in Emerging Petascale Applications," in *ACM/IEEE Conference on Supercomputing (SC'09)*. ACM, 2009, pp. 1–11.

[40] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary Analysis for Measurement and Attribution of Program Performance," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*, 2009, pp. 441–452.

[41] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD Cores," in *ACM/IEEE Conference on Supercomputing (SC'09)*, 2009.

[42] C. Weaver, K. C. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance Analysis Using Pipeline Visualization," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE Int'l Symp. on*, 2001, pp. 18–21.

[43] G. Yuan, "Personal Communication," 2009.

[44] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *Proc. 42th IEEE/ACM Int'l Symp. on Microarchitecture*, 2009.