Computer Science and Information Systems 00(0):0000–0000 https://doi.org/10.2298/CSIS123456789X

Demystifying Power and Performance Variations in GPU Systems through Microarchitectural Analysis ⋆

Burak Topcu⋆⋆[1] Deniz Karabacak[2] and Işıl Öz[3]

[1] The Pennsylvania State University
Department of Computer Science and Engineering, State College, PA, USA
topcuuburak@gmail.com
[2] Izmir Institute of Technology
Electrical and Electronics Engineering Department, Izmir, Turkey
denizkarabacak@std.iyte.edu.tr
[3] Izmir Institute of Technology
Computer Engineering Department, Izmir, Turkey
isiloz@iyte.edu.tr

Abstract. Graphics Processing Units (GPUs) serve efficient parallel execution for general-purpose computations at high-performance computing and embedded systems. While performance concerns guide the main optimization efforts, power issues become significant for energy-efficient and sustainable GPU executions. Profilers and simulators report statistics about the target execution; however, they either present only performance metrics in a coarse kernel function level or lack visualization support that can enable microarchitectural performance analysis or performance-power consumption comparison. Evaluating runtime performance and power consumption dynamically across GPU components enables a comprehensive tradeoff analysis for GPU architects and software developers. In this work, we present a novel memory performance and power monitoring tool for GPU programs, GPPRMon, which performs a systematic metric collection and provides useful visualization views to guide power and performance analysis for target executions. Our simulation-based framework dynamically gathers SM and memory-related microarchitectural metrics by monitoring individual instructions, and reports achieved performance and power consumption information at runtime. Our visualization interface presents spatial and temporal views of the execution. While the first demonstrates the performance and power metrics across GPU memory components, including global memory, cache, and SMs, the latter shows the corresponding information at the instruction granularity in a timeline. Based on our framework, we demonstrate performance and power analysis for memory-bound graph applications and resource-critical embedded programs from GPU benchmark suites. Our case studies reveal potential usages of our tool in memory-bound kernel identification, performance bottleneck analysis of a memory-intensive workload, performance-power evaluation of an embedded application, and the impact of input size on the memory structures of an embedded system.

Keywords: GPU Computing, Performance monitoring, Power consumption

⋆ This article is an extended version of our previously published workshop paper, "GPPRMon: GPU Runtime Memory Performance and Power Monitoring Tool, Burak Topçu, Işıl Öz, Workshop on Resource AWareness of Systems and Society (RAW), co-located with International European Conference on Parallel and Distributed Computing (Euro-Par), 2023."

⋆⋆ Work was done when the author was at Izmir Institute of Technology.

1. Introduction

As high-performance and energy-efficient computation requirements increase in data processing tasks, parallel GPU architectures, with heterogeneous components, play an essential role in accelerating parallel workloads such as streaming and graph applications [40, 47]. Since GPU devices have evolved into more complex systems with the recent technological developments, efficiency and throughput utilization require more detailed research effort. As a result of high computational capacity, energy consumption and power issues have become crucial in GPU-based systems [22].

While GPU devices have large computational power with multiple processing units, the performance and energy efficiency may decline for memory-intensive workloads due to the high pressure on memory units by concurrently executing multiple threads. While several works discuss the impacts of the memory wall problem for the GPUs [10, 12], there are also studies explaining the root causes of memory bottleneck points for GPU applications [20, 32] and proposing various improvements for the problem [13, 16, 46, 50]. Additionally, the researchers propose energy-efficient methods by considering power consumption of the memory operations on GPU programs [6, 36]. While both performance and energy improvements contribute to the efficient execution of GPU programs, they may compete with each other, and the design decisions become critical and get complicated, requiring the evaluation of the tradeoffs between performance and energy efficiency [4, 9, 17, 42].

Performance and energy efficiency analysis for GPU execution requires low-level measurements at runtime and detailed evaluations of the performance bottlenecks and power consumption. Besides data-parallel programs utilizing SIMD parallelism in GPU resources, general-purpose task-parallel applications introduce more sophisticated algorithm implementations. Therefore, evaluating the execution for performance and power consumption at the kernel function level hides most of the clear evidence for conducting a baseline analysis from many perspectives. More fine-grained level analysis, where the individual instructions belonging to a warp, the smallest execution element in the GPU execution context, should be performed throughout the execution on SM resources. However, profiling and simulation tools collect and report GPU performance results at the kernel level. For instance, the NVIDIA Nsight Compute Tool [27], which presents occupancy, IPC, and memory utilization metrics, operates on a kernel basis. Similarly, the state-of-the-art GPU simulation tools [15, 45] report the performance and hardware metrics at the kernel level. None of the tools directly reports GPU programs' dynamic performance, memory access behavior, and power consumption at runtime. While profiling tools and simulation frameworks report runtime statistics, the software developers and researchers spend additional effort to collect related metrics from the experiments. In other words, several in-house target-specific works still exist to monitor the runtime performance and power consumption for a GPU execution, and repetitive studies cause redundant effort.

While a set of profiling tools helps software developers to understand performance and power consumption information about GPU execution, microarchitecture-level simulators are quite significant in modeling hardware and monitoring the runtime application behaviors. The simulators target micro and macro scales by collecting performance and energy metrics. In GPU-related research, GPGPU-Sim [15] and Multi2Sim [45] simulators have been prominent in offering accurate hardware models for NVIDIA and AMD GPUs, re-

spectively, among the other simulators [3, 7, 34]. Moreover, the developer communities of these simulators have provided continuity by incorporating new GPU architectures and optimizations introduced in GPU hardware and software. Among the top five hundred computer systems [41], 179 use NVIDIA-based GPUs as the accelerator/co-processor technology; specifically, 84 include NVIDIA Volta, and 64 use NVIDIA Ampere devices.

In this work, we build and implement, **GPPRMon**, a performance and power monitoring tool, for GPU programs executing on top of a simulation environment. We aim to close the gap between the profilers' high-level static results and the cycle-accurate simulators' large-volume raw data about the execution. Not only do we set up GPPRMon built on configurable simulation execution, but we also generate abstractions and provide visualization views that are easy to capture and comprehend large amounts of data. Our tool presents both dynamic and configurable simulation execution, and rich architectural profiling views by combining the best of both worlds.

As potential users of the GPPRMon, we target program developers, system architects, and researchers, who are working to optimize GPU software or hardware, considering both performance improvement and energy efficiency. Our simulation-based framework dynamically collects microarchitectural metrics by monitoring individual instructions' issues/completions and reports achieved performance and power consumption information at runtime. Hence, it enables the users to analyze the dynamic behavior of memory accesses and thread blocks during the target program execution. Based on the detailed characteristics collected at runtime, our visualization interface presents both spatial and temporal views of the execution, where the first demonstrates the performance and power metrics for each hardware component, including memory units and SMs; and the latter shows the corresponding information at the instruction granularity in a cycle-based timeline. Our tool enables the users to perform a fine-granularity evaluation of the target execution by observing instruction-level microarchitectural features related to performance and power consumption at runtime. In this article, we extend our previously published workshop paper [44] by including additional case studies that demonstrate the usage scenarios of our tool. To the best of our knowledge, this is the first work monitoring a GPU kernel's performance by visualizing the execution of instructions for multiple user-configurable scenarios, relating memory hierarchy utilization with performance, and tracking power dissipation at runtime. Our main contributions are as follows:

- We design a systematic microarchitectural metric collection methodology that keeps track of instruction-per-cycle (IPC) per streaming multiprocessor (SM) as a performance metric, instruction execution records for each warp to observe issue and completion cycles, memory statistics per each component in the memory hierarchy to understand the possible impacts on performance and power-related statistics per each GPU component at runtime. We build our configurable collection framework on top of the GPGPU-Sim simulation environment [15], which provides cycle-accurate information about the execution.
- Based on the information gathered by our metric collection module, we design and build a visualization framework that executes in parallel to our collection module and generates displays and charts for each kernel execution with the following three perspectives:

1. *General View* displays the average IPC among SMs, access statistics of L1 and L2 caches, row buffer utilization of DRAM partitions, and dissipated power by the main components for any execution cycle interval.
2. *Temporal View* shows the details of the instructions with issue and completion cycles for each thread block at warp level. In addition to power consumption statistics for the sub-components in an SM, we include L1 Data (L1D) cache access statistics, which are local for each SM, to relate the thread block's performance in the same execution interval.
3. *Spatial View* demonstrates the access information for each on-chip L1D cache, L2 cache in each sub-partition, and row buffers of DRAM banks in each memory partition. Additionally, it shows the power consumption distribution among the memory components in the execution interval.

– We present case studies to demonstrate the potential usages of our framework and its visualizations by performing experiments for memory-bound graph workloads and resource-critical embedded applications. Our tool enables us to perform detailed performance and power analysis for the target GPU executions.

The remainder of this paper is organized as follows: Section 2 presents some background on GPU architecture, CUDA programming model, and the simulator. We explain our design and implementation details for tool construction in Section 3. Then, we present case studies in Section 4, demonstrating usage scenarios of the GPPRMon. Section 5 presents the existent performance and power evaluation studies for GPGPU applications. Finally, Section 6 summarizes the work with some conclusive remarks.

2. Background

2.1. GPU Hardware

Modern GPU architectures employ a single instruction multiple thread (SIMT) execution in the Streaming Multiprocessor (SM) units. Each SM includes multiple warp schedulers, instruction dispatchers, a register file, multiple single and double precision ALUs, tensor cores, special function units (SFU), and load/store units with on-chip (on-SM) local memory. An interconnection network connects SMs to off-chip memory partitions on which DRAM and Last-Level caches (LLC) are placed. While the cores inside the same SM can access the private L1 cache, all the cores can communicate via the L2 cache structure. Load/store instructions may require off-chip accesses whenever requested data cannot be found in the L1D cache. Furthermore, data access gets slower for memory instructions as moving down the hierarchy. GPU architectures have been evolving, with each new generation introducing enhancements in performance, power efficiency, and specialized features. This overview provides a general understanding of GPU architecture, but specific details and terminology may vary based on the GPU device model and manufacturer. We specify the architectural and resource specifications for the GPUs in our experimental setup parts.

2.2. CUDA Programming Model

Compute Unified Device Architecture (CUDA) [26] is an API to execute a function, namely kernel function, on GPUs. A GPU kernel consists of a 3D grid space, where each

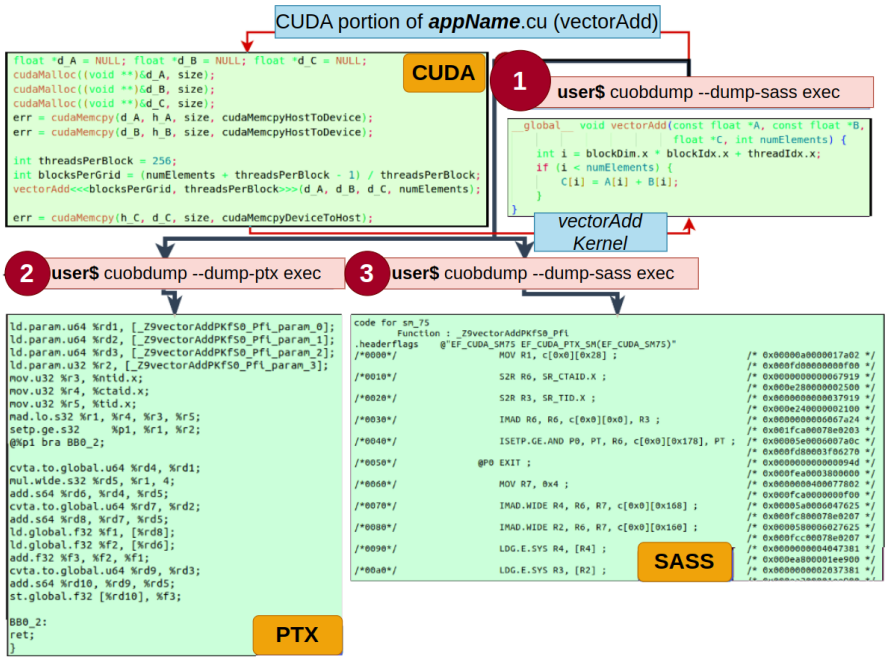


Fig. 1: CUDA, PTX, and SASS code snippets for *vectorAdd* code.

grid has a 3D thread block with multiple threads. Each sequential 32-thread set forms a *warp* within a thread block in CUDA. When a kernel function launches, the Gigathread engine (thread block scheduler) schedules thread blocks to SMs in the Round-Robin fashion. Register resources on SMs determine the number of active thread blocks, some of which may be issued to the same SMs. Figure 1 demonstrates a GPU kernel code, *vectorAdd*, for the vector addition operation. Part 1 presents a compilation command by nvcc [29], which is the CUDA compiler to generate the executable. Part 2 demonstrates the Parallel Thread Execution (PTX) [30] instructions, which represent a virtual machine ISA generated by *nvcc*, and Part 3 includes the SASS machine instructions, which represent low-level machine assembly that compiles to binary code executing on NVIDIA GPU hardware.

2.3. GPGPU-Sim Simulation Framework

A cycle-level microarchitectural simulator GPGPU-Sim [15] (hereafter referred to as the simulator) has been heavily utilized by researchers working on GPU software and hardware optimizations. Figure 2 displays the workflow of the simulator, which configures a traditional NVIDIA GPU and simulates CUDA-based applications. The simulator provides functional and performance-driven modes such that functional mode enables developers to check the kernel's functional correctness, while the performance mode simulates the kernel for the configured GPU in a cycle-accurate manner. The simulator officially supports Volta-based Titan V, V100, RTX2060 GPU series, Pascal-based Titan X, Kepler-based TITAN, and Fermi-based GTX480. Additionally, AccelWattch developers introduced Volta-based RTX2060 S and Ampere-based RTX3070 GPUs to the official

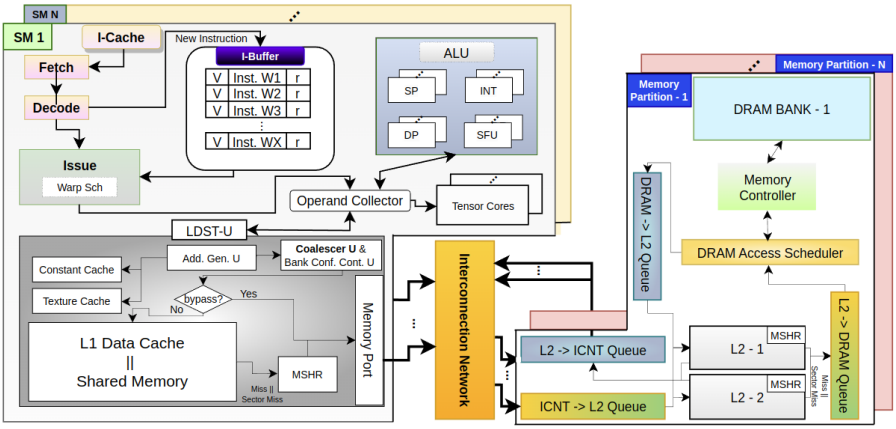


Fig. 2: A general workflow simulation model for a modern GPU.

simulator configurations. Beyond these, one can reconfigure any GPU with different resources on top of these architecture models. Additionally, the simulator reports achieving 15% performance accuracy error rate with its virtual ISA implementation [15].

The AccelWattch [14], a power model extension of the simulator, is an analytical model formulating power dissipation and utilizing validated power coefficients of microarchitectural components such as functional units, CUDA core lanes, and memory components, which are gathered through a comprehensive set of experiments. AccelWattch, which supports Dynamic Voltage-Frequency Scaling (DVFS), estimates the energy consumption for V100 with 90% accuracy. In addition to V100, AccelWattch is validated for TITAN X and Turing RTX GPU series through a large set of applications from Rodinia, Parboil, CUTLASS, and DeepBench benchmark suites and NVIDIA CUDA Samples, enables tracking detailed power dissipation of any GPU kernel execution.

Since we build our GPPRMon tool completely on top of GPGPU-Sim and its AccelWattch power model, its accuracy, architectural support and scalability limitations can be considered parallel to the support of the simulation framework.

3. Methodology

GPPRMon tool enables monitoring and visualizing runtime GPU execution performance and power consumption. Figure 3 displays GPPRMon workflow consisting of two main parts: 1 Metric Collection, 2 Visualization. For any execution interval, GPPRMon systematically calculates IPC rates of SMs, records warp instruction's issues and completions, collects memory access statistics across the memory hierarchy, and tracks dissipated power on sub-hardware units. Parts 1-a and 1-b demonstrate examples of the power and performance metrics, respectively, and Section 3.1 details what these metrics are and configuration options for users. Furthermore, Part 2 reveals GPPRMon's visualizer that contains three views to show general performance, memory access statistics, instruction monitoring, and their power dissipation by processing the collected metrics. We build our framework on top of the simulator, which is compatible with many GPU

models mentioned in Section 2.3. The GPPRMon framework is available as open-source software [4].

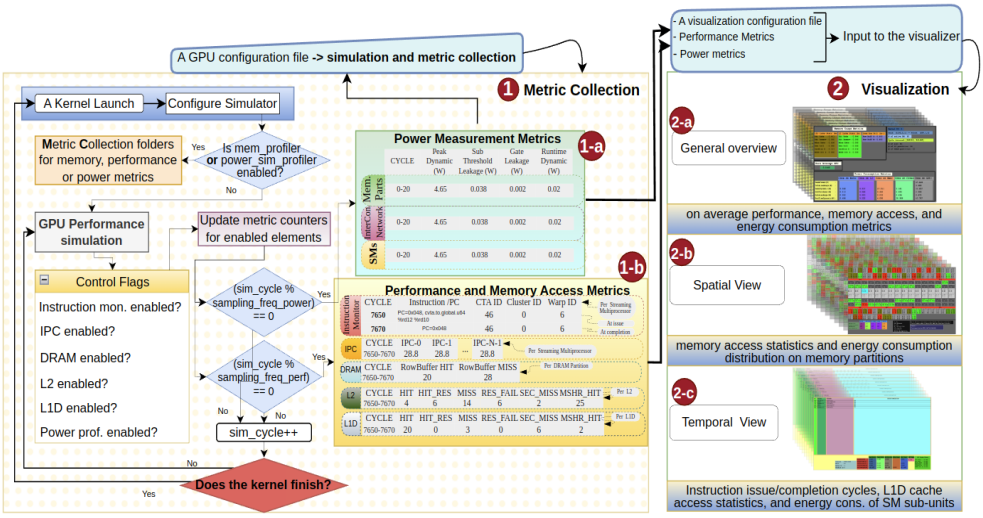


Fig. 3: A general workflow overview of **GPPRMon** framework.

3.1. Metric Collection

The *Metric Collection* phase of GPPRMon, shown in Part 1 in Figure 3, conducts the systematical recording of performance and power metrics during the execution. Performance metrics mainly consist of hardware utilization and execution statistics of the memory hierarchy and SMs, such as cache usage efficiency and SMs IPC values. The metric collector extension to the simulator cumulatively tracks the hardware performance, execution statistics, and power counters during a sampling interval, which is determined by *sampling_freq_perf* in Figure 3. At the end of each observation interval, the metric collector exports the tracked metrics to their respective files and clears counters. To enable the performance metric collection feature, the user needs to specify the *mem_profiler* flag in the simulation configuration file. Similarly, power metrics, such as dynamic runtime and peak power, can be collected by setting the *power_sim_profiler* flag.

GPPRMon's metric collection is multi-functional, such that users can separately track warp instruction issue/completion for each thread block, runtime IPC values of each SM, and runtime memory hierarchy utilization statistics (i.e., L1, L2, and DRAM row buffers). GPPRMon allows users to either accumulate or independently collect metrics for each observation interval. Furthermore, users can discard store operations from the memory hierarchy utilization metrics for their runtime target observation. Users can configure these features through metric collection specifications, as depicted *Control Flags* in Figure 3. GPPRMon creates separate folders for each component's statistics and distinct files for

[4] https://github.com/parsiyte/GPPRMon

each sub-component with IDs. To reduce storage and access overheads during the kernel's execution, we utilize CSV file format for recording metrics. Our visualizer, shown in Part 2, can execute parallel to Part 1 and processes the collected metrics to generate runtime visualizations. The following subsections will briefly describe the collected microarchitectural performance and power metrics and how to configure each metric collection separately. More detailed information about how to build GPPRMon and deploy multi-functional metric collection is available on the tool's GitHub page.

Performance Metrics

L1 Data and L2 Caches. A memory request's access status on caches can be one of the possible states: i) Hit: Data resides on the cache line; ii) Hit Reserved: The data is requested, and the corresponding cache line is allocated, but the data is still invalid; iii) Miss: The corresponding cache line causes several sector misses, resulting in a line eviction; iv) Reservation Failure: The situations in which a line allocation, MSHR entry allocation, or merging an entry with an existing in MSHR fail, or the miss queue does not have any slot to hold new requests result in reservation failure; v) Sector Miss: A memory request cannot find the data in the sector of a cache line (i.e., a sector is 32B, whereas a cache line is 128B); vi) MSHR hit: When the upcoming request's sector miss has already been recorded, and request can merge with the existing entry, MSHR hit occurs.

GPPRMon can observe runtime access statistics of each L1D and L2 cache separately. Users can activate the metric collection feature of GPPRMon for L1D and L2 caches with regarding *control flags* as depicted in Figure 3. Tracking runtime cache utilization helps researchers evaluate the application's memory access behavior and relate it to the overall application performance. While cache hits indicate small access latencies, misses generally refer to longer latencies and increasing active memory traffic in the hierarchy. Furthermore, reservation failures result in a memory pipeline stall, directly delaying the subsequent load/store operations. Handling intensive misses requires detailed analytic observations to deduct behavioral interpretations across applications and cache utilization. In this manner, GPPRMon can meet the runtime analytic observation necessity for micro-architectures on GPU to identify the memory performance and power issues understandably.

Row Buffers of DRAM Banks. A row buffer hit occurs when an L2 miss request finds the requested data in the row buffer of the target DRAM bank. A row buffer miss results in a longer access service time than those hits because handling a row buffer miss requires scheduling a memory request to the correct address on DRAM and activation latencies. Row buffer utilization with cache access statistics completes the runtime memory hierarchy behavior exploitation, crucial for describing overall memory performance, especially for sparse data applications. GPPRMon provides separate metric collections for the row buffers in each DRAM partition, and users can activate this feature by enabling *DRAM control flag* as depicted in Figure 3.

SM IPC. An IPC rate mainly describes an SM's performance, which consists of various functional units with varying lane depths. For example, V100 SMs include four single-precision (SP) ALUs with four pipe depths, as presented in Figure 2. When a thread

block occupies only SP-ALU lanes, assuming an operation takes one cycle, the ideal IPC for each SM should be sixteen without other functional units' contribution. However, IPC oscillates during the execution depending on SM and memory hierarchy utilization. GPPRMon tracks the runtime IPC rate for each SM separately for each sampling interval, and one can active per SM IPC tracking by enabling *IPC control flag* as in Figure 3. With GPPRMon, developers can analyze runtime IPC variations and investigate the root causes of IPC suffering.

Instruction Monitor. The instruction monitor feature of GPPRMon records the issue/completion cycles of warp instructions within each thread block together with their opcode, operand, and PC separately. Since GPUs execute instructions for multiple threads concurrently by a common PC within a warp, we design tracking instruction issues/completions at the warp level. While the first row of the *Instruction Monitor* in Part 1-b shows the issue statistics, the second displays the completion. Even if the dispatcher unit may issue the same instruction multiple times for a warp, it is guaranteed that any two instructions, of which *CTA_ID*, *SM_ID*, *local Warp_ID*, and *PC* are the same, cannot change the issue/completion sequence. Hence, we can obtain the correct issue/completion matching for each instruction. Users can activate the instruction monitoring utility for each thread block with *instruction mon. control flag* as in Figure 3.

Power Metrics

The comprehensive results of the dissipated power on GPU yield the analytical observation that gains significance, especially on embedded systems. Therefore, we develop GPPRMon to systematically collect the power distribution on the sub-units of SMs, memory partitions, and the interconnection network, in addition to performance metrics. Moreover, SMs are GPUs' most impactful hardware units in terms of runtime power dissipation with their dense compute units. Thus, GPPRMon further classifies SM's power distribution through execution units, including the register file and the beginning of the instruction pipeline, functional units, and load/store units involving the L1D cache.

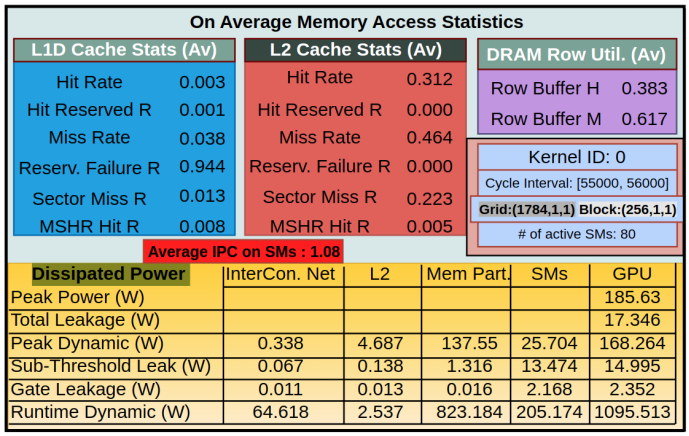
We implement the collection of power metrics utility on top of the AccelWattch [14], built upon McPAT [21], and the accuracy and theoretical limitations of AccelWattch yet reside. However, GPPRMon's power metric collection feature is still a reliable and practical tool since we extend various configurability options of AccelWattch to GPPRMon, such as DVFS. GPPRMon assures the following runtime power measurements for each component apart from idle SM: *Peak Dynamic*(W), the maximum momentary power within the interval, *Sub-threshold Leakage (W)* and *Gate Leakage(W)*, the leaked power (due to current leakage) from the junctions of MOSFETs, and *Runtime Dynamic (W)*, the total consumed power. Moreover, GPPRMon supports collecting power metrics either cumulatively or distinctly for each sampling interval, starting from a kernel's execution. For instance, power dissipation results in Figure 4 show the *cumulatively* collected results for the interval between [55000, 56000], which means aggregating power results for each sampling interval. The cumulative power metric collection option eases determining average power dissipation for different execution intervals. While V100's TDP is 300W [35], aggregated *Runtime Dynamic* power for 1000 cycles is 1095.5W, which corresponds to an average of 109.5W per sampling interval with the cycling frequency of 100 on the sim-

ulator. Users can configure the power metric collection by first enabling power profiling and determining other configurations as detailed in the tool's source code.

3.2. Visualization

By processing the collected metrics (Part 1 in Figure 3), GPPRMon depicts performance and power dissipation with three perspectives at runtime, as represented in Part 2 in Figure 3, and enables pointing out detailed interaction of the program with the underlying hardware.

i **General View**, Part 2-a, presents the overall IPC of GPU, the average memory access statistics, and dissipated power among the major components with application- and architecture-specific information;
ii **Spatial View**, Part 2-b, presents the detailed access statistics of the GPU memory hierarchy and dissipated overall power among the memory partitions by enabling the monitoring of the entire GPU memory space;
iii **Temporal View**, Part 2-c, demonstrates instruction execution statistics with activation intervals at warp-level for user-specified thread blocks, L1D cache access characteristics, and power distribution among the sub-components of SMs by activating the execution monitoring feature.



Dissipated Power	InterCon. Net	L2	Mem Part.	SMs	GPU
Peak Power (W)					185.63
Total Leakage (W)					17.346
Peak Dynamic (W)	0.338	4.687	137.55	25.704	168.264
Sub-Threshold Leak (W)	0.067	0.138	1.316	13.474	14.995
Gate Leakage (W)	0.011	0.013	0.016	2.168	2.352
Runtime Dynamic (W)	64.618	2.537	823.184	205.174	1095.513

Fig. 4: *General View* with average performance and power consumption metrics collected at runtime.

GPPRMon includes three configuration options for different visualization perspectives and an interval sampling cycle to divide runtime execution into portions. Since GPPRMon's *Temporal View* may require scanning of many statistics for all thread blocks, especially in large architectures, GPPRMon provides a *Temporal View* option among the thread blocks determined with IDs. Depending on the configuration, GPPRMon starts tracking collected metrics for each kernel and systematically saves images in PNG format per execution interval.

Figure 4, an example of our *General View*, presents the overall measurement of the first kernel for the *SPMV* from *Gardenia* benchmark [48] executed on V100 GPU [23]. It displays average memory access statistics among the active L1D caches, L2 caches, and DRAM banks; average IPC value among the active SMs; dissipated power on major sub-GPU components within the [55000, 60000] cycle interval. The view includes grid (i.e., 1764 thread blocks) and block dimensions (i.e., 256 threads per block) with the number of actively used SMs so that the users can relate active SMs and workloads at runtime. To illustrate, Figure 4 shows that the kernel executes with the IPC rate of 1.08 and utilizes the memory hierarchy inefficiently due to high miss and reservation failure rates on V100 in this interval, where the Volta SM architecture supports concurrent execution of 2048 threads per SM. Considering that V100 SMs contain 16 SP/INT/DP ALUs, SFUs, and Tensor Cores, we can notice the low performance since the ideal IPC should be much more than 1.08 with 640 thread blocks (8 thread blocks per SM) in the given interval. Long-latency memory operations may slow instruction completion and result in low IPC. Moreover, memory partitions consume 75% of total power dissipation, which validates that SMs mostly stay idle for the execution interval given in Figure 4.

Our *General View* supports two additional visuals that show the time spanning of memory access statistics and the relationship between IPC and power metrics for longer runtime intervals as in Figure 11 and Figure 14 (the examples given as part of our case studies), respectively.

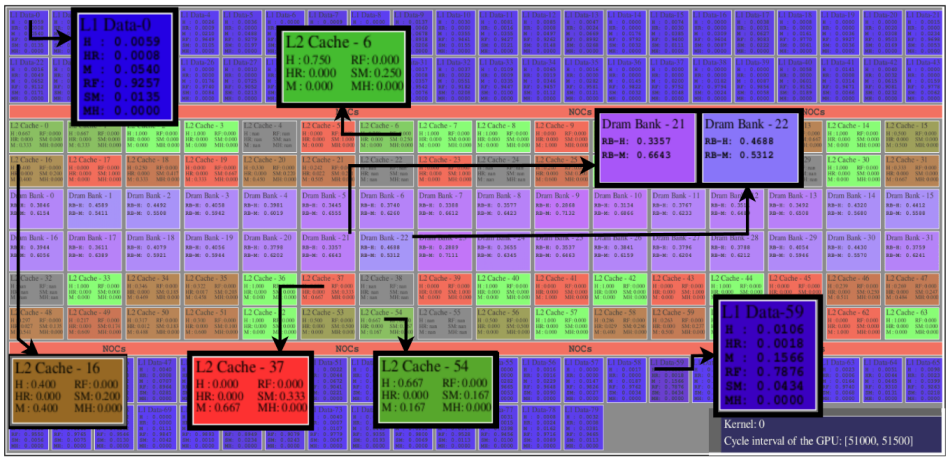


Fig. 5: *Spatial View* displaying memory access statistic at runtime.

Figure 5, an example of our *Spatial View*, shows the memory access statistics across the GPU memory hierarchy on L1D caches, L2 caches, and DRAM. On caches, the green emphasizes hit and hit reserved accesses, the red indicates miss and sector miss, and the blue states the reservation failures through miss queues or MSHR. Similarly, DRAM bank pixels are colored with a mixture of red and blue to specify the row buffer misses and hits, respectively. GPPRMon supports memory hierarchy visualization for all official GPU configurations of the simulator, even if some disable the L1D cache for load/store operations. *Spatial View* includes detailed data about access statistics, resource quanti-

ties, architecture types, and dissipated power on memory partitions. To detail the view, we zoom in on some memory units in Figure 5, which presents statistics in the cycles of [51000, 51500] for the kernel execution. Different L1D caches behave similarly in that interval, such that almost all L1D caches turn blue due to reservation failure concentration. To illustrate, the reservation failure rates are 0.92 and 0.78 on *L1D-0* and *L1D-1*, respectively. On the contrary, statistics on L2 caches imply heterogeneous data accesses. While *L2 Cache-6* and *L2 Cache-54* bring hit rates of 0.75 and 0.67, respectively, *L2 Cache-37* causes 0.67 miss and 0.33 sector miss rates, and L2 cache is intermediate in utilizing data locality among the others. Gray color among the units regards no access occurring on the *Spatial View*.

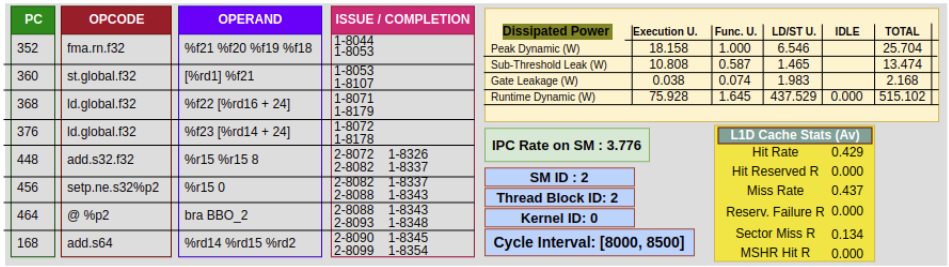


Fig. 6: *Temporal View* monitoring instruction executions together with the performance and power consumption of the corresponding SM.

Figure 6, an example of our *Temporal View*, displays a thread block's execution statistics at warp-level, L1D cache statistics, and dissipated power of core components configurable execution intervals. It presents each warp's PTX instruction sequence, with opcodes, operands (source/destination registers and immediate values if they exist), and the program counter (PC). The *Issue/Completion* column indicates the execution start and writeback times of warp instruction segments within any thread block. For instance, Figure 6 reveals the execution monitoring of the *Thread Block 2* on *SM 2* in the cycle range of [8000, 8500]. The instruction dispatcher unit issues two SP global loads with PC=368 and PC=376 at *Cycle 8071* and *Cycle 8072*, and they are completed at *Cycle 8179* and *Cycle 8178*, respectively. *Temporal View* allows tracking execution duration per instruction in this manner. Since L1D cache hits result in low latency, these load instructions lasting above 100 cycles are among the misses or sector misses of L1D cache statistics. In addition, the view enables us to relate IPC, instruction statistics, and power metrics. The fact that the rate of memory instructions is 0.37 and inefficient use of the L1D cache within the given interval significantly degrades the IPC on SM2. Furthermore, the load/store unit dissipates nearly 92% of SMs total power in the corresponding interval because of the pressure on the L1D cache. As a result, one can analyze the data locality in a multi-perspective by utilizing access statistics of caches and row buffers on *Spatial View*, tracing the issue/completion times of memory operations on Temporal View at runtime.

GPPRMon Overheads. GPPRMon execution performance mainly depends on ***i.*** *the metric collection sampling frequency*, ***ii.*** *visualizer intervals*, and ***iii.*** view types. Firstly,

each sampling operation during simulation conducts multiple I/O operations to the metric collection files, such as performance, memory, and power trace files, which are significantly slow compared to the simulation execution. Hence, widening the simulation runtime sampling interval directly reduces the number of I/O operations for exporting results to output files, and the simulation duration decreases accordingly. For instance, simulating the *SPMV* benchmark [48] takes 98 minutes for the *Higgs Twitter Mention* data by recording both power and performance metrics per 5000 simulation cycles, while the baseline simulation (i.e., not collecting runtime performance and power metrics) completes the execution at 88 minutes in our local infrastructure. Furthermore, the visualization interval mainly determines the spanning range of the collected metrics. That is, lower visualization intervals search for fewer sampled metrics, reducing generating views to demonstrate runtime performance and power observations on the GPU. Since each view type requires different metrics, composing each figure takes various amounts of time. For example, generating *General View* in Figure 4 necessitates spanning all statistics on L1D and L2 caches, row buffers, SMs, and power dissipation for a given internal while *Spatial View* only displays memory access statistics, and is much easier to compose. Additionally, tracking the instruction monitoring of limited thread blocks through *Temporal View* is comparably easy, whereas increasing the number of thread blocks for instruction monitoring raises the spanning overhead and takes longer to generate those views.

4. Usage Scenarios

We evaluate a set of CUDA programs from two benchmark suites and demonstrate the case studies that can use our framework. Specifically, we utilize graph workloads from the Gardenia benchmark suite [48] and embedded applications from the GPU4S embedded benchmark suite [37]. Since graph workloads target large-scale systems, we configure the GPPRMon to simulate Volta architecture-based V100, commonly used in HPC systems and GPU architecture research. Table 1 presents salient characteristics of the V100 device. On the other hand, for embedded applications, we configure our tool to simulate the GPU device on Jetson AGX Xavier, which provides a System-on-Module with a Volta-based GPU and contains an 8GB/16GB unified memory with a high-bandwidth interface to the GPU, and a 512KB shared L2 cache exists in the memory hierarchy [25]. Its GPU includes 512 cores corresponding to 8 SMs involving a 128KB on-chip cache per SM.

4.1. Determining Memory-Bound Kernels and Spatial Characteristics

We evaluate the *Betweenness Centrality (BC)* program (from Gardenia suite [48]) with multiple kernel executions. By analyzing the kernels with the largest cycles, we classify them as kernels with no memory pressure and kernels with memory bottleneck. Additionally, we review SM distribution and utilization of the target executions.

Figure 7 presents memory-related data and IPC values during the execution time intervals for the kernel function, *Kernel 1*, and demonstrates L1, L2, and row buffer statistics and the corresponding IPC/power behavior. While our spatial and temporal views present fine-grained values for each hardware component across the GPU device, those statistics (Figure 7 and Figure 9 afterward) demonstrate the dynamic average values for overall GPU SMs. While there are a few oscillations in the rates, the general behavior

Table 1: V100 GPU configuration specifications based on Volta architecture.

SM (80) Specifications	Registers, Register Banks	65536, 16
	SP-U, SF-U, DP-U, INT-U, TC-U, LD/ST-U (WB-PipeDepth)	4,4,4,4,4,1(8)
	Warp Scheduler	4
	on-chip L1I Cache, NoF banks, access latency, cache line	128KB (64 sets, 16-way), 1, 20 cycles, 128B
Memory Parititon (32) Specifications	L2 Cache, NoF banks, access latency, cache line	96KB (32 sets and 24-way), 2, 160 cycles, 128B
	DRAM, NoF banks, access latency (after L2)	1GB, 16 banks, 100 cycles, 128B
	DRAM scheduler	First-ready, first-come first-service

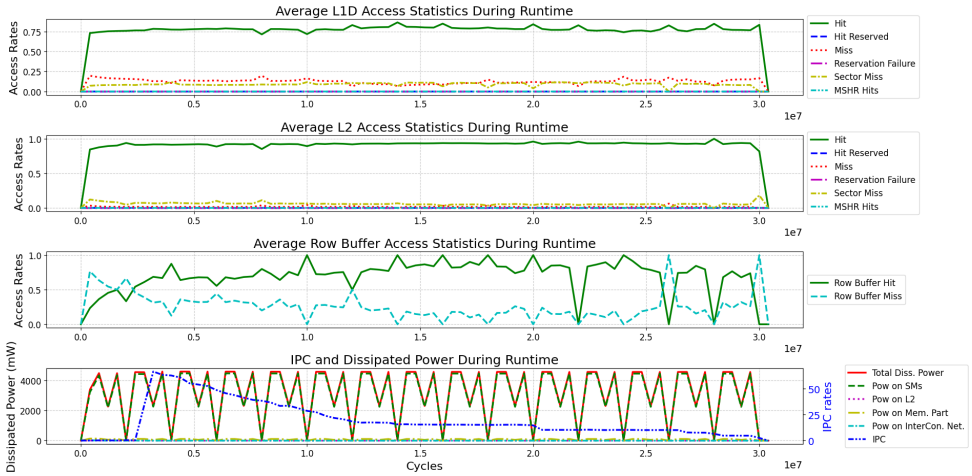


Fig. 7: Average memory and power consumption statistics for *BC-Kernel 1*.

demonstrates steadily high hit rates. Moreover, we can see the peak IPC value during the intensive computations at the beginning of the kernel execution. Then, IPC values tend to decrease as the computations end. We can observe that the kernel has no memory pressure during the execution. After warming the caches at the very beginning of the execution, the kernel could perform its operations with high hit rates and compatible IPC rates.

After getting the memory behavior of our target kernel based on our average timing statistics, we can utilize our *Spatial View* to understand its SM utilization. Figure 8 presents the partial view that includes the L1 cache structures of each SM at time intervals [5000-10000], [15000-20000], and [245000-250000], respectively. The application launches the kernel function with 64 thread blocks and 256 threads per block, such that the thread blocks are scheduled to 64 SMs (out of 80 in the V100 device). Therefore, only the corresponding L1 caches exhibit non-zero values. Since all thread blocks are active and the caches do not hold data at the beginning (5K-10K time interval), all 64 L1 cache structures employ low hit rates (red color in our representation, while inactive L1s are gray). At the next time interval, the execution starts warming the caches, and hit rates

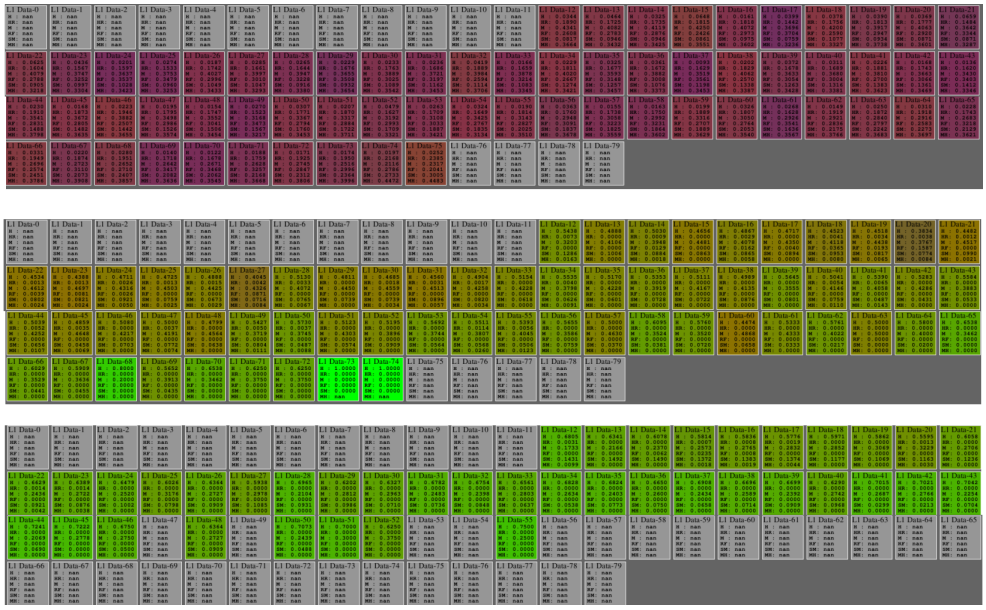


Fig. 8: L1 cache status for *BC-Kernel 1* for successive time intervals.

increase (green color in our representation). Eventually, as the thread blocks complete execution and they retire, L1 caches on the corresponding SMs do not collect statistics (gray as the other L1 caches at the beginning). We can track all thread blocks' scheduling and their L1 cache utilization during the execution for the given time intervals. For the specific kernel execution, observing the high hit rates (green colors) in all L1 caches of the active SMs and corresponding IPC rates demonstrates efficient L1 utilization and low memory pressure.

Figure 9 demonstrates memory-related data and IPC values for another kernel, *Kernel 2*. Since *Kernel 2* has been launched by much more threads than available SMs, all SMs are active during the execution, and the threads compete for both L1 and L2 caches. The reservation failure and miss rates at the first part of the execution are significant for private L1 and shared L2 cache, in turn, the kernel exhibits low IPC values. Whenever L2 and especially L1 hit rates become more than 0.5 (around cycle 600K), IPC values also increase, and the kernel function performs most of the target computations (until cycle 800K). We can observe the corresponding cache status in our *Spatial View*. For this case, we include both L1 and L2 cache structures in our aggregated visualization in Figure 10. Different from *Kernel 1*, we can see that all L1 caches exhibit non-zero values from the beginning of the execution since the available threads utilize all SMs in the device. The figure demonstrates the warming of the caches through the execution cycles, where the kernel function has pressure on both L1 and L2 by highly utilizing those structures.

4.2. Performance Bottleneck Analysis and its Power Impacts for a Memory-Intensive Workload

We evaluate CUDA implementation of the *Page Ranking (PR)* algorithm given in Gardenia suite [48] to analyze a memory-bound GPU program and irregular memory access

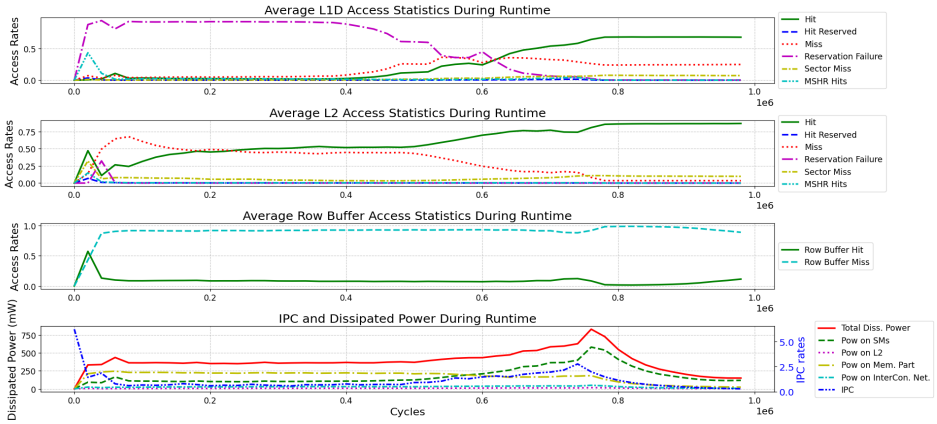


Fig. 9: Average memory and power consumption statistics for *BC-Kernel 2*.

statistics through GPPRMon. *PR* assigns weights to graph nodes describing relative importance among nodes.

The *PR* execution iterates with the *Contribution Step (K0)*, *Pull Step (K1)*, and *Linear Normalization (K2)* kernels, and the total number of iterations varies depending on the data size. Table 2 presents the performance overview of the kernel statistics. At the beginning of the execution, *K0* causes a high miss rate on caches since all memory operations are completed before warming up. No row buffer locality information is provided across DRAM accesses, as the required data mostly fits into the L2 cache. In other words, L2 misses do not access the same row of DRAM banks within any memory partition. Since total elapsed cycles indicate that *K1* dominates the execution at 99.7% and directly affects the application performance and overall power consumption, we focus on that kernel execution.

Table 2: *Page Ranking (PR)* kernel performance statistics.

Kernel	**GPU IPC**	**GPU Occupancy**	**L1D**		**L2**		**DRAM**		**Total Cycle**
			Miss Rate	Res. Fail Rate	Miss Rate	Res. Fail Rate	Row B. Loc. (L+S)	Row B. Loc. (Load)	
Page Ranking - Contrib K0	715.59	82.76%	1.000	0.819	0.333	0.0	NA	NA	8670
Page Ranking - PullStep K1	3.007	5.55%	0.584	0.400	0.156	0.011	0.658	0.667	8677889
Page Ranking - LinNorm K2	1297.68	77.108%	0.501	0.285	0.457	0.001	0.724	0.732	11718

V100 includes 870GB/s bandwidth migrating 217.5 Giga SP float to the SMs and 14.8-SP/7.4-DP TFLOPS peak computational power [23]. With these specifications, we can state that the time consumed for one load complies with the execution of 68 SP float operations on SMs, ideally (i.e., without L1D and L2 caches). On the other hand, *K1* has a memory instruction intensity of around 0.2 in its PTX code, which validates that

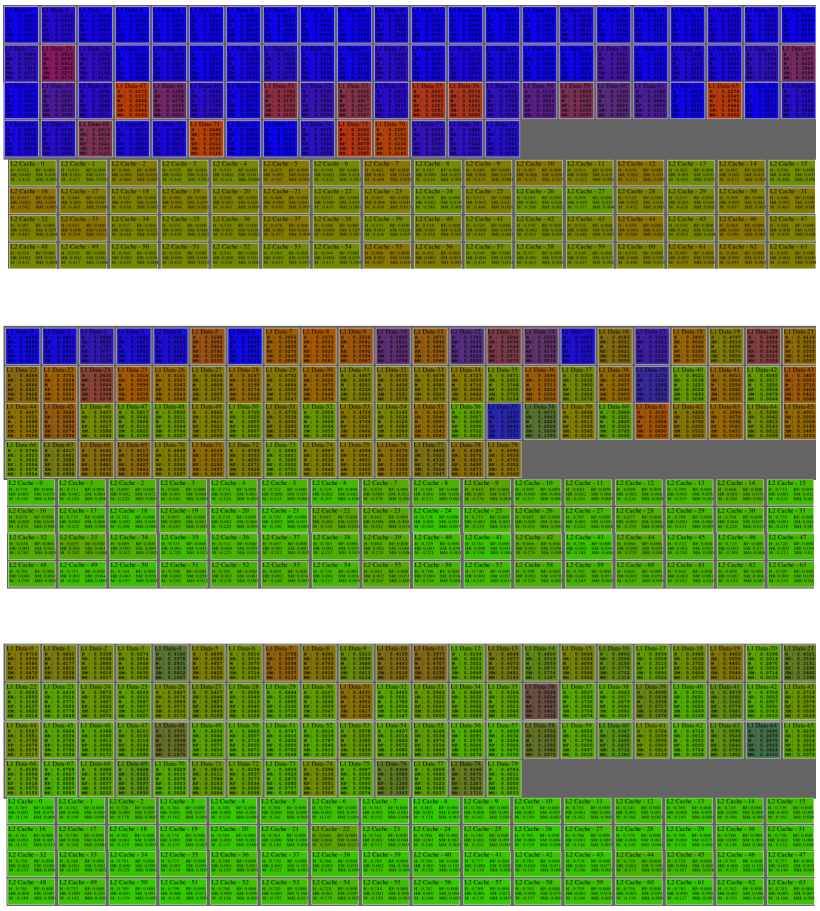


Fig. 10: L1 and L2 cache status for *BC-Kernel 2* from cycles 400K, 600K, 700K.

K1 bounds the performance due to intrinsic memory workload. Not only is *K1* memory-bound, but also inefficient memory usage severely impacts performance. To see performance-critical points in the execution, we monitor runtime performance-degrading factors with low-frequency execution snapshots provided as part of our tool.

While the overall IPC is around 0.3 for *K1*, *General View* results obtained for every 500 cycles during the execution indicate that IPC oscillates in the range of [0.1, 9.54], with the highest peak of 53.44. Figure 11, which is part of our *General Overview*, shows average access statistics on memory units in [5000, 100000] cycles. After caches warming up (around cycles 10000), while the average miss rate on L1D caches oscillates in [0.14, 0.51], sector misses, which the simulator does not provide separately, vary in [0.05, 0.31] with the metrics collected in every 20 cycles. We can understand the data pollution on the L1D caches, which prevents the execution from exploiting cache locality. For instance, *K1* does not utilize spatial locality on the L1D cache since the MSHR hits oscillate

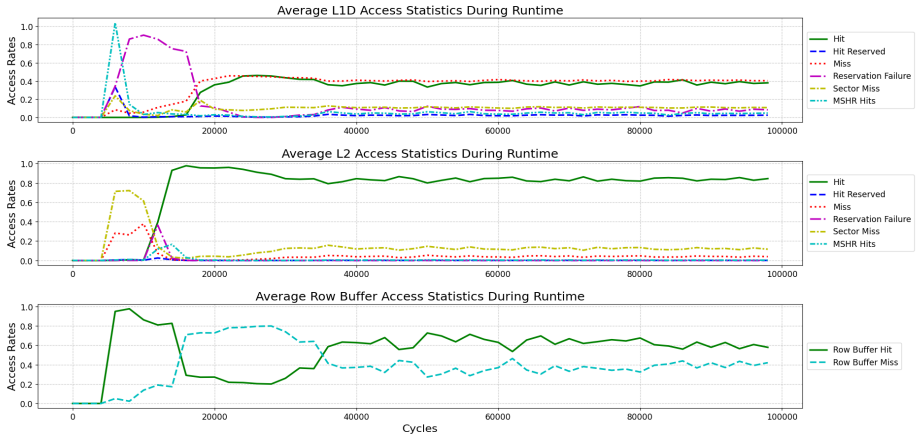


Fig. 11: Average memory access statistics in the cycle range of [5000, 100000] for *PR-K1*.

slightly in [0.03, 0.08] during the execution. Hence, the overall hit rate on L2 caches is quite high when we use the web-Stanford dataset [19], which occupies memory space five times larger than the L2 cache size. While the performance metrics in Table 2 hide the L2 statistics as it counts misses before warming up at kernel launch, the actual L2 hit rate oscillates in [0.82, 0.95] at runtime with sampling per 500 cycles. Additionally, the row buffer hits and misses vary in [0.2, 0.85] in an unstable manner, which verifies data sparsity throughout the execution.

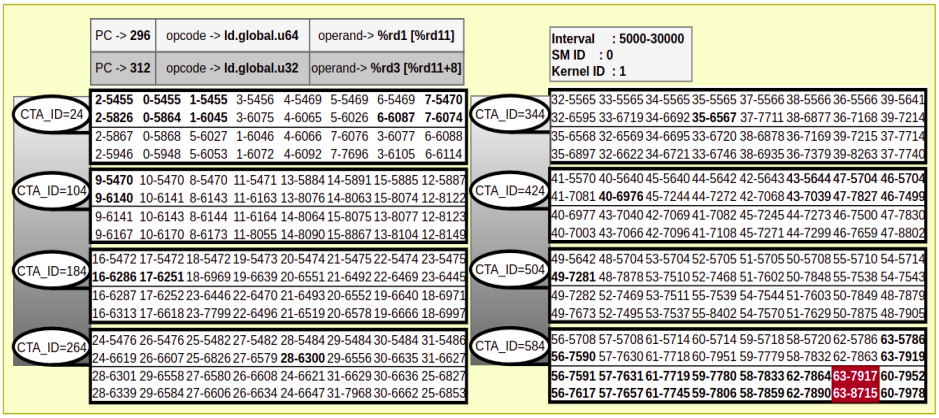


Fig. 12: Instruction monitoring for the load instructions on *SM 0* in the cycle range of [5000, 30000] for *PR-K1*.

Figure 12 presents the instruction issue/completion cycles of 8 *K1* thread blocks running on *SM 0*. We merge multiple snapshots of our *Temporal View* that belong to thread blocks in *SM 0* to evaluate the performance of all load instructions together. The first and second lines point to the load instructions of which the program counter (PC) equals 296

(loads DP) and 312 (loads SP), respectively. Figure 13, a snapshot of our *Spatial View*, demonstrates the memory access statistics of representative components within the same interval. After the kernel launch, each thread collects thread-specific information from

Fig. 13: Memory performance overview in the cycle range of [5000, 9500] for *PR-K1*.

parameter memory, which takes 250-450 cycles to process target data addressed with the thread's private registers. The warp schedulers dispatch the load instructions (i.e., at PC=296 *ld.global.u64*), and all eight warps at *Thread Block 24* start executing the instruction after *Cycle 5455*. Furthermore, Figure 12 reveals that SM dispatches load instructions from the remaining thread blocks in the interval of [5470, 5786] after issuing the load instructions of *Thread Block 24*. Figure 13 reveals that no access occurs on some of the L1D caches, while none of the L2 caches and DRAM banks are accessed during the preparation time in [5000, 5500] in Part 1. No data brought to the L1D cache of *SM 0* by the warps of *Thread Block 24* after *Cycle 6087* (*Warp 6*) enables the early completion of the instruction pointed with PC=296, belonging to the thread blocks 104, 184, 264, 344, 424, 504, and 584. We highlight this observation with the bold fonts representing the earliest completion times within each thread block in the second row of the first instructions. Additionally, a high reservation failure and no MSHR hit rates on the L1D cache of *SM 0* in Part 2 confirms that the locality utilization between thread blocks is quite low for the first load. If the L1D cache locality was utilized, we should have observed larger MSHR hit rates and completion of the same instructions just after *Cycle 6087*. Parts 2,3,4,5 reveal that the

reservation failed requests pointed by PC=296 cause a miss on L2 caches without MSHR merging. Thus, memory requests of the same instruction from different SMs cannot benefit from L2 locality and cause more traffic in the memory hierarchy. Additionally, Parts 3,4,5,6 reveal that the L1 access status mostly turns to the hit after *Cycle 6000*. Unlike the load instructions at PC=296, ones at PC=312 (*ld.global.u32*) usually hit on L1. The second line for each thread block shown in Figure 12 indicates that the completion takes much fewer cycles for the loads at PC=312. To illustrate, while *Thread Block 504* completes the first load instructions within 2133 cycles, except *Warp 63*, it takes 26 cycles for the second instructions, whose requests result in a miss on both the L1D and L2 caches. While the loads at PC=296 complete the execution in the range of [350, 2250] cycles, the loads at PC=312 take less than 50 cycles for most of the warps due to the increasing hits on the L1D cache. However, the loads at PC=296 delay the issue of the second load instructions due to long latency. The remaining thread blocks (from *Thread Block 641*) are assigned to SMs after *Cycle 55000*. With the observation that a thread block occupies 50000 cycles on an SM (for the web-Stanford graph, which can easily fit into DRAM), the schedule of any waiting thread block delays around 2000 cycles. Such delays affect the performance of a thread block by 4% in addition to affecting the memory traffic negatively and degrading the overall performance of *K1*. In this manner, an approach such as adaptive thread block scheduling by throttling the load/store unit issue amount depending on the access statistics of caches can reduce the side effects and increase the overall performance.

Table 3: Dissipated average power in milliWatt in the cycle range of [5000,10000] for *PR-K1*.

Cycles	Streaming Multiprocessors					Memory Partitions						NoC	TOTAL
	Exec. Units	Func. Units	LD/ST Unit	Idle	Total	MC FEE	PHY	MC TE	Dram	L2	Total		
5k, 5.5k	2637.5	54.3	35.6	23.7	2751.3	3.7	8.2	4.6	0	0	16.5	0.7	2768.5
5.5k, 6k	597	6.3	860	0	1463.7	177.2	17.8	9.4	557.2	3.4	764.9	26.9	2501.6
6k, 6.5k	614.4	12.4	399.9	0	1026.7	56.1	31.3	16.1	1346.4	3.0	1452.9	92.6	2577.3
6.5k, 7k	708.6	14.4	464.3	0	1187.3	65.5	31.4	16.2	1354.3	3.1	1470.5	94.2	2755.1
7k, 7.5k	686.8	13.9	463.9	0	1164.6	65.6	31.8	16.2	1354.9	3.1	1471.2	94.4	2733.2
7.5k, 8k	795.8	16.3	487.4	0	1299.4	69.9	29.7	15.3	1264.3	3.7	1383.1	96.7	2782.9
8k, 8.5k	543	10.2	335	0	888.2	60.4	30.5	15.7	1341.4	4.4	1452.0	124.7	2469.3
8.5k, 9k	354.5	5.6	249.3	0	609.3	52	31.1	16.1	1362.6	19	1480.7	148	2257.2
9k, 9.5k	474.9	5.3	455.4	0	935.7	80.2	27.8	14.4	1096.9	66.1	1284.4	216.8	2503
9.5k, 10k	446.8	4.8	475.5	0	927	78.2	23.8	12.4	843.2	41	968.7	153.8	2090.4

Table 3 presents power measurements for *K1* execution throughout the hardware structures. After kernel launch, (i.e., *Cycle 5000*), the threads load thread-identifier data from the parameter memory, causing higher power consumption on SMs. The power values of the memory partitions in the following cycles get higher than the SMs. Additionally, the power consumption by the LD/ST unit is high due to the intense memory operations and pressure on the L1D cache, and other units apart from the register file portion of the execution unit get lower after *Cycle 5500*. We can see that DRAM consumes the most power in the memory partitions with intense usage of high-bandwidth [24].

4.3. Performance-Power Analysis of an Embedded Application

Embedded applications targeting artificial intelligence, computer vision, and advanced graphics require high computational power. Jetson AGX Xavier provides a System-on-Module that meets these demands with a Volta-based GPU. We execute CUDA implementation of the *Fast Fourier Transform (FFT)* from the GPU4S suite [37].

Table 4 presents the overall performance metrics for executing the first three kernels. The thread blocks consist of 256 threads, and a maximum of 8 thread blocks can run in parallel on the SMs since the register file limits the number of simultaneous thread block execution. Since the kernels other than the *K0* utilize the hardware similarly, they result in similar performance as shown for *K1* and *K2*. Thus, we explain the relationship between performance and power consumption for *K0*, which employs diverse characteristics.

Table 4: Fast Fourier kernel performance statistics.

Kernel	**GPU IPC**	**GPU Occ.**	**L1D**		**L2**		**DRAM**		**Total Cycle**
			Miss Rate	Res. Fail Rate	Miss Rate	Res. Fail Rate	Row Buffer Loc. (L+S)	Row Buffer Loc. (Load)	
FFT - K0	31.76	62.52 %	0.67	0.92	0.65	0	0.21	0.89	990276
FFT - K1	100.22	80.25 %	0.1	0,75	0.20	0	0.33	0.92	235390
FFT - K2	49.20	85.28 %	0.1	0.837	0.21	0	0.41	0.92	239755

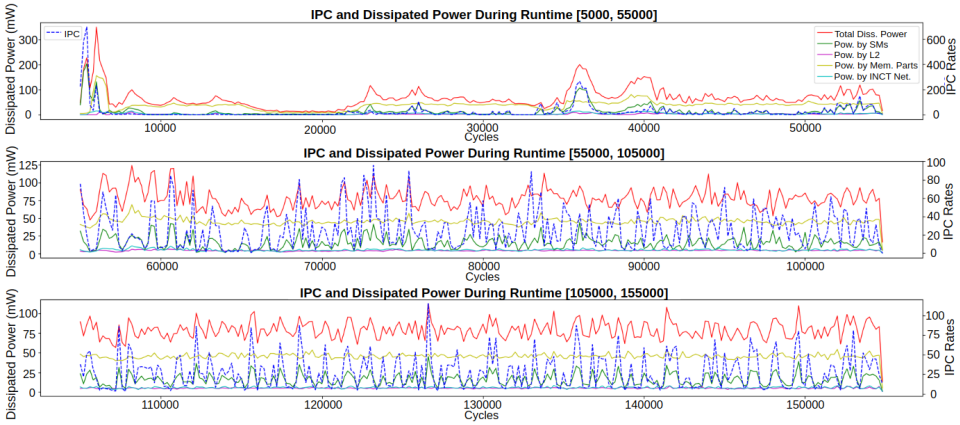


Fig. 14: IPC with dissipated power metrics in the cycle range of [5000, 155000] for *FFT-K0*.

Figure 14 displays the IPC and power metrics measurements for *K0* at different execution cycle intervals. To avoid losing observation details related to IPC and power metrics of the kernel, we report the relationship in three execution intervals separately. At cycles [5000, 6500], the power per cycle and IPC values increase significantly since each SM

registers thread-identifier information in their private registers, causing a high activation on register files and functional units. Figure 15, (as part of our *General View*), shows four loads, three data movements, and one multiply-add instruction executed by 16384 threads concurrently (with 64 thread blocks where each contains 256 threads) in that interval. Throughout the *K0* execution, we have high L1D and L2 miss rates. Figure 14 reveals the effect of those accesses on power dissipation (yellow line). The memory partitions dissipate half of the total power, around 50W, due to intensive activation at runtime. IPC and power dissipation increase instantly with SM activation points. By tracking the points, where IPC increases between [55000, 105000] and [105000, 155000] cycles, we can see parallel increments in power metrics dissipated by SMs and GPU. While the overall IPC value for *K0* equals 31.76 on SMs, runtime IPC varies in the range of [5, 75]. The concurrent load and store instructions cause small latencies and slight computational loads on the SMs.

PC	OPCODE	OPERAND
0	ld.param.u64	%rd1 [_Z21binary_reverse_kernelPKfPfli_param_0]
8	ld.param.u64	%rd2 [_Z21binary_reverse_kernelPKfPfli_param_1]
16	ld.param.u64	%rd3 [_Z21binary_reverse_kernelPKfPfli_param_2]
24	ld.param.u32	%r2 [_Z21binary_reverse_kernelPKfPfli_param_3]
32	mov.u32	%r3 %ntid.x
40	mov.u32	%r4 %ctaid.x
48	mov.u32	%r5 %tid.x
56	mad.lo.s32	%r1 %r3 %r4 %r5

Fig. 15: Instructions preparing threads for execution with thread-specific data in the cycle range of [5000, 6500] for *FFT-K0*.

4.4. Analyzing the Impact of the Input Size on Resource-Critical Embedded System

Since embedded devices employ relatively smaller computational and memory resources, the target programs' resource utilization becomes more important and influential on the execution performance. Besides program characteristics, the input size significantly affects the pressure on the memory structures and the obtained performance. To understand the impacts of the input size on memory behavior and execution performance, we utilize the memory statistics view of our tool. We execute *softmax* program from GPU4S suite [37] for two matrix sizes (1024 and 4096).

Figure 16 presents the memory statistics of the first 500K cycles. The execution employs high L1 reservation failure rates and L2 miss rates for the first 300K cycles for both executions. However, the small input (1024) can fit within the L2 and L1 caches around *Cycle 300K* and *Cycle 350K*, respectively. On the other hand, the large input (4096) execution still employs low hit L1 and L2 rates.

Figure 17 demonstrates the corresponding IPC behavior, which is compatible with cache rates. While the IPC for the large input case oscillates at small values (0.2-0.8), the IPC values for the small input case are steadily low but increase after L2 and L1 cache

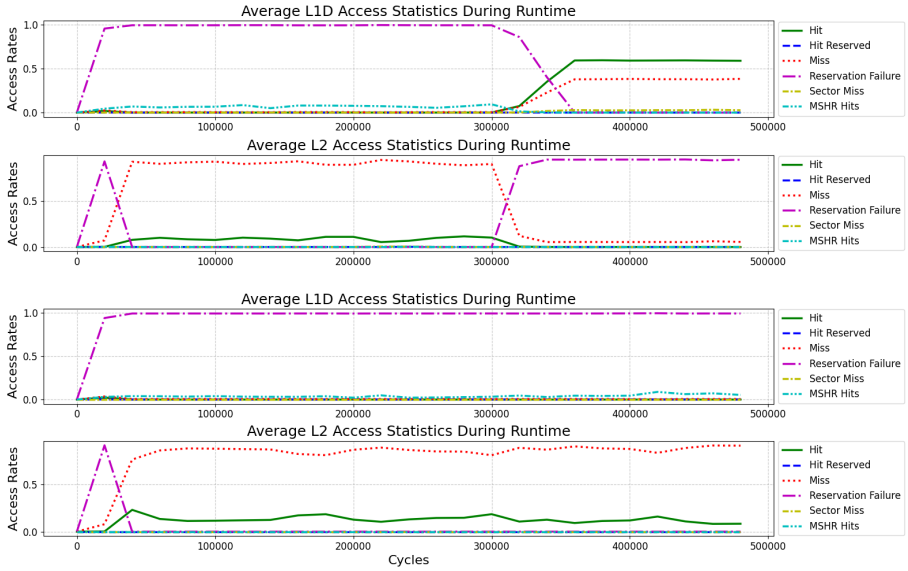


Fig. 16: Average memory access statistics at first 500K cycles for *softmax* kernel (matrix sizes with 1024 and 4096).

breakpoints. With this analysis, we can decide on the cache utilization of the target input dimension for the program.

5. Related Work

AerialVision [1] visualizes runtime warp divergence, dynamic IPC, global memory access statistics, active thread count, and a mapping window between source code and exposed pipeline latency metrics of kernel execution. While GPPRMon enables developers to dig into details of runtime GPU execution within specific microarchitectural units, AerialVision profiles the run time metrics on a much longer execution scale and visualizes overall GPU performance without per-component performance analysis. GPPRMon is similar to AerialVision in terms of providing runtime performance metrics, but GPPRMon can uncover more detailed behavioral insight inside kernel execution with more detailed runtime observation opportunities. Furthermore, AerialVision does not support displaying dissipated power.

Nsight Compute Tool [27] runs an application on an NVIDIA GPU device and collects average hardware usage statistics for the main components on a kernel basis. Nsight System Tool [28] is the other tool that is mostly preferred to analyze end-to-end kernel execution performance, including CPU-GPU communications. Profiling through these tools eases interpreting the overall kernel performance and hardware utilization with well-designed GUIs. In addition to performance analysis tools, GPU users can track instant power dissipation of deployed GPU through *System Management Interface* (SMI) library (i.e., nvidia-smi) [31]. Different from all NVIDIA tools and library frameworks, GPPRMon can track and visualize execution and hardware utilization statistics at run-

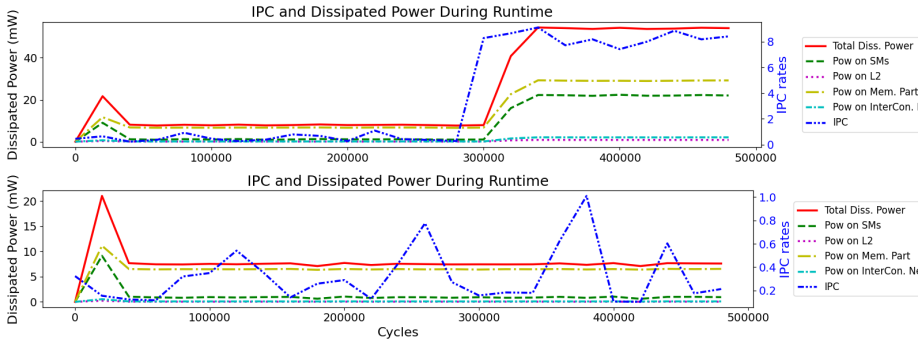


Fig. 17: IPC values for first 500K cycles for *softmax* kernel (matrix sizes with 1024 and 4096).

time for each component separately. GPPRMon further monitors warp-instruction issues/completions for each thread block, which provides an on-point analysis opportunity in a highly parallel execution. Hence, GPPRMon provides multi-functional performance and power tracking options together with various configurations, and it supports these features for all official GPU models described in Section 2.3.

TAU Performance System [39] is a performance profiling tool for hybrid parallel programs such as CUDA and OpenCL by intercepting the execution and inserting metric collection calls. After gathering the performance results, TAU integrates them with data through instrumentation to display a performance picture of the execution. Like Nsight Systems Tool [28], which provides performance traces for high-level CUDA functions such as *cudaMemcpy*, TAU does not address detailed hardware usage and performance during runtime as GPPRMon does. While TAU is insufficient to reveal the hardware's energy consumption, GPPRMon provides runtime power breakdown of GPU subcomponents during the execution.

Daisen [43] displays the overall occupancy on SM pipeline stages and memory components during the simulation. The authors aim to propose a performance-improving architecture by iterating an algorithm that benefits from the previously collected performance and hardware usage metrics. In other words, Daisen does not highlight the performance degrading points analytically. Instead, their approach addresses general bottlenecks, shares with the user, and tries to optimize performance in each iteration. Similar to [33], their approach offers more systematic performance optimization points on GPU executions. On the other hand, GPPRMon focuses on monitoring the execution at the PTX instruction level and relating the execution with the memory occupancy during execution. That is, GPPRMon offers a runtime analytical observation environment to evaluate both architecture and application inefficiencies. Moreover, while GPPRMon supports runtime power tracking and overall energy consumption metrics, Daisen does not provide them.

CHAMPVis [33] offers a web-supported architectural performance monitoring tool that provides a hierarchical analysis of trends and bottlenecks for varying applications. The tool aims to analyze performance by evaluating metrics from a system view and automatically generates predictive optimization speculations for the applications. Unlike GPPRMon, CHAMPVis does not employ any dissipated power tracking. GPPRMon yields detailed execution statistics, whereas CHAMPVis highlights the overall execution trends.

Francisco et al. [5] model a portion of the memory hierarchy of the AMD GPUs accurately by investigating the behavior of MSHRs, coalescing for vector (warp for our case) memory requests by extending Multi2Sim [45]. The authors find that the size and switching frequencies of MSHRs affect performance directly, especially for irregular workloads. Additionally, coalesced memory accesses, implemented in the simulator, reduce the repeated overheads of memory requests, significantly affecting performance for global memory accesses. GPPRMon offers more comprehensive evaluation opportunity besides the impact of MSHRs and memory request coalescing by being capable of runtime monitoring of both all memory hierarchy and SMs. Furthermore, GPPRMon extends relating the performance analysis options together with the dissipated power for performance/power trade-off evaluations at any execution scale.

Tanzima et al. [11] present an analysis and visualization framework (DASHING) targeting exascale computing consisting of multi-core architectures. The authors provide user interaction to compare varying configuration models by providing environment configuration options for analysis and visualization. Similarly, GPPRMon supports official GPU configurations of the GPGPU-Sim for multiple performance analyses and includes multi-functional metric collection and visualization perspectives depending on the user's demand. However, we obtain a more low-level performance analysis tool as described in *Temporal* and *Spatial View* Sections. In addition to runtime monitoring support, GPPRMon allows tracking the power dissipation across microarchitectural GPU components at runtime, which enables evaluating performance/power together, especially for energy-critical applications.

MemAxes [8] proposes an analysis framework for memory performance with various inspections on multi-core architectures. Its interface displays the analysis by obtaining performance metric samples from different simulations and mapping them into a single visual. In addition, the authors offer memory utilization-based clustering research among the benchmark applications. On the contrary, GPPRMon enables the analysis of GPU execution at runtime with performance and power dissipation features through *Spatial*, *Temporal*, and *General Views*.

To the best of our knowledge, GPPRMon differs from existing works by targeting hardware from both embedded and large-scale GPUs and providing runtime performance and power analysis opportunities. GPPRMon offers a detailed micro-architectural and power dissipation analysis for any GPU application. With observations through GPPRMon, developers can identify performance and power issues for both applications and architectures. For instance, data sparsity, irregular memory access behaviors, compute-bound behaviors, and the execution interval of these bottlenecks can be easily recognized via GPPRMon. Moreover, since GPPRMon supports runtime power dissipation together with performance monitoring, developers can analyze the energy impact of those performance issues and make their decisions. The studies conducting performance-energy tradeoff analysis [2, 38] and power capping strategies accordingly [49, 18] can potentially benefit from our fine-grained dynamic performance and power evaluations. We believe GPPRMon will be useful in fulfilling the micro-architectural performance and power analysis of GPUs for diverse application domains.

6. Conclusion

To conclude, we design and build a systematic runtime metric collection of instruction monitoring, performance, memory access, and power consumption metrics. Our tool provides a multi-perspective visualization framework that displays performance, execution statistics of the workload, occupancy of the memory hierarchy, and dissipated power results to conduct baseline analysis on GPUs at runtime. Since GPPRMon reliably reveals all the interactions between hardware and application at runtime, it potentially helps the researchers and software developers understand the dynamic behavior of the target GPU execution. While the researchers can propose architectural optimizations based on the detailed information, the software developers can deal with performance bottlenecks by analyzing our visualizations and fixing the target GPU code accordingly. Additionally, the low-power embedded system developers can utilize dynamic performance and power considerations to balance between two important design points.

We believe that GPPRMon will help to conduct baseline analysis for the literature concerning GPU performance and power dissipation and eliminate the need for additional in-house efforts that involve real-time monitoring and profiling support. Since GPPRMon's metric collection and visualization components are independent, other microarchitectural metrics obtained during runtime from either GPUs or other simulators can be displayed by exploiting the GPPRMon visualizer. The integration of other simulators with GPPRMon results in a system capable of runtime execution, memory statistics, and power dissipation tracker among all the possible GPUs. Alternatively, extending the runtime visualization by deepening the scope of the runtime metric collection and visualization phases of GPPRMon is another future direction.

7. Acknowledgement

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Grant No: 122E395. This work is partially supported by CERCIRAS COST Action CA19135 funded by COST Association.

References

1. Ariel, A., Fung, W.W.L., Turner, A.E., Aamodt, T.M.: Visualizing complex dynamics in many-core accelerator architectures. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). pp. 164–174 (2010)
2. Aslan, B., Yilmazer-Metin, A.: A study on power and energy measurement of nvidia jetson embedded gpus using built-in sensor. In: 2022 7th International Conference on Computer Science and Engineering (UBMK). pp. 1–6 (2022)
3. del Barrio, V., Gonzalez, C., Roca, J., Fernandez, A., E, E.: Attila: a cycle-level execution-driven simulator for modern gpu architectures. In: 2006 IEEE International Symposium on Performance Analysis of Systems and Software. pp. 231–241 (2006)
4. Becker, P.H.E., Arnau, J.M., González, A.: Demystifying power and performance bottlenecks in autonomous driving systems. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). pp. 205–215 (2020)
5. Candel, F., Petit, S., Sahuquillo, J., Duato, J.: Accurately modeling the gpu memory subsystem. In: 2015 International Conference on High Performance Computing & Simulation (HPCS). pp. 179–186 (2015)

6. Chen, X., Chang, L.W., Rodrigues, C.I., Lv, J., Wang, Z., Hwu, W.M.: Adaptive cache management for energy-efficient gpu computing. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 343–355 (2014)
7. Collange, C., Daumas, M., Defour, D., Parello, D.: Barra: A parallel functional simulator for gpgpu. In: 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. pp. 351–360 (2010)
8. Giménez, A., Gamblin, T., Jusufi, I., Bhatele, A., Schulz, M., Bremer, P.T., Hamann, B.: Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. IEEE Transactions on Visualization and Computer Graphics 24(7), 2180–2193 (2018)
9. Guerreiro, J., Ilic, A., Roma, N., Tomás, P.: Dvfs-aware application classification to improve gpgpus energy efficiency. Parallel Computing 83, 93–117 (2019), https://www.sciencedirect.com/science/article/pii/S0167819118300243
10. Hong, J., Cho, S., Kim, G.: Overcoming memory capacity wall of gpus with heterogeneous memory stack. IEEE Computer Architecture Letters 21(2), 61–64 (2022)
11. Islam, T., Ayala, A., Jensen, Q., Ibrahim, K.: Toward a programmable analysis and visualization framework for interactive performance analytics. In: 2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools). pp. 70–77 (2019)
12. Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., Gonzalez, J.E.: Checkmate: Breaking the memory wall with optimal tensor rematerialization. CoRR abs/1910.02653 (2019), http://arxiv.org/abs/1910.02653
13. Jog, A., Kayiran, O., Nachiappan, N.C., Mishra, A.K., Kandemir, M.T., Mutlu, O., Iyer, R.R., Das, C.R.: OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In: Sarkar, V., Bodík, R. (eds.) Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013. pp. 395–406. ACM (2013), https://doi.org/10.1145/2451116.2451158
14. Kandiah, V., Peverelle, S., Khairy, M., Pan, J., Manjunath, A., Rogers, T.G., Aamodt, T.M., Hardavellas, N.: Accelwattch: A power modeling framework for modern gpus. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. p. 738–753. MICRO '21, Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3466752.3480063
15. Khairy, M., Shen, Z., Aamodt, T.M., Rogers, T.G.: Accel-sim: An extensible simulation framework for validated gpu modeling. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 473–486 (2020)
16. Koo, G., Oh, Y., Ro, W.W., Annavaram, M.: Access pattern-aware cache management for improving data utilization in gpu. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). pp. 307–319 (2017)
17. Krzywaniak, A., Czarnul, P., Proficz, J.: Gpu power capping for energy-performance trade-offs in training of deep convolutional neural networks for image recognition. In: Computational Science – ICCS 2022. pp. 667–681. Springer International Publishing, Cham (2022)
18. Krzywaniak, A., Czarnul, P., Proficz, J.: Dynamic gpu power capping with online performance tracing for energy efficient gpu computing using depo tool. Future Generation Computer Systems 145, 396–414 (2023), https://www.sciencedirect.com/science/article/pii/S0167739X23001267
19. Leskovec, J., Lang, K., Dasgupta, A., Mahoney, M.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics 6 (11 2008)
20. Lew, J., Shah, D.A., Pati, S., Cattell, S., Zhang, M., Sandhupatla, A., Ng, C., Goli, N., Sinclair, M.D., Rogers, T.G., Aamodt, T.M.: Analyzing machine learning workloads using a detailed gpu simulator. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 151–152 (2019)

21. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 469–480 (2009)
22. Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Comput. Surv. 47(2) (aug 2014), https://doi.org/10.1145/2636342
23. NVIDIA: Quadro gv100 data sheet (2018), https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-volta-gv100-a4-nvidia-704619-r3-web.pdf
24. NVIDIA: Volta architecture white paper (March 2018), https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
25. NVIDIA: Jetson agx xavier and the new era of autonomous machines (2019), https://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf
26. NVIDIA: Cuda toolkit documentation (Jan 2023), https://docs.nvidia.com/cuda
27. NVIDIA: Nsight compute kernel profiling guide (2024), https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html
28. NVIDIA: Nsight systems profiling guide (2024), https://developer.nvidia.com/nsight-systems
29. NVIDIA: Nvidia cuda compiler driver nvcc (2024), https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/
30. NVIDIA: Parallel thread execution isa version 8.4 (2024), https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
31. NVIDIA: System management interface (2024), https://developer.nvidia.com/system-management-interface
32. O'Neil, M.A., Burtscher, M.: Microarchitectural performance characterization of irregular gpu kernels. In: 2014 IEEE International Symposium on Workload Characterization (IISWC). pp. 130–139 (2014)
33. Pentecost, L., Gupta, U., Ngan, E., Beyer, J., Wei, G.Y., Brooks, D., Behrisch, M.: Champvis: Comparative hierarchical analysis of microarchitectural performance. In: 2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools). pp. 55–61 (2019)
34. Power, J., Hestness, J., Orr, M.S., Hill, M.D., Wood, D.A.: gem5-gpu: A heterogeneous cpu-gpu simulator. IEEE Computer Architecture Letters 14(1), 34–36 (2015)
35. PowerUP, T.: V100 tech powerup (2018), https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-16-gb.c2957
36. Rhu, M., Sullivan, M., Leng, J., Erez, M.: A locality-aware memory hierarchy for energy-efficient gpu architectures. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. p. 86–98. MICRO-46, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2540708.2540717
37. Rodriguez, I., Kosmidis, L., Lachaize, J., Notebaert, O., Steenari, D.: Gpu4s bench: Design and implementation of an open gpu benchmarking suite for space on-board processing. Universitat Politecnica de Catalunya (2019)
38. Sezgin, Y., Öz, I.: Performance-reliability tradeoff analysis for safety-critical embedded systems with gpus. In: Ulusal Yüksek Başarımlı Hesaplama Konferansı (BAŞARIM) (2024), https://indico.truba.gov.tr/event/140/attachments/310/642/BASARIM2024-BildiriKitabi.pdf
39. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (may 2006), https://doi.org/10.1177/1094342006064482
40. Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B., Hua, Q.S.: Graph processing on gpus: A survey. ACM Comput. Surv. 50(6) (jan 2018), https://doi.org/10.1145/3128571
41. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: Highlights - november 2022 (2022), https://www.top500.org/lists/top500/2022/11/highs/

42. Sun, Y., Mukherjee, S., Baruah, T., Dong, S., Gutierrez, J., Mohan, P., Kaeli, D.: Evaluating performance tradeoffs on the radeon open compute platform. In: 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 209–218 (2018)
43. Sun, Y., Zhang, Y., Mosallaei, A., Shah, M.D., Dunne, C., Kaeli, D.R.: Daisen: A framework for visualizing detailed GPU execution. Comput. Graph. Forum 40(3), 239–250 (2021), https://doi.org/10.1111/cgf.14303
44. Topçu, B., Öz, I.: Gpprmon: Gpu runtime memory performance and power monitoring tool. In: Euro-Par 2023: Parallel Processing Workshops. pp. 17–29. Springer Nature Switzerland (2024)
45. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2sim: A simulation framework for cpu-gpu computing. In: 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 335–344 (2012)
46. Vijaykumar, N., Ebrahimi, E., Hsieh, K., Gibbons, P.B., Mutlu, O.: The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). pp. 829–842 (2018)
47. Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A.T., Owens, J.D.: Gunrock: Gpu graph analytics. ACM Trans. Parallel Comput. 4(1) (aug 2017), https://doi.org/10.1145/3108140
48. Xu, Z., Chen, X., Shen, J., Zhang, Y., Chen, C., Yang, C.: Gardenia: A graph processing benchmark suite for next-generation accelerators. ACM Journal on Emerging Technologies in Computing Systems 15(1) (jan 2019), https://doi.org/10.1145/3283450
49. Zhao, D., Samsi, S., McDonald, J., Li, B., Bestor, D., Jones, M., Tiwari, D., Gadepally, V.: Sustainable supercomputing for ai: Gpu power capping at hpc scale. In: Proceedings of the 2023 ACM Symposium on Cloud Computing. p. 588–596. SoCC '23, Association for Computing Machinery, New York, NY, USA (2023), https://doi.org/10.1145/3620678.3624793
50. Zhao, X., Adileh, A., Yu, Z., Wang, Z., Jaleel, A., Eeckhout, L.: Adaptive memory-side last-level gpu caching. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). pp. 411–423 (2019)