Photon: A Fine-grained Sampled Simulation Methodology for GPU Workloads

Changxi Liu
National University of Singapore
Singapore

Yifan Sun
College of William & Mary
USA

Trevor E. Carlson
National University of Singapore
Singapore

ABSTRACT

GPUs, due to their massively-parallel computing architectures, provide high performance for data-parallel applications. However, existing GPU simulators are too slow to enable architects to quickly evaluate their hardware designs and software analysis studies. Sampled simulation methodologies are one common way to speed up CPU simulation. However, GPUs apply drastically different execution models that challenge the sampled simulation methods designed for CPU simulations. Recent GPU sampled simulation methodologies do not fully take advantage of the GPU's special architecture features, such as limited types of basic blocks or warps. Moreover, these methods depend on up-front analysis via profiling tools or functional simulation, making them difficult to use.

To address this, we extensively studied the execution patterns of a variety of GPU workloads and propose Photon, a sampled simulation methodology tailored to GPUs. Photon incorporates methodologies that automatically consider different levels of GPU execution, such as kernels, warps, and basic blocks. Photon does not require up-front profiling of GPU workloads and utilizes a light-weight online analysis method based on the identification of highly repetitive software behavior. We evaluate Photon using a variety of GPU workloads, including real-world applications like VGG and ResNet. The final result shows that Photon reduces the simulation time needed to perform one inference of ResNet-152 with batch size 1 from 7.05 days to just 1.7 hours with a low sampling error of 10.7%.

CCS CONCEPTS

• **Computing methodologies** → **Modeling and simulation**; **Graphics processors**; **Cluster analysis**.

KEYWORDS

GPU, Workload sampling, Simulation

ACM Reference Format:
Changxi Liu, Yifan Sun, and Trevor E. Carlson. 2023. Photon: A Fine-grained Sampled Simulation Methodology for GPU Workloads. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3623773

1 INTRODUCTION

General-Purpose Graphics Processing Units (GPUs) are widely used to accelerate data-parallel applications in domains such as big data analysis [12, 30], machine learning [1, 14], and physics simulation [17, 34]. Because of their massively-parallel computing architectures, GPUs have become one of the most important types of accelerators that power today's datacenters [25] and supercomputers [55].

A GPU functions by grouping a subset of threads (usually 32 or 64) into a warp to execute in parallel on shared execution units to maximize parallelism. One GPU usually contains hundreds to thousands of execution units while each unit processes multiple warps at the same time, allowing them to overlap long-latency memory operations. This feature enables GPUs to achieve extremely high throughput, with today's GPUs achieving nearly 134 TFLOPs [41] for single-precision floating point operations.

In the past decades, researchers from academia and industry have proposed new hardware designs to improve GPU throughput [6, 7, 16, 24, 39, 54, 63]. To validate these proposed designs, architects depend on GPU simulators to profile applications, examine hardware behaviors, and evaluate overall performance. Simulators are also useful for existing GPU designs, as they allow performance architects to build a detailed understanding of GPU workloads, and to find and understand bottlenecks to help software developers improve their existing algorithms. However, the performance of open-source GPU simulators [5, 32, 52, 56] is quite low, especially considering the huge workload processed by GPUs. With performance levels below 30 KIPS, these simulators are over 1,000,000,000× slower than the real-time performance of a GPU. The disparity between the high throughput of GPUs, and the extremely low simulation rate handicaps hardware researchers and hinders innovation in this domain. To give one concrete example, an inference of the VGG-16 [50] convolutional neural network (CNN), with an input image size of 224×224, requires ≃ 15.5 billion FLOPs, requiring a simulation time of 3.44 days to evaluate the complete workload. Therefore, alternatives that allow researchers to simulate workloads faster are needed to evaluate next-generation hardware and software.

Sampled simulation methodologies are one common way to speed up workload simulation. Sampling single-threaded workloads for CPU architectures are mainly derived from two representative works: SimPoint [48] and SMARTS [60] (profile-driven and statistical sampling simulation methods, respectively). Moreover, modern sampled simulation methodologies, including LoopPoint [45], FlexPoints [59], and BarrierPoint [11], have been proposed for multi-threaded applications. However, GPUs apply a drastically different execution model that involves a significantly larger number of computing cores and threads, challenging the sampled simulation methods designed for CPU simulations. As the number of threads increases, the phase behavior of these workloads can vary due to the interaction among threads. Therefore, a systematic sampled simulation solution tailored for GPU execution is needed to enable faster GPU simulation.

Recently, sampled simulation methodologies [4, 29, 31, 40] have been introduced targeting GPUs. However, these methods use sampling methodologies similar to CPUs, such as monitoring for a stable IPC during execution intervals [4, 29], or they do not fully explore GPU-specific sampling directions, such as intra-kernel sampling [31, 40]. They can potentially miss-out on opportunities to improve performance as they have not taken into account unique features of the GPU, like warp-based execution. Moreover, it can be challenging to utilize existing methodologies if profiling tools are unavailable for new GPU architectures, or if benchmarks are too large even for functional simulation [4]. Earlier sampled simulation techniques that require profiling tools or functional simulation to conduct up-front analysis could significantly increase end-to-end simulation time for researchers working to optimize systems via compiler-hardware co-design [58], optimization algorithms for GPUs [13, 38]. They would now be required to repeat the pre-profiling process each time the binary is updated.

To facilitate building a more practical sampled simulation methodology for GPUs, we have extensively studied the execution patterns seen in GPUs. We find that GPU architecture, different from CPU architecture, (1) requires specific methodologies as hundreds to thousands of basic units of work from different levels of GPU workloads, including warps, basic blocks, and instructions, simultaneously execute and interact with other work at the same level. We also find that (2) sampled simulation methods should occur at different levels of interest, as no single method can handle all GPU workloads. This is because some GPU workloads have a huge number of warps that only contain tens of instructions each, while other GPU workloads have limited numbers of warps with each warp executing a significant number of instructions. (3) We also find that significant, time-consuming up-front profiling is not required as GPU workloads contain limited types of kernels, warps, basic blocks, and instructions. These units are often highly repetitive and demonstrate similar behaviors, creating the opportunity to avoid analyzing and simulating all of them in a highly detailed manner.

Based on our insights into GPU simulation, we propose Photon, a systematic methodology to accelerate GPU simulation. Photon requires no upfront analysis and fully relies on online analysis to accelerate GPU simulation with sampling methods. Photon combines sampling-based simulation solutions at three different levels, including kernel-sampling, warp-sampling, and basic-block-sampling.

Photon provides comprehensive solutions for inter- and intra-kernel sampling according to different types of GPU workloads. For inter-kernel sampling, Photon skips detailed simulation of GPU kernels if one similar kernel has been simulated before. Instead of using up-front analysis results to decide the similarity, we online functionally simulate and analyze a sample of warps to understand the kernel's phase behavior. GPU kernels are typically combined with sufficient numbers of warps with limited types, and a subset of warps can often represent the whole kernel. For intra-kernel sampling, Photon performs sampled simulation of one GPU kernel via warp-sampling and basic-block-sampling. Photon automatically switches between sampling different levels of the program execution, according to runtime information, to achieve a good compromise between high performance and low simulation error.

We implement Photon on MGPUSim [52], a cycle-level GPU simulator. The goal of Photon is to enable large-scale system simulation and empower the community to perform GPU hardware research and software analysis for different types of GPU workloads with different inputs. Overall, in this work, we make the following contributions:

- We propose Photon, the first sampled GPU simulator that requires no up-front analysis. Photon supports basic-block-sampling, warp-sampling and kernel-sampling (as well as a combination of the three), so that Photon can use sampling to simulate a large variety of GPU workloads with both high performance and low sampling errors.
- We evaluate the kernel-, warp-, and basic block-level sampled simulation of Photon, and the results show that our method achieves up to 24.65× for the single kernel GPU workloads over the full detailed simulation with the average sampling error rate 6.83%.
- We evaluate Photon on real-world applications, including PageRank, VGG, and ResNet. The final result shows that Photon reduces the simulation time needed to perform one inference of ResNet-152 with batch size 1 from 7.05 days to 1.7 hours. Photon achieves 39.1× speedup with a very low error rate of 10.7% for ResNet-152.

2 SAMPLING GPU WORKLOADS

In this section, we describe the main features of GPU workloads that enable fast and accurate sampling in Photon. We also discuss prior sampled simulation methodologies and some of their limitations.

GPU workloads. A GPU program is usually comprised of a CPU and GPU portion. A function call from the CPU side to the GPU side is called a *kernel launch*. Usually, one kernel contains a large number of threads that execute the same program but across different sets of data. Moreover, the GPU groups a number of threads (typically 32-64) into warps. Threads within a warp always execute the same instruction at the same time. Additionally, a programmer-defined number (usually 1-16) of warps can be further organized into blocks. On a GPU, threads from a block are executed in a single streaming multiprocessor (SM, which is a computing core on a GPU that executes instructions).

Similar to CPU programs, GPU programs are also organized as a list of instructions. These instructions form basic blocks (a group of instructions with one entry and one exit point). Note that as GPUs execute instructions at the warp level, basic blocks are also defined at the warp level, and all of the threads will enter and exit the basic block at the same point. Basic blocks can be differentiated by their start PC address and length (the instruction count).

GPU Architecture. The architecture of the GPU has received increasing attention from both academic and industry researchers, as it is a powerful platform for parallel processing data [1, 12, 19, 46]. The GPU architecture uses a hierarchy to arrange threads. For example, AMD GPUs are designed with command processors (CPs), as well as compute units (CUs) and SIMD units. NVIDIA GPUs are built from streaming multiprocessors (SMs) which contain shader processors (SPs). Compute units (or SPs) are basic calculation units for GPUs. Threads on GPUs are dynamically scheduled to one compute unit (CU) and all threads communicate and compete for memory bandwidth, the use of cache capacity, as well as for other resources. This can result in a variable, or unpredictable, execution pattern even if their instructions and inputs are the same.

GPU Simulators. GPU simulators are essential for GPU architecture exploration and GPU workload analysis. Open-source GPU simulators can be clustered into execution-driven and trace-driven based on how the front end of the simulator has been built. Trace-driven GPU simulators, such as MacSim [21], execute functionally-generated traces with a timing model to generate simulation results, such as GPU performance. While trace-driven simulators do not rely on the functional model, they do require pre-generated traces, which can make it challenging to evaluate certain design features such as dynamic synchronization mechanisms [51] and value-dependent optimization [54]. On the other hand, execution-driven GPU simulators, including MGPUSim [52], gem5 APU [8, 23] and GPGPUSim [5], directly execute the binary, but at the cost of lower performance due to the need for functional simulation during performance simulation. Other simulators, including Accel-Sim [32] and NVArchSim [57], support both execution- and trace-driven modes. Current GPU simulators have shown simulation performance that is lower than 30 KIPS [32], as simulators simulate GPU workloads on far fewer CPU cores than exist in real GPU cores.

Sampled Simulation Methodologies. Sampled simulation methodologies extrapolate whole application performance through detailed simulation of key, representative pieces of an application. To perform sampled simulation, simulators must support two modes, fast-forward mode, which allows for functional simulation only, and detailed mode, which enables the timing model. Sampled simulators switch between these two modes and split the whole application into fast-forward mode regions and detailed mode regions. The final results of fast-forward regions are predicted by other regions in detailed mode.

Sampled simulation methodologies for CPUs can be clustered into two main directions: statistical sampling and profile-driven sampling simulation. Statistical sampled simulation [59, 60] utilizes a number of small regions in detailed mode to represent the whole application as long as these regions are enough to cover all application behaviors. On the other hand, profile-driven sampled simulation [11, 45, 48] utilizes up-front analysis and clustering methods to select representative regions to simulate in detail mode. These regions are then used to determine the overall performance of the workload. However, with the number of threads increasing, these applications exhibit complex behaviors, making it challenging to accurately simulate their performance using traditional sampling methods.

Sampled simulation for GPU workloads. Prior work has attempted to use sampling methods to accelerate GPU simulation. GT-Pin [31] utilizes the kernel name, arguments, and basic block statistics to select representative portions of GPU programs at a *kernel-level* granularity. Sieve [40] points out that using both the kernel name and instruction count allows for both sampling speedups and low errors. GT-Pin and Sieve only focus on the inter-kernel level, but we find that speeding-up intra-kernel simulation is also very important for GPU simulators, as simulating one GPU kernel takes hours to days if the problem size is large. Yu et al. [61, 62] utilize a SimPoint-like [49] methodology to detect representative loops to extrapolate a *kernel* performance. As this work uses proxy applications, it can be difficult to extend. Principal Kernel Analysis (PKA) [4] and TBPoint [29] enable sampled simulation of GPU workloads at both the inter- and intra-kernel levels. One of the main limitations of these two works is that, to speed up simulation, they require stable values for intra-kernel IPCs. As we will show in Section 3, there are a number of applications where this does not hold, preventing these works from improving simulation performance significantly. In addition, their intra-kernel methods analyze the instructions executed in the same time interval as a whole, similar to how multi-threaded sampled simulation methodologies work [10, 11, 45]. Ignoring the hierarchical nature of GPU workloads can result in missed opportunities to accelerate sampled simulation. Apart from performance, these previous works can also experience low accuracy because of their assumption of a stable IPC during execution. For example, GPU workloads, like AES, exhibit stable IPC at the start of their kernel. However, this measurement does not account for the instructions that have yet to be executed in the subsequent steps of these kernels, which may no longer exhibit the same stability, leading to incorrect results.

Besides these works, analytical modeling methodologies [27, 28, 36] are another choice to evaluate new GPU architectures. Analytical models are faster than simulator methodologies but these models are often unable to model the newest architectures. Performance model methodologies are also not sufficient to show the hardware execution details.

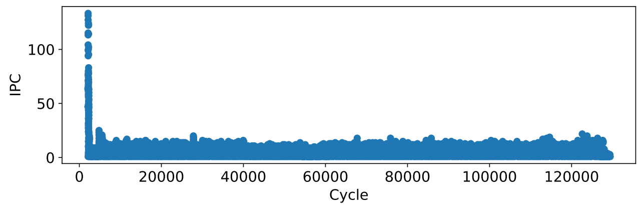
3 OBSERVATIONS

We observe GPU workload executions to gain insights that can support our decisions in designing the sampled GPU simulator.

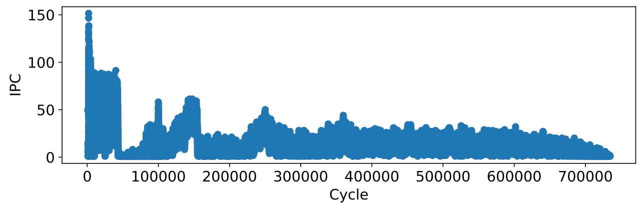
Observation 1: GPU execution does not demonstrate phases. Contrary to the conclusion mentioned by several research projects [37, 44], we find most GPU workloads do not demonstrate clear intra-kernel execution phases due to the interaction between threads, as shown in Figure 1b. While there may be software phases for each thread, concurrently executing thousands of threads blurs the boundary of the phases during execution, creating a swarm effect. This is a major difference with CPU execution. Existing sampled CPU simulations mainly rely on phase detection methods [10, 22, 45, 49, 59, 60]. However, our observation shows that not all CPU simulation methods apply directly to GPUs.

Observation 2: IPC may not remain stable during the entire workload execution. While it is expected that the swarm effect of GPU execution can lead to a stable IPC, our investigation into a large number of GPU workloads shows that IPC only stabilizes for some GPU kernels. For example, Figure 1a shows how the IPC stabilizes over time for ReLU benchmark [35]. However, the IPC in some other kernels, such as Matrix Multiplication (MM) shown in Figure 1b, changes frequently. Closely examining the MM workload suggests that its IPC variance comes from complex interactions between warps, such as resource competition and synchronization.

Prior sampled simulation methods [4, 10, 29, 48] are mainly based on the assumption that application phases repeat, allowing

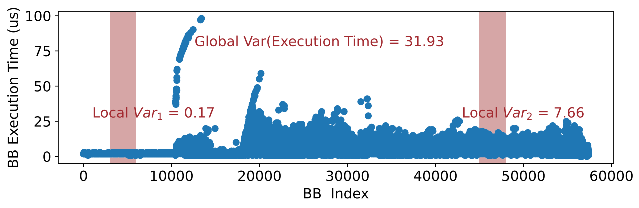


(a) Rectified Linear Unit (ReLU)

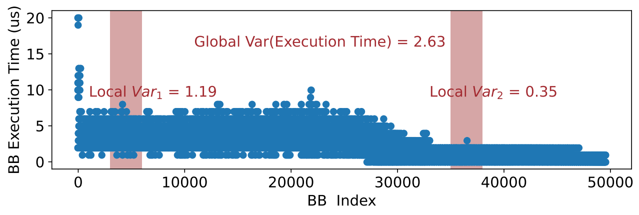


(b) Matrix Multiplication (MM)

Figure 1: A GPUs instructions per cycle (IPC) may (see (a)) or may not (see (b)) stabilize over time. These results show that IPC may not remain stable during the entire workload execution.



(a) MM : A regular application with a large kernel.



(b) SpMV : An irregular application with irregular memory accesses.

Figure 2: Execution time and variance of the dominating (in terms of execution time) basic block in the MM and SpMV applications. These results demonstrate that the variance, which is used as the threshold by prior works [4, 29] to determine run time stability, cannot always successfully predict stabilization as some applications' basic blocks stabilize at multiple stable regions.

one to extrapolate the execution time of one region based on the execution time of another region. For example, PKA [4] detects the IPC and checks for its stability. The two observations above demonstrate that relying on phase detection and stable IPCs may miss opportunities to skip detailed simulation. While it still may be possible to rely on IPC values, this observation suggests that a multi-layer sampled execution methodology may be needed.

Observation 3: Basic blocks' execution time can be stable over time. Basic blocks are consecutive instructions with only one entry point (e.g., a redirectable label) and one exit point (e.g., a branch instruction) [47, 48]. Basic blocks are labeled by the PC of their first instructions. For GPU workloads, the basic block we define is not the same as the one traditionally defined by compilers (where basic blocks end with a branch instruction). In addition, our methodology ends basic blocks using instructions that cause inter-warp synchronization. Specifically, Photon uses `s_barrier` in addition to branch instructions as the ending of basic blocks. We do not consider atomic instructions as MGPUSim does not currently support these instructions, but future work could evaluate these instructions as well. We consider these extra instructions to distribute the latency caused by interactions among warps into their respective basic blocks.

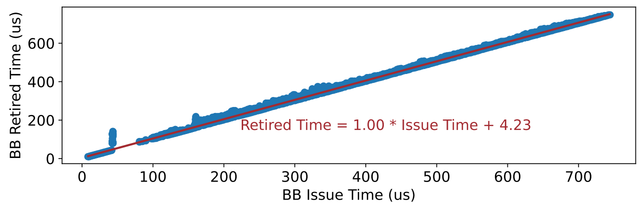
We also considered additional special instructions that synchronize threads inside the warp. For example, the instruction, `s_waitcnt`, isolates memory accesses so that a single basic block will not contain different sets of unrelated memory accesses. The evaluation of these instructions is left for future work.

We select two kernels, which are shown in Figure 2 and Figure 3, as examples of regular and irregular applications, and we see that other benchmarks evaluated also demonstrate similar behavior. Irregular applications, which are discussed by PKA [4], are difficult to sampled simulate due to their complicated memory access patterns and the workload imbalance among warps. Regular applications, on the other hand, have balanced workloads and regular memory access patterns across warps.

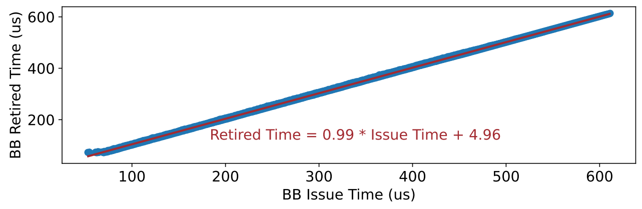
Figure 2 shows that the execution time of basic blocks can vary along with the application execution. We initialize the block indices in an ascending order based on the retirement time of basic blocks, as this facilitates the collection of online execution information once the basic blocks have completed their processing. Note that we define the execution time of basic blocks by the time interval between the issue time of the first instructions for the basic block and its next basic block. We find that utilizing the variance directly, as employed in prior works [4, 29] as a threshold, misses opportunities to detect if the basic block's execution time is stable. For example, the global variances of MM and SpMV in Figure 2 are 31.93 and 2.63, respectively. In addition, using variance, without examining the entire execution, could lead to the determination of false stable regions, as seen in Figure 2.

We also observe that the relationship between the issue and the retired time of basic blocks can help us to better understand the workload. As shown in Figure 3, the dots, which represent the issue and retired time for basic blocks, show a regular pattern that can be used, regardless of whether they belong to regular or irregular applications. These regular patterns are because a large number of threads average out of the influence of complicated interactions among threads, including memory access patterns and thread communication.

Observation 4: Warps for regular application can be stable overtime. Warps are the basic task unit for GPU workloads and warps usually process the same instructions but with different input data. The performance of warps executing the same instructions

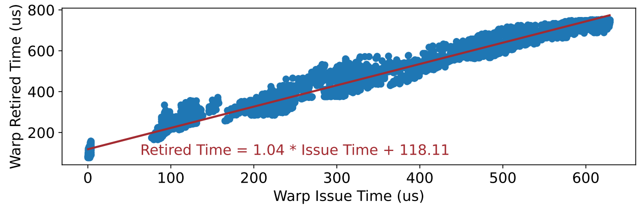


(a) MM : A regular application with a large kernel.

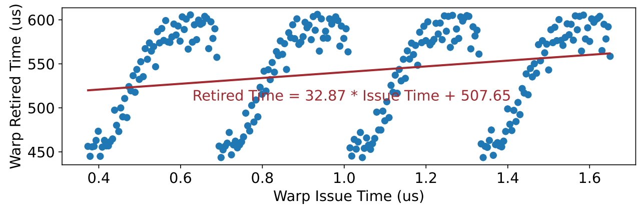


(b) SpMV : An irregular application with irregular memory accesses.

Figure 3: The relationship between the issue and retired time of the dominating (in terms of execution time) basic block in the MM and SpMV applications. These results show that this relationship can be utilized to detect if the basic blocks' execution time is stable.



(a) MM : A regular application with a large kernel.



(b) SpMV : An irregular application.

Figure 4: The corresponding relationship between the issue time and retired time of warps each benchmark. Plot (a) shows that this relationship can be used to detect if the warps' execution time is stable for regular applications. Plot (b) shows that we can utilize this relationship to identify if the execution time of warps is variable for irregular applications.

and inputs can also be different due to memory contention, and micro-architecture status, like TLB contents, which is affected by prior and current warps. We define warps of the same type as those that execute identical sequences of instructions during their execution, independent of whether threads inside a warp are masked as being executed or ignored. Specifically, if two warps have the same

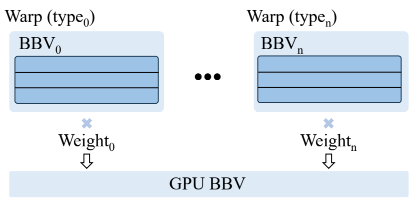


Figure 5: GPU BBV: the feature vector to capture the behavior of one GPU kernel. Warps of the same type are those that execute identical sequences of instructions during their execution and share the same BBV. Weights are the portion of warps in each type with respect to the total number of warps in the kernel. GPU BBVs are combined with the weighted BBVs.

Basic Block Vectors (BBVs), they belong to the same type. BBVs are often used in CPU-based sampling methodologies [11, 45, 48] to determine phase similarity. A BBV is a list of basic blocks executed at least once during program execution, identified by the program counter of the first instruction of the basic block. SimPoint [48] points out that if two portions of a program have similar BBVs, they share similar execution behavior. For GPU workloads, warps executing identical sequences of instructions have the same BBVs.

Regular applications, like ReLU, have only one or a small number of warp types, whereas irregular applications, like SpMV, have a large number of warp types due to different loop iterations, data-dependent branch divergence, etc. A warp is issued from the GPU scheduler to the calculation units and then sends the completion message to the GPU scheduler to release its resources after it finishes all instructions. We consider the issue time and retired time of warps by the time it is scheduled to the calculation unit and the time it finishes all instructions, respectively.

Referring to the basic block level method, we analyze the relationship between the issue time and retired time for regular and irregular applications, and the results are shown in Figure 4. We observe that the retired time of warps for regular applications, like MM, has the same pattern as that of the basic block level. This is because warps for regular applications tend to execute the same instructions, but with different data. However, for irregular applications like SpMV, warps have different tasks and memory access patterns are also different from each other.

Observation 5: GPU Kernels with similar GPU BBV have similar IPC. Similarities between kernel invocations, as well as their implication in sampled simulation, as been studied in prior work [4, 40]. All prior methods focus on counting the number of handpicked features (e.g., each type of instructions, warps). However, we find limitations with these methods as it can often lead to mis-clustering: 1) completely different kernels may be clustered together due to similar feature counts and 2) similar kernels may not be able to be grouped due to count difference. Therefore, a more sophisticated method beyond simply using feature counting needs to be designed.

For multi-threaded CPU workloads, prior methodologies, including BarrierPoint [11] and LoopPoint [45], directly merge BBVs of

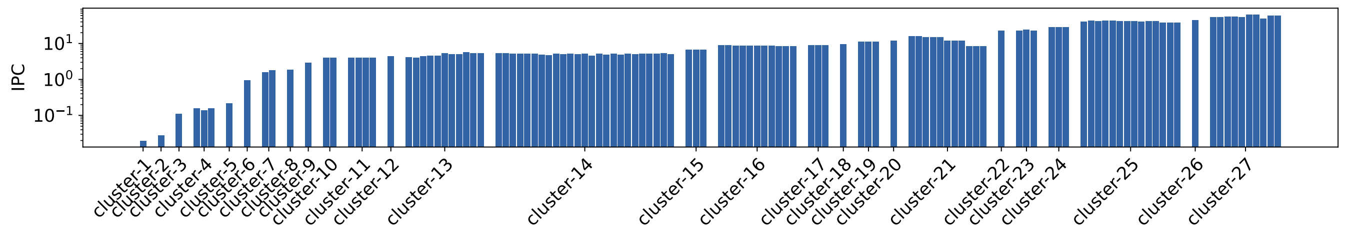


Figure 6: The IPC of kernels from all convolution, pooling, and dense layers of VGG-16. We cluster them based on GPU BBVs and the results show that kernels in the same cluster have similar IPC.

each thread together. However, these methods do not directly apply to GPU workloads as GPU workloads contain a significantly larger number of threads that are dynamically scheduled. We find that directly using multi-core CPU BBV clusters GPU workloads with similar execution behaviors into different groups due to this execution model.

GPU BBVs, as shown in Figure 5, are designed based on BBVs to capture the phase behavior of one GPU kernel. We first independently calculate BBVs for each warp and then project these BBVs into fixed-size BBVs (We use 16) to improve online clustering performance. Since warps with similar BBVs indicate similar execution behavior, we can easily cluster the warps if they have the same BBV. With the warp clustered, we then calculate the portion of warps in each cluster with respect to the total number of warps in the kernel ({weight_0, ..., weight_n}). Finally, the weighted BBVs are ordered according to a descending order based on the weights, and the GPU BBV is formed by listing the weighted (multiply the weight of BBVs with each BBV value) BBVs.

Figure 6 shows that Photon utilizes GPU BBVs to cluster all kernels from convolution, pooling, and dense layers of VGG-16. We can see kernels with similar GPU BBVs tend to have similar IPC. We also find that GPU kernels with similar BBVs and the same number of warps have a higher similarity than kernels with solely similar BBVs.

4 THE PHOTON METHODOLOGY

In this section, we propose a sampled GPU simulation methodology, Photon, which aims to enable sampled simulation for a large variety of GPU workloads. Photon employs a multi-tiered solution that includes kernel-sampling, warp-sampling, and basic-block-sampling. As described previously, GPU kernels are highly diverse and one method may not always be applicable. The multi-tiered solution ensures a wide coverage of applications that can be accelerated by Photon. Additionally, Photon does not require any up-front analysis. We believe any up-front analysis harms generality and can limit use cases (e.g., adding a new instruction or using different compilation options). Photon is embedded into our simulator so that researchers can easily use it in the same way as they would when running a simulator without sampling.

When starting the simulation of each kernel, Photon first decides if the kernel execution time can be predicted using kernel-sampling. If not, Photon will then attempt to use basic-block-sampling as it takes less time to detect. Meanwhile, the warp-sampling detector runs in parallel and Photon switches to warp-sampling when the criteria are satisfied as warp-sampling is faster.

Sampling Warps. Sampling GPU workloads at the warp level avoids functionally emulating instructions inside warps and can significantly reduce simulation time. Warp-sampling applies to applications with a large number of warps, with most warps executing the same instructions (according to **Observation 4**).

Sampling Basic Blocks. In large-warp kernels, the finishing time of the first warp is usually close to the finishing time of the GPU kernel, leaving limited opportunities to extrapolate the rest of the kernel execution. For example, for several inputs, the first warp of MM finishes after most instructions are completed. Also, basic-block-sampling works for irregular applications (e.g. SPMV) as it breaks down the kernel into smaller elements and can better utilize the statistical properties.

Sampling Kernels. GPU workloads, especially real-world applications like VGG-16, usually contain more than one GPU kernel. Photon avoids detailed simulation of duplicate GPU kernels by utilizing **Observation 5** and GPU BBVs.

Basic-block-sampling effectively accelerates simulation of irregular workloads, by considering divergence and synchronization in the statistical model. For divergence, single-threaded or 64-thread execution will have similar latencies in most cases, and hence, we do not need to take special consideration for divergence, at least for AMD GPUs. For synchronization, as most of the warps will perform synchronization at the same point, the synchronization latency will also stabilize, which can lead to stabilization of the execution time of warps. Therefore, our statistical model can also cover variable synchronization latency to a certain degree. However, Photon currently does not accelerate GPU workloads with extremely complex or long patterns of repeated behaviors as the methodology currently only keeps track of the last 1024 warps. These patterns could be analyzed at the cost of reducing sampling performance. However, according to our experience, 1024 is sufficient for the workloads evaluated.

We do not consider basic blocks with different synchronization points. In theory, it is possible that basic blocks may have distinct synchronization points and may have a distinct execution time. However, in practice, we do not find applications that can not be handled by our statistical model with three-level sampling. Photon falls back to full detailed simulation to ensure sampling accuracy in cases where stable regions are not detected at any level. Basic blocks with distinct synchronization points can be discerned via

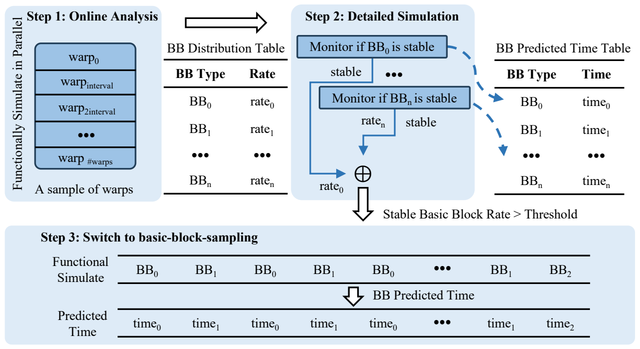
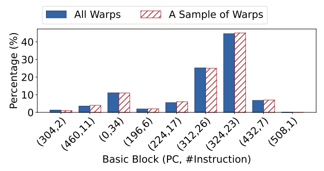
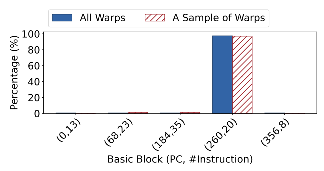


Figure 7: Photon online analyzes the distribution of basic block distributions via parallel functional simulating a sample of warps (Step 1). Then Photon detailed simulates GPU workloads, accumulates the rate of stable basic blocks, and records their execution time (Step 2). Finally, Photon switches to basic-block-sampling if the rate of stable basic blocks is larger than the threshold. Photon functional simulates warps and predicts their execution time at the basic block level (Step 3).



(a) SC : A regular workload.



(b) SpMV : An irregular workload.

Figure 8: The distribution of basic blocks of all warps and a sample of warps for regular and irregular applications.

monitoring special registers, like *VCC* and *EXEC* for AMD GPUs, during functional simulation. We leave this for future work.

4.1 Sampling Basic Blocks

The workflow of sampling basic blocks is represented in Figure 7. In the beginning, Photon functional simulates a percentage (we use 1%) of warps in parallel to collect the basic block distribution for the GPU kernel. Then Photon starts detailed simulation of the GPU workload, accumulates the rate of stable basic blocks, and records the runtime of each type of basic block. Finally, Photon switches to basic-block-sampling when the rate is larger than the threshold (we use 95%). Photon functionally simulates the remaining warps and predicts their execution time by accumulating the predicted time of their basic blocks.

The online analysis provides the distribution of basic blocks in advance so that basic-block-sampling does not require waiting for rare basic blocks that do not affect GPU workload performance significantly, to finish. For example, SpMV, at the end of its execution, stores its results into the target vector, and that task is finished by one basic block executed at the end after all warp execution. However, this basic block does not substantially affect the runtime of SpMV warps since it only accounts for about 0.6% of all the basic blocks executed by the SpMV kernel in our experiments. Moreover, we analyze a sample of warps to ensure that online analysis will not introduce a large overhead. Figure 8 shows the distribution of basic blocks for both all warps and a sample of warps. The percentage is calculated by the total instruction count of these basic blocks to the instructions of whole kernels. We can see that no matter whether the applications are irregular or regular, online analysis of a sample of warps' basic block distribution is enough.

We then use online detection to determine whether the execution of each type of basic block is stable. Photon utilizes the least-squares method to detect their stable status. The least-squares method [9] is used to find the best-fit line or curve that passes through a set of data points by minimizing the sum of the squared residuals between the observed data and the predicted values of the model. Assume that we have a set of data $\{(x_0,y_0),...,(x_n,y_n)\}$, we calculate the best-fit line, $y = a \cdot x + b$, using Equation 1.

$$\begin{aligned} a &= \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2} \\ b &= \frac{1}{n}\sum_{i=1}^{n} y_i - a\frac{1}{n}\sum_{i=1}^{n} x_i \end{aligned} \tag{1}$$

As we can see from Figure 3a and Figure 3b, the least-squares lines for the issue time and retired time of the most frequent basic blocks are $Retired\ time = 1.00 \cdot Issue\ time + 4.23$, and $Retired\ time = 0.99 \cdot Issue\ Time + 4.96$, respectively. We observe that the slope value a calculated by the least-squares method is always close to one for basic blocks no matter which benchmarks we evaluate. We investigate this observation and conclude that this is because the execution time of basic blocks is not related to its issue time since the memory competition and others are stable for GPU systems. In Figure 3, we can find that at the beginning of GPU workloads, the a of the least square line for dots ($x = issue\ time$, $y = retired\ time$) is not close to one. And then the a of these dots is close to one after competition among warps becomes stable.

Photon calculates the rolling slope value a of the least-squares method for the last n basic blocks from the same type (we use 2048 across all GPU workloads), in order to leverage the regular pattern in **Observation 3**. If the value of $|1 - a|$ is less than the threshold δ (we use 3%), this type of basic block is considered to be stable. Moreover, we find that some applications might trigger a false stable signal as a falls into a local optimum [33]. We propose to solve this by also checking the average number of executions with the last $2n$ basic blocks and ensuring that their relative difference is also less than the threshold δ.

Rare Basic Blocks: Rare basic blocks are basic blocks that are rarely triggered during execution. Basic-block-sampling requires predicting the execution of these basic blocks but hard to collect their data. Rare basic blocks occur in several cases. One example is that basic blocks in some benchmarks to handle special cases. For example, one basic block (PC=508, number of instructions=1) for the benchmark SC is the last instruction of one warp and it is triggered as this warp is an empty task. Another case is that the final basic block for handling final results write or others for large kernel applications, like SpMV.

We process rare basic blocks via a performance modeling method based on interval analysis, as shown in Figure 9. Firstly, we collect

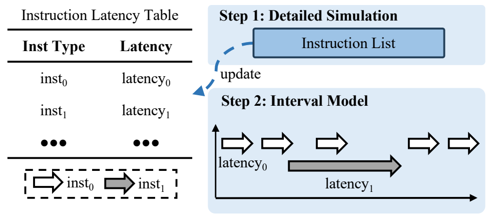


Figure 9: Photon utilizes the interval model and online instruction latency table generated during detailed simulation to predict the runtime of rare basic blocks.

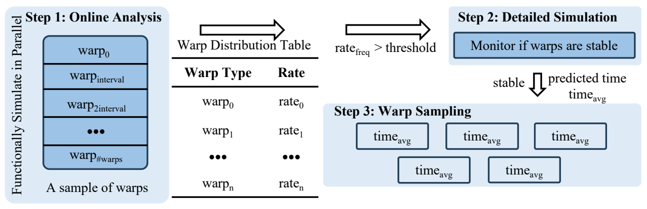


Figure 10: Photon online analyzes the distribution of warp distributions via parallel functional simulating a sample of warps (Step 1). Warp-sampling can be enabled only when the rate $rate_{freq}$ of the most frequent warps $warp_{freq}$ is larger than the threshold. Then Photon detailed simulates GPU workloads (Step 2). Finally, Photon switches to warp-sampling if warps are stable and predicts the execution time of following warps by the average execution time of the last n warps (Step 3).

the latency for each type of instruction during the detailed simulation so that Photon is not required to consider the multi-warp model like GPUMech [28]. During the basic block sampling period, Photon predicts the runtime of rare basic blocks via the interval model. Photon models the issue and retired time of instructions and when the dependency happens, postpones the issue time of that instruction to be after the retire time of the dependent instruction. For these rare instructions that we never collect, we set their initial value according to the latency of caches and ALUs.

4.2 Sampling Warps

Warps represent the basic unit of computation in GPU kernels that are scheduled to compute units during execution. Different types of warps have different BBVs. Figure 10 shows the workflow of warp-sampling. Photon first collects the distribution for each type of warp. If there is one warp dominating the GPU kernel (this type of warp accounts for more than 95% for all warps), we start to detect if the execution of the dominant warps is stable during detailed simulation. Once this warp is stable, Photon switches to warp-sampling and only simulates the scheduler.

Photon firstly completes online analysis of the distribution of each warp. We derive this by analyzing a sample of warps to ensure both accuracy and performance. Note that the online analysis overhead of warp sampling is similar to the online analysis for both basic-block- and warp-sampling as most of the online analysis time

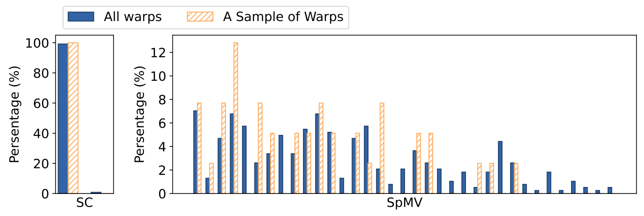


Figure 11: The distribution of different warp types for a regular application (SC) and an irregular application (SpMV) of all warps and a sample of warps.

is spent on functional simulation. Figure 11 illustrates the distribution for each type of warp of regular and irregular applications for both all warps and a sample of warps. For regular applications, like SC, the distribution of warps for all warps and a sample of warps is similar so that we can use a subset of warps (we use 1% warps) to decide if warp-sampling should be enabled. For irregular applications, like SpMV, both a sample of warps and all warps have no dominant warp, and therefore, we can also disable warp-sampling by analyzing a subset of warps.

The relationship between the issue and retired time of warps for regular applications, like MM, has the same pattern as seen at the basic block level, as shown in Figure 4a. Regular applications exhibit a predominant warp type, resulting in their warps being primarily influenced by the microarchitecture status. As a result, the a of regular applications approaches one, which means that the execution time of the warps in that type is stable. However, in the case of irregular applications, the execution of warps is influenced by various factors, such as the instructions executed and memory access patterns. Consequently, the value of a for these applications tends to deviate significantly from one. For example, the value for a of the least squares fitted value for SpMV, shown in Figure 4b, deviate significantly from one, so the methodology will not enable sampling this irregular application on the warp level.

Photon starts the detailed simulation and collects the issue and retired time for all warps for applications with a dominant warp type based on **Observation 4**. Once Photon detects the a of the least square method on the last n warps (we set n=1024) is close to 1 ($|a - 1| < \delta$), Photon will then switch to warp-sampling as shown in Figure 10, **Step 3**. By doing so, warp-sampling is automatically disabled for irregular applications as they do not have a dominant warp type and the value of a deviates significantly from 1. However, basic-block-sampling (Section 4.1) and kernel-sampling (Section 4.3) can still be applied for these irregular applications. Similar to basic-block-sampling, we also check the relative difference for the average execution time of n and 2n warps to avoid issue of finding a local optimum. Photon only simulates the scheduler and all execution time of warps is predicted as the average time of the last n warps during warp-sampling.

4.3 Sampling Kernels

GPU kernels are the basic units of execution that are scheduled to execute on the GPU. Instead of depending on offline analysis to

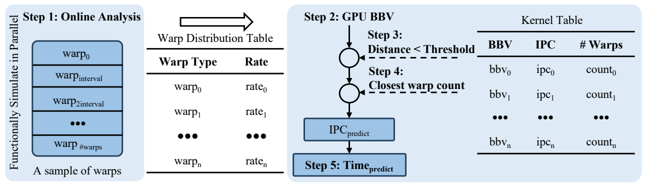


Figure 12: Photon online generates GPU BBV via parallel functional simulating a sample of warps (Step 1). Then Photon finds all prior kernels whose distance is less than the threshold (Step 2). Furthermore, Photon selects kernels whose warp count is closest (Step 3). Finally, Photon predicts the execution time of the kernel via the predicted IPC and instruction count (Step 4). Note that the number of warps must be equal for kernels whose warp count is less than GPU cores.

decide the similarity of GPU kernels, we provide an online kernel-sampling method based on **Observation 5**.

The workflow of Photon for kernel-sampling is shown in Figure 12. We first collect the distribution and BBVs for each type of warp, which is the same as sampling-warps. If both warp-sampling and kernel-sampling are enabled, we can reuse the analysis data of warp-sampling. Then we generate online GPU BBV as shown in Figure 5. As shown in Figure 6, GPU-BBVs generated from different types of kernels can be clustered into different groups attributed to the higher variance of warps' type and distribution.

Furthermore, we select all prior kernels that have similar GPU BBVs. We select the kernel that has the closest number of warps since kernels with a similar number of warps usually have similar IPC. Noted that we notice that for kernels with warps that are less than the number of GPU cores, their IPC is different compared to kernels with a large number of warps. That is due to these kernels having less competition for resources and also less parallelism. Kernel-sampling requires the warp number to be the same for sampling accuracy when the number of warps of the kernels is less than the number of GPU cores.

Finally, we predict the execution time of one kernel K via the predicted instruction count and IPC via the prior GPU kernel, K'. The instruction count of the entire kernel can be predicted by $\#insts = \frac{\#insts^{K'} \cdot \#insts_{sample}}{\#insts^{K'}_{sample}}$, where $\#insts$ and $\#insts^{K'}$ are the total number of instructions executed for K and K', and $\#insts^{K'}_{sample}$ and $\#insts_{sample}$ are the number of instructions executed of a sample of warps during online analysis for K and K', respectively. The IPC of K is predicted to be the same as K'.

5 EXPERIMENTAL SETUP

This section describes the experimental setup to evaluate Photon. We first provide the details of the simulation framework and micro-architectures that Photon validates. Then we provide the benchmarks we used to validate the accuracy and performance of Photon.

Simulation Configuration. We modify MGPUSim [52] to support switching between the detailed mode and fast-forward mode simulation and implement Photon and PKA inside the simulator.

Table 1: The configuration parameters used for R9 Nano and MI100 GPU on MGPUSim to evaluate Photon.

Component	R9 Nano	MI100 [3]
CU	1.0GHz, 64 per GPU	1.0GHz, 120 per GPU
L1 Vector Cache	16KB 4-way 64 per GPU	16KB 4-way 120 per GPU
L1 Inst Cache	32KB 4-way 16 per GPU	32KB 4-way 30 per GPU
L1 Scalar Cache	16KB 4-way 16 per GPU	16KB 4-way 30 per GPU
L2 Cache	256KB 16-way 8 per GPU	8MB 16-way 32 per GPU
DRAM	4GB	32GB

We choose MGPUSim as it is an open-source GPU simulator and provides a range of tools to monitor GPU workload behavior. We validate Photon on two different architectures, R9 Nano and MI100, as shown in Table 1. We use the default scheduling algorithm provided by MGPUSim. R9 Nano and MI100 are both CDNA architectures that are for computing. We do not use RDNA because RDNA GPUs are mainly for gaming and 3D rendering and CDNA is the default architecture for MGPUSim. The MI100 is already the newest CDNA-architecture-based GPU that is commercially available today. The MI200 and MI300 have been announced but cannot yet be purchased at this time. We evaluate all experiments on the platform using the Intel® Xeon® Gold 6132 CPU. The operating system used is Ubuntu 20.04.4.

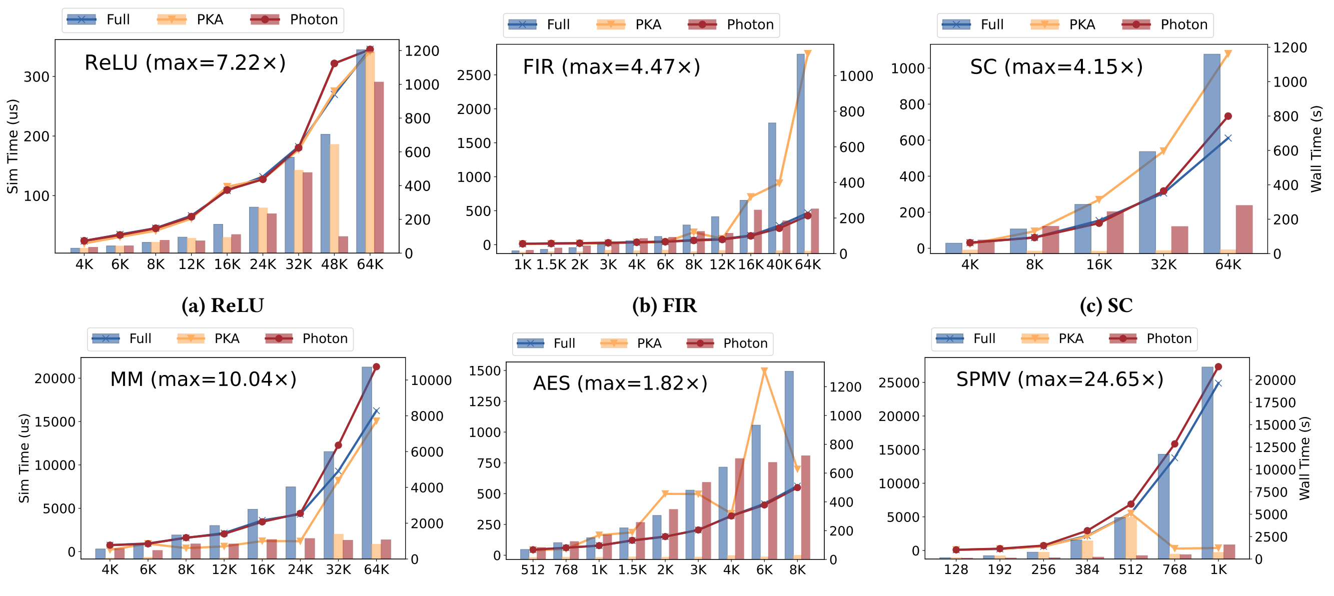
GPU workloads. We evaluate Photon on both regular and irregular applications including simple GPU kernels, like ReLU, as well as larger, real-world applications, like VGG-16, as shown in Table 2. All kernels are written in OpenCL and then compiled by the AMD ROCm compiler. The compiler is set to use the default options. To fully validate each benchmark, we run all benchmarks using various problem sizes, which are defined by the number of warps for GPU workloads. Each benchmark is executed over 10 runs, and the kernel execution kernel and wall time of simulating are averaged.

Accuracy. We select kernel execution time (Sim Time) instead of the IPC of TBPoint [29] and PKA [4] as the kernel execution time is the most important feature that GPU users care about when executing workloads. We validate the absolute error rate of sampled simulation by comparing the execution time of GPU workloads with full detailed mode via $\frac{|T_{full} - T_{sampled}|}{T_{full}} \times 100\%$, where T_{full} and $T_{sampled}$ are the GPU workloads execution time of full detailed mode and sampled methods, respectively.

Performance. We utilize the wall time (Wall Time) of finishing simulation for GPU workloads to evaluate the performance of Photon and other methods. Another metric that we used is speedup, which is calculated by $Speedup = \frac{Wall\ Time_{full}}{Wall\ Time_{sampled}}$, where $Wall\ Time_{full}$ and $Wall\ Time_{sampled}$ are the wall time of completing simulating GPU workloads using full detailed mode and sampled methods, respectively.

6 EVALUATION

In this section, we present a comprehensive evaluation of Photon. We first present the performance and accuracy of Photon by comparing it with the state-of-the-art work, PKA [4] on different problem sizes for single kernel GPU workloads. Then we evaluate Photon using both the R9 Nano and MI100 configurations to show



(a) ReLU (b) FIR (c) SC

(d) MM (e) AES (f) SPMV

Figure 13: The kernel execution time (left y-axis with lines) and wall time (right y-axis with bars) for Full detailed MGPUSim, PKA, and Photon. The numbers in the parentheses represent the maximum speedup among all problem sizes for each application.

Table 2: The benchmarks used to evaluate Photon.

Abbr.	Suite	Workload Description
AES	Hetero-Mark [53]	AES-256 Encryption
FIR	Hetero-Mark	FIR filter
SC	AMD APP SDK [2]	Simple Convolution
MM	AMD APP SDK	Matrix Multiplication
ReLU	DNNMark [18]	Rectified Linear Unit
SPMV	SHOC [15]	Sparse Matrix-Vector Multiplication
PR-X	Hetero-Mark	PageRank with X nodes.
VGG [50]		VGG-16 and VGG-19; batchsize=1
ResNet [26]		ResNet-18 (34, 50, 101, 152); batchsize=1

that Photon is robust to microarchitecture changes. Thereafter, we independently evaluate the effect of different levels of sampling in Photon both independently as well as together as a complete simulator. Finally, we evaluate Photon on the larger, real-world applications.

6.1 Overall Effectiveness

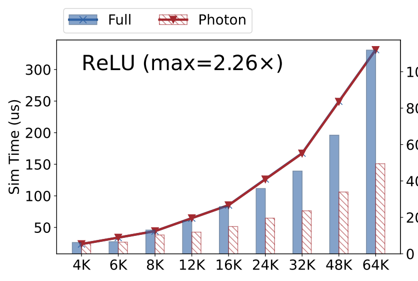
In this section, we compare Photon with the prior work, PKA, using MGPUSim modeling the R9 Nano. Figure 13 illustrates the results of MGPUSim configured to use full detailed mode, PKA and Photon, respectively. Figure 13 demonstrates that Photon achieves up to 24.65× speedup (average speedup 1.87×) with an average simulation error of 6.83%, while PKA results shows either a significant sampling error or low performance.

We implement PKA in MGPUSim by referencing the PKA source code and we use the PKA default parameters: $s = 0.25$ where s is the threshold of the IPC variance over the last 3000 cycles. Sampling simulation speedup is related not only to applications but their inputs, implementation, underlying architecture, and underlying simulator. It is therefore expected that different speedups can be reported with PKA in another environment. Although our PKA implementation shows a lower sampling speedup for SpMV, our results also show that PKA achieves a 13× speedup for Matrix Multiplication, while PKA [4] only reports a speedup of 1.3×.

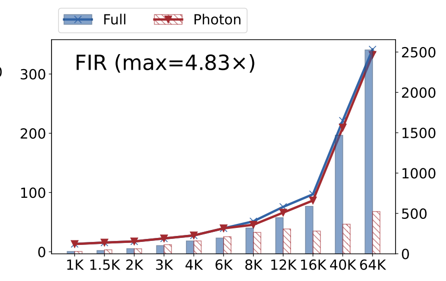
Small kernel GPU workloads. Small kernel GPU workloads, including FIR and ReLU, have a large number of warps but their warps only contain tens to hundreds of instructions each. Figure 13a and Figure 13b show that both Photon and PKA archive low error rates and high performance. PKA, especially, outperforms Photon on FIR benchmarks for problem size including 3K, and 6K. However, Photon maintains both low sampling errors and high performance when the problem size is larger than 16K, while PKA is unable to. This is because that Photon utilizes basic blocks and warps as basic units so that its basic granularity is larger than PKA, which allows Photon to have lower performance than PKA with better accuracy.

Complex GPU workloads. Complex GPU workloads, including SC and MM, are other common GPU kernels. Their number of warp and instructions per warp are both large. As shown in Figure 13c and Figure 13d, Photon ensures low sampling errors for all problem sizes, whereas PKA exhibits significantly high sampling errors for some problem sizes in SC and MM. For problem sizes 32K and 64K of MM, Photon achieves a higher error rate as we still have local optimum issues for some special cases.

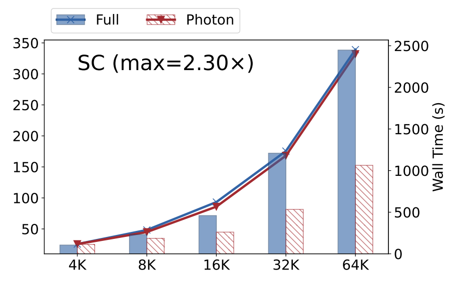
Additionally, AES represents another set of complex GPU workloads. Unlike SC and MM which have large instruction counts due to loops, AES has a long instruction sequence, about 400 instructions, to process different steps of its algorithm. PKA fails to ensure the sampling accuracy for some problem sizes like 2K and 6K as



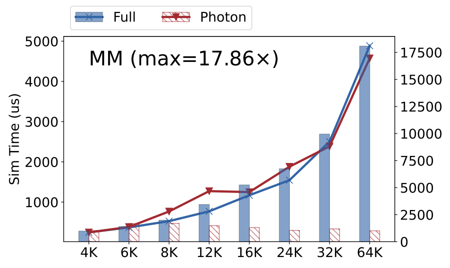
(a) ReLU



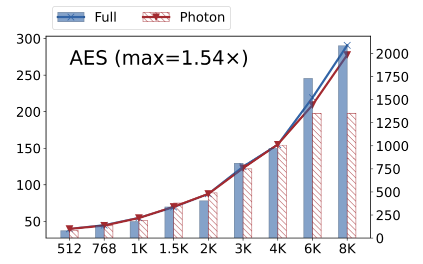
(b) FIR



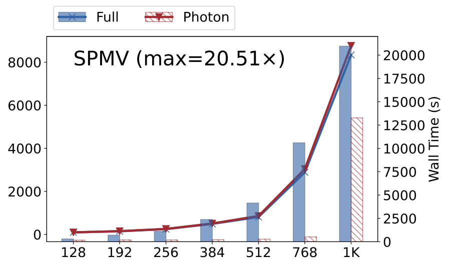
(c) SC



(d) Matrix Multiplication



(e) AES



(f) SPMV

Figure 14: The kernel execution time (left y-axis with lines) and wall time (right y-axis with bars) for Full detailed MGPUSim, Photon for MI100. The numbers in the parentheses represent the maximum speedup among all problem sizes for each application.

it does not collect all instructions inside the kernel, while Photon achieves very low sampling errors for all problem sizes.

Irregular GPU workloads. Irregular applications are a set of applications whose kernels contain different types of warps. These applications are difficult for PKA to handle due to their fluctuating IPC. As shown in Figure 13f, PKA has either low performance or high error rates for all problem sizes, while Photon achieves up to 24.65× speedup with an average sampling error 18.9% due to its granularity in basic blocks instead of cycles.

Micro-architecture Independent. We evaluate all benchmarks with the same problem sizes on MI100 and the results indicate that Photon is a micro-architecture independent methodology. Figure 14 shows the kernel runtime and wall time of Photon compared to full detailed simulation. For all benchmarks and problem sizes, Photon achieves similar performance and low sampling errors, as sampling R9 Nano. The benchmark MM with the 12K problem size shows a slightly higher error rate due to the local optimum issue.

We conclude that Photon outperforms PKA and other prior works due to these methods mostly focusing on kernel-samping simulation. While PKA's assumption of stable IPC may hold true for some GPU workloads, it may not be an accurate predictor of performance for all types of workloads.

6.2 Effects of Different Sampling Levels

Figure 15 shows the performance of each sampling level and their combination for Photon. The average error rates for basic-block-sampling, warp-sampling and Photon are 9.70%, 1.75%, and 6.83%, respectively.

Small kernel GPU workloads. Small kernels, like ReLU and FIR, contain tens or fewer basic blocks. For example, ReLu only has two basic blocks so the threshold of basic-block sampling is easier to satisfy. Figure 15a and Figure 15e show that basic-block-sampling contributes most performance improvement for Photon.

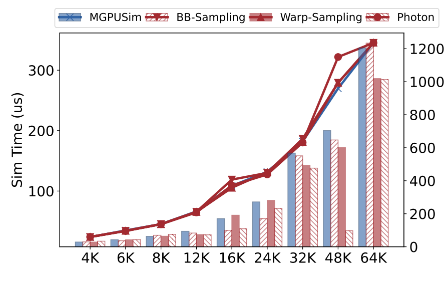
Complex kernel GPU workloads. For complex kernels, including SC and MM, whose main instructions come from loops, basic-block-sampling, and warp-sampling simulation work well individually, and their combination provides an even larger speedup. For AES, most of the performance improvement comes from warp-sampling as this kernel contains a long instruction sequence, which allows warps of AES to stabilize more easily than when basic-block-sampling.

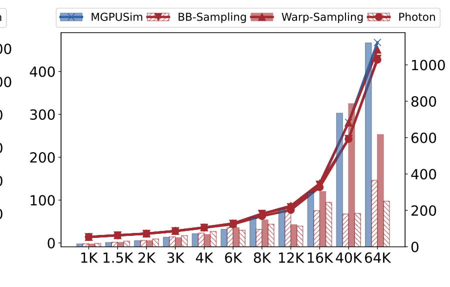
Irregular GPU workloads. For irregular applications, basic-block-sampling contributes to the vast majority of the performance improvement. We notice that for some problem sizes, like a warp count of 384, basic-block-sampling is worse than Photon and we determined that it is due to the variance across different executions. MGPUSim dynamically schedules warps into GPU cores and the memory access order also causes irregular applications to become more variable.

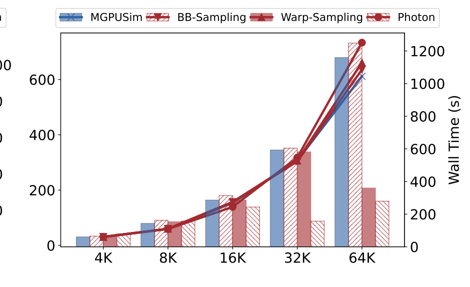
We conclude that basic-block-sampling, warp-sampling, and the combination of both provide both accuracy and performance for a variety of different types of GPU kernels. We can not depend on one single method to solve all GPU sampling simulation workloads.

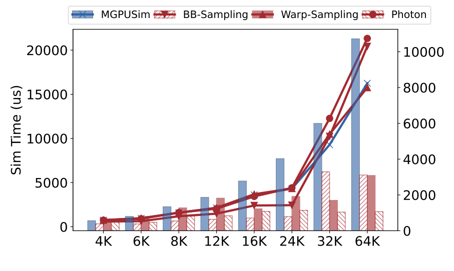
6.3 Real-world Applications

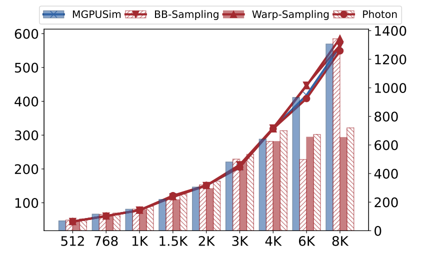
In this section, we present larger, real-world GPU workloads, as shown in Figure 16, to help demonstrate that Photon yields both


(a) ReLU


(b) FIR


(c) SC


(d) MM


(e) AES

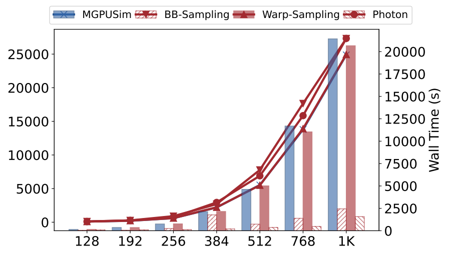

(f) SPMV

Figure 15: The kernel execution time (left y-axis with lines) and wall time (right y-axis with bars) for full detailed MGPUSim, basic-block-sampling (BB-sampling), warp-sampling, and Photon.

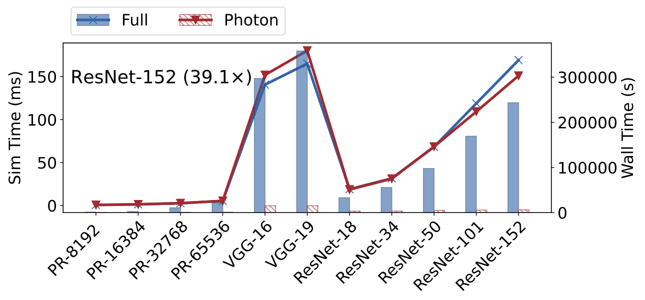
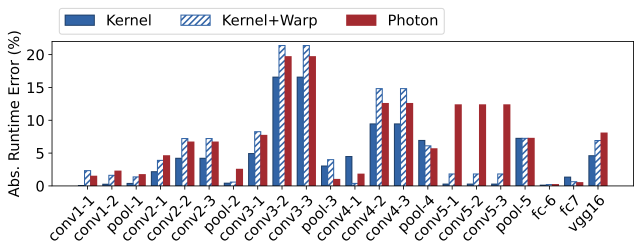


Figure 16: The kernel execution time (left y-axis with lines) and wall time (right y-axis with bars) for full detailed MG-PUSim, and Photon. The numbers in the parentheses represent the maximum speedup for all real-world applications. The average sampling accuracy for all applications is 4.3%.


(a) Absolute runtime error

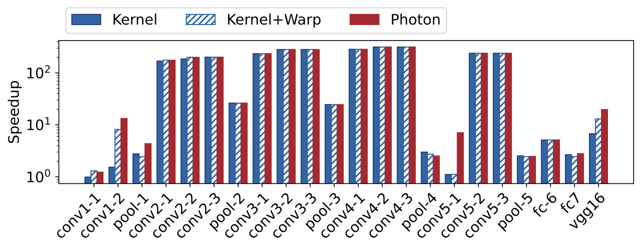

(b) Speedup

Figure 17: The absolute runtime error and speedup of kernel-sampling, kernel+warp-sampling, and Photon for each layer and whole inference runtime of VGG-16 with batch size 1. Noted that each layer contains multiple GPU kernels.

high accuracy and performance for a wide range of applications. We also present the detailed results of VGG-16[1] in Figure 17 to evaluate the kernel-sampling, kernel+warp sampling, and Photon, which enables kernel-, warp- and basic-block-sampling. Note that prior works provide solutions for kernel-sampling but their methods depend on pre-processing. Photon utilizes online-analysis and we include this analysis time into the runtime of Photon. Figure 16 shows that sampling GPU workloads without up-front analysis still achieves up to a 39.1× speedup with a low average sampling error rate, 4.3%.

[1]Only kernel-level sampling is enabled for fc-6 because of a bug in online analysis.

Accuracy. Figure 17a shows the sampling error rate for each layer of VGG-16 and its full runtime. The result shows that kernel-sampling is the most accurate sampling method. The online GPU BBVs can capture the behavior of GPU kernels to achieve sampling accuracy. Adding basic-block- and warp-sampling increases the sampling error rate from 4.60% to 8.05%. This error is mainly from conv5-1, 5-2, and 5-3 as we select larger execution time basic blocks to predict the kernel runtime.

Speedup. The speedup numbers for the whole VGG-16 using kernel-sampling, kernel+warp-sampling and Photon are 6.76, 13.08 and 19.71×, respectively. Warp-sampling improves performance by about 1.9× over kernel-sampling, and basic-block-sampling continues to increase the sampling performance by 1.5× over the prior two sampling methods. The result illustrates that basic-block-, warp- and kernel-sampling can work together to improve the performance.

Online/Offline Tradeoff. Although we consider having a high-speed online profiling solution that allows for a fast turn-around time between new application updates and simulation results, Photon also supports offline analysis to further improving sampling simulation performance. All data that is generated by the online analysis of kernel-, basic-block- and warp-sampling sampling, is micro-architecture agnostic. Hardware researchers can reuse these data to enable sampled simulation of GPU workloads. For example, our results show that offline Photon reduces the sampling simulation time of VGG-16 from 4.19 hours to 3.76 hours, compared with the original, online version of Photon.

7 CONCLUSION

In this paper, we propose a new sampled simulation methodology, Photon, that can optimize sampling based on application behavior across the basic block, warp, and kernel levels. The evaluation of Photon shows that basic-block-sampling, warp-sampling, and kernel-sampling (both independently, as well as together in Photon) can successfully simulate a variety of GPU workloads. Finally, we evaluate Photon on a wide range of real-world applications to demonstrate its high sampling accuracy and performance. For example, Photon reduces the simulation time needed to perform one inference of ResNet-152 with batch size 1 from 7.05 days to just 1.7 hours, with a very low sampling error of 10.7%; previous works [4] tend to show higher errors for large real-world applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and the AEC for evaluating our artifacts. We also like to thank Alen Sabu for helping us review the paper and providing valuable feedback. We also thank AMD, Intel, and VMware for supporting this research.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides all of the information that is required to use the Photon methodology. We provide a simulator that can run both detailed and sampled simulation. We also provide scripts to help simulate and collect all of the results presented in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm: Photon**
- **Program: Go, Python**
- **Compilation: Go**
- **Run-time environment: Docker**
- **Hardware: Machine with at least 32 cores.**
- **Metrics: Wall time and simulation time**
- **Output: Plain text, tables, figures**
- **Experiments: Run benchmarks with the simulator and collect data**
- **How much disk space required : ≈8GB**
- **How much time is needed to complete experiments ?: ≈ 24 hours**
- **Publicly available?: Yes**
- **Workflow framework used?: MGPUSim**

A.3 Description

We provide a Dockerfile to help users easily set up the experimental environment. Within the Dockerfile, we set up the Go and Python environments, as well as all necessary environment variables (PATH, etc.). Note that MGPUSim is a multi-threaded simulator and our Python script maintains a thread pool (set to 8) to help evaluate multiple benchmarks at the same time. We suggest evaluating `N` benchmarks at the same time, where `N` is the $\frac{1}{4}$ of the hardware core count. The default output data is in JSON format and stored in the `${HOME}/gpudata` directory. The Python script also includes a `-check` option to read the results and display the results.

A.3.1 How to access. The Dockerfile and its related scripts are included at Zenodo [42, 43]. We provide the Dockerfile to automatically download all software and set up the execution environment.

A.3.2 Hardware dependencies. We suggest using a server with at least 32 cores and 128GB memory to evaluate the results of Photon.

A.3.3 Software dependencies. The Internet is required for MGPUSim to download all dependencies.

A.3.4 Data sets. The data sets used in this work are included with Photon.

A.4 Installation

Once the Dockerfile and its related scripts are downloaded, build the Docker environment via the `make_docker.sh` script.

```
$ ./make_docker.sh
```

This script will pull all code that we modified to build Photon. All the path variables are set via this script.

Once the setup is done, use the `run_docker.sh` to run the container that is generated via the previous script `make_docker.sh`.

```
$ ./run_docker.sh
```

A.5 Evaluation and expected results

Single kernel applications. The Python script `testallbench.py` inside the directory `sampledrunner` of the `sampled-mgpu-sim` repository is can be used to generate all results in the paper for single GPU kernels (Figure 13, Figure 14 and Figure 15). Note that the first time this is run, it will automatically download and install the go dependencies for Photon, and compile the benchmark binaries for evaluation.

For example, to generate the data of Figure 13 and Figure 15, run the command:

```
$ ./testallbench.py
```

Once the execution finishes, use the command:

```
$ ./testallbench.py -check
```

to collect results and export the result file `r9nano.xlsx` into the `micro2023_figures` repository for further steps to generate Figure 13 and Figure 15.

To generate the data and export the result file `mi100.xlsx` for Figure 14, add the parameter `-arch=mi100`. The `-n` option is used to set how many applications to evaluate at the same time. And `-h` gives help information.

Real-world application. For the results of real-world applications in Figure 16 and Figure 17, we provide scripts for deep learning applications (`testdlapps.py`) and PageRank (`testpagerank.py`). For example, to run VGG-16, use the command `./testdlapps.py -bench=vgg16` and then utilize `./testdlapps.py -bench=vgg16 -check` to generate the results of VGG-16 and export the result file `vgg16.xlsx`. Other applications are similar.

Results. We provide both the scripts and the data we used to create all the result figures within the `micro2023_figures` repository. Prior steps will export files of new data and overwrite the original files inside the `micro2023_figures` repository. To generate figures for the new data, simply run scripts after overwriting files via prior steps. To generate Figure 13, Figure 14 and Figure 15, execute the python scripts `r9nano.py`, `mi100.py` and `r9nanolevels.py`, respectively. Running `vgg16.py` and `vgg16speedup.py` will automatically generate Figure 17. Generating Figure 16 is different with others. New data should be manually copied into the file `all.xlsx` for each application before running `all.py`.

A.6 MLCommons

We have added support to evaluate Photon using the MLCommons CM automation language [20]. Check the README inside the `mlcommons` directory for details.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] AMD. 2017. AMD APP SDK 3.0 Getting Started.

[3] AMD. 2020. AMD Instinct MI100 Accelerators. www.AMD.com/InstinctMI100.

[4] Cesar Avalos Baddouh, Mahmoud Khairy, Roland N Green, Mathias Payer, and Timothy G Rogers. 2021. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 724–737.

[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.

[6] Aaron Barnes, Fangjia Shen, and Timothy G Rogers. 2023. Mitigating GPU Core Partitioning Performance Effects. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 530–542.

[7] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–609.

[8] Bradford M Beckmann and Anthony Gutierrez. 2015. The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*.

[9] Åke Björck. 1990. Least squares methods. *Handbook of numerical analysis* 1 (1990), 465–652.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout. 2013. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

[11] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. 2014. BarrierPoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

[12] Cen Chen, Kenli Li, Aijia Ouyang, Zhuo Tang, and Keqin Li. 2017. Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 10 (2017), 2740–2753.

[13] Guoyang Chen, Yufei Ding, and Xipeng Shen. 2017. Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity. In *IEEE 33rd International Conference on Data Engineering (ICDE)*. 621–632.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.

[15] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the General Purpose GPUs (GPGPU)*. 63–74.

[16] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu, and Hamid Sarbazi-Azad. 2022. Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 228–244.

[17] Sambit Das, Phani Motamarri, Vishal Subramanian, David M Rogers, and Vikram Gavini. 2022. DFT-FE 1.0: A massively parallel hybrid CPU-GPU density functional theory code using finite-element discretization. *Computer Physics Communications* 280 (2022), 108473.

[18] Shi Dong and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of the General Purpose GPUs (GPGPU)*. 63–72.

[19] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391–407.

[20] Grigori Fursin. 2023. Toward a common language to facilitate reproducible research and technology transfer: challenges and solutions. https://doi.org/10.5281/zenodo.8105339

[21] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung Luk. 2018. Performance characterisation and simulation of Intel's integrated GPU architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 139–148.

[22] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. 2016. TaskPoint: Sampled simulation of task-based programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 296–306.

[23] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy G. Rogers. 2018. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 608–619.

[24] Dongho Ha, Yunho Oh, and Won Woo Ro. 2023. R2D2: Removing ReDunDancy Utilizing Linearity of Address Generation in GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. 1–14.

[25] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[27] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. 152–163.

[28] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S Lee. 2014. GPUMech: GPU performance modeling technique based on interval analysis. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 268–279.

[29] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S Lee. 2014. TB-Point: Reducing simulation time for large-scale GPGPU kernels. In *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*. 437–446.

[30] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. 2015. Scaling up MapReduce-based big data processing on multi-GPU systems. *Cluster Computing* 18 (2015), 369–383.

[31] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A Kim. 2015. Fast computational gpu design with gt-pin. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*. 76–86.

[32] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486.

[33] Joshua D Knowles, Richard A Watson, and David W Corne. 2001. Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary Multi-Criterion Optimization: First International Conference (EMO)*. 269–283.

[34] Boris Krasnopolsky and Alexey Medvedev. 2016. Acceleration of large scale OpenFOAM simulations on distributed systems with multicore CPUs and GPUs. In *Parallel Computing: On the Road to Exascale*. 93–102.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[36] Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. 2022. GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. 424–436.

[37] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 49–63.

[38] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*. 339–350.

[39] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. 2023. FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 516–529.

[40] Mahmood Naderan-Tahan, Hossein SeyyedAghaei, and Lieven Eeckhout. 2023. Sieve: Stratified GPU-Compute Workload Sampling. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 224–234.

[41] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. https://www.nvidia.com/en-us/data-center/h100/.

[42] Photon 2023. MICRO - Artifact - Photon Github Repo. https://github.com/nus-comparch/photon.git.

[43] Photon 2023. *MICRO - Artifact - Photon Zenodo Repo.* https://doi.org/10.5281/zenodo.8357867

[44] Fritz Previlon, Charu Kalra, Devesh Tiwari, and David Kaeli. 2020. Characterizing and exploiting soft error vulnerability phase behavior in gpu applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 19, 1 (2020), 288–300.

[45] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. 2022. LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[46] Jason Sanders and Edward Kandrot. 2011. *CUDA by example: an introduction to general purpose GPU programming.*

[47] Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 3–14.

[48] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.

[49] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *30th International Symposium on Computer Architecture (ISCA))*. 336–349.

[50] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations (ICLR)*.

[51] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2017. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 161–174.

[52] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David R. Kaeli. 2019. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 197–209.

[53] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *IEEE International Symposium on Workload Characterization (IISWC)*. 1–10.

[54] Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli. 2019. Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-GPU systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 664–674.

[55] TOP500. 2022. TOP500 Supercomputer Sites. https://www.top500.org/. Accessed on November 16, 2022.

[56] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 335–344.

[57] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 868–880.

[58] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S Vetter. 2013. Exploring hybrid memory for GPU energy efficiency through software-hardware co-design. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 93–102.

[59] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[60] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *International Symposium on Computer Architecture (ISCA)*. 84–97.

[61] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy John, Hai Jin, and Chengzhong Xu. 2013. Accelerating GPGPU architecture simulation. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 331–332.

[62] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy K John, Hai Jin, Chengzhong Xu, and Junmin Wu. 2015. GPGPU-MiniBench: accelerating GPGPU micro-architecture simulation. *IEEE Trans. Comput.* 64, 11 (2015), 3153–3166.

[63] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. SAC: Sharing-Aware Caching in Multi-Chip GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. 1–13.