Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads

Cesar A. Baddouh
Purdue University

Mahmoud Khairy
Purdue University

Roland Green*
Cerebras

Mathias Payer
EPFL

Timothy G. Rogers
Purdue University

ABSTRACT

Simulating all threads in a scaled GPU workload results in prohibitive simulation cost. Cycle-level simulation is orders of magnitude slower than native silicon, the only solution is to reduce the amount of work simulated while accurately representing the program.

Existing solutions to simulate GPU programs either scale the input size, simulate the first several billion instructions, or simulate a portion of both the GPU and the workload. These solutions lack validation against scaled systems, produce unrealistic contention conditions and frequently miss critical code sections. Existing CPU sampling mechanisms, like SimPoint, reduce per-thread workload, and are ill-suited to GPU programs where reducing the number of threads is critical. Sampling solutions on GPUs space lack silicon validation, require per-workload parameter tuning, and do not scale.

A tractable solution, validated on contemporary scaled workloads, is needed to provide credible simulation results. By studying scaled workloads with centuries-long simulation times, we uncover practical and algorithmic limitations of existing solutions and propose Principal Kernel Analysis: a hierarchical program sampling methodology that concisely represents GPU programs by selecting representative kernel portions using a scalable profiling methodology, tractable clustering algorithm and detection of intra-kernel IPC stability. We validate Principal Kernel Analysis across 147 workloads and three GPU generations using the Accel-Sim simulator, demonstrating a better performance/error tradeoff than prior work and that century-long MLPerf simulations are reduced to hours with an average cycle error of 27% versus silicon.

CCS CONCEPTS

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

KEYWORDS

GPU, Workload sampling, Simulation methodology

*Work was completed during his time at Purdue University.

ACM Reference Format:
Cesar A. Baddouh, Mahmoud Khairy, Roland Green, Mathias Payer, and Timothy G. Rogers. 2021. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480100

1 INTRODUCTION

There is no real ending. It's just the place where you stop the story.

Frank Herbert

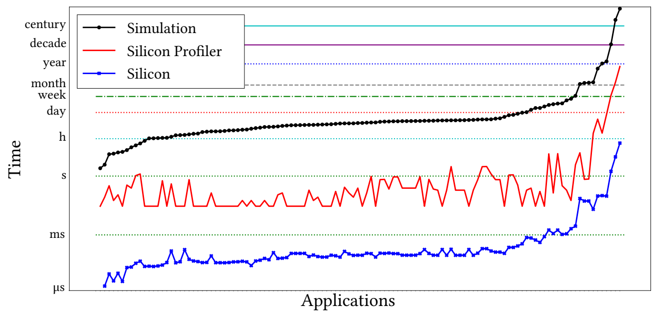


Figure 1: Projected hours to simulate, profile 12 statistics (Table 2) in-silicon [41] and raw execution time of the 147 workloads we study (Section 4) on a Volta V100. Projected simulation times are based on Accel-Sim's [32] simulation rate.

Simulators carry out a cycle-level introspection of GPU workloads. These simulators are highly configurable and enable different analyses such as (i) debugging of workloads on specific simulated hardware to detect implementation errors [11], (ii) profiling of workloads to analyze performance bottlenecks [1, 8], or (iii) reconfiguration of the simulated hardware to analyze model changes [2]. Many such use cases are impossible with silicon profiling [41].

The flexibility and introspection capabilities of simulation come at a price. Simulating complex hardware like modern GPUs incurs orders of magnitude more time for each simulated instruction. What takes seconds on a GPU would take millennia on a simulator. Due to the overhead in simulation, existing simulators cannot reasonably execute real GPU programs. Even high-performance industrial simulators used by GPU companies are still not fast enough to simulate applications that take seconds to minutes on contemporary GPUs [61]. *Any simulation platform must therefore restrict the number of executed instructions.*

Existing approaches restrict either the workload or the simulated platform, introducing various limitations. Common approaches include: (i) scaling workload inputs [14, 23, 56], which reduces the applicability of the simulation results due to the extremely short runtimes (ignoring scaling effects), (ii) simulating the first several billion instructions of a scaled workload [10, 32, 60], which restricts the insights to a limited horizon of simulation (often limiting measurements to the warmup phase), and (iii) reducing the size of the GPU simulated [29, 42, 49], which forces the workload to adapt to different hardware. Each of these methodologies has limitations, and there is no public work validating them against the scaled workloads and systems they aspire to represent. Furthermore, no prior work has attempted to simulate or validate representative large-scale workloads from MLPerf [36, 48] to completion.

To demonstrate the extent of workload realism and simulation slowdown, Figure 1 plots the silicon execution time and projected simulation time for 147 workloads from both contemporary GPU application suites typically used in simulation and 7 applications from the MLPerf benchmark suite where the datasets are publicly available. Benchmark authors often adjust the size of traditional workloads so that simulations complete in a matter of hours or days. However, these workloads execute in microseconds in silicon, rendering them impractical. The realistic workloads we study from MLPerf take several seconds to run in real silicon, and centuries to simulate. A *representative portion of the scaled GPU program* must be selected to make simulation practical.

Traditional mechanisms to select representative portions of CPU programs [13, 22, 24, 45, 64, 66] focus on selecting basic blocks from a single thread. This approach is ill-suited to GPU programs. The control-flow graphs of individual threads in GPU programs are relatively small, as each thread performs a limited amount of work compared to a CPU thread. As a result, selecting per-thread-basic block vectors without curtailing the number of threads does not significantly reduce simulation time. Prior work on selecting representative portions of GPU programs [26, 67, 68] lacks silicon validation, requires per-workload tuning, and does not scale.

Intelligent solutions like TBPoint [26] require full functional simulation to produce thread-block-level profiling information and rely on inter-kernel clustering mechanisms. This approach does not scale to modern workloads such as MLPerf. As a result, no GPU workload sampling methodology has achieved general acceptance. In addition, no existing work has evaluated the practical implications of silicon profiling at scale. Figure 1 quantifies the slowdown experienced by detailed in-silicon profiling using Nvidia's latest profiling tools [41]. Collecting even a limited number of statistics from long-running workloads quickly becomes impractical and any sampling methodology that relies on detailed profiling of the entire program does not scale to contemporary workloads.

We propose the automated *Principal Kernel Analysis* (PKA) methodology which reduces the number and length of kernels used to represent fully scaled GPU applications. We base *Principal Kernel Analysis* on three key observations. First, even though realistic workloads can launch millions of kernel instances (5.3 million in MLPerf's SSD Training), these kernels can be characterized and grouped by a set of architecture-independent metrics using principal component analysis obtained from detailed silicon profiling.

Second, detailed silicon profiling is impractical in scaled workloads, making two-level profiling necessary. For workloads with impractical profiling times, PKA performs detailed profiling on a subset of the application's kernels and lightweight profiling on the rest. Using a variety of classifying algorithms (Stochastic Gradient Descent, Gaussian Naive Bayes, Multilayer Perceptrons) PKA maps the lightly profiled kernels into the groups identified in the detailed profiling phase. Using this per-kernel analysis, we perform *Principal Kernel Selection* (PKS) to extract the minimum set of kernels necessary to obtain a target projected execution time error.

Our third observation comes from the behavior of individual kernels. We observe that the instantaneous Instructions Per Cycle (IPC) within one kernel often stabilizes around a value that will be representative of the kernel's final IPC. Borrowing a method from the financial analysis sector that attempts to predict stock price stabilization over time [20], we track the standard deviation of a kernel's IPC throughout simulation. Once our online stabilization calculation reaches an appropriate confidence interval, we use occupancy information about the running kernel to make a *Principal Kernel Projection* (PKP) based on the amount of work remaining. Our online mechanism can be executed quickly in simulation and validated against lightweight silicon profiling.

To evaluate the effectiveness of *Principal Kernel Analysis* and its effect on accuracy, we apply our selection and projection mechanisms to the cycle-level GPU simulator Accel-Sim [32]. Using silicon profiling data, we select a workload's representative kernels. These kernels are then simulated until IPC stabilization is detected in the simulator, at which point the resultant statistics are projected.

This work makes the following contributions:

(1) We perform the first silicon-validated simulation analysis with scaled GPU workloads, identifying characteristics we exploit to create concise representations of GPU programs, reducing simulation time.
(2) We introduce *Principal Kernel Selection* an inter-kernel, architecture independent principal component analysis that automatically clusters kernels with similar behavior. Using metrics obtained from two-level silicon profiling, we select a representative subset of kernels that we use to project the entire application's behavior. We demonstrate that the kernels we select from profiling one GPU generation generalize across Nvidia's Volta, Turing, and Ampere platforms.
(3) To reduce intra-kernel simulation time with low-overhead, we leverage an observation that the instantaneous IPC in many scaled, real-world kernels stabilizes near its final average. Inspired by methods that predict stock-price stability, we propose *Principal Kernel Projection*, which detects IPC stability and projects per-kernel metrics based on occupancy.
(4) We propose a fully automated characterization and simulation methodology *Principal Kernel Analysis* that combines principal kernel selection and projection. Evaluated on Accel-Sim using 147 workloads, we demonstrate that PKA greatly reduces simulation time, while maintaining an error rate close to the baseline simulator. Centuries-long simulation times from MLPerf are reduced to hours with an average cycle error of 27% versus silicon.

Table 1: Landscape of Sampled Simulation Literature.

Sampling Methodologies	Control-Flow Reduction [24, 45], [54, 64, 66]	Synchronization Regions [13, 22]	GPGPU -MiniBench [67, 68]	GT-Pin[30]	TBPoint [26], Clustering [21]	***Principal Kernel Analysis***
Threaded	Single	CPU Multi-Threaded	GPU Multi-Threaded	GPU Multi-Threaded	GPU Multi-Threaded	GPU Multi-Threaded
Mechanism	Identify common basic blocks	Inter-barrier regions	Intra-thread-block control flow analysis	Unique kernels & control flow analysis	Thread block reduction [26], kernel clustering	Thread block/kernel reduction
Inter-kernel	NA	NA	X	✓	✓	✓
Intra-kernel	NA	NA	✓	X	[26]* Requires full functional simulation	✓
Sampling Clustering	Automated	Automated	Automated	Automated	Hierarchical hand-tuned	Automated
# GPU Workloads	NA	NA	23	25	12	147
Silicon Validated vs Century-Long Full-Simulation	X	X	X	X	X	✓

2 BACKGROUND AND MOTIVATION

Characterizing programs to reduce simulation time through sampling techniques is a decades-old research area. Table 1 presents a survey of sampling techniques proposed for single-threaded CPU [24, 45, 64, 66], multithreaded CPU [13, 22, 35] and GPU [26, 67, 68] applications. Fundamental differences in the nature of CPU programs makes the direct application CPU techniques to GPUs difficult. CPU programs contain, at most, tens of threads. These techniques focus on reducing the work done by each thread, reducing the scope of the dynamic control-flow graph [24, 35, 45, 64, 66] or selecting portions of each thread between synchronization points [13, 22], without decreasing the number of threads.

Prior work has also explored sampling GPU applications. Kambadur et al. introduce GT-Pin [30], a dynamic binary instrumentation tool for OpenCL workloads on Intel GPUs that can be used to select representative portions of GPU programs. Using a clustering algorithm based on kernel name, arguments and basic block statistics, GT-Pin selects representative portions of the program with a kernel as the smallest granularity. In contrast, *Principal Kernel Analysis* focuses on both inter- and intra-kernel reduction, while basing its inter-kernel clustering on a name-independent feature-vector.

Zhibin et al. [67, 68] proposed GPGPU-MiniBench in which they profile an application's control-flow divergence. Their analysis is similar to SimPoint [24] in that they analyze and define the minimum number of intra-thread-block loops needed to represent a kernel. However, the MiniBench analysis does not address inter-kernel reduction, inter-thread-block reduction, involves the creation of proxy-applications, and focused on legacy workloads.

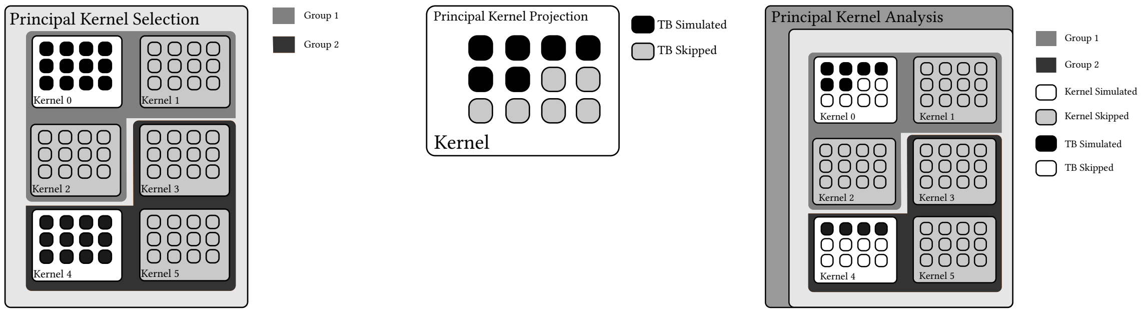
More closely related to PKA is TBPoint [26], which uses statistical modeling and pure-simulation results to reduce both the number and length of kernels used to represent a workload. Using 12 legacy workloads and statistics gathered from full functional simulation [18], TBPoint uses hierarchical clustering to group and reduce the number of kernels in the program representation.

Although TBPoint attacks both inter- and intra-kernel reduction, the mechanisms do not scale to the century-long workloads we study in this paper, and require per-application parameter tuning. To reduce the number of kernels simulated, TBPoint performs hierarchical clustering on a feature vector derived from simulation. To reduce intra-kernel runtimes, per-thread-block simulation statistics are required for the entire kernel. Although precise, TBPoint cannot be applied to applications that cannot be fully simulated. To handle contemporary, scaled workloads we argue that the analysis must be done online, then validated against silicon execution. To demonstrate the efficacy of PKA on workloads where TBPoint is tractable, we perform a quantitative comparison against TBPoint in Section 5.

Despite the well-reasoned related work in this space, a GPU equivalent of SimPoint has yet to receive widespread adoption. We aim to fill this gap by introducing the *Principal Kernel Analysis* methodology and toolset [1], designed to have the following characteristics:

(1) **Scalable:** By using two-level profiler data from real silicon execution of full-scale workloads, *Principal Kernel Analysis* is able to characterize workloads that would be untenable using prior GPU-centric sampling techniques like TBPoint.
(2) **Automatic:** The inputs to *Principal Kernel Analysis* are the profiled results from silicon, a desired maximum error from the *Principal Kernel Selection* phase and a confidence interval for the *Principal Kernel Projection* phase. For all the applications we study, we apply the same desired error and confidence interval such that no per-workload tuning of opaque clustering parameters is required.
(3) **Tunable:** There is always a tradeoff between simulation time and simulation fidelity. The work simulated using *Principal Kernel Selection* and *Principal Kernel Projection* can be tuned to the user's desired error and confidence interval respectively.
(4) **Verified Against Silicon:** Simulator versus simulator validation limits the workloads that can be validated to those possible to simulate in their entirety. Applying *Principal Kernel Analysis* to Accel-Sim, we compare sampled simulation results to real silicon results, demonstrating that an open-source simulator can achieve an acceptable absolute and relative error on real workloads.

[1]Fully-automated scripts, profiler data and simulator integration are included in the paper's appendix and corresponding Zenodo record [4].



(a) ***Principal Kernel Selection.*** (b) ***Principal Kernel Projection.*** (c) ***Principal Kernel Analysis***

Figure 2: An illustration of *Principal Kernel Analysis* by reducing the number of executed kernels and thread blocks.

Table 2: A list of microarchitecture-agnostic characteristics collected for PCA analysis.

Metric	Nsight metric name
Coalesced global loads	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum
Coalesced global stores	l1tex__t_sectors_pipe_lsu_mem_global_op_st.sum
Coalesced local loads	l1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum
Thread global loads	smsp__inst_executed_op_global_ld.sum
Thread global stores	smsp__inst_executed_op_global_st.sum
Thread local loads	smsp__inst_executed_op_local_ld.sum
Thread shared loads	smsp__inst_executed_op_shared_ld.sum
Thread shared stores	smsp__inst_executed_op_shared_st.sum
Thread global atomics	smsp__sass_inst_executed_op_global_atom.sum
#Instructions	smsp__inst_executed.sum
Divergence efficiency	smsp__thread_inst_executed_per_inst_executed.ratio
#thread blocks	launch_grid_size

3 PRINCIPAL KERNEL ANALYSIS

In this section, we introduce our two-level hierarchical method to create concise representations for GPU programs and speedup simulation. There are two families of techniques to speedup GPU simulation, inter-kernel and intra-kernel, one acting at kernel-granularity and the other at thread block-granularity. Our first method, called *Principal Kernel Selection*, depicted in Figure 2a, uses several microarchitecture-agnostic metrics to cluster similar kernels together and simulates only one representative kernel per cluster. The second method, called *Principal Kernel Projection*, depicted in Figure 2b, reduces the number of simulated thread blocks in the grid by detecting IPC stability. Figure 2c illustrates *Principal Kernel Analysis*, which combines both techniques together to reduce both the number of kernels and the number of thread blocks within those kernels.

3.1 Principal Kernel Selection

To reduce the number of simulated kernels, we group similar kernels together and only simulate the most representative (or principal) kernels. To achieve this, we profile each application in silicon. We tell the profiler only to report certain microarchitecture-agnostic features, such as the number of global loads, stores, and atomic operations (Table 2). Note that these statistics are dependent only on the generated GPU code, not the specific GPU being profiled. One caveat is that different GPU generations use different machine ISA representations, therefore; the number of instructions and makeup of specific instructions can vary slightly across generations. We note that classic CPU methodologies have a similar issue between the x86 representation of the program and the micro-ops used by the pipeline. To further demonstrate that these metrics hold across different GPU generations, Section 5 evaluates the efficacy of *Principal Kernel Selection* across the three most recent Nvidia GPU generations (Volta, Turing and Ampere). In our evaluation, *Principal Kernel Selection* is performed for Volta and those same representative kernels are selected to project the execution time in Turing and Ampere, without running *Principal Kernel Selection* on each machine.

Since GPGPU workloads may have different categories and special characteristics, we perform non-supervised machine learning techniques—Principal Component Analysis (PCA) and K-Means—to reduce these microarchitecture-agnostic features' dimensions to a more manageable number from the components in Table 2. The PCA data are then clustered using K-Means. As a result, K groups of similar kernels are formed. It should be noted that clustered kernels often do not have the same name, groups are usually composed of several instances of differently named kernels. For each of these K groups, a single representative kernel is chosen. Because this kernel summarizes the entire group, we scale its runtime (in cycles) by the number of kernels (elements) to obtain a projection of each group's total runtime. This process is aggregated across all groups to obtain a projection for all the kernels in the program.

Algorithm for choosing K-Groups: Part of the reason why we choose PCA+K-Means is explain-ability. By applying PCA, we can trivially consider a broader set of characteristics. We know that the principal dimensions will have the most variance. With K-Means, we can directly change the number of groups. By combining PCA and K-Means, we avoid the curse of dimensionality when clustering. Another benefit is that the K parameter represents a less abstract notion than other clustering techniques, like sigma and hierarchy clustering used by TBPoint [26]. Most importantly, k-means clustering can scale to the millions of kernels in our large workloads, where hierarchical clustering demands an impractical amount of memory and runtime.

By varying the K parameter in K-means, the trade-offs are apparent. We start by sweeping across different values of K, typically from 1 to 20, and generating different clustering configurations. For

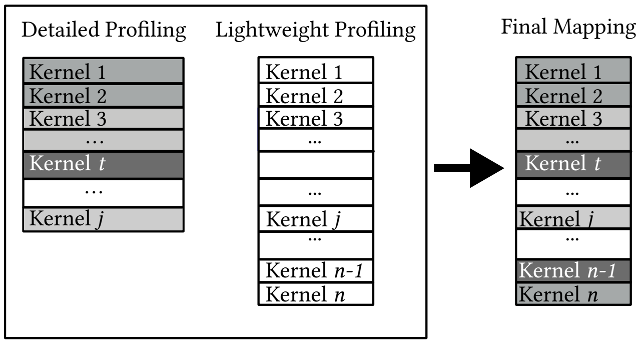


Figure 3: PKA's two-level profiling mechanism

each clustering configuration, we find the most representative kernels. We scale the number of cycles of each representative kernel by the number of elements in the group and aggregate them to obtain a total runtime projection. We end with K different projections. For each projection, we calculate its error with respect to the total silicon number of cycles. The smallest K value whose projected error falls below a threshold is chosen. A smaller K is preferred, since fewer groups results in a greater reduction in the number of kernels required to represent the program. The only input the user gives to the process is the desired execution time error. In all the data we present, we set the *Principal Kernel Selection* cycle error threshold to 5%. Less error will require more groups. Therefore, linked to choosing the number of groups is selecting the most representative kernel within each group.

To determine which kernels should be selected as the principal one we experimented with random selection, selecting the closest to the center and selecting the first chronologically. We empirically determined that random selection has an inconsistent error rate. In contrast, the performance difference between choosing the cluster-center and first-chronologically kernels is negligible. Selecting the first chronological kernel has practical advantages in reducing tracing and profiling times, thus we use it for *Principal Kernel Selection.*

Two-level profiling: For workloads where detailed silicon profiling is intractable (i.e. if the profiling takes more than one week), we propose a novel two-level profiling approach. Figure 3 shows a visualization of our approach. We perform detailed profiling on the first j kernels and create our k-groups based on their characteristics (i.e. we apply *Principal Kernel Selection*). We profile the remaining kernels using the low-overhead Nsight-systems profiler, where only the kernel name and grid dimensions are collected. For the MLPerf workloads, we augment Nsight-systems with information supplied by Nvidia's PyProf. PyProf is a PyTorch-compatible tool that provides an additional layer of per-kernel logging via NVTX annotations. The extra information is tensor dimensions, a program trace, and a pointer to which layer of the neural network is associated with each kernel. The $n - j$ kernels where only lightweight information is available are then labeled with one of the k groups identified from detailed profiling. We use three different classification models (Stochastic Gradient Descent, Naive Bayes Gaussian, and Multi-Layer perceptrons) to map the augmented data to groups.

Table 3: An example of *Principal Kernel Selection*'s output. The last two columns show the set of kernel IDs selected to represent each group and the number of kernels per group.

Suite	Workload	Selected Kernel ID	Group Count
Rodinia	gaussian_208	0	414
	bfs 65k	0	20
Parboil	histogram	0,1,2,3	20,20,20,20
	cutcp	0,1,2	2,3,6
Polybench	fdtd2d	0,2	1000,500
	gramschmidt	0, 1, 2, 1439, 2783, 4127	2048, 2273, 479 448, 448, 448
Cutlass	2560 x 128 x 2560 wmma	0	7
	4096 x 4096 x 4096 sgemm	0	7

Group Selection Examples: Table 3 depicts an example of *Principal Kernel Selection* analysis output for a few selected workloads with a target error of 5%. In the example the analysis of the application *gaussian* clusters 414 similar kernels into only one group. In this scenario, only one kernel (kernel id=0) is selected. In *gramschmidt*, the clustering analysis determines six different groups are necessary (ranging in size from 448 kernels to 2273 kernels), and thus six representative kernels (out of 6411 kernels) are selected.

In Figure 4, we depict the per-group kernel composition after applying *Principal Kernel Selection* to the long-running ResNet 5.0 workload from the MLPerf suite. As shown in the figure, we end up with nine different groups and each group contains hundreds of kernels. It is worth mentioning that kernels in the same cluster have different names/code implementations. We notice that compute-intensive kernels (e.g., convolution operations and fully connected layers) are combined in the same group, whereas memory-intensive kernels (e.g., element-wise operations) are clustered in the same group. In addition, some kernels with the same name are sorted into different groups. This often happens when the kernel is launched several thousand times with different grid and/or thread block dimensions. Since we use unsupervised learning to create this clustering, these groupings naturally fell out from silicon profiling.

3.2 Principal Kernel Projection

Principal Kernel Selection only addresses the number of kernels an application runs; long-running kernels may still be a bottleneck. To reduce the runtime of individual kernels, we introduce *Principal Kernel Projection.* The key insight exploited in *Principal Kernel Projection* is that each thread in the grid executes the same code and the code tends to have few phases, as the lifetime of threads are much shorter than their CPU peers. As a result, we have observed that the IPC of most GPU kernels stabilizes around the final average, even in some irregular applications like graph processing. Therefore, *Principal Kernel Projection* is designed to detect a stable IPC when it occurs, ending simulation then and projecting final statistics.

To detect stability, we calculate the rolling average and standard deviation of the last n cycles (we use 3000 across all our workloads). If the standard deviation falls below another user-defined variable s, the IPC is considered quasi-stable. The standard deviation variable s is the only user-facing input variable to *Principal Kernel Projection* and the user can select a value of s to reflect the confidence interval

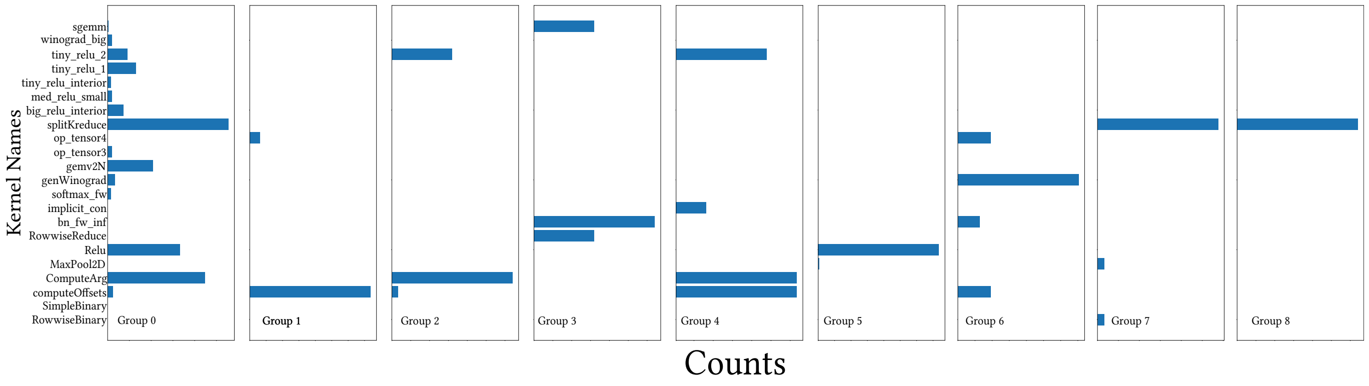
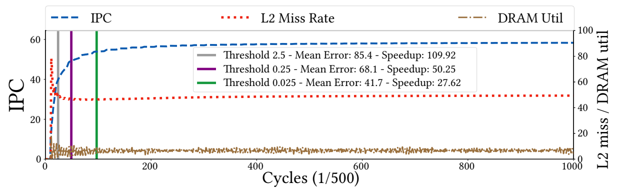


Figure 4: Per-group kernel composition after applying PKS to ResNet. The x-axis shows the number of kernel instances for each named kernel in the 9 groups identified by PKS.

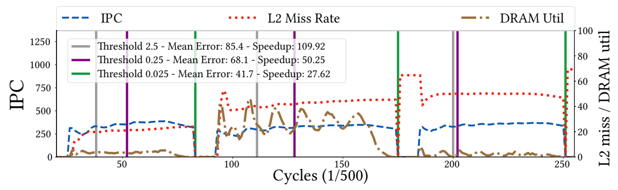
desired in IPC stabilization. To show that *Principal Kernel Projection* does not require extensive hand-tuning to generate reasonable results, we select an *s* value of 0.25 for all our experiments. A smaller value will increase the confidence that the IPC has stabilized at the expense of more simulation time.

To ensure resource contention is properly captured, we impose an additional constraint on classifying the signal as stable; the number of finished thread blocks must be more than the amount that fills the GPU's cores, i.e., enough thread blocks to reach the highest-possible occupancy of the kernel in question. We call this quantity a wave. Once enough thread blocks have finished to complete the wave where quasi-stability occurs, we consider the signal stable. To project the number of cycles it would take to finish the kernel, we take the number of unfinished thread blocks and linearly project the number of cycles left. If a kernel launches less thread blocks then a wave, we ignore this conditional and stop the kernel as soon as stability occurs. Since kernels with few CTAs do not experience CTA ending/beginning phases, we find that removing this condition for low-CTA kernels results in acceptable error.

To illustrate the operation of *Principal Kernel Projection*, Figure 5 presents a visualization of application IPC versus time for a regular workload (ATAX in Figure 5a) and an irregular workload (BFS in Figure 5b). Also included in the graph are the L2 Miss rate and DRAM util as different time series. The stopping conditions for *Principal Kernel Projection* at different *s* values are indicated with vertical lines. The results in the regular workload are unsurprising in that it quickly ramps up to its peak IPC value and stays there for the duration of each kernel. For each of the three kernels shown for the irregular workload, the results are more surprising. Despite having significant control and memory divergence, over time the IPC, L2 miss rate and DRAM utilization of BFS does stabilize. Despite the fact that each thread in the system is performing different amounts of work, in the aggregate a collision of irregularity in all the threads results in stability. Figure 5 also illustrates the effect different threshold values have on the stopping point of *Principal Kernel Projection*. We empirically find that 0.25 results in a good compromise between accuracy and speedup across all our apps.



(a) A single kernel from atax: A regular application.



(b) Three kernels from BFS: An irregular application.

Figure 5: IPC, L2 miss rate and DRAM utilization vs time for kernels from two applications. Solid lines show Principal Kernel Projection stopping points at different *s* values.

4 MEASUREMENT SETUP

To evaluate *Principal Kernel Analysis* we combine experiments on both silicon and simulation, using three generations of Nvidia GPUs. *Principal Kernel Selection* requires silicon profiling to infer kernel similarity. For detailed and lightweight profiling, we use Nsight Compute and Nsight Systems respectively [41]. We evaluate *Principal Kernel Selection* in both silicon and simulation, while we show *Principal Kernel Projection* for simulation only. We obtain the principal kernels by applying *Principal Kernel Selection* to a V100 [15]. We use these kernels to evaluate their effectiveness on a Turing RTX 2060 [12] and an Ampere RTX 3070 [16]. To evaluate both selection and projection, we use Accel-Sim [1, 9, 32, 63].

We evaluate our technique with the complete benchmark sets of Rodinia [14], Parboil [56], Polybench [23], the machine learning suite DeepBench [5], and the GeMM-based CuTLASS [40] benchmark suite. We also evaluate the subset of the reference implementations of the applications in MLPerf [48] for which we could get realistic datasets and confirm correct functionality in

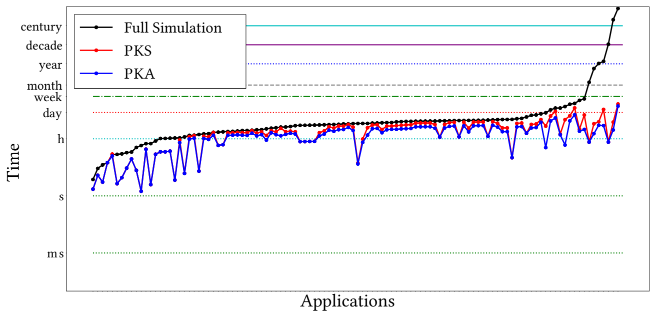


Figure 6: Simulation time using full simulation, PKS and PKA. Y-axis is log scale hours.

silicon. Specifically those are: ResNet [25] using the ImageNet dataset [50], SSD [34] using the COCO dataset [33], GNMT [65] using the German and English euro database, BERT [17] using the SQUAD dataset [47], the medical imaging 3D-Unet [28] using the BRATS dataset [37] [6] [7]. Application are compiled using CUDA 11.1 and cuDNN 8.0.2.

5 EVALUATION

We stipulate that the combination of *Principal Kernel Selection* and *Principal Kernel Projection* drastically reduce the expected simulation time with an acceptable loss of accuracy. In this evaluation section we set out to answer the following research questions:

- RQ1: How accurately do *Principal Kernel Selection* and *Projection* individually and collectively predict performance and reduce simulation time and how do they compare to prior work (Section 5.1)?
- RQ2: How do the characteristics of applications affect the efficacy of *Principal Kernel Analysis*, and do our results generalize across architectures (Section 5.2)?
- RQ3: Given that architects care most about the relative accuracy of a simulator, how well do *Principal Kernel Selection* and *Projection* predict the relative performance of the different architectures we study (Section 5.3)?

In section 5.1, we analyze and compare the achieved simulation speedup and accuracy of *Principal Kernel Analysis* versus: (1) a commonly-used technique to only simulate the first 1 billion instructions, and (2) state-of-the-art sampled simulation of GPU applications, TBPoint [26] in simulation using Accel-Sim version 1.1 [32]. In Section 5.2, we perform a deep-dive into the data for each application suite and configuration we study in both silicon and in simulation. In section 5.3 we present two case studies of how *Principal Kernel Analysis* predicts the relative performance of different architectures.

5.1 Overall Effectiveness

In this subsection, we compare the efficacy of PKA to prior work in simulation using Accel-Sim modeling an Nvidia Volta V100. Figure 6 plots the simulation times we achieve versus the original simulation times originally shown in Figure 1 across our benchmarks sorted by their simulation time. The figure demonstrates that *Principal Kernel Analysis* is able to reduce the simulation time of every workload we study from up to several centuries down to less than one week. The figure demonstrates that many applications see significant reduction from PKS. The effect of the intra-kernel reduction is more skewed. There is a significant constant-factor speedup on most of the longer-running workloads (reducing simulation time from days to hours in some cases). However, the bulk of the reduction in simulation comes from PKS. Most complex applications are divided into multiple kernel launches, hence PKS shows the most benefit when application runtimes are long. The full error results with this speedup are shown in Table 4 and discussed in Section 5.2.

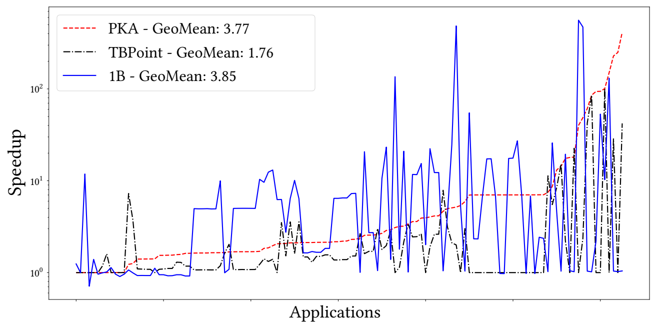


Figure 7: Simulation speed up of PKA, TBPoint and 1B instructions over full simulation in all the applications that can complete in simulation.

To evaluate the effectiveness of PKA compared to previous work, Figure 7 plots the reduction in cycles of PKA, TBPoint [26], and the commonly used practice of executing the first 1 billion instructions. Only the applications that complete in full simulation (and are hence possible using TBPoint) are plotted here. In lieu of the hand-tuned threshold setting that TBPoint originally required, our implementation of TBPoint sweeps across 20 threshold values between 0.01 and 0.2 and follows the same criteria *Principal Kernel Selection* does to decide the best one. In these classic workloads, PKA is able to reduce the number of cycles almost as much as the simple, high-error mechanism of executing the first 1 billion instructions. Although TBPoint is able reduce simulation time significantly, it requires 2.19× more simulation than PKA.

Figure 8 plots the absolute IPC error for the same three mechanisms, sorted by the baseline error of full simulation. Although 1B instructions provides a significant speedup, the error is 5.4× higher than full simulation. TBPoint's relatively conservative reduction mechanism results in an error 0.56 points higher than full simulation, while PKA's error is 4.44 points higher. Comparing TBPoint to PKA, PKA provides a 2.19× reduction in simulation time over TBPoint for only slightly more error. While this tradeoff is appealing, the main advantages of PKA over TBPoint is the ability to evaluate scaled workloads and the automated nature of PKA's selection mechanisms. We discuss the error and speedup of PKA on the scaled MLPerf applications in Section 5.2.

5.2 Results Analysis

To help perform a side-by-side analysis of our various configurations, hardware platforms and applications, Table 4 aggregates the

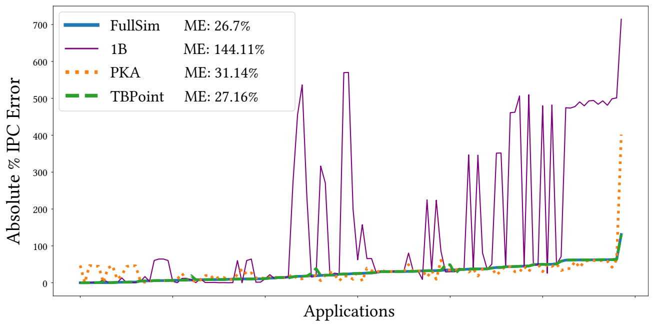


Figure 8: Accel-Sim simulation error using full simulation, 1B instructions, PKA and TBPoint.

raw results of *Principal Kernel Analysis* in both silicon and in simulation. The first 6 columns of the table present the error and speedup results for the three GPUs we study when applying *Principal Kernel Selection* to silicon-only results. Speedup (SU) represents the reduction in total execution time that is achieved with the adjacent error. The next 5 columns are the simulated Volta error and speedup results for both *Principal Kernel Selection* alone and when combined with *Principal Kernel Projection* (i.e. PKA).

Only the Volta V100 GPU has enough memory to run the MLPerf workloads we study, hence we subdivide this section into first discussing the silicon Volta results (Section 5.2.1) using *Principal Kernel Selection* in isolation, contrasting those results with Turing and Ampere's silicon results in Section 5.2.2 and finish with a discussion of the full volta simulation results in Section 5.2.3.

5.2.1 Volta Silicon Results. In this subsection we discuss the first two columns of Table 4. The Rodinia application suite commonly used in architecture studies has inputs sized such that they finish in simulation, resulting in real execution times of under a millisecond. However, *Principal Kernel Selection* is able to significantly reduce the simulation time of many of the applications that launch a large number of kernels. Overall, the error introduced by *Principal Kernel Selection* in Rodinia is 1.6% with a geomean speedup of 7.2×. Parboil and Polybench are similar (1.3% error/5.8× speedup and 0.8% error/4.2× speedup respectively). Single-kernel applications see no benefit, while apps with many kernels see speedups of up 711.1×, with little accuracy loss.

Cutlass and the various implementations of Deepbench represent highly-tuned machine-learning kernels evaluated in isolation and are used as a reasonable proxy for the matrix-multiplication kernels found in neural networks. They are hand-tuned by Nvidia engineers, and make use of tensor cores. Although the error rates remain low, the speedup is also muted, in comparison to the other suites (ranging between 1 and 7×). Since these applications launch fewer, targeted kernels, *Principal Kernel Selection* is less effective.

The final suite of applications are from the MLPerf suite. Among all the considered suites, these are the longest workloads. We are running the reference implementations of MLPerf release 1.0. We should note that the input sizes utilized for both inference and training workloads were either above or equal to the minimum sizes dictated by the MLPerf 1.0 inference [38] and training [39] submission policies. Running the BERT inference pass using the Offline scenario takes roughly 10 minutes in silicon. Profiling these workloads was challenging, as the number of kernels is orders of magnitude larger than the other suites. For the vision and classification inference workloads, ResNet and 3D-Unet, a complete profiling was achievable using Nsight Compute. For the larger workloads, the hierarchical clustering technique was used. The effects of which is clear because of the penalty said technique incurs in the mean error. The largest workload is SSD training, with 5.3 million kernels, of which 20 thousand kernels were profiled in detail. The average error across the runs of MLPerf benchmarks is 10.0%, and the geomean speedup is 1987×.

5.2.2 Turing and Ampere Silicon Results. In this subsection, we discuss columns 3-6 in Table 4. To validate our hypothesis that the principal kernels identified using the volta are representative, regardless of architecture, we use them to evaluate *Principal Kernel Analysis*'s accuracy in other generations.

For Rodinia, Parboil and Polybench, the overall error and performance trends are maintained inter-generation; if applying *Principal Kernel Selection* to Rodinia yields a speedup above 400 × in Volta, we see the same trend in Turing and Ampere. The performance of Turing and Ampere in the Cutlass Perf Suite SGEMM presents a negligible mean error and a geomean speedup of 6×. The Tensor Core version keeps the mean error under 1% for both Turing and Ampere and yields the same speedup of 7× as the Volta card did. The reduced kernel projection of Turing and Ampere performs roughly the same for Parboil and Polybench suites as the Volta GPU did. If the selected kernels work in Volta, they work in Turing and Ampere, and the complement is also true.

The next suite of workloads is Deepbench. Starting with the convolution inference workloads, both the CUDA and Tensor Core variants. We see the same sensible errors and modest speedups across all GPU generations, a mean error of 0.8% and a speedup of 1.5×. In training, things get interesting due to a quirk with the cuDNN libraries. Out of the five workloads, only one workload per card had the same number of kernels as the Volta card. One of the cuDNN functions selects the best performing backward and forward propagation algorithms based on some metrics at runtime. Introducing the profiler in the mix resulted in several different combinations of algorithms being used, therefore the work being done was no longer guaranteed to be the same across multiple runs. The Turing card had an error of 51.3% and a speedup of 5×, while the Ampere card had an error of 0.5% and a speedup of 3.6×. The GEMM bench and RNN bench results are similar for both cards in both CUDA and Tensor Core variants.

5.2.3 Simulation Results. Finally, we discuss columns 7-11 in Table 4 which evaluate PKS and PKS+PKP (i.e. PKA) in simulation using Accel-Sim and the V100 model [32]. In Table 4 we report the simulator error with respect to silicon ("SimError") so PKS and PKA can be put into context.

We start with the Rodinia, Parboil and Polybench suites. The average error between the baseline simulator and PKS is consistently very close, with the speedups tracking what we saw in silicon (Section 5.2.1). Applying full-PKA to these applications is a mixed bag, where many of the applications see the bulk of their speedup from PKS. There are a few exceptions to the rule, in-particular fdtd2d,

Table 4: Cycle error and speedup for Principal Kernel Selection (PKS) in silicon and using Accel-Sim. Principal Kerenel Analysis (PKA) results shown for simulation. "*"= no data (explained in Section 5). SU=Speedup (in ×). Errors are in %. H=Hours.

Application	Silicon						Simulation					Metrics	
	Volta		Turing		Ampere		Volta					DRAM Util	
	Error [%]	SU	Error [%]	SU	Error [%]	SU	SimError	PKS Error	PKS SimTime [H] (SU)	PKA Error	PKA SimTime [H] (SU)	Full	PKA
Rodinia Suite													
b+tree	0	1	0	1	0	1	5.8	5.8	0.4 H (1.0)	3.5	0.2 H (1.7)	14.3	14.2
backprop	0	1	0	1	0	1	4.3	4.3	0.1 H (1.0)	4.3	0.1 H (1.0)	35.0	55.0
bfs1MW	5	1.5	0.4	1.2	0.7	1.3	36.7	34.5	1.4 H (1.5)	12.1	1.0 H (1.7)	24.0	30.4
bfs4096	1.6	1.2	2	1.2	1.8	1.2	15.5	23.0	0.1 H (1.2)	23.0	0.1 H (1.2)	0	0
bfs65536	1.9	19.6	35.6	31.1	2.8	19.4	14.2	12.1	0.0 H (21.4)	12.5	0.0 H (22.2)	0	0
dwt2d_192	1.2	3.5	1.4	3.2	6.3	3.3	45.2	48.3	0.0 H (3.5)	48.3	0.0 H (3.5)	0	0
dwt2d_rgb	0.3	2.3	1.2	2	0.1	2	1.6	0.1	0.1 H (2.4)	0.1	0.1 H (2.4)	25.4	41.36
gauss_208	5	435.6	7.8	449	7.2	446.1	56.7	63	0.0 H (429.6)	51	0.0 H (431.1)	0	0
gauss_mat4	1.8	5.9	0.9	5.9	1.1	6.1	77.8	86.8	0.0 H (6.0)	86.8	0.0 H (6.0)	0	0
gauss_s16	2.5	14.9	2.9	14.8	0.1	14.5	73.5	84.5	0.0 H (15.0)	73.5	0.0 H (20.1)	0	0
gauss_s64	0.7	60.1	1.6	61.3	2.4	62	69.8	79.0	0.0 H (63.7)	67.9	0.0 H (74.0)	0	0
gauss_s256	0.4	226.3	8.5	167.9	3.8	232.4	53.4	65.8	0.0 H (248.0)	50.8	0.0 H (258.4)	0	0
hots_1024	0	1	0	1	0	1	3.9	3.1	0.2 H (1.0)	9.1	0.1 H (1.3)	23.5	20.4
hots_512	0	1	0	1	0	1	16.1	16.1	0.0 H (1.0)	16.1	0.0 H (1.0)	0	0
hstort_500k	4.8	4.4	6	4.6	3.9	4.4	45.1	46.5	0.3 H (4.3)	46.5	0.3 H (4.3)	1.0	1.28
hstort_r	4.6	5.6	7.8	6.6	5.9	6	49.5	47.8	2.3 H (5.6)	45.4	2.2 H (5.7)	14.1	34.9
kmeans_28k	1.4	1.6	0	1.3	0	1.6	15.8	16.6	17 M (1.6)	16.6	17 M (1.6)	9.4	6.6
kmeans_819k	0	1.2	0	1.3	0.1	1.4	60.8	38.9	5.1 H (1.1)	3	1.5 H (3)	31.2	32.6
kmeans_oi	0.1	1.2	0	1.3	0.1	1.4	57.6	32.8	3.8 H (1.1)	0.2	1.8 H (2.0)	29.8	32.0
lavaMD	0	1	0	1	0	1	13.2	13.2	8.0 H (1.0)	0.1	6.7 H (1.2)	*	*
lud_i	2	19.5	6.7	13.2	4	16	10.6	15.8	0.0 H (18.2)	11.6	0.0 H (18.7)	0.4	0.0
lud_256	0.4	8.5	0.5	7.8	0.6	8	11.8	15.7	0.0 H (7.6)	11.8	0.0 H (7.2)	0.1	0.0
myocyte	*	*	*	*	*	*	*	*	*	*	*	*	*
nn	0	1	0	1	0	1	38	38	0.0 H (1.0)	38	0.0 H (1.0)	0	0
nw	3.6	88.2	7.7	92.1	2.9	87.5	0.1	1.3	0.0 H (87.1)	2.5	0.0 H (87.6)	0	0
scluster	0.9	128.9	1.9	127.5	1.2	128.5	25.9	30.4	0.0 H (125.5)	30.4	0.0 H (119.5)	*	*
srad_v1	2	98.2	0.9	99.2	0.6	99.5	2	2.3	0.1 H (101.8)	2.3	0.1 H (101.8)	0	0
Parboil Suite													
bfs	4.2	1.1	3.9	1.1	4	1.1	37.8	40.4	0.9 H (1.1)	40.4	0.9 H (1.1)	*	*
cutcp	3.3	4.1	2.9	4	3	4	17.5	19.5	0.9 H (4.0)	19.5	0.9 H (4.0)	*	*
histo	0.4	20.1	0.2	20	0.3	19.9	60.9	57.4	0.2 H (18.4)	57.4	0.2 H (18.4)	14.0	14.5
mri	0.4	3	0.2	3	0.3	3	8.2	8.2	0.2 H (2.9)	8.2	0.2 H (2.9)	0.3	2.1
sad	0	1	0	1	0	1	7.8	7.8	0.3 H (1.0)	7.8	0.3 H (1.0)	10.0	10.0
sgemm	0	1	0	1	0	1	153.9	153.9	2.9 H (1.0)	153.9	2.9 H (1.0)	5.1	5.1
spmv	2.2	48.9	0.8	50.4	0.5	50.3	14.2	12.4	0.1 H (50.9)	12.4	0.1 H (50.9)	*	*
stencil	0	100	1.3	101.3	0.3	99.7	30.1	30.1	0.0 H (1)	30.1	0.0 H (1)	0.1	5
Polybench Suite													
2Dcnn	0	1	0	1	0	1	12	17	1.3 H (1.0)	42	0.2 H (4.6)	53.5	36.0
2mm	0	2	0.1	2	0	2	6.8	1.7	99.7 H (2.0)	15	3.8 H (1.3)	*	*
3dconvolution	4.6	242.9	2.2	259.8	0.4	253	50.3	56.6	0.0 H (243.7)	56.6	0.0 H (249.7)	0	0
3mm	0.4	3	0.1	3	0.5	3	11.4	11.6	1.7 H (3.0)	7.9	1.3 H (4.0)	0.4	0.6
atax	0	1	0	1	0	1	22.4	22.4	2.3 H (1.0)	22.4	2.3 H (1.0)	6.5	6.5
bicg	0	1	0	1	0	1	23	23	2.2 H (1.0)	23	2.2 H (1.0)	6.5	6.5
correlation	0	1	0	1	0	1	42.8	42.8	494.4 H (1.0)	42.8	494.4 H (1.0)	*	*
covariance	0	1	0	1	0	1	43.4	43.4	502.6 H (1.0)	43.4	502.6 H (1.0)	*	*
fdtd2d	1.6	711.1	1.3	722.5	1.6	706.9	6.5	2.6	0.3 H (725.6)	2.6	0.1 H (2725.5)	*	*
gemm	0	1	0	1	0	1	12.8	12.8	1.9 H (1.0)	7.5	1.5 H (1.3)	0.5	0.7
gsummv	0	1	0	1	0	1	0.1	0.1	2.5 H (1.0)	0.1	2.5 H (1.0)	6.7	5.9
gramschmidt	4.9	498.2	6.8	507.1	4.3	494.5	27.8	26.3	1.1 H (500)	26.3	1.1 H (500)	*	*
mvt	0	1	0	1	0	1	22.9	22.9	2.3 H (1.0)	22.9	2.3 H (1.0)	6.5	6.5
syr2k	0	1	0	1	0	1	119	188	50 D (1.0)	11.0	24 H (50)	0.1	0.2
syrk	0	1	0	1	0	1	1.7	1.7	45.2 H (1.0)	17.6	8.2 H (5.5)	*	*
Cutlass Perf Suite SGEMM (10 inputs)													
Mean	0.3	6.0	0.0	6.0	0.0	6.0	1.9	1.9	4.9 H (6.1)	3.7	2.4 H (7.6)	6.1	5.3
Cutlass Perf Suite WGEMM (TensorCore) (10 inputs)													
Mean	0.3	7.0	0.7	7.0	0.1	7.0	44.9	45.0	1.8 H (7.0)	42.7	0.4 H (12.3)	11.0	10.3
Deepbench Suite - Convolution - Inference (5 inputs)													
Mean	0.8	1.5	0.9	1.5	0.6	1.6	13.4	13.5	2.3 H (1.4)	13.6	2.1 H (1.5)	1.2	0.6
Deepbench Suite - Convolution - Training (5 inputs)													
Mean	1.3	2.8	51.3	5.0	0.5	3.6	*	*	*	*	*	1.8	6.1
Deepbench Suite - Convolution - Inference (TensorCore) (5 inputs)													
Mean	0.9	1.5	0.2	1.5	0.2	1.5	11.1	11.9	2.9 H (1.4)	13.0	2.5 H (1.6)	1.8	0.8
Deepbench Suite - Convolution - Training (TensorCore) (5 inputs)													
Mean	2.1	1.9	*	*	*	*	21.6	25.8	14.8 H (1.7)	28.3	12.5 H (2.9)	0.6	2.0
Deepbench Suite - GEMM bench - Inference (5 inputs)													
Mean	2.4	1.1	4.1	1.2	4.2	1.2	10.3	12.4	2.2 H (1.2)	12.4	2.2 H (1.3)	21.1	38.0
Deepbench Suite - GEMM bench - Training (5 inputs)													
Mean	0.9	1.3	0.2	1.6	0.6	1.5	12.6	11.6	3.5 H (1.3)	11.6	3.4 H (1.4)	23.4	29.3
Deepbench Suite - GEMM bench - Inference (TensorCore) (5 inputs)													
Mean	2.4	1.1	4.0	1.2	4.0	1.2	10.4	12.5	3.1 H (1.2)	12.5	3.1 H (1.2)	21.1	38.1

Application	Silicon						Simulation					Metrics	
	Volta		Turing		Ampere		Volta					DRAM Util	
	Error [%]	SU	Error [%]	SU	Error [%]	SU	SimError	PKS Error	PKS SimTime [H] (SU)	PKA Error	PKA SimTime [H] (SU)	Full	PKA
Deepbench Suite - GEMM bench - Train (TensorCore) (5 inputs)													
Mean	0.8	1.3	0.1	1.5	0.8	1.5	12.7	11.8	4.2 H (1.3)	11.8	4.1 H (1.3)	25.2	27.0
Deepbench Suite - RNN bench - Inference (9 inputs)													
Mean	3.3	3.0	5.6	5.3	3.2	4.5	18.7	13.0	6.1 H (1.9)	13.0	6.1 H (1.9)	0.1	6.0
Deepbench Suite - RNN bench - Train (5 inputs)													
Mean	0.5	1.1	1.5	1.2	1.1	1.1	19.4	18.8	6.3 H (1.2)	18.8	6.3 H (1.2)	0.3	5.8
Deepbench Suite - RNN bench - Inference (TensorCore) (10 inputs)													
Mean	3.4	3.2	6.6	5.0	3.6	4.3	18.8	13.3	5.7 H (2.1)	13.3	5.7 H (2.1)	0.1	6.0
Deepbench Suite - RNN bench - Train (TensorCore) (5 inputs)													
Mean	0.6	1.1	1.6	1.2	0.7	1.1	19.6	19.0	6.0 H (1.2)	19.0	6.0 H (1.2)	0.3	5.0
MLPerf Suite													
BERT Offline Inference	12.5	21564	*	*	*	*	*	29.51	0.4 H	29.51	0.4 H (1)	*	*
SSD Training	32.5	13000	*	*	*	*	*	35.9	4.5 H	28	0.5 M (500)	*	*
ResNet 50 64b Inference	3.2	1144	*	*	*	*	*	6.4	10 H	18	1.3 H (17)	*	*
ResNet 50 128b Inference	3.8	851	*	*	*	*	*	3.5	8 H	12	1.5 H (5)	*	*
ResNet 50 256b Inference	0.7	330	*	*	*	*	*	2.2	18 H	24	1.6 H (11)	*	*
GNMT Training	16.2	9630	*	*	*	*	*	17.0	36 H	39	25 H (1.4)	*	*
3D-Unet Inference	2.8	141	*	*	*	*	*	49.3	0.1 H	49.3	0.1 H (1)	*	*

syr2k, syrk, 2mm, 2Dcnn and others show large reductions in simulation time when PKP is applied. We experienced some issues with myocyte, where the profiling and tracing runs (necessary for Accel-Sim) ran a mismatched number of kernels.

Cutlass and Deepbench show little additional accuracy loss between full-simulation and PKS. The average simulation time of these workloads is 17 hours, while enabling PKS drops the average simulation time to 3 hours. The mean error across all Deepbench workloads is 15.0% in Accel-Sim, and 14.5% with PKS enabled, cutting simulation time in half. Here, PKA is generally more effective, since the kernels tend to be longer and there are fewer of them. In the Deepbench convolution training application, we experienced the same kernel-id mismatch we did with myocyte.

For the MLPerf workloads we can identify two situations. The first one is when the workload can be completely profiled with detail using Nsight Compute. This is the case with ResNets Inference and the 3D-Unet Inference, which in conjunction present a low average *Principal Kernel Selection* silicon error of 2% and a geomean speedup of 460×. The simulations show an average error of 15%, no speedup is reported, because these workloads cannot be simulated to completion. Therefore we just present the time to simulate in hours for *Principal Kernel Selection*, while the speedup of *Principal Kernel Analysis* is presented relative to *Principal Kernel Selection*. The second situation is when we cannot profile with detail, and use our two-level technique. These are Single Stage Detector (SSD) training, GNMT training (RNN translation), and BERT inference. These workloads have millions of kernels. The error is higher, an average of 20%, and the speedup is considerably larger.

Finally, the last two columns in Table 4 show the DRAM utilization (as a percentage) reported by full simulation and by using PKA to project DRAM utilization respectively. This analysis demonstrates that PKA can be used to project metrics other than just execution time. Only applications that complete in full simulation are listed to enable accuracy comparison between PKA and the simulator. With a few exceptions, the DRAM utilization predicted by PKA very closely matches the full simulation report.

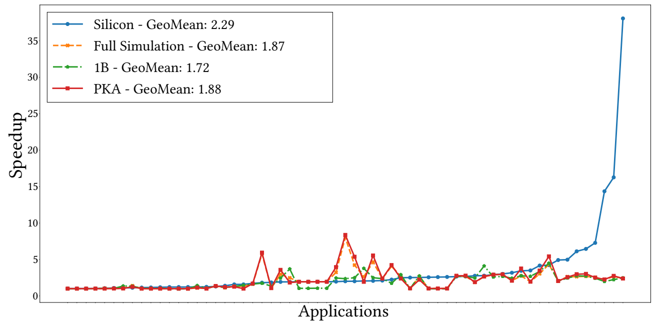


Figure 9: Volta V100 speedup over Turing RTX 2060 in silicon, when using full simulation and when using *Principal Kernel Analysis.*

5.3 Case Studies on Relative Accuracy

In this subsection, we explore the particular use-case architects care about when using simulators: If a change is introduced to the architecture, does the trend of the simulator match the trend of final hardware? To evaluate the effectiveness of *Principal Kernel Analysis* in this scenario, we perform two case studies where we measure the speedup of an architectural change and calculate the corresponding speedup measured by *Principal Kernel Analysis.*

In Figure 9, we evaluate the relative speedup of a high-end Volta V100 over a lower-end (but newer architecture) Turing RTX 2060. Note that not all the workloads are executable on the Turing card, in particular MLPerf, due to its limited memory capacity. Figure 9 demonstrates that *Principal Kernel Analysis* closely matches the predicted speedup of full simulation (although the baseline simulator has some inaccuracy). The simulator's inaccuracy is independent from *Principal Kernel Analysis*'s effectiveness.

To cover all the workloads, while still evaluating a silicon-validated architectural change, we halve the number of SMs on the V100 using Nvidia's Multi-Process Service (MPS). Figure 10 plots the resulting speedup of using 100% of the SMs over using 50% for silicon, full

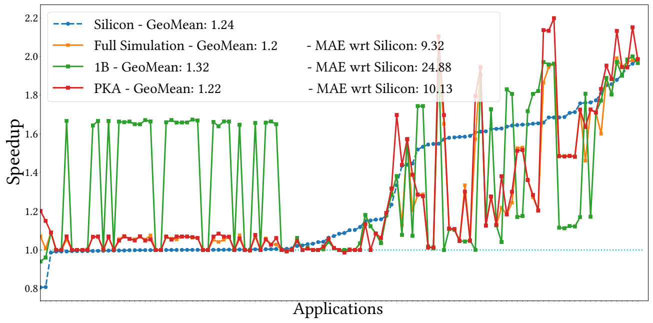


Figure 10: Speedup of using 80 cores over 40 cores in a Volta V100 evaluated in silicon, full simulation and using *Principal Kernel Analysis.*

simulation and *Principal Kernel Analysis*. Again, despite perturbations in the baseline simulator's accuracy, *Principal Kernel Analysis* tracks very closely to full simulation for the workloads that can be fully simulated. For the MLPerf workloads (for which there are no full simulations results), PKA's speedup error is less than 10%.

6 RELATED WORK

Prior work on CPUs showed that single-threaded applications could have varying performance characteristics over time [52] and that Basic Block Vectors can be used as an architecture-independent way for capturing these features [19, 46, 53]. Harmony [31] introduced parallel block vectors for summarizing the degrees of parallelism at the granularity of basic blocks. Eeckhout et al. [19] propose PCA plus clustering based workload characterization for CPU SPEC workloads. Phansalkar et al. [46] pointed out the importance of selecting microarchitecture-independent characteristics.

Later work on CPUs showed that sampling based on basic block vectors could be used to automatically characterize large-scale behaviors of programs with a direct application for improving simulation time [54]. Furthermore, this understanding of program behavior could be implemented in hardware at a minimal cost, and be used to detect phase-based program behavior [55].

Furthermore, Sherwood et al. [27, 54] showed that CPU phases are often stable over millions of cycles. This idea has been used by later works such as the Application Slowdown Model [57] for sampling performance metrics over time to represent large-scale behaviors. Tallam et al. [59] showed that traces of control flow could be used to capture the data dependencies in applications faster, for use in code optimization and dynamic program slicing.

Recent work on accelerating GPU simulation showed that parallelization of the simulator itself could improve simulation time between 2.5x and 3.5x compared to serial simulation for multiple GPUs [58]. Parallelization has also been used CPUs [51]. Pai et al. [43] predict kernel execution to improve performance and fairness at runtime for preemptive multi-kernel execution. They show that kernel execution time can be obtained by sampling, possibly as soon as a single thread block finishes execution. Recent work by Pati et al. [44] select representative iterations for sequence-based neural networks to drastically reduce the simulation time. Similarly, work by Villa et al. [62] details NVIDIA's solution to the same problem, unreasonably long simulation times. While the techniques presented in the paper are mostly focused on their proprietary hybrid execution-trace driven simulator, NVArchSim, their simulation methodology on MLPerf can be compared to *Principal Kernel Analysis*. Instead of running the entire MLPerf application, the authors run a single iteration and scale the results appropriately. We evaluated this methodology on ResNet from MLPerf and find that it yields accuracy comparable to *Principal Kernel Analysis*, but requires significantly longer simulation time. Simulating a single iteration would roughly take 3× times as long versus *Principal Kernel Selection* and 48× times as long versus *Principal Kernel Analysis* at a comparable accuracy. Although intuitive and practical in some cases, using a single iteration for machine learning workloads is not a general solution, as it requires contextual knowledge of the application, and has scaling issues as the network size increases.

7 CONCLUSION

This paper presents *Principal Kernel Analysis*, a silicon-verified mechanism to concisely represent GPU applications such that simulation times for scaled, real-world workloads are tractable. *Principal Kernel Analysis* is a hierarchical mechanism that reduces both the number of kernels executed through *Principal Kernel Selection*, and decreases the number of thread blocks executed in each kernel with *Principal Kernel Projection*. We demonstrate the effectiveness and generality of *Principal Kernel Selection* over 147 workloads across the three most recent Nvidia GPU generations, resulting in average speedup and error rates that range from an average of 7.2× @12.6% error across Rodinia to 1987× @28.5% error across 7 MLPerf applications. We then perform a case study applying *Principal Kernel Analysis* to Accel-Sim, demonstrating that the centuries-long simulation times for MLPerf can be reduced to a matter of hours with error rates that are in-line the baseline simulator.

ACKNOWLEDGMENTS

We thank the reviewers for precise and detailed feedback and the AEC for evaluating our artifacts. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), DARPA under award HR001119S0089-AMP-FP-034, the National Science Foundation (NSF) CCF #1910924, the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA, and generous gifts from Huawei and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides the entire environment to perform Principal Kernel Analysis (PKA). The simulation framework, benchmarks, and tools necessary to run PKA are all either included or generated by the Dockerfile. Scripts are offered to validate the results presented in the paper.

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

- **Algorithm: Principal kernel analysis**
- **Program: Python, CUDA**
- **Compilation: GCC 7**
- **Data set: Rodinia, Parboil, Polybench, MLPerf**
- **Run-time environment: Dockerfile**
- **Hardware: Nvidia V100, Nvidia RTX 2060, Nvidia RTX 3070**
- **Metrics: cycles**
- **Output: Table**
- **How much disk space required (approximately)?: 2 TB regular (+8 TB complete)**
- **How much time is needed to prepare workflow (approximately)?: 5 hours**
- **How much time is needed to complete experiments (approximately)?: 3 hours**
- **Publicly available?: Yes**
- **Workflow framework used?: GPGPU-Sim, Accel-Sim**

A.3 Description

We provide a Dockerfile to compose the experimental environment. Within the Dockerfile we offer options via commented blocks to download additional benchmarks and their associated traces. Since running all applications is impossible, we by default only enable all of those which can be simulated in under 3 hours. We note that the combined size of the traces and the visualizer logs for all suites can weigh in the order of 10 TB.

A.3.1 How to access. The scripts and the Dockerfile are included in the Zenodo record [3, 4]. The Dockerfile will generate an environment with the CUDA toolkit 11.2. It will pull the Accel-Sim-Framework and gpu-app-collection repositories from the Accel-Sim repository, set all the paths variables correctly and compile.

A.3.2 Hardware dependencies. For the silicon results, the experiment utilized three different GPUs: Nvidia V100, Nvidia RTX 2060 and Nvidia RTX 3070. For this artifact, we assume only the V100 is present.

For the simulation results a 1000-threaded XEON server was used.

A.3.3 Software dependencies. The host computer should have the CUDA 11.2 toolkit and compatible drivers installed, alongside nvidia-docker2. The newer versions of the profiler software Nsight Compute and Nsight Systems require special permissions to run, we defer to Nvidia's documentation on the matter.

A.3.4 Data sets. The data sets associated with the classic benchmarks will be downloaded. Certain MLPerf applications require special data sets that cannot be freely distributed, we defer to the documentation by MLPerf on steps to obtaining them.

A.4 Installation

Once the requirements of the CUDA toolkit 11.2, compatible drivers and nvidia-docker2 have been met, build the Docker environment by running the included make-docker.sh script.

```
$ . make_docker.sh
```

The dockerfile will pull an Nvidia container with CUDA-11.2 pre-installed and configured. The script installs everything required to run Accel-Sim and the benchmarks, including fetching data sets and compiling the benchmarks.

Once this initial setup is done, the traces will be downloaded. The total (uncompressed) size of the traces for the entire classic benchmarks suite is 5.5 TB, only the smaller ones will be downloaded ($\approx$ 9 GB compressed, 220 GB uncompressed). The rest of the traces can be obtained by uncommenting the option inside the Dockerfile.gpu file, and running make-docker.sh again. To run the benchmarks we use the run_docker.sh shell script, which runs the previously generated container.

```
$ . run_docker.sh
```

A.5 Evaluation and expected results

Running the shell script file Run_PKA.sh will generate the big table included in the paper (Table 4), alongside several pkl files containing the number of principal groups, the principal kernels associated with each group and their respective weights.

Note that since we assume only the V100 is included, only the columns associated with it will be populated. Some applications take months to simulate, so expect fewer applications to appear in the table. Simulating (almost) everything will require continuously running the container for a few months.

As explained in the paper, some cuDNN applications are expected to generate a different amount of kernels depending on the level of overhead experienced by the application. We believe that the cuDNN function cudnnFindConvolutionForwardAlgorithmEx is somewhat responsible. As a rule of thumb, if the number of kernels don't match, we exclude the workload.

A.6 Experiment customization

Because of the aforementioned issue of simulation run-time, we are by default only selecting those smaller classic workloads (Rodinia 3.1).

If the user wants to include more benchmarks, they can uncomment the blocks inside the Dockerfile.gpu associated with said benchmark, and re-run the make_docker.sh script. Once the other benchmarks are downloaded, the user can modify the Run_Updateable.sh shell script follwing the instructions indicated in said file. The user can also start the docker in interactive shell mode and configure things manually. The command is

```
$ Nvidia-docker run -it micro-2021-pka /bin/bash
```

A.7 Methodology

Submission, reviewing and badging methodology:

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

REFERENCES

[1] Aaron Ariel, Wilson WL Fung, Andrew E Turner, and Tor M Aamodt. 2010. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 164–174.
[2] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.
[3] Cesar Avalos. 2021. Micro2021 - Artifact - Principal Kernel Analysis Github Repo. https://github.com/cesar-avalos3/micro-2021-artifact
[4] Cesar Avalos. 2021. Micro2021 - Artifact - Principal Kernel Analysis Zenodo Repo. https://doi.org/10.5281/zenodo.5150378
[5] Baidu. 2017. DeepBench: Benchmarking Deep Learning Operations on Different Hardware. https://github.com/baidu-research/DeepBench
[6] Spyridon Bakas, Hamed Akbari, Aristeidis Sotiras, Michel Bilello, Martin Rozycki, Justin S. Kirby, John B. Freymann, Keyvan Farahani, and Christos Davatzikos. 2017. Advancing The Cancer Genome Atlas glioma MRI collections with expert

segmentation labels and radiomic features. *Scientific data* 4 (5 Sept. 2017). https://doi.org/10.1038/sdata.2017.117

[7] Spyridon Bakas, Mauricio Reyes, András Jakab, Stefan Bauer, Markus Rempfler, Alessandro Crimi, Russell Takeshi Shinohara, Christoph Berger, Sung Min Ha, Martin Rozycki, Marcel Prastawa, Esther Alberts, Jana Lipková, John B. Freymann, Justin S. Kirby, Michel Bilello, Hassan M. Fathallah-Shaykh, Roland Wiest, Jan Kirschke, Benedikt Wiestler, Rivka R. Colen, Aikaterini Kotrotsou, Pamela LaMontagne, Daniel S. Marcus, Mikhail Milchenko, Arash Nazeri, Marc-André Weber, Abhishek Mahajan, Ujjwal Baid, Dongjin Kwon, Manu Agarwal, Mahbubul Alam, Alberto Albiol, Antonio Albiol, Alex Varghese, Tran Anh Tuan, Tal Arbel, Aaron Avery, Pranjal B., Subhashis Banerjee, Thomas Batchelder, Kayhan N. Batmanghelich, Enzo Battistella, Martin Bendszus, Eze Benson, José Bernal, George Biros, Mariano Cabezas, Siddhartha Chandra, and Yi-Ju Chang. 2018. Identifying the Best Machine Learning Algorithms for Brain Tumor Segmentation, Progression Assessment, and Overall Survival Prediction in the BRATS Challenge. *CoRR* abs/1811.02629 (2018). arXiv:1811.02629 http://arxiv.org/abs/1811.02629

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[10] BM Beckmann and A Gutierrez. 2015. The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, and Somayeh Sardashti. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[12] John Burgess. 2020. Rtx on the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.

[13] Trevor Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. 2014. BarrierPoint: sampled simulation of multi-threaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software-ISPASS*. IEEE, 2–12. http://dx.doi.org/10.1109/ISPASS.2014.6844456

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[15] Jack Choquette. 2017. Volta: Programmability and performance. In *Hot Chips: A Symposium on High Performance Chips*.

[16] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany. 2021. The A100 Datacenter GPU and Ampere Architecture. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*.

[17] J Devlin and MW Chang. [n. d.]. M, K. Lee and K. Toutanova,Bert: Pretraining of deep bidirectional transformers for language understanding, 2018. *arXiv preprint arXiv:1810.04805* ([n. d.]).

[18] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 353–364.

[19] Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere. 2002. Workload design: Selecting representative program-input pairs. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 83–94.

[20] Igor V Evstigneev, Thorsten Hens, and Klaus Reiner Schenk-Hoppé. 2006. Evolutionary stable stock markets. *Economic Theory* 27, 2 (2006), 449–468.

[21] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. 2010. Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE, 1–10.

[22] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. 2016. TaskPoint: Sampled simulation of task-based programs. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 296–306. https://doi.org/10.1109/ISPASS.2016.7482104

[23] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012.*

[24] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism.*

[25] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[26] J. Huang, L. Nai, H. Kim, and H. S. Lee. 2014. TBPoint: Reducing Simulation Time for Large-Scale GPGPU Kernels. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 437–446. https://doi.org/10.1109/IPDPS.2014.53

[27] Canturk Isci and Margaret Martonosi. 2003. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 93–. http://dl.acm.org/citation.cfm?id=956417.956567

[28] Fabian Isensee, Jens Petersen, Simon A. A. Kohl, Paul F. Jäger, and Klaus Maier-Hein. 2019. nnU-Net: Breaking the Spell on Successful Medical Image Segmentation. *ArXiv* abs/1904.08128 (2019).

[29] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–24.

[30] 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*. IEEE, 76–86.

[31] Melanie Kambadur, Kui Tang, and Martha A Kim. 2012. Harmony: Collection and analysis of parallel block vectors. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 452–463.

[32] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[33] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollar. 2014. Microsoft COCO: Common Objects in Context. http://arxiv.org/abs/1405.0312

[34] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. https://doi.org/10.1007/978-3-319-46448-0_2 To appear.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[36] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, and Victor Bittorf. 2020. Mlperf training benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.

[37] B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, Y. Burren, N. Porz, J. Slotboom, R. Wiest, L. Lanczi, E. Gerstner, M. A. Weber, T. Arbel, B. B. Avants, N. Ayache, P. Buendia, D. L. Collins, N. Cordier, J. J. Corso, A. Criminisi, T. Das, H. Delingette, Ç. Demiralp, C. R. Durst, M. Dojat, S. Doyle, J. Festa, F. Forbes, E. Geremia, B. Glocker, P. Golland, X. Guo, A. Hamamci, K. M. Iftekharuddin, R. Jena, N. M. John, E. Konukoglu, D. Lashkari, J. A. Mariz, R. Meier, S. Pereira, D. Precup, S. J. Price, T. R. Raviv, S. M. S. Reza, M. Ryan, D. Sarikaya, L. Schwartz, H. C. Shin, J. Shotton, C. A. Silva, N. Sousa, N. K. Subbanna, G. Szekely, T. J. Taylor, O. M. Thomas, N. J. Tustison, G. Unal, F. Vasseur, M. Wintermark, D. H. Ye, L. Zhao, B. Zhao, D. Zikic, M. Prastawa, M. Reyes, and K. Van Leemput. 2015. The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS). *IEEE Transactions on Medical Imaging* 34, 10 (2015), 1993–2024. https://doi.org/10.1109/TMI.2014.2377694

[38] MLCommons. 2021. MLPerf v1.0 DataCenter Inference Result Guidelines. (2021). https://mlcommons.org/en/inference-datacenter-10

[39] MLCommons. 2021. MLPerf v1.0 Training Result Guidelines. (2021). https://mlcommons.org/en/training-normal-10/

[40] NVIDIA. 2018. CUTLASS: CUDA template library for dense linear algebra at all levels and scales. https://github.com/NVIDIA/cutlass.

[41] NVIDIA. 2019. NVIDIA Nsight CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[42] Mike O Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 41–54.

[43] Sreepathi Pai, R Govindarajan, and Matthew J Thazhuthaveetil. 2014. Preemptive thread block scheduling with online structural runtime prediction for concurrent GPGPU kernels. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 483–484.

[44] Suchita Pati, Shaizeen Aga, Matthew D. Sinclair, and Nuwan Jayasena. 2020. SeqPoint: Identifying Representative Iterations of Sequence-Based Neural Networks. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 69–80. https://doi.org/10.1109/ISPASS48437.2020.00017

[45] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, and Andrew Sun. 2004. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *In International Symposium on Microarchitecture*. IEEE Computer Society, 81–92.

[46] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy Kurian John. 2005. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005*. IEEE, 10–20.

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *EMNLP*.

[48] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, and William Chou. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[49] Timothy G Rogers, Daniel R Johnson, Mike O'Connor, and Stephen W Keckler. 2015. A variable warp size architecture. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 489–501.

[50] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vision* 115, 3 (Dec. 2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[51] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 475–486. https://doi.org/10.1145/2508148.2485963

[52] Timothy Sherwood and Brad Calder. 1999. Time Varying Behavior of Programs.

[53] T. Sherwood, E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 3–14. https://doi.org/10.1109/PACT.2001.953283

[54] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[55] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase Tracking and Prediction. *SIGARCH Comput. Archit. News* 31, 2 (May 2003), 336–349. https://doi.org/10.1145/871656.859657

[56] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[57] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–75. https://doi.org/10.1145/2830772.2830803

[58] 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 Kaeli. 2019. MGPUSim: Enabling multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[59] Sriraman Tallam and Rajiv Gupta. 2007. Unified Control Flow and Data Dependence Traces. *ACM Trans. Archit. Code Optim.* 4, 3, Article 19 (Sept. 2007). https://doi.org/10.1145/1275937.1275943

[60] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 335–344.

[61] 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 *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[62] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David W. Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021*. IEEE, 868–880. https://doi.org/10.1109/HPCA51647.2021.00077

[63] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[64] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. 2005. *TurboSMARTS: Accurate microarchitecture simulation sampling in minutes*. Technical Report. SIGMETRICS Performance Evaluation Review.

[65] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, and Klaus Macherey. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[66] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *in Proceedings of the 30th annual international symposium on Computer architecture*. 84–97.

[67] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy John, Hai Jin, and Chengzhong Xu. 2013. Accelerating GPGPU architecture simulation. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. 331–332.

[68] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy K John, Hai Jin, Cheng-Zhong Xu, and Junmin Wu. 2015. GPGPU-MiniBench: accelerating GPGPU micro-architecture simulation. *IEEE TRANSACTIONS ON COMPUTERS* 64, 11 (2015), 3153–3166. http://dx.doi.org/10.1109/TC.2015.2395427