Sieve: Stratified GPU-Compute Workload Sampling

Mahmood Naderan-Tahan Hossein SeyyedAghaei Lieven Eeckhout
Ghent University, Belgium

***Abstract*—To exploit the ever increasing compute capabilities offered by GPU hardware, GPU-compute workloads have evolved from simple computational kernels to large-scale programs with complex software stacks and numerous kernels. Driving architecture exploration using real workloads hence becomes increasingly challenging, up to the point of becoming intractable because of extremely long simulation times using existing architecture simulators. Sampling is a widely used technique to speed up simulation, however, the state-of-the-art sampling method for GPU-compute workloads, Principal Kernel Selection (PKS), falls short for challenging GPU-compute workloads with a large number of kernels and kernel invocations.**

This paper presents *Sieve*, an accurate and low-overhead stratified sampling methodology for GPU-compute workloads that groups kernel invocations based on their instruction count, with the goal of minimizing the execution time variability within strata. For the challenging Cactus and MLPerf workloads, we report that *Sieve* achieves an average prediction error of 1.2% (and at most 3.2%) versus 16.5% (and up to 60.4%) for PKS on real hardware (Nvidia Ampere GPU), while maintaining a similar simulation speedup of three orders of magnitude. We further demonstrate that *Sieve* reduces profiling time by a factor of 8× (and up to 98×) compared to PKS.

I. Introduction

GPUs are by far the most popular hardware accelerators today. General-purpose programming interfaces such as CUDA and OpenCL have opened up the vast compute capabilities GPUs offer towards GPU-compute applications beyond traditional graphics processing [30], [46], including high-performance computing (HPC) [9], [14], graph analytics [10], [48], and machine learning (ML) [28], [32], [34]. The rapid growth in raw hardware compute capabilities and software support has rendered the workloads ever more complex, consisting of multiple tens of kernels that are invoked multiple (tens or even hundreds of) thousands of times.

As GPU-compute workloads become increasingly more complex, they stress the performance evaluation methodologies computer architects use during early stages of the design cycle to their limit. At a typical simulation speed of 6 KIPS rate, the state-of-the-art GPU simulator, Accel-sim [26], can only simulate workload executions on the order of a couple milliseconds within a simulation time budget of a few hours. Unfortunately, real applications easily run for minutes, if not hours, rendering current simulation methodologies inadequate. As argued by Baddouh et al. [11], some of the recent MLPerf [34] workloads would take a century to simulate on current simulators. This is clearly impractical.

Researchers are well aware of the architecture simulation challenge and have developed a variety of techniques to speed up architectural simulation. Sampling, through which a limited number of representative regions are simulated, is a widely used methodology. While there exists a large body of work on sampled simulation for CPUs [16]–[20], [24], [38], [39], [51], sampling techniques specifically developed and tailored for speeding up GPU simulation have only recently received attention, see in particular [23], [25], [41], [44]. The state-of-the-art GPU workload sampling methodology, and most closely related work compared to ours, is Principal Kernel Selection (PKS) [11] which was shown to yield high accuracy and high speed for a variety of GPU-compute workloads. PKS first profiles all kernel invocations of a workload using a set of microarchitecture-independent execution characteristics, after which it groups kernel invocations in clusters based on similarity. PKS then selects a representative kernel invocation per cluster for simulation. Overall application performance is then estimated by computing a weighted performance figure across all representative kernel invocations.

Unfortunately, we find that although PKS is accurate and effective for many workloads, it fails to yield high accuracy for all workloads, and especially the more challenging workloads with a large number of kernels and kernel invocations. The reason is that, for these workloads, there is too high variability in the clusters from which PKS selects a representative kernel invocation, i.e., the different kernel invocations grouped within a cluster exhibit too high variability in execution cycle count for PKS to select an invocation that is representative for all invocations within the cluster. In addition, profiling the workload or collecting the dozen execution characteristics that serve as input to PKS, is time-consuming up to the point of becoming impractical.

In this paper, we propose *Sieve*, an accurate and low-overhead sampling methodology for GPU-compute workloads. *Sieve* is based on the simple and intuitive observation that different invocations of the same kernel most often lead to a (very) similar or even the same instruction count. *Sieve* groups kernel invocations into strata based on instruction count variability. The stratification process effectively 'sieves' kernel invocations into strata such that the execution time variability within the strata is minimized. Moreover, *Sieve* relies on a low-overhead profiling phase in which only a single microarchitecture-independent execution characteristic is collected per kernel invocation, namely instruction count.

We evaluate *Sieve* on two real hardware platforms (contemporary Nvidia Ampere and Turing GPUs), and demonstrate significantly higher accuracy for the challenging Cactus [31] and MLPerf [34] workloads compared to PKS, while achieving a similarly high (∼1,000×) simulation speedup. In particular, *Sieve* achieves an average prediction error of 1.2% (and at most 3.2%), versus 16.5% (and up to 60.4%) for PKS. Moreover,

profiling time is on average $8\times$ (and up to $98\times$) faster than PKS. We further show that *Sieve* accurately predicts relative performance differences across architectures, in contrast to PKS which yields misleading relative performance predictions in some cases. We provide the *Sieve* scripts and the identified representative kernel invocations for the Cactus [31], MLPerf [34], Rodinia [14], Parboil [40] and CUDA SDK [6] benchmark suites here: https://github.com/gpubench/sieve.

II. Prior Work and its Limitations

GPU simulation acceleration is not a new topic and some prior work has been done in this space. Principal Kernel Selection (PKS) [11] is the current state-of-the-art, and is most closely related to *Sieve*. We now describe the PKS method and its limitations. We discuss other related work in Section VI.

A. Principal Kernel Selection (PKS)

PKS made a substantial leap forward in GPU sampling compared to its own prior work [23], [25], [42], [44]. PKS operates as follows. It first profiles a workload of interest by collecting a wide variety of execution characteristics for each kernel invocation. The 12 characteristics that PKS collects are all microarchitecture-independent. PKS subsequently applies Principal Component Analysis (PCA) to reduce the dimensionality of the data set, and then uses Cluster Analysis (i.e., k-means clustering) to group the kernel invocations in this (reduced) multi-dimensional workload space. The basic intuition is to group kernel invocations into clusters such that all kernel invocations within a cluster feature similar execution characteristics. Note that different kernel invocations in the same cluster may originate from different kernels, as long as the execution characteristics are similar.

Once the clusters are formed, a representative kernel invocation is identified per cluster, and its relative weight is computed. The weight per cluster (representative) is computed based on the number of kernel invocations within the cluster. In other words, the more kernel invocations a cluster contains, the higher its weight.

Performance for the full workload execution is then estimated based on the detailed simulation of the representative kernel invocations only. Indeed, the representative kernel invocations are simulated, their respective performance numbers are computed (i.e., cycle count), and an overall performance prediction is obtained by computing a weighted sum across the performance numbers obtained for the representative kernel invocations. In other words, PKS computes a weighted sum of cycle counts across all representative kernel invocations, with the weights being the number of kernel invocations per respective cluster.

To further speed up the simulation, Baddouh et al. [11] complement PKS with a technique which they call Principal Kernel Projection (PKP) — PKS plus PKP then is their final proposal, namely Principal Kernel Analysis (PKA). PKP builds upon the observation that performance quickly converges within a kernel invocation, i.e., as the execution of a kernel invocation progresses over time, the overall performance number (e.g., instructions executed per cycle or IPC) quickly converges to its steady-state value. Hence, we do not need to simulate the entire kernel invocation, and a significant simulation speedup can be achieved by stopping simulation once the performance number has converged. According to the results obtained by Baddouh et al. [11], PKA is receiving the bulk of its speedup from PKS, however, for a few workloads PKP leads to additional speedups. In this work, we hence focus on PKS and discard PKP because (1) it is less effective, and (2) it is orthogonal to both *Sieve* and PKS, i.e., PKP can be applied to both techniques with similar benefits.

B. Limitations

Although Baddouh et al. [11] convincingly demonstrate PKS' accuracy and speed, there are a couple limitations which we overcome with *Sieve*. First, PKS is effective for many workloads, particularly relatively simple GPU-compute workloads from the Parboil [40], Rodinia [14] and CUDA SDK [6] benchmark suites. However, we find that PKS is inaccurate for more challenging workloads with a large number of kernels and kernel invocations. In particular, for the non-trivial workloads from Cactus [31] and MLPerf [34], we find that PKS incurs significant inaccuracy. The reason relates to how PKS characterizes kernel invocations. PKS groups kernel invocations based on microarchitecture-independent characteristics, and hence invocations from different kernels may be grouped within the same cluster. Moreover, because PKS predicts application cycle count as a weighted sum of the cycle counts of the cluster representatives, with the weights being the number of kernel invocations per cluster, PKS essentially assumes that all kernel invocations in the same cluster have the same execution time. We find though that there is significant variability in cycle count within clusters. Selecting a kernel invocation that is representative for the rest of the invocations in the cluster is hence challenging, or even (close to) impossible, as we will demonstrate in Section V.

Second, PKS, as mentioned before, characterizes kernel invocations by collecting 12 microarchitecture-independent execution characteristics. Collecting this many characteristics is time-consuming using existing profiling tools such as Nvidia's Nsight Compute [3]. For long-running workloads, profiling takes multiple days, and in some cases even several weeks. Although this is a one-time cost only, it is impractical because one needs to reserve an entire machine for multiple days or weeks to profile a single workload. To overcome this challenge, Baddouh et al. [11] propose two-level profiling in which they perform detailed profiling collecting the 12 characteristics for a first batch of kernels, followed by low-overhead profiling to collect the kernel names and grid dimensions for the remaining kernels in the workload. In our experiments we find that PKS profiling is indeed time-consuming, and substantially shorter profiling times can be achieved by collecting only a single execution characteristic, namely instruction count, and still achieve high accuracy, as we will demonstrate in Section V.

A last and more technical concern is that PKS relies on a golden reference obtained on a real hardware device to select the representative kernel invocations. More specifically, PKS relies on k-means clustering to group kernel invocations. Choosing the optimal number of clusters k is determined by calculating the error for each k (up to a maximum k of 20), and then selecting the one that minimizes the prediction error. The latter step, i.e., computing the prediction error for all k's, relies on a golden reference cycle count obtained on real hardware. This leads to the following concern. Although PKS uses microarchitecture-independent execution characteristics during profiling, it uses a specific hardware platform to perform the clustering, which makes the final selection of representative kernel invocations not truly microarchitecture-independent. In contrast, *Sieve* profiles kernel invocations and selects representatives *only* using a microarchitecture-independent characteristic, namely instruction count; by consequence, the representative kernel invocations are truly microarchitecture-independent. In other words, *Sieve* does not rely on a golden reference cycle count obtained on a real hardware platform. There is a caveat though that instruction count, as well as other microarchitecture-independent execution characteristics, might vary slightly across different GPU generations. However, it is expected that the variation in instruction count will be less than the variation in performance. In that sense, *Sieve* is more microarchitecture-independent compared to PKS.

C. Overcoming the Limitations

Sieve overcomes these limitations by only using a single microarchitecture-independent execution characteristic, namely instruction count, to characterize and categorize kernel invocations. Because of the low variability within each cluster, *Sieve*'s accuracy is high — significantly higher than PKS. Because we need to collect only a single execution characteristic, profiling is also fast — significantly faster than PKS. Furthermore, the achieved speedup for simulation of the representative kernel invocations obtained through *Sieve* remains high, and is comparable to PKS.

III. Sieve Sampling Method

We now describe how *Sieve* operates, see also Figure 1 for a high-level block diagram of the workflow. The first step is to profile the workload of interest. The profile information serves as input to stratification which selects and outputs the kernel invocations that are most representative along with their respective weights. These representative kernel invocations are then used for detailed simulation or workload analysis. We now discuss the various parts of the *Sieve* methodology.

A. Profiling

A GPU program typically consists of multiple kernels, and each kernel can be executed multiple times. We refer to different executions of the same kernel as *kernel invocations*. The profiling phase of *Sieve* collects the following information for each kernel invocation: kernel name, kernel invocation ID, and number of dynamically executed instructions. In other words, the profile essentially is a big table with as many rows as there are kernel invocations in the workload, and three columns per row (kernel name, kernel invocation ID, and instruction count).

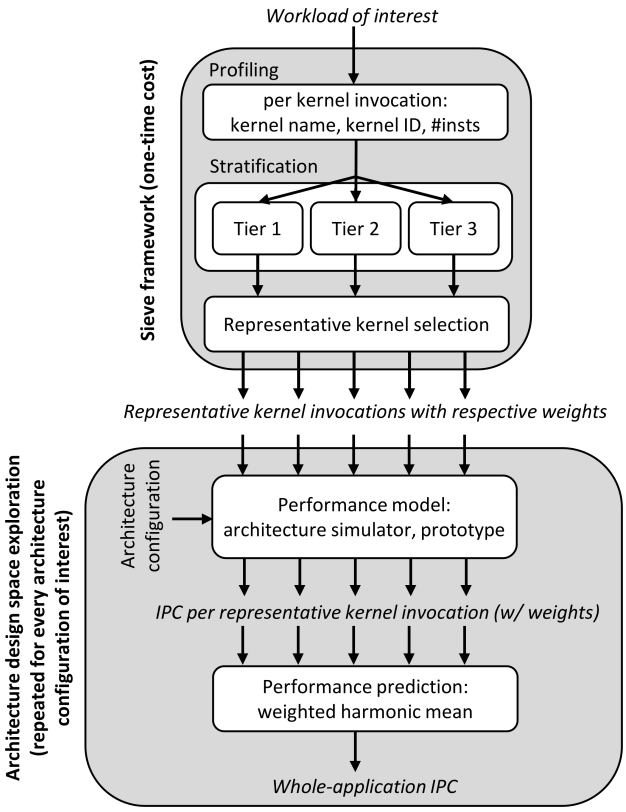

Fig. 1. Block diagram for *Sieve*. *Sieve selects representative kernel invocations with their respective weights for driving architecture design space exploration.*

Note that profiling is a one-time cost, and its purpose is to feed the *Sieve* back-end to select representative kernel invocations. The representative kernel invocations are then simulated multiple times during architecture design space exploration. Although profiling is a one-time cost, it can be extremely time-consuming. *Sieve* makes profiling practical by measuring only a single program characteristic that is easy to profile, namely dynamic instruction count. PKS on the other hand collects a dozen execution characteristics, which leads to impractically long profiling times of more than a month for some of the complex workloads considered in this work. *Sieve* substantially reduces profiling time compared to PKS as we will quantify in Section V.

B. Stratification

Once the profile information has been collected, the *Sieve* back-end selects the most representative kernel invocations. *Sieve* uses a simple, yet effective method to categorize kernel invocations into *strata*. *Sieve* considers all invocations of a kernel and categorizes them based on instruction count in such a way that each stratum consists of a collection of kernel invocations of the same kernel with the same or similar

dynamic instruction count. By doing so, *Sieve* effectively 'sieves' kernel invocations into strata with the following shared attribute: (1) being from the same kernel, and (2) featuring the same or similar instruction count. The intuition is that strata selected in this way show limited execution time variability because all kernel invocations within a stratum are executions of the same kernel *and* execute roughly the same number of instructions, i.e., the work performed by the kernel invocations within the same stratum is similar.

The stratification per kernel is based on the simple observation that different invocations of the same kernel often lead to the same (or very similar) instruction count. We hence categorize kernel invocations in three so-called *tiers*:

- **Tier-1:** There is no variation in the number of instructions across invocations. In other words, the kernel executes the exact same number of instructions across invocations.
- **Tier-2:** There is *little* variation in the number of executed instructions across invocations of the same kernel.
- **Tier-3:** There is *large* variation in the number of executed instructions across invocations of the same kernel.

Note that Tier-2 and Tier-3 represent variable instruction count across invocations, and they are distinguished by a threshold. We use the *Coefficient of Variation (CoV)* to quantify the variability in instruction count across invocations. CoV is defined as the standard deviation σ (i.e., the average squared differences with the mean) divided by the mean instruction count μ:

$$CoV = \frac{\sigma}{\mu}.$$

If the CoV is below a predefined and user-set threshold θ, the kernel belongs to Tier-2; otherwise, it belongs to Tier-3. The smaller the threshold θ, the less the variability within strata. This suggests higher accuracy but lower speed. We evaluate *Sieve*'s sensitivity to θ for accuracy and speed, and find that a threshold of $\theta = 0.4$ strikes a good balance between accuracy and speed. We hence consider $\theta = 0.4$ in our setup, unless mentioned otherwise.

Figure 2 categorizes the kernel invocations across the three tiers for the various Cactus and MLPerf workloads considered in this work for three threshold values θ. (See Section IV for details about our experimental setup.) Interestingly, the majority of kernel invocations fall within the Tier-1 and Tier-2 groups, meaning that there is relatively little variability in instruction count across invocations of the same kernel. On average, 41% of kernel invocations belong to Tier-1; 22%, 42% and 49% belong to Tier-2 for $\theta = 0.1$, $\theta = 0.5$ and $\theta = 1.0$, respectively. For some workloads, all kernel invocations belong to Tier-1 and Tier-2, as is the case for gms and lmr, even for the smallest thresholds; for gru, lmc, bert, resnet50, all kernel invocations belong to Tier-1 and 2 for the larger thresholds θ at 1.0 (or 0.5). The gst benchmark has the largest Tier-3 fraction above 50%. A couple other benchmarks have a Tier-3 fraction in the 10% to 50% range depending on the threshold θ.

Because of the large variability within Tier-3, we need to further stratify the invocations so that the variability within each stratum is small. We therefore use *Kernel Density Estimation (KDE)* [8], [36] which essentially groups kernel invocations in such a way that it (1) minimizes the number of strata, and (2) ensures that the variability in instruction count is less than a preset threshold. Again, we rely on the CoV for instruction count and we use the same threshold θ as before. The end result for the stratification process is that we end up with a number of strata or groups of kernel invocations that all feature the same or similar instruction count *and* originate from the same kernel. For this reason, we expect the execution behavior to be similar across all invocations within a stratum.

C. Representative Kernel Invocation Selection

Once the kernel invocations have been categorized, the next step is to select a representative invocation for each stratum. For Tier-1, this is straightforward because all kernel invocations have the same dynamic instruction count. We hence simply select the first-chronological kernel invocation. For Tier-2 and Tier-3, selecting a representative kernel invocation is a little more complicated because there is some variability within a stratum (although variability is small). We select the first-chronological invocation with the most dominant CTA[1] size, meaning that the selected kernel invocation occupies the available hardware resources in a representative way for the rest of stratum. (We also considered selecting the invocation with the maximum CTA size to better stress the GPU architecture, but we found this to be less accurate.)

In addition to identifying a representative invocation per stratum, we also need to compute its relative weight. We do this by computing the sum of the instruction count for all invocations within the stratum. Dividing the total instruction count per stratum to the total instruction count for the entire workload yields the stratum's weight, and thus the weight of its representative kernel invocation. As such, an invocation that represents a stratum with a high instruction count receives a higher weight.

D. Performance Prediction

The output provided by *Sieve* is the representative kernel invocations and their respective weights, which we subsequently use to drive architecture design space exploration. To predict overall application performance, the following two steps need to be conducted for each architecture configuration of interest. We first execute or simulate the representative kernel invocations and we compute their respective performance number, e.g., the number of instructions executed per cycle (IPC). Second, we compute the overall, application-level performance number by weighting the per-stratum IPC numbers with their respective weights. This is done by computing the weighted harmonic mean IPC as follows:

$$IPC = \frac{1}{\sum_{i=1}^{N} \frac{w_i}{IPC_i}},$$

[1]In Nvidia's terminology, a Cooperative Thread Array (CTA) refers to a thread block. This is similar to a workgroup in OpenCL's terminology.

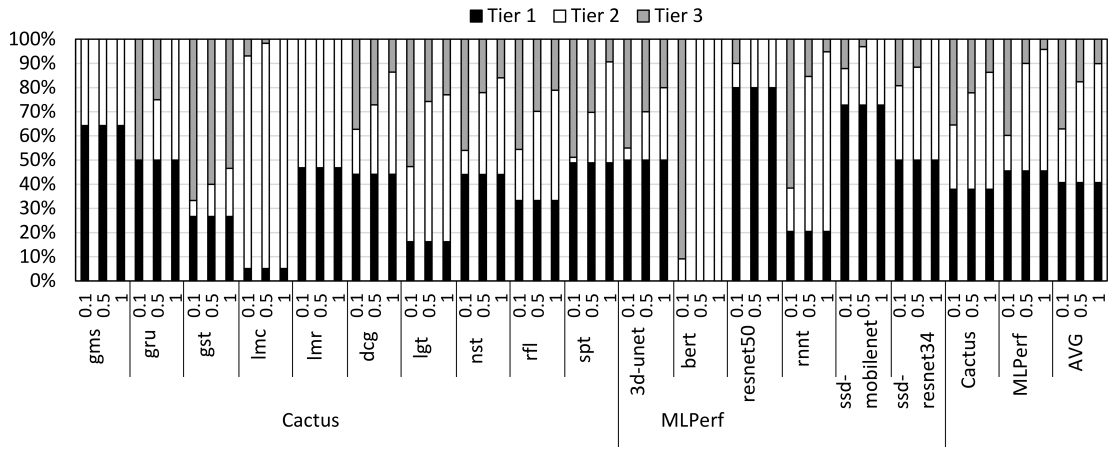

Fig. 2. Fraction kernel invocations belonging to Tier-1, Tier-2 and Tier-3 as a function of the threshold $\theta = 0.1$, $\theta = 0.5$ and $\theta = 1.0$. *Most kernel invocations below to Tier-1 and Tier-2, meaning that there is little to no variability in instruction count across invocations of the same kernel.*

with N the number of strata, and IPC_i and w_i the IPC and weight per stratum, respectively. Of course, the weights add up to one, i.e., $\sum_{i=1}^{N} w_i = 1$. Note that if the performance number per stratum would be CPI, we would need to compute the weighted arithmetic mean across the per-stratum CPI numbers with the weights based on instruction count.

E. Discussion

Note that *Sieve* categorizes kernel invocations into strata based on (1) kernel name and (2) similar (or same) instruction count. Hence, a kernel that is invoked only once (or only a few times) will be identified as a kernel to select (a) representative kernel invocation(s) from, even if its contribution to the overall execution time is small. PKS on the other hand may possibly cluster such kernels and invocations with invocations from other kernels because it does not rely on kernel names but on microarchitecture-independent characteristics. In other words, PKS may cluster invocations from different kernels but with similar characteristics. By doing so, PKS may possibly select fewer representative kernel invocations, thereby achieving higher simulation speedup compared to *Sieve*. In practice though, we find that *Sieve* achieves similar simulation speedup as PKS while being substantially more accurate.

IV. Experimental Setup

1) GPU Platform: To evaluate *Sieve*, we consider a modern-day high-end Nvidia RTX 3080 GPU system featuring 68 SMs, 10 GB of memory, and 760 GB/s of DRAM bandwidth. This GPU implements Nvidia's Ampere architecture [5]. In addition to this baseline architecture, we also consider a previous generation Nvidia RTX 2080Ti GPU with 68 SMs, 11 GB of memory, and 616 GB/s of DRAM bandwidth in some of our experiments. This GPU implements Nvidia's Turing architecture [7], and is used in addition to our baseline architecture to evaluate *Sieve*'s accuracy for predicting relative performance differences across architectures. The main libraries and tools we use in this work include the CUDA-11 driver, cuDNN-8 library and Nsight Compute 2022.

2) Benchmarks: We use a wide variety of (randomly selected) workloads from different benchmark suites, including Parboil [40], Rodinia [14], CUDA SDK [6], Cactus [31] and MLPerf inference [34], see Table I. In the evaluation, we will focus mostly on the Cactus and MLPerf benchmarks because they are the most challenging to sample featuring a large number of kernels and a large number of invocations per kernel. The benchmarks in the other suites, namely Parboil, Rodinia and SDK, are relatively easy to sample, and prior work (PKS) — alike *Sieve* — achieves high accuracy for those workloads because of the small number of kernels and invocations.

Due to infrastructure limitations, and because profiling is extremely time-consuming, especially for PKS, we do not consider full-application runs for the Cactus and MLPerf benchmarks — full-workload profiling for PKS is estimated (based on extrapolation) to take more than three weeks. In contrast, we consider and limit the total number of kernel invocations profiled for the Cactus and MLPerf benchmarks as listed in Table I; there are two exceptions, namely gst and gru, which are the shortest running benchmarks and for which we consider full runs. Overall, the large invocation count provides confidence that the Cactus and MLPerf benchmark experiments are representative. We profile all kernel invocations for the Parboil, Rodinia and CUDA SDK benchmarks.

3) Evaluation Methodology: We collect the data as follows. For Cactus, all applications are built and compiled following their documentation and makefiles, see [2]. For MLPerf, we use the Nvidia docker image v2.0 which is publicly available in the MLPerf repository [4]. Table II shows all the metrics used by PKS, versus the one metric, instruction count, used by *Sieve*. For each application, we run the profiler which produces a list of values (one for each metric of interest) per kernel invocation. The data is converted into a readable CSV file which serves as input to PKS and *Sieve*.

To compute sampling accuracy, we use cycle count per kernel invocation obtained on real hardware. This implicitly assumes perfect warmup, i.e., the cache and microarchitecture

TABLE I
WORKLOADS CONSIDERED IN THIS WORK FROM THE PARBOIL [40], RODINIA [14], CUDA SDK [6], CACTUS [31] AND MLPERF [34] BENCHMARK SUITES. THE NUMBER OF KERNELS AND KERNEL INVOCATIONS IS ALSO LISTED.

Suite	Workload	#Kernels	#Invocations
Parboil	bfs_ny	2	11
	histo	4	252
	lbm	1	3,000
	mri-g	9	51
	stencil	1	100
Rodinia	cfd	4	14,003
	dwt2d	4	10
	gaussian	2	16,382
	heartwall	1	20
	hotspot3d	1	100
	huffman	6	46
	lud	3	22
	nw	2	255
	srad	6	502
SDK	blackscholes	1	512
	cholesky	25	143
	gradient	7	84
	dct8x8	8	118
	histogram	4	68
	hsopticalflow	6	7,576
	mergesort	4	49
	nvjpeg	2	32
	random	2	42
	sortingnet	4	290
Cactus	gru	8	43,837
	gst	15	175
	gms	14	92,520
	lmc	58	248,548
	lmr	62	74,765
	dcg	59	414,585
	lgt	74	532,707
	nst	50	1,072,246
	rfl	57	206,407
	spt	43	112,668
MLPerf	3d-unet	20	113,183
	bert	11	141,964
	resnet50	20	78,825
	rnnt	39	205,440
	ssd-mobilenet	33	64,138
	ssd-resnet34	26	57,267

TABLE II
EXECUTION CHARACTERISTICS PROFILED BY PKS VERSUS SIEVE.

Execution characteristic	PKS	Sieve
Coalesced global loads	✓	
Coalesced global stores	✓	
Coalesced local loads	✓	
Thread global loads	✓	
Thread global stores	✓	
Thread local loads	✓	
Thread shared loads	✓	
Thread shared stores	✓	
Thread global atomics	✓	
Instruction count	✓	✓
Divergence efficiency	✓	
Number of thread blocks	✓	

state is perfectly warmed up at the beginning of each sample or representative kernel invocation. This is a reasonable assumption for long-running kernel invocations, which is the case for our workloads. (Studying the impact of warmup on sampling accuracy is left for future work.) PKS uses cycle count for the representative kernel invocations to predict cycle count for the entire application, as described in Section II-A. For *Sieve*, we combine cycle count and instruction count for the representative kernel invocations to predict the application's IPC, as described in Section III-D; dividing total instruction count with the predicted IPC yields the predicted cycle count.

We use the PKS scripts as made publicly available [1], and we implement *Sieve* through a separate set of scripts, which are available at https://github.com/gpubench/sieve. The error metric to quantify sampling accuracy is the same for PKS and *Sieve*, namely the absolute cycle count difference between the predicted cycle count and the total recorded cycle count normalized to the total cycle count:

$$Error = \frac{|C_{predicted} - C_{measured}|}{C_{measured}}.$$

Recall that *Sieve* predicts application IPC as the weighted harmonic mean of the IPC values across all representative kernel invocations, with the weights being total instruction count per cluster; predicted cycle count is then obtained by dividing total instruction count (which is known) with the predicted IPC. PKS predicts application cycle count as a weighted sum of the cycle counts for each cluster representative, with the weights being the number of kernel invocations per cluster. We obtain cycle count and IPC for each kernel invocation from real hardware execution; the golden reference, total cycle count, is also collected on real hardware. We thus evaluate Sieve (and compare it against PKS) through real silicon validation — not through simulation.

We quantify speedup as the ratio of the total cycle count for the entire workload execution divided by the total cycle count for all representative kernel invocations — speedup quantified as such is a measure for the speedup to expect when simulating the representative kernel invocations as opposed to the entire workload execution. Profiling time is defined as the total time it takes to collect all the metrics needed as input for PKS versus *Sieve*.

V. RESULTS

We now evaluate *Sieve* along a number of dimensions, in particular accuracy, speedup, and profiling time. We also evaluate *Sieve*'s accuracy for predicting relative performance differences across architectures. We mostly evaluate *Sieve* using the more challenging Cactus and MLPerf benchmarks, but also provide results for the other benchmark suites Parboil, Rodinia, and CUDA SDK. We compare *Sieve* against the state-of-the-art Principal Kernel Selection (PKS) approach [11] — more specifically, we compare against PKS-first which selects the first chronologically occurring kernel invocation per cluster, as advocated by Baddouh et al., and we evaluate sensitivity to other selection policies.

A. Accuracy

Figure 3 reports the prediction error for *Sieve* versus PKS. The overall conclusion is that *Sieve* is substantially more accurate than PKS. While PKS yields an average error of 20.4% and 16.0% for Cactus and MLPerf, and a maximum

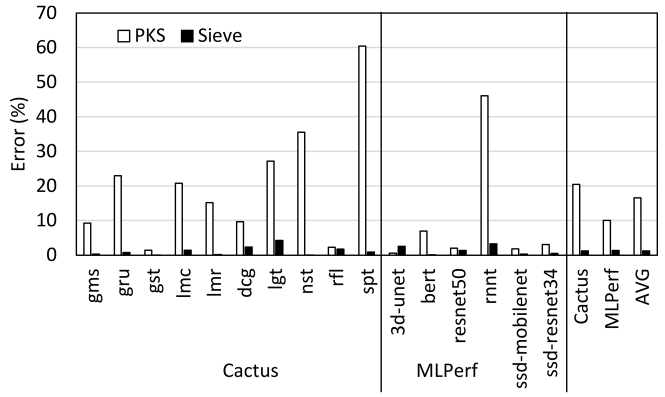

Fig. 3. Prediction error for *Sieve* and PKS. *Sieve is substantially more accurate than PKS: average error of 1.2% (at most 3.2%) for Sieve versus 16.5% (at most 60.4%) for PKS.*

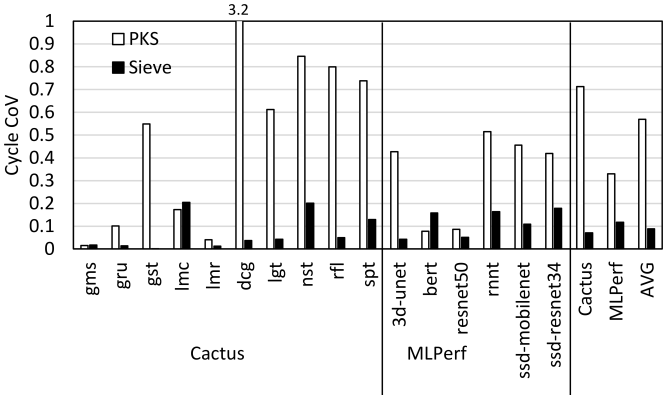

Fig. 4. Cycle count variability (CoV) within a cluster. *The degree of dispersion within each cluster is substantially smaller for Sieve compared to PKS.*

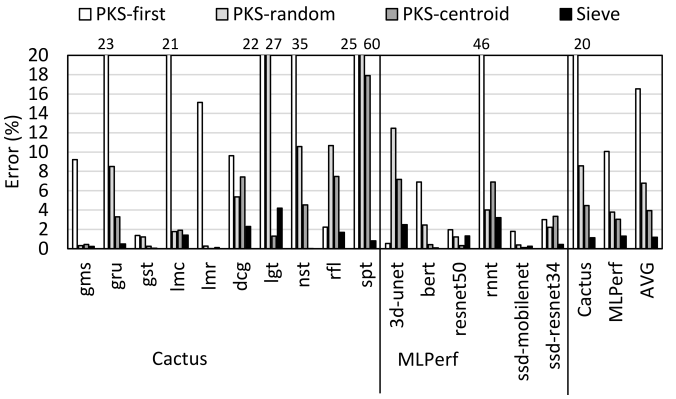

Fig. 5. Prediction error for different representative kernel invocation selection mechanisms for PKS. *Selecting kernel representatives differently does not close the accuracy gap between PKS and Sieve.*

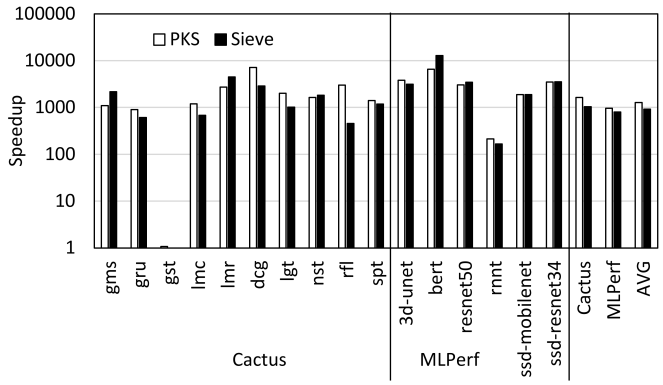

Fig. 6. Speedup for *Sieve* and PKS on a logarithmic scale. *Sieve and PKS achieve comparable speedup typically in the 100× – 10,000× range.*

error of 60.4% (spt) and 46.0% (rnnt), respectively, the error is substantially lower for *Sieve*: average error of 1.1% (and at most 4.1% in lgt) for Cactus, and 1.3% average error (and at most 3.2% in rnnt) for MLPerf. Across both Cactus and MLPerf, *Sieve* achieves an average error of 1.2% (at most 3.2%) versus 16.5% (and up to 60.4%) for PKS. The improvement in accuracy is remarkably consistent across workloads, especially for the Cactus workloads.

Figures 4 and 5 explain why *Sieve* is more accurate than PKS. Figure 4 reports the (weighted) average coefficient of variation (CoV) of cycle count within each cluster for PKS versus *Sieve*. The coefficient of variation is defined as the standard deviation divided by the mean, and is a measure for the degree of cycle count variability or dispersion within each cluster. The smaller the CoV, the smaller the dispersion, and hence the closer the values within a cluster are to the mean or centroid of the cluster. The CoV is substantially smaller for *Sieve* compared to PKS: the average CoV for *Sieve* equals 0.09 (at most 0.2 in lmc and 0.17 in ssd-resnet34), versus 0.57 (and up to 3.25 in dcg and 0.51 in rnnt) for PKS.

High dispersion within a cluster is only part of the reason for PKS' high prediction error. Figure 5 reports the error for different ways of selecting a representative kernel invocation per cluster. PKS by default selects the first kernel invocation per cluster as it occurs chronologically during workload execution [11]. The authors argue that the first chronological kernel invocation 'has practical advantages in reducing tracing and profiling time' while being equally accurate as selecting a representative kernel invocation closest to a cluster centroid. They further found that selecting of a cluster representative randomly leads to inconsistent errors. In contrast, we find that, at least for the Cactus and MLPerf workloads, the first chronological kernel invocation leads to a substantially higher error with an average error of 16.5% (and up to 60.4%), see also Figure 5. Note there is some correlation between the degree of dispersion and a high error for the first-chronological representative selection, see lgt, nst, spt and rnnt. Random selection leads to lower error: 6.8% average error (and up to 25.3% in spt), and so does centroid selection: 3.9% average error (and up to 17.9% in spt).

The overall conclusion is that *Sieve* is substantially more accurate than PKS, for two reasons. First, *Sieve* groups kernel invocations in clusters such that the per-cluster dispersion is substantially less. Second, the default selection of a cluster representative by PKS (first-chronological) is inaccurate for clusters with high dispersion. Improved representative kernel invocation selection techniques, such as random and centroid selection, are insufficient to close the gap with *Sieve* though.

B. Speedup

Sieve and PKS achieve approximately the same speedup, see Figure 6. The harmonic mean speedup across all benchmarks (excluding gst) achieved through *Sieve* equals 922×, versus 1,272× for PKS. Sieve selects one representative invocation for Tier-1 and Tier-2 kernels, and two or more representative invocations for Tier-3 kernels; hence there are more representative kernel invocations selected by *Sieve* than the number of kernels listed in Table I. PKS limits the number of representative kernels to at most 20, however, this is insufficient for achieving high accuracy. As a result, *Sieve* yields higher speedup for some workloads, while PKS achieves higher speedup for others. It is worth noting that the benchmarks for which PKS achieves a (slightly) higher speedup are also the benchmarks for which PKS yields a high error, see in particular spt (1,398× speedup for PKS vs. 1,183× for *Sieve* at an error of 60% for PKS vs. 0.83% for *Sieve*), and rnnt (213× speedup for PKS vs. 166× for *Sieve* at an error of 46% for PKS vs. 3.2% for *Sieve*). Overall, the speedup for both *Sieve* and PKS varies between 100× and 10,000×. The only exception is gst, which is why we excluded this benchmark when computing the mean speedup; the reason for the low speedup is that gst spends most (85%) of its execution time in a single kernel invocation with high variability in terms of instruction count and execution characteristics, and hence both *Sieve* and PKS are equally ineffective at achieving a significant speedup, i.e., both techniques select all invocations of that dominant kernel — a technique such as Principal Kernel Projection [11] could possibly be effective for both *Sieve* and PKS to reduce the runtime of individual representative kernel invocations.

C. Profiling Time

The third metric of key importance is profiling time. As previously argued, PKS collects as many as 12 execution characteristics for each kernel invocation, while *Sieve* only measures a single one, namely instruction count, see also Table II. Although profiling needs to be done only once for each workload, it can be a matter of practical concern if profiling time ends up requiring weeks or months of experimentation, as is the case for PKS. In particular, for the MLPerf workloads, PKS spends more than one month to collect the profile. *Sieve* is substantially faster with an average (harmonic mean) speedup of 8× and up to 98×, see Figure 7. It is further worth mentioning that the instruction count collected for *Sieve* can be calculated using a light-weight (and thus faster) instrumentation tool such as NVBit [45], versus a more complex (and thus slower) detailed profiler such as Nsight Compute [3]. (In particular, we observed that profiling using Nsight Compute becomes progressively slower as we profile an increasing number of kernels, i.e., profiling time increases super-linearly with the number of kernel invocations profiled.) Interestingly, the profiling time improvement for *Sieve* is higher for MLPerf compared to Cactus. The reason is the larger number of instruction types for the MLPerf benchmarks, which, combined with the multiple runs needed to collect all

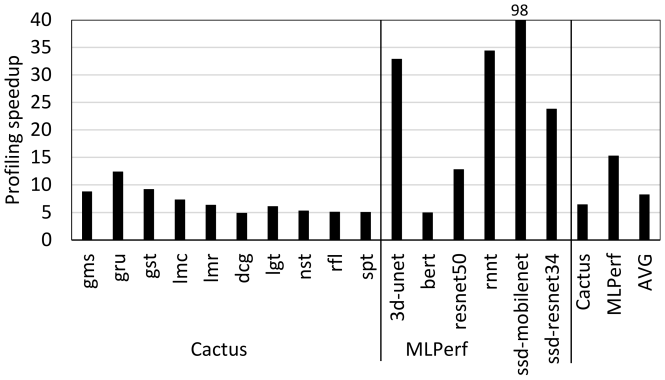

Fig. 7. Profiling time speedup. *Profiling takes substantially less time for Sieve compared to PKS.*

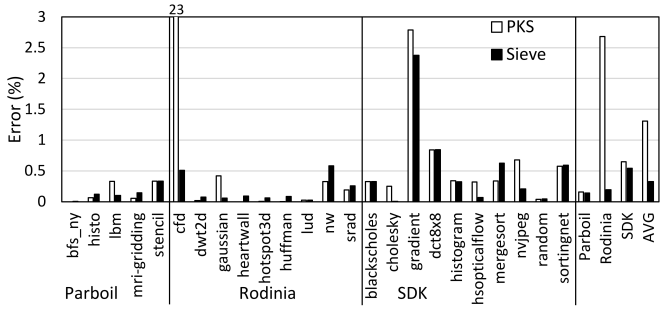

Fig. 8. Prediction error for *Sieve* and PKS in traditional benchmark suites. *Both methods achieve high accuracy: 0.32% average error (at most 2.3%) for Sieve versus 1.3% (and at most 23%) for PKS.*

12 execution characteristics using Nsight Compute and the overhead for saving and restoring memory across those runs, leads to a higher runtime overhead for PKS.

D. Other Workloads

So far, we focused on the more challenging workloads from the Cactus and MLPerf benchmark suites. Figure 8 reports the prediction error for the other workloads from Parboil, Rodinia and CUDA SDK. The overall conclusion is that while *Sieve* is more accurate than PKS, the error for PKS is also small except for one workload, namely cfd from Rodinia. We note an average error of 0.32% (at most 2.3%) for *Sieve* versus a 1.3% average error (and at most 23%) for PKS. The reason for the high accuracy for both *Sieve* and PKS is that these workloads are relatively simple compared to Cactus and MLPerf: they feature few kernels (for some workloads even a single kernel) and relatively few invocations per kernel, see also Table I. As a result, these workloads are relatively easy to sample and select a representative kernel invocation from, and both *Sieve* and PKS are adequate.

E. Relative Accuracy

So far, we focused on *Sieve*'s accuracy for a single GPU architecture, namely our baseline Nvidia Ampere GPU architecture. Of particular interest to computer architects is relative accuracy, i.e., how accurately can a performance analysis technique predict the relative performance difference between

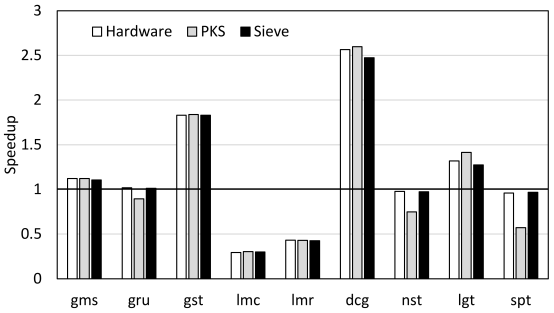

Fig. 9. Speedup for Ampere versus Turing. *PKS provides misleading relative performance results for some benchmarks in contrast to Sieve.*

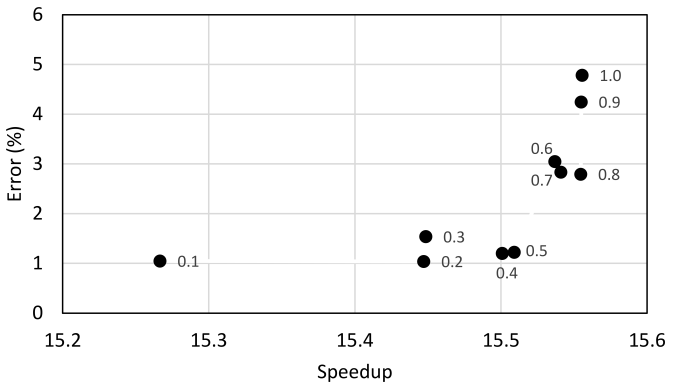

Fig. 10. Prediction error for *Sieve* as a function of speedup for different θ threshold values. *A threshold value below 0.5 yields low error.*

architectures. Figure 9 reports the relative performance difference between the Ampere architecture (RTX 3080) relative to the Turing architecture (RTX 2080Ti) for real hardware (the golden reference), PKS and *Sieve*. (Due to infrastructure limitations on the RTX 20280Ti we were unable to run the MLPerf workloads as well as Cactus' rfl.) The Ampere architecture achieves substantially higher performance than Turing for gst, dcg and lgt while being slower for lmc and lmr. *Sieve* accurately tracks the golden reference, in contrast to PKS, see in particular the spt, nst and gru benchmarks. *Sieve* predicts the speedup with an average error of 1.5% (and at most 3.5% for dcg), versus 9.8% for PKS, and up to 12.1% (gru), 23.5% (nst) and 40.3% (spt). In other words, while PKS may yield accurate speedup predictions for some workloads, it leads to (largely) inaccurate speedup predictions for others. The problem of course is that there is no way of knowing, i.e., the implication is that a computer architect may be misguided about the relative performance differences between architectures when using PKS, in contrast to *Sieve* which delivers accurate relative performance predictions.

F. Sensitivity Analysis

As aforementioned, *Sieve* relies on a threshold θ to decide whether a kernel invocation belongs to Tier-2 versus Tier-3. Figure 10 reports the average prediction error as a function of speedup for different values of the threshold θ obtained on our baseline RTX 3080 GPU system (similar results were obtained on the RTX 2080). We find that prediction error is sensitive to the θ value while speedup is (much) less sensitive. In particular, a threshold below 0.5 yields a low average prediction error below 1.6%. A threshold in the $[0.6 - 0.8]$ range leads to an average error around 3% while a threshold value of 1.0 leads to an average error of 4.8%. To balance prediction error and speedup, we assumed a threshold value of $\theta = 0.4$ in this work.

G. Simulation

The ultimate goal of a sampling method like *Sieve* is to prepare a reduced set of kernel invocations for simulation as opposed to simulating full-application runs. To this end, *Sieve*'s output is the selected kernel invocations. We have modified the Accel-sim [26] tracer, which uses the NVBit instrumentation tool, to only create the SASS trace of the selected kernel invocations. The traces are simple plain text files which are then simulated by Accel-sim on conventional CPUs. We have collected the traces for Ampere (both Cactus and MLPerf) and Turing (only Cactus), which we made available at the aforementioned *Sieve* repository. As each kernel invocation is a plain text file, it is possible to simulate a workload by dispatching each trace file to a separate core (i.e., parallel simulation), or simulate them one by one on a single core (i.e., serial simulation). For serial simulation, our experiments show that the simulation time of each workloads is less than 2 days (expect gst which takes 2.5 days) with instructions counts of 1 B as an average. For parallel simulation, however, the total simulation time is reduced significantly, and is determined by the longest-running kernel. Our experiments show that simulating the representative kernel invocations for most of the Cactus workloads (gms, gru, lmc, lgt, rfl and spt) takes less than one hour. Simulating each of the MLPerf workloads takes around 10 hours, and the longest-running workload is gst which takes around 30 hours.

VI. Related Work

Sampled simulation is a topic that has received broad attention over the past few decades, especially for CPUs. Various proposals have been proposed for sampling single-threaded workloads (i.e., random sampling [15], periodic sampling [50], and targeted sampling a.k.a. SimPoint [37]) as well as multi-threaded workloads (i.e., for server workloads [49] and synchronization-intensive workloads [12], [13], [35]). While most sampling methods consider fixed-length samples [43], some consider variable-length samples [27]. SimPoint computes basic block distributions for fixed-length instruction intervals. Similarities across instruction intervals based on the basic block distribution yields a limited set of representative instruction intervals or so-called simulation points. The execution characteristic used by *Sieve* is even simpler than what SimPoint uses, namely instruction count per kernel invocation.

Methodologies have been developed to identify similarities across CPU benchmarks. In particular, Eeckhout et al. [20] profile workloads using microarchitecture-independent characteristics and then use Principal Component Analysis (PCA)

and Cluster Analysis to identify a select number of representative benchmarks in the large workload space. Eeckhout et al. [19] combine PCA-based workload analysis with SimPoint to identify representative simulation points across benchmarks. The methodology proposed by PKS [11] is similar except that it targets GPU workloads and identifies representative kernel invocations across kernels from the same workload.

Prior work in GPU simulation acceleration has received recent attention. Yu et al. [52] generate synthetic miniature, yet representative, GPU-compute workloads based on basic block profiles. Kambadur et al. [25] use GTPin, a dynamic binary instrumentation tool for Intel GPUs, to identify representative $\sim$100 M instruction regions within OpenCL workloads using a broad set of features including kernel name, basic blocks executed, number of bytes read or written, etc. TBPoint [23] collects a broad set of execution characteristics (related to branch and memory divergence as well as thread block variation) obtained through functional simulation to then group kernel invocations through hierarchical clustering. PKS [11] follows a similar workflow while collecting a dozen microarchitecture-independent execution characteristics through profiling on native hardware, and while leveraging k-means clustering to scale to larger workloads. PKS suffers from curse-of-dimensionality issues where all kernel invocations are far away from each other in the 12-D workload space, making it hard to identify representative kernel invocations. We find that the only critical execution characteristic to profile is instruction count per kernel invocation as done by *Sieve*: this not only reduces profiling time, it also leads to substantially higher accuracy while maintaining similarly high simulation speedups. SeqPoint [33] identifies representative iterations in sequence-based deep neural network training workloads based on the iterations' input sequence length, which can be obtained without incurring profiling nor simulation overhead. While SeqPoint was developed to specifically accelerate DNN training simulation, *Sieve* is a more generally applicable sampling technique.

GPU modeling approaches other than synthetic workload generation and sampling-based simulation include analytical modeling, hybrid-abstraction simulation, and parallel simulation. A string of analytical GPU performance models have been proposed over the past decade with different capabilities, see in particular [21], [22], [29], [47]. NVArchSim (NVAS) [44] is the proprietary hybrid trace-driven simulator used by Nvidia in which different levels of abstraction (detailed versus high-abstraction timing models) are deployed to balance simulation speed and accuracy. MGPUSim [41] is a parallel simulator for modeling multi-GPU systems.

VII. Conclusion

Simulating contemporary GPU-compute workloads is a major challenge for academia and industry. This paper presented *Sieve*, a sampling methodology for GPU-compute workloads that (1) is substantially more accurate than the state-of-the-art PKS (average error of 1.2% versus 16.5%, and max error of 3.2% versus 60.4%, respectively); (2) yields comparably high simulation speedups (on the order of $\sim$1,000$\times$); and (3) incurs significantly less profiling overhead (8$\times$ average reduction and up to 98$\times$). While PKS is effective and accurate for many workloads, *Sieve* yields superior results for all workloads, in particular the more challenging workloads with a large number of kernels and kernel invocations.

Acknowledgements

We thank the reviewers for their valuable feedback. This work is supported in part by the UGent-BOF-GOA grant No. 01G01421, the Research Foundation Flanders (FWO) grant No. G018722N, and the European Research Council (ERC) Advanced Grant agreement No. 741097.

References

[1] "Artifact - principal kernel analysis github repp." https://github.com/cesar-avalos3/micro-2021-artifact, accessed: 2021.

[2] "Cactus benchmark suite for GPGPU," https://github.com/gpubench/cactus, accessed: 2022.

[3] "An interactive kernel profiler for CUDA applications," https://developer.nvidia.com/nsight-compute, accessed: 2022.

[4] "MLPerf inference v2.0 nvidia-optimized implementations," https://github.com/mlcommons/inference_results_v2.0/tree/master/closed/NVIDIA, accessed: 2022.

[5] "Nvidia Ampere GA102 GPU architecture," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, accessed: 2022.

[6] "Nvidia CUDA SDK code sample," https://docs.nvidia.com/cuda/cuda-samples/index.html, accessed: 2022.

[7] "Nvidia Turing GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, accessed: 2022.

[8] "Simple 1D kernel density estimation," https://scikit-learn.org/stable/auto_examples/neighbors/plot_kde_1d.html, accessed: 2021.

[9] M. Abraham, T. Murtola, R. Schulz, S. Páll, J. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19–25, 2015.

[10] M. Ahmad and O. Khan, "GPU concurrency choices in graph analytics," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[11] C. A. Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled GPU workloads," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021, pp. 724–737.

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

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

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[15] T. M. Conte, M. A. Hirsch, and K. N. Meneze, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the International Conference on Computer Design (ICCD)*, 1996, pp. 468–477.

[16] A. S. Dhodapkar and J. E. Smith., "Dynamic microarchitecture adaptation via co-designed virtual machines," in *Proceedings of the International Solid State Circuits Conference*, 2002, pp. 198–199.

[17] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002, pp. 233–244.

[18] L. Eeckhout, K. De Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2000, pp. 1–6.

[19] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2005, pp. 2–12.

[20] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload design: Selecting representative program-input pairs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002, pp. 83–94.

[21] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 280–289.

[22] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU performance modeling technique based on interval analysis," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014, pp. 268–279.

[23] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "TBPoint: Reducing simulation time for large-scale GPGPU kernels," in *Proceedings of the International Conference on Parallel and Distributed Processing Symposium*, 2014, pp. 437–446.

[24] A. Joshi, J. Yi, R. H. Bell, L. Eeckhout, L. John, and D. Lilja, "Evaluating the efficacy of statistical simulation for design space exploration," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006, pp. 70–79.

[25] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, "Fast computational GPU design with GT-Pin," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2015, pp. 76–86.

[26] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[27] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005, pp. 135–146.

[28] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[29] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022, pp. 424–436.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[31] M. Naderan-Tahan and L. Eeckhout, "Cactus: Top-down GPU-compute benchmarking using real-life applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2021, pp. 176–188.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Proceedings of the International Conference on Neural Information Processing Systems*, 2019, pp. 8024–8035.

[33] S. Pati, S. Aga, M. D. Sinclair, and N. Jayasena, "SeqPoint: Identifying representative iterations of sequence-based neural networks," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 69–80.

[34] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, M. J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf inference benchmark," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[35] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2022, pp. 604–618.

[36] D. W. Scott, *Multivariate Density Estimation: Theory, Practice and Visualization*. John Wiley & Sons, Inc., 1992.

[37] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 45—57.

[38] T. Sherwood, E. P. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001, pp. 3–14.

[39] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003, pp. 336–349.

[40] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[41] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGPUSim: Enabling multi-GPU performance modeling and optimization," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

[42] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.

[43] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Efficient sampling startup for sampled processor simulation," in *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, 2005, pp. 47–67.

[44] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chat-terjee, N. Jiang, and D. W. Nellans, "Need for speed: Experiences building a trustworthy system-level GPU simulator," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2021, pp. 868–880.

[45] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for nvidia GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.

[46] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the International Conference on Supercomputing*, 2008, pp. 1–11.

[47] L. Wang, M. Jahre, A. Adileho, Huawei, and L. Eeckhout, "MDM: The GPU memory divergence model," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014, pp. 1009–1021.

[48] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 2, pp. 1–49, 2017.

[49] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[50] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–97.

[51] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. John, H. Jin, and C. Xu, "Accelerating GPGPU architecture simulation," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013, pp. 331–332.

[52] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, "GPGPU-MiniBench: Accelerating gpgpu micro-architecture simulation," *IEEE Transactions on Computers*, vol. 64, no. 11, pp. 3153–3166, 2015.