MEGsim: A Novel Methodology for Efficient Simulation of Graphics Workloads in GPUs

Jorge Ortiz
Universidad de Murcia
Murcia, Spain
jorge.ortize@um.es

D. Corbalán-Navarro
Universidad de Murcia
Murcia, Spain
david.corbalan2@um.es

Juan L. Aragón
Universidad de Murcia
Murcia, Spain
jlaragon@um.es

Antonio González
Universitat Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

***Abstract*—An important drawback of cycle-accurate microarchitectural simulators is that they are several orders of magnitude slower than the system they model. This becomes an important issue when simulations have to be repeated multiple times sweeping over the desired design space. In the specific context of graphics workloads, performing cycle-accurate simulations are even more demanding due to the high number of triangles that have to be shaded, lighted and textured to compose a single frame. As a result, simulating a few minutes of a video game sequence is extremely time-consuming.**

In this paper, we make the observation that collecting information about the vertices and primitives that are processed, along with the times that shader programs are invoked, allows us to characterize the activity performed on a given frame. Based on that, we propose a novel methodology for the efficient simulation of graphics workloads called MEGsim, an approach that is capable of accurately characterizing entire video sequences by using a small subset of selected frames which substantially drops the simulation time. For a set of popular Android games, we show that MEGsim achieves an average simulation speedup of 126×, achieving remarkably accurate results for the estimated final statistics, e.g., with average relative errors of just 0.84% for the total number of cycles, 0.99% for the number of DRAM accesses, 1.2% for the number of L2 cache accesses, and 0.86% for the number of L1 (tile cache) accesses.

***Index Terms*—Simulation, GPUs, Graphics pipeline, Statistical simulation, Sampling, Clustering**

I. Introduction

Simulation tools and frameworks play an important role in computer architecture research. The main goal of simulation is to model and characterize new research ideas, estimate their performance improvements, measure their power consumption and evaluate, debug and more precisely understand the behavior of existing systems. A key factor of architectural simulations is the excessively long simulation time. In particular, cycle-accurate simulations require a huge amount of time. In the case of graphics processors and the simulation of graphics workloads, which involve complex scenes made up of hundreds of thousands of triangles (primitives) to be rendered, the simulation of a 500-frame video sequence (8 seconds of gaming at a conventional frame rate of 60 frames per second) of a representative Android game can take up to one day in a state-of-the-art simulator [1]. This becomes an important concern, especially when hundreds of simulations have to be carried out to explore a desired design space.

One common approach to address the long simulation times in architecture-level simulations of conventional CPUs is relying on sampling techniques to identify repetitive parts of the workload and then use a single sample of each repeating part to model the complete behavior of the application. SimPoint [2] is a well-known approach aimed at general-purpose processors that automatically finds a small set of simulation points that represent the complete execution of a general-purpose program. SimPoint builds on the idea that the behavior of a program at a given time is directly related to the code executed during that interval. In order to get this behavior information, SimPoint relies on the use of *basic blocks* by dividing the execution of a program in *intervals*, of e.g. 100 millions of instructions, and collecting the number of times each basic block is executed within each execution interval.

In the case of GPUs, which present a totally different architecture, techniques like SimPoint cannot be directly applied for several reasons. First, not all the stages of the graphics pipeline are programmable. Therefore, just by looking at the program code would be impossible to characterize the execution of some important phases like the *Tiling Engine*. Moreover, as a video game sequence is composed of a number of frames that have to be rendered, the use of frames is a more appropriate division when it comes to analyzing the execution of a graphics workload, instead of using fixed intervals of instructions.

Based on the above considerations, we propose a methodology to efficiently simulate graphics workloads in GPUs called MEGsim, a novel approach inspired by the underlying idea of SimPoint that is capable of automatically determining similarities between frames. MEGsim makes use of different parameters to characterize the execution of a graphics workload which are representative of the activity carried out along the different stages of the graphics pipeline (depicted in Figure 1).

To accomplish this, the information about the execution of *program shaders* is used. A shader is a user-defined program designed to run on a particular stage of the graphics processor. There are two types of shaders: *vertex shaders* and *fragment shaders*. The former can manipulate the attributes of vertices and determine the activity happening in the Geometry stages of the graphics pipeline. On the other hand, fragment shaders apply a shading and a lighting model to produce the final color of a pixel on the screen, so they determine the activity

 DOI: 10.1109/ISPASS55109.2022.00007

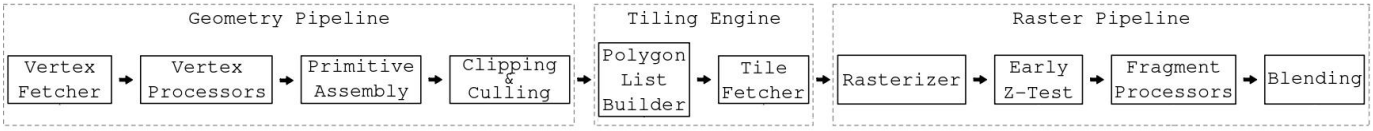

Fig. 1: Overview of the graphics pipeline of a mobile GPU implementing a Tile-Based Rendering architecture. For MEGsim, one parameter is selected from the different stages to characterize the execution of a graphics workload.

of the Raster stages in the graphics pipeline. We argue that the number of times that the different program shaders have been executed allows to characterize the behavior of each frame. In contrast with SimPoint, that relies on the use of basic blocks, the main difference between a shader and a basic block is that a shader can contain branches (i.e., more than one control flow). In fact, program shaders can be considered as supersets of basic blocks. In the case of GPUs, control flow analysis is not critical because threads are grouped into warps that proceed in a lock-step manner, and normally, both paths of conditional branches are executed, each one for a different subset of threads.

We also argue that information about program shaders alone is not enough for an accurate characterization of a frame, since it would lack information about another critical part of the graphics pipeline: the Tiling Engine. Therefore, MEGsim uses an additional parameter to characterize the activity of the Tiling Engine stages. In particular, the number of *primitives* in a frame has been determined to be a good proxy to characterize the activity done by the Tiling Engine on each frame.

A correlation study was carried out to ensure that all these parameters have an impact on the simulation metrics and our experimental results showed that the selected characterizing parameters have a high correlation coefficient with several key performance metrics such as the total number of cycles. Once the parameters to characterize a frame are identified, another key factor of the proposed simulation methodology is the weight given to each one of these parameters. We argue that these weights should be proportional to the activity performed on the different stages of the graphics pipeline for an average graphics workload.

The selected characterizing parameters are then combined to create a vector of characteristics that can be used to compare how similar two frames are. This information is used to group similar frames together by using a clustering algorithm. Our approach relies on the *k-means* method, which uses an iterative refinement technique that tries to minimize the variance. MEGsim also makes use of the Bayesian Information Criterion (BIC) [3] to measure the goodness of a fitting.

Different studies have been carried out to evaluate the accuracy and speedup achieved by MEGsim. Our experimental results show that, thanks to the use of the proposed MEGsim methodology, it is possible to reduce the amount of frames to simulate by a factor of 63× in the worst case, whereas in some applications the number of frames to be simulated is reduced by a factor as high as 165×. On average, MEGsim is capable of reducing the simulation time by two orders of magnitude (a factor of 126×) with a negligible impact on the accuracy of the simulation results. In particular, the average relative error of several key performance metrics is extremely low: 0.84% for the execution time (i.e., number of cycles), 0.99% for the number of main memory accesses, 1.2% for the number of L2 cache accesses and 0.86% for the number of Tile cache accesses. This shows the high accuracy and robustness of the proposed simulation methodology, given the extremely high accuracy in a variety of key performance metrics. As a comparison point, we have evaluated the accuracy and performance of simple random sub-sampling, showing that it requires 58.5× more frames to reach the same accuracy as MEGsim.

To summarize, the main contributions of this paper are:

1) The observation that it is possible to characterize the frames in a scene by using the input information of the different stages of the graphics pipeline.
2) A detailed proposal of MEGsim, a novel methodology that is able to exploit the aforementioned observation to perform efficient simulations of graphics workloads in GPU architectures.
3) An experimental evaluation of MEGsim that shows an average reduction of 126× in the number of frames to simulate with a relative error lower than 1% for different simulation output statistics.

The rest of the paper is organized as follows. Section II provides some background on mobile GPUs and simulation techniques, and presents some related work. Section III describes the proposed MEGsim simulation methodology whereas Section IV explains the evaluation approach we have followed. Section V reports the experimental results of our simulation methodology. Finally, Section VI summarizes the main conclusions of this work.

II. Background and Related Work

A. The Graphics Pipeline of GPUs

As depicted in Figure 1, the graphics pipeline in a typical GPU consists of two parts: the Geometry Pipeline, which is a front-end that transforms all the vertices of the scene from object-space coordinates to screen-space coordinates, generating all the corresponding *primitives* (typically triangles) that compose the whole scene; and the Raster Pipeline, a back-end that discretizes each one of the primitives into pixel-sized units called *fragments*, which are then applied a shading and a lighting model, texturized, and blended with other transparent fragments falling in the same screen position, to generate their final output color which is written into a frame buffer.

There are two main rendering modes: Immediate-Mode Rendering (IMR) and Tile-Based Rendering (TBR) [4]. In IMR, each primitive generated in the Geometry Pipeline is immediately sent to the Raster Pipeline for further pixel processing. Graphical objects that overlap with others generate a lot of fragments that are not visible in the final frame. Part of them are culled during a visibility checking stage like the Z-Test, which is based on depth checking. However, many other fragments cannot be culled incurring a large amount of useless work, an effect known as *overdraw* [5], [6]. In IMR, the problem is even worse because the colors of those eventually occluded fragments are not only computed but also written into main memory multiple times, thus increasing the main memory traffic and wasting energy. In contrast, TBR architectures completely avoid this useless memory traffic by splitting the screen into equally-sized small square regions called *tiles*, and processing them one at a time. Since each tile is small enough so that all its pixels may be stored in local on-chip memory, each pixel color is not transferred to main memory until the whole tile is rendered, and hence, each pixel is written only once.

Overall, Tile-Based Rendering (TBR) reduces off-chip memory accesses and is more energy efficient [7], [8], making TBR architectures more suitable for mobile GPUs. To be more precise, Figure 1 depicts the graphics pipeline of a TBR architecture. A brief description of the different stages in the pipeline follows. The first stage corresponds to the Vertex Fetcher which loads the vertices that compose the scene from main memory. Vertices are then sent to the Vertex Processors which apply a *vertex shader* to transform them from model-space coordinates to screen-space coordinates. The transformed vertices are sent to the Primitive Assembly stage where they are appropriately grouped into primitives (triangles) which are then clipped and culled before being passed over to the Tiling Engine.

In the Tiling Engine, the Polygon List Builder identifies the screen tiles overlapped by each primitive to generate, for each tile, a list of overlapped primitives. For each tile, its primitives are passed over to the Raster Pipeline. In the Rasterizer stage, primitives are discretized into fragments by interpolating the attributes of their vertices. Fragments are checked for visibility in the Early Z-Test stage [9] using depth information and those fragments which are determined to be visible are processed in the Fragment Processors by executing a *fragment shader* that applies a shading and a lighting model to produce their final color. Finally, output colors are processed in the Blending Unit to properly overlap with transparent, non-occluded fragments.

B. Overview of Simulation Techniques

Computer architecture simulators play an important role in advancing computer architecture research. A work in 2006 already highlighted the growth in the percentage of papers that were simulation-related [10]. Simulation techniques can be classified according to different criteria such as the detail of simulation, the scope of the target system, or the input to the simulator.

Regarding the simulation level of detail, two main classes can be distinguished: functional and timing simulators. A functional simulator implements the architecture of a processor at a high level and focuses on achieving the correct functionality and behavior of the processor that it models. On the other hand, timing simulators (also known as performance simulators) simulate the microarchitecture of a processor and produce detailed statistics about the performance of the target system. Finally, both timing and functional simulators can be integrated altogether to achieve a more comprehensive, flexible and accurate simulation model.

Depending on the scope of the target system to be simulated, simulators can be also classified into two types: full-system and application-level simulators. A full-system simulator is able to boot an OS and run application benchmarks on it, simulating all needed I/O devices, memory, network connections, etc. Alternatively, application-level simulators run only user-level applications instead of simulating the full stack, resulting in lighter and faster simulation times.

Simulators can be also categorized into two types depending on the input used to feed the simulator. Trace-driven simulators use trace files as input, i.e., pre-recorded streams of instructions (or any other information) generated during the actual execution of a benchmark. Differently, execution-driven simulators use binaries or executable files of benchmarks compiled for the simulated target machines.

Approximation techniques have also been used over the years to accelerate the simulation of individual components [11], [12]. E.g., by using simple core models (also known as 1-IPC core models) such as those implemented in Sniper [13] and CMP$im [14] if the interest is placed on evaluating the cache hierarchy or the memory system. Interval simulation has also been proposed to accelerate simulation times [15] by modeling particular miss events (cache misses, branch mispredictions) along with an analytical model to estimate the duration for every interval of instructions [16]–[18]. Sampling is another well-established technique to simulate a small but representative portion of a program's execution [2], [19], [20]. The next Section further elaborates on sampling-based approaches that can be found in the literature.

C. Related Work on Sampling-based Simulation

Sampling-based simulation measures only chosen sections (called sampling units) from a benchmark's full execution stream. The selection of sampling points can be done in two different ways [21]. On the one hand, statistical sampling can be applied to obtain periodic samples, as it is the case of SMARTS [19] and SimFlex [20] simulators. Alternatively, targeted sampling can also be used to obtain sampling points after analyzing the program's behavior. SimPoint [2] is a good example of the latter. It is a widely-known technique that calculates phases for a program/input pair, and then chooses a single representative from each phase. It relies on the use of basic blocks such that an interval of a program execution is characterized by the number of basic blocks that have been executed. A cluster of intervals constitutes a program phase

which can be characterized by a single interval designed to be representative of that phase. Sampling has proven to be a successful approach to summarize the behavior of a program, thus, allowing to effectively reduce the amount of information that has to be simulated.

In [22] it was analyzed the accuracy and speedup of different techniques that fall into these categories: reduced input sets, truncated execution and sampling. This work showed that SimPoint and SMARTS were the two sampling techniques that had the best trade-off between speed and accuracy. However, providing the Architectural State Starting Image (ASSI) is an important challenge associated with sampling-based simulation techniques. Some simulators make use of *checkpoints* [23] to avoid the use of functional warming, one of the main performance bottlenecks of sampling-based simulation. Other approaches rely on *fast forwarding*, by means of using a quick functional simulator to construct the ASSI [21].

Argollo *et al.* developed COTSon [24], an infrastructure for full-system simulation aimed at simulating a cluster with their associated devices connected through a standard communication network. More recently, Pati *et al.* developed SeqPoint [25], an approach that accurately characterizes the behavior of sequence-based neural networks by identifying some representative iterations. Flolid *et al.* proposed SimTrace [26], a methodology for representing a program's large scale over time phase behavior. Baddouh *et al.* proposed a methodology that is targeted for GPGPU to reduce the amount of work simulated in scaled GPU workloads [27]. They successfully applied a mechanism called *Principal Kernel Analysis* that not only reduces the number of executed kernels but also decreases the number of thread blocks executed in each kernel.

Different from the aforementioned simulation approaches, our proposed MEGsim is the very first sampling-based methodology for the efficient simulation of graphics workloads in GPUs.

III. MEGsim: a Novel Sampling-based Simulation Methodology for Graphics Workloads

A. Overview

This paper proposes a novel Methodology for Efficient GPU simulation (MEGsim), a fast and automatic approach that accurately determines statistically representative frames in a whole video sequence rendered by the execution of a graphics workload. MEGsim makes use of several parameters to characterize the activity carried out along the different stages of the graphics pipeline, which are combined to create a matrix of characteristics for each frame of the sequence. This information is used to group similar frames together by using a clustering algorithm. Finally, the best clustering that characterizes the full execution of the graphics workload is determined.

B. Input Parameters

In order to characterize the activity of the graphic pipeline, some information about the execution of *program shaders* is used. As described in Section II-A, a shader is a program designed to run on some stages of the graphics pipeline, having two types of shaders depending on where they are used. Vertex shaders manipulate the input geometry of a scene, therefore, they determine the activity that takes place along the Geometry Pipeline, whereas fragment shaders calculate the output color of each fragment inside a primitive, therefore, they determine the activity happening along the Raster Pipeline (refer to Figure 1).

In a first step, information about all shaders that are used in a given graphics workload is collected by performing a fast functional simulation. This includes the total number of instructions executed by each shader. In addition to that, the number of times that each shader is executed in every frame is also collected and stored into two vectors: VSCV (*Vertex Shader Count Vector*) and FSCV (*Fragment Shader Count Vector*). Each element in the combined vector corresponds to the count of how many times a shader has been executed in a specific frame, multiplied by the number of instructions in that shader. It is important to note that texture accesses are weighted according to the number of memory accesses they generate, depending on the texture filtering type they utilize. I.e., texture accesses using linear filtering are weighted with a value of 2 (as they perform 2 memory accesses); bilinear-filtering texture accesses use a weight of 4; whereas trilinear-filtering texture accesses use a weight of 8.

However, if only information about program shaders were used, there would be no proper characterization about another critical part of the graphics pipeline: the Tiling Engine. To account for the activity of the Tiling stages, the number of primitives within a frame is utilized. As depicted in Figure 2, this information (PRIM) is collected and appended to the other two vectors (VSCV and FSCV) to form a single vector of characteristics for a given frame. Therefore, assuming a graphics workload of N frames and a vector of characteristics of length D, the input dataset for MEGsim is a $N \times D$ matrix.

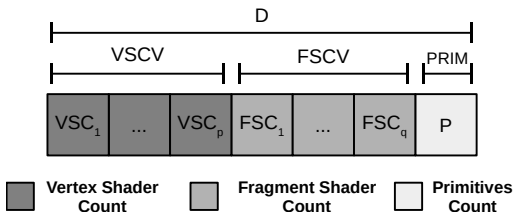

Fig. 2: Vector of characteristics for a particular frame where p corresponds to the number of vertex shaders and q to the number of fragment shaders.

An important advantage of choosing these parameters to characterize graphics workloads is that they are totally independent of the underlying GPU architecture and they can be easily obtained by performing a functional simulation. These kinds of simulators have much lower execution times compared to cycle-accurate simulators. This way, obtaining the input parameters needed by MEGsim is a very fast process and does not impose a critical burden.

A correlation study has been carried out to ensure that all these input parameters have an impact on one of the key simulation metrics: the total number of cycles. In the case of primitives, the correlation could be calculated by using only the Pearson's correlation coefficient, as they are defined as one-dimensional column vectors ($\text{PRIM}_{1..N}$). The Pearson's correlation coefficient is calculated as follows:

$$\rho = \frac{\text{cov}(X, Y)}{\sigma_x \sigma_y} \tag{1}$$

However, the information about shaders execution is represented using multi-dimensional vectors. Therefore, a different method has to be used: the coefficient of multiple correlation. This coefficient is a measure of how well a given variable can be predicted using a linear function of a set of other variables and can be calculated with the following formula:

$$R^2 = c^T * {R_{xx}}^{-1} * c \tag{2}$$

where c is the vector of correlations $r_{x_n y}$ between the predictor variables x_n (shader count vectors) and the target variable y (the number of total cycles); and ${R_{xx}}^{-1}$ is the inverse of the following matrix:

$$R_{xx} = \begin{pmatrix} r_{x_1x_1} & r_{x_1x_2} & \cdots & r_{x_1x_N} \\ r_{x_2x_1} & \ddots & & \vdots \\ \vdots & & \ddots & \\ r_{x_Nx_1} & \cdots & & r_{x_Nx_N} \end{pmatrix} \tag{3}$$

where $r_{x_i x_j}$ represents the correlation coefficient between the shader counts for frames i and j.

Figure 3 shows the results obtained in this correlation study. It can be observed that the characterizing input parameters, especially the shader count, have a high correlation with the total number of cycles of the simulation. However, the number of primitives (PRIM) has a more limited impact on such metric. Therefore, the three components of the per-frame vector of characteristics (VSCV, FSCV, PRIM) must be weighted differently as they represent a different amount of activity along the graphics pipeline.

C. Input Parameters Normalization

To determine the weight for each component of the vector of characteristics, we have measured the dissipated power in each of the three main phases of the graphics pipeline (Geometry Pipeline, Tiling Engine and Raster Pipeline) as a proxy of the activity performed on each pipeline's phase. Figure 4 shows the results obtained for the evaluated benchmarks. As expected, the Raster stages (those in charge of rendering and calculating the final color of each individual fragment) are the ones with the highest power dissipation, with an average power fraction of 74.5%. The Geometry stages are responsible for 10.8% of the power dissipation on average, whereas the Tiling stages are responsible for 14.7% of the power dissipation. Therefore, these values are used as weights (0.108, 0.745, 0.147) for the three components of the vector

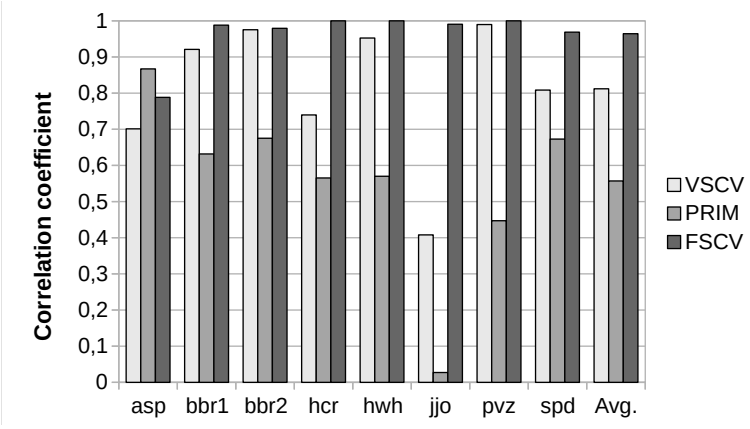

Fig. 3: Correlation coefficient between the input parameters and the total number of cycles for the evaluated benchmarks (described in Table II).

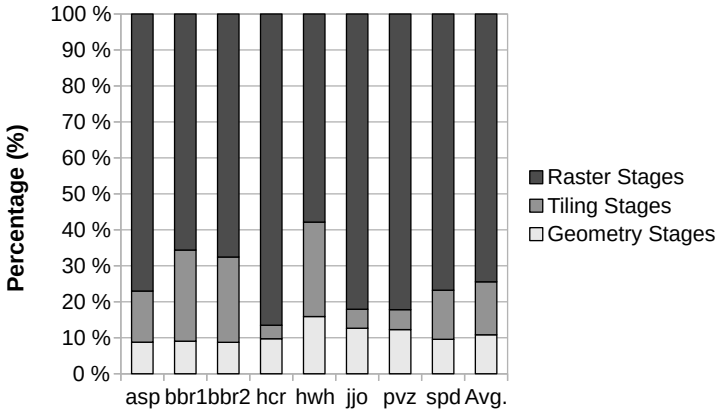

Fig. 4: Fraction of dissipated power in the three main phases of the graphics pipeline (Geometry, Tiling and Raster).

of characteristics (VSCV, FSCV, PRIM), respectively. To do that, a per-column normalization is performed by adding all the values within each group of characteristics which are then weighted accordingly.

D. Similarity Between Frames

After collecting the vector of characteristics for the different frames, it is possible to start finding similarity patterns in the video sequence. To determine how similar two frames are, we calculate the Euclidean distance between their vectors of characteristics. Two similar frames will have an Euclidean distance close to 0.

Given a sequence of N frames, a *Similarity Matrix* can be created in order to find how frames relate to each other. A Similarity Matrix is an upper triangular $N \times N$ matrix, similar to the one used in SimPoint, where the cell (x, y) represents the Euclidean distance between the vector of characteristics of frame x and that of frame y. The Similarity Matrix can be graphically plotted as shown in Figure 5 where the darker the points, the more similar the two corresponding frames are

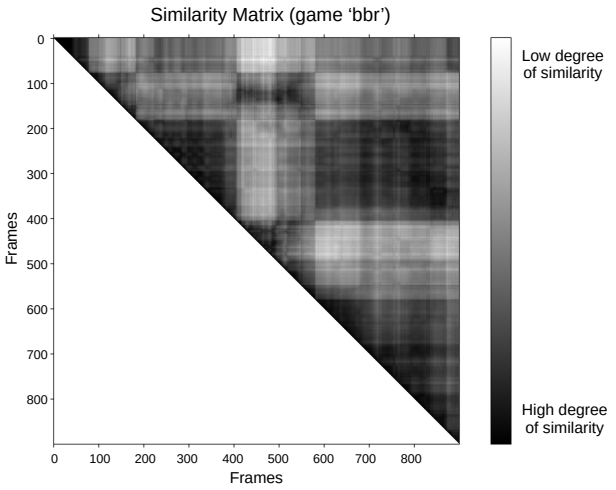

Fig. 5: Similarity Matrix for the benchmark Beach Buggy Racing (*bbr*) with 900 frames analyzed. The darker a point is, the more similar the two corresponding frames are.

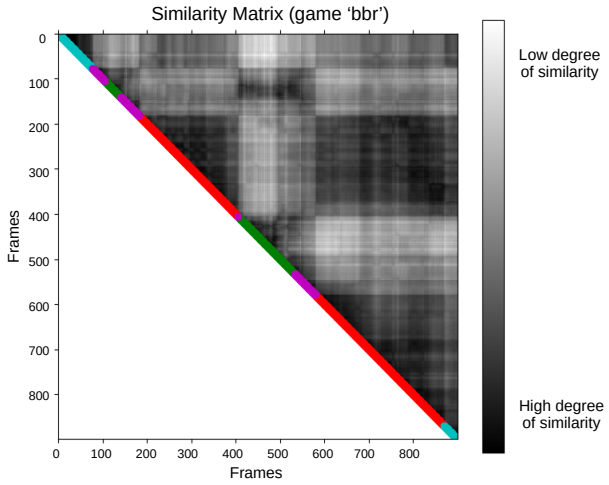

Fig. 6: Clusters obtained for the *bbr* benchmark by *k-means*. The 4 identified clusters are drawn along the matrix diagonal.

(i.e., the Euclidean distance is closer to 0). Note that points along the diagonal axis are always 0 (plotted in black) as they represent the similarity between a frame and itself.

As it can be seen in Figure 5, there are different regions or patterns in the benchmark Beach Buggy Racing (*bbr*). For example, starting from point $(200, 200)$ and tracing a horizontal line, we can see a dark region until we reach the point $(400, 200)$. This means that all the frames between 200 and 400 are very similar. Conversely, starting from point $(400, 300)$ and tracing another horizontal line, we can see a light region until we reach the point $(600, 300)$. This means that frame 300 is very different from frames 400 to 600.

E. Clustering

Looking at the example of the *bbr* benchmark represented in Figure 5, it can be seen that there exist different regions of similar frames. The idea of clustering is to group frames that exhibit similar behavior.

Clustering algorithms are used to partition *N* observations into k clusters. MEGsim uses the method *k-means* to cluster frames. *K-means* uses an iterative refinement technique that tries to minimize the within-cluster sum of squares (WCSS), i.e., the cluster variance. Given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional vector of characteristics, *k-means* aims to partition the n observations into k $(<= n)$ sets $(S_1, S_2, ..., S_k)$, so as to minimize their WCSS. Formally, the objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i \quad (4)$$

where $\boldsymbol{\mu}_i$ is the mean (a.k.a. centroid) of the points in set S_i.

In our case, we would obtain: 1) a vector of length k with all the clusters' centroids; and 2) a vector of length N containing all the labels for the different frames (i.e., the assignation of the frames to the corresponding cluster). Next, the distances between the centroids and all of the frames within each cluster are calculated. The selected frame for a cluster is the one with the lowest distance and it will constitute the cluster's representative, i.e., only this frame needs to be simulated and the obtained output statistics will be scaled according to the total number of frames that are included in that cluster. Figure 6 shows the clustering result for the *bbr* benchmark, where 4 clusters have been identified, represented with different colors along the diagonal.

F. Scoring and Selecting Clusters

There are many techniques to determine if a set of clusters is a good fit for the data. MEGsim makes use of the Bayesian Information Criterion (BIC) [3]. BIC is a penalized likelihood that offers a measure of the goodness of a fit. By adding new parameters to a model, e.g., by increasing the number of clusters, the likelihood of having a better fit increases. However, it is often desirable to obtain a fit that also minimizes the number of selected clusters to represent the whole dataset. To address this issue and achieve a good trade-off, BIC introduces a *penalty* term that grows as the number of clusters increases. We calculate the BIC score using the formulation given in [28], [29] as:

$$BIC(\phi) = \hat{l}_\phi(D) - \frac{p_\phi}{2} \cdot \log R \quad (5)$$

where $\hat{l}_\phi(D)$ is the likelihood (defined below) and p_ϕ is the number of parameters to estimate (defined as $K * (d + 1)$). The likelihood is calculated as:

$$\hat{l}(D) = \sum_{n=1}^{K} R_n \log R_n - R \log R - \frac{RM}{2} \log\left(2\pi\sigma^2\right) - \frac{M}{2}(R - K) \quad (6)$$

where R is the number of points in the data, R_i is the number of points included in the i^{th} cluster, K is the number of clusters, d is the dimension of the vector of characteristics, and σ^2 is the average variance of the Euclidean distance from each point to its cluster's centroid.

To determine the number of clusters, MEGsim starts with a single cluster (comprising all the frames) and iteratively increases this value. For every cluster, the BIC score is calculated and the algorithm stops when a BIC score lower than the previous one is obtained. Finally, the algorithm chooses the clustering that achieves a BIC score that is at least 85% of the spread between the largest and the smallest BIC score. We have denoted this threshold value as T. This threshold is empirically chosen and represents a trade-off between the achieved accuracy and the number of clusters. A higher value of T will result in a higher BIC score, leading to more accurate results at the expense of increasing the number of clusters. On the other hand, smaller values of T will lead to fewer clusters but it will also reduce the achieved accuracy.

IV. Evaluation Methodology

A. GPU Simulation Framework

To evaluate our proposal we employ TEAPOT [1] a cycle-accurate GPU simulation framework that allows to run unmodified Android applications and evaluate the performance and energy consumption of the GPU and the memory system. Table I shows the parameters employed in our simulations, which model an architecture resembling an Arm Mali-450 GPU [30]. We have chosen this particular mobile GPU as a proof-of-concept to evaluate MEGsim since the Mali 400 MP series is one of the most deployed Mali GPUs and still represents a good fraction of the mobile GPU market. Note, however, that our methodology relies on parameters that are application dependent and related to the scene's geometry and the program shaders, which makes MEGsim independent of the underlying GPU architecture. As such, it can be easily extended to support newer GPUs by incorporating information about additional pipeline phases (e.g., a Hidden Surface Removal phase as implemented by TBDR –deferred rendering– GPUs [5]) or recent GPUs that incorporate *mesh shaders*.

TEAPOT is comprised of three main components: an OpenGL trace generator, a GPU functional simulator, and a GPU cycle-accurate simulator. The workloads are executed in the Android Emulator deployed in the Android Studio [31]. While the application is running, the OpenGL trace generator intercepts and stores all the OpenGL commands that the Android Emulator sends to the GPU driver. The generated OpenGL commands trace is then used to feed an instrumented version of Softpipe. Softpipe is a software renderer included in Gallium3D, a well-known architecture for building 3D graphics drivers. Our instrumented Softpipe executes the OpenGL commands and creates a GPU trace including information of the different stages of the graphics pipeline (memory accesses, shader instructions, vertices, fragments, etc). The GPU trace is used by the cycle-accurate simulator, which gathers activity factors of all the components included in the modeled TBR architecture and reports timing as well as power consumption. Regarding the power model, McPAT [32] provides energy estimations for the processors and the caches included in the GPU. The main memory and the memory controller are simulated with DRAMsim2 [33].

TABLE I: GPU simulation parameters.

Baseline GPU parameters	
Frequency	600 MHz
Voltage	1.0 V
Technology node	22 nm
Screen Resolution	1440x720
Tile Size	32x32 pixels
Main memory	
Frequency	400 MHz
Voltage	1.5 V
Technology node	32 nm
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Line Size	64 bytes
Size	1 GiB, 8 banks
Queues	
Vertex (Input & Output)	16 entries, 136 bytes/entry
Triangle & Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Color	64 entries, 24 bytes/entry
Caches All of 64 bytes/line, 2-way associativity	
Vertex Cache	4 KiB, 1 bank, 1 cycle
Texture Caches (x4)	8 KiB, 1 bank, 2 cycles
Tile Cache	32 KiB, 1 bank, 2 cycles
L2 Cache	256 KiB, 8 banks, 18 cycles
Color Buffer	1 KiB, 1 bank, 1 cycle
Depth Buffer	1 KiB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 vertex/cycle
Rasterizer	1 attribute/cycle
Early Z-Test	8 in-flight quad-fragments
Programmable stages	
Vertex Processor	4 vertex processors
Fragment Processor	4 fragment processors

B. Benchmarks

Table II shows the set of benchmarks utilized to evaluate the proposed MEGsim methodology, which consists of eight commercial Android graphics applications. Our set of benchmarks includes both 2D and 3D games, applications that stress the GPU further than other commonly used applications in battery-operated devices. Among the 3D games, we include workloads with simple 3D models (such as *hwh*) and workloads with more sophisticated 3D models and scenes (such as *asp*, *bbr1* or *bbr2*). The workloads included in our set of benchmarks are representative of the current landscape of smartphone games as it includes popular Android games with millions of downloads according to Google Play [34], some of them surpassing 500 million downloads. The evaluated video sequences, with several thousands of frames, have been carefully selected in order to get a representative, common and realistic use-case scenario for each game.

TABLE II: Evaluated benchmark set

Benchmark	Alias	Description	Type	Downloads (Mill.)	Frames	Vertex shaders	Fragment shaders	Cycles (Mill.)	IPC
Asphalt 9: Legends	asp	Racing	3D	50–100	4000	42	45	107811	4.34
Beach Buggy Racing	bbr1	Racing	3D	100–500	2500	73	62	39839	4.91
Beach Buggy Racing	bbr2	Racing	3D	100–500	4000	66	59	58317	4.95
Hill Climb Racing	hcr	Platforms	2D	500–1000	2000	5	5	10111	6.51
Hot Wheels	hwh	Racing	3D	50–100	4000	30	30	86791	4.71
Jetpack Joyride	jjo	Side-scrolling endless runner	2D	100–500	5000	4	5	41219	5.61
Plants *vs* Zombies	pvz	Tower defense	2D	100–500	5000	4	5	39534	4.66
Spider-Man Unlimited	spd	Side-scrolling endless runner	3D	1–5	5000	16	26	75938	6.10

TABLE III: Reduction factor in the number of frames.

Benchmark	Actual frames	MEGsim frames	Reduction factor
asp	4000	23	174×
bbr1	2500	40	63×
bbr2	4000	47	85×
hcr	2000	27	74×
hwh	4000	30	133×
jjo	5000	28	179×
pvz	5000	30	167×
spd	5000	37	135×
Average	3938	33	126×

V. Experimental Results

This section presents the main results obtained by MEGsim. Different studies have been carried out to evaluate both the achieved accuracy and simulation time improvement of the proposed methodology.

A. Simulation Time Improvement

As mentioned before, the selected scenes of each benchmark contain thousands of frames (see Table II) which have been simulated using the cycle-accurate simulator TEAPOT. As expected, completing these simulations took a significant amount of time, in fact, more than one week. However, the use of MEGsim allows to effectively reduce the number of frames to be simulated with a negligible effect on the simulation accuracy.

Table III reports the achieved reduction factor in the number of simulated frames. As it can be seen, the evaluated graphics workloads can be properly characterized just by simulating an average of 33 frames out of 3938 frames, obtaining an average reduction factor of 126× in the number of frames. This means that less than 0.8% of the actual frames have to be simulated which translates into an impressive reduction of more than two orders of magnitude in the simulation time.

B. Simulation Accuracy

As for the simulation accuracy, our experimental results show that it is possible to characterize the full simulation of a video sequence comprised of a high number of frames with a very small subset of selected representatives with a negligible error. To measure the accuracy of MEGsim, both the entire sequence and the subset of representative frames have been simulated using TEAPOT. For the sake of space, we focus our accuracy analysis on four key performance metrics: 1) the total number of cycles, which offers a measure directly proportional to the execution time; 2) the number of accesses to main memory; 3) the number of accesses to the L2 cache; and 4) the number of accesses to the Tile cache, which has an important influence on the overall energy consumption. The accuracy was calculated by comparing the aforementioned output metrics when simulating the whole video sequence against the values obtained when simulating the subset of representative frames, and calculating the relative error which is plotted in Figure 7.

As it can be observed, the higher complexity of 3D games slightly reduces the accuracy of the analyzed metrics. Similarly as with the reduction factor in the number of frames, the relative error for the set of 3D games is slightly higher than the one obtained for 2D games. Regarding the total number of execution cycles, the average relative error is a mere 0.84% (with a maximum error of 2.6% in only one of the games), being this one the metric with the highest accuracy. Regarding the number of main memory accesses, the average relative error is a mere 0.99% whereas the number of L2 cache accesses results in an average relative error of 1.2%. Finally, the average relative error for the number of accesses to the Tile cache is 0.86%, and not exceeding a 2.1% error for any of the evaluated games.

These results demonstrate that it is possible to accurately characterize the activity carried out along the graphics pipeline. Our novel simulation methodology is able to select a small number of frames that accurately represent the whole scene, leading to an impressive reduction of the simulation time with an almost negligible error.

C. Comparison with Random Sub-sampling

To provide a better understanding of how well MEGsim behaves, a naive technique based on random sub-sampling has been implemented to compare with MEGsim. This technique randomly selects the representative frames for the whole sequence. However, the main drawback of random sub-sampling is that, *a priori*, there is no way of determining how many representatives should be selected to properly characterize the whole sequence. Therefore, for a graphics workload of N frames, this technique starts selecting one random frame (representative) and iteratively increases the number of repre-

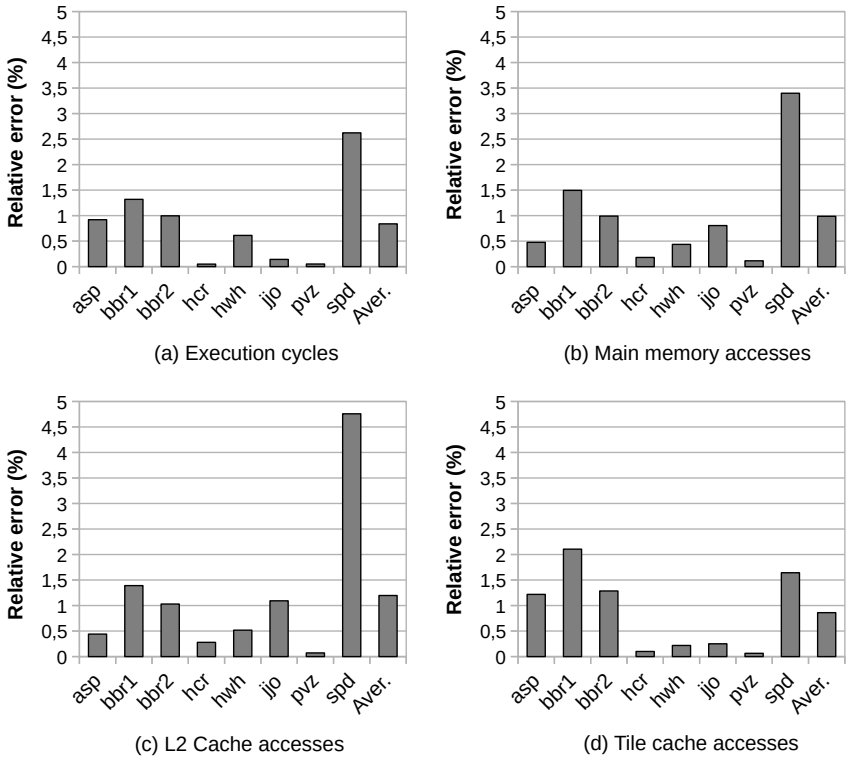

Fig. 7: Relative error obtained in the different performance metrics that have been analyzed.

sentatives (k) such that each of these randomly selected frames represents a fixed range of $\frac{k}{N}$ frames. This is one difference between random sub-sampling and MEGsim since the clusters generated by MEGsim contain a variable number of frames.

Another important difference between the two approaches is that MEGsim automatically stops when a clustering that is considered good enough is found, thanks to using the BIC score. However, random sub-sampling cannot use the BIC score since there are no clusters involved. As it is not possible to know the number of representatives to choose from, for random sub-sampling we iteratively increase the number of representatives until the relative error for the estimated metric (e.g., cycles) is as good as the one obtained by MEGsim.

To properly evaluate the accuracy of this random methodology, the tests have been repeated 1000 times. A similar approach has been followed with MEGsim but repeating the tests 100 times and varying the initialization values of *k-means* to obtain different results. We have measured the maximum relative error for the total number of cycles obtained by both techniques in an interval of confidence of 95% (i.e., the maximum relative error has been calculated after removing the 5% of the results with the worst estimation in both techniques).

As reported in Table IV, to obtain the same accuracy as MEGsim, random sub-sampling needs to select, on average, 58.5× more frames than MEGsim. In particular, random sub-sampling needs to simulate an average of 1686.3 frames to estimate the total number of cycles with the same accuracy as MEGsim, which only needs to simulate an average of 32.8 frames. Finally, we can highlight the extraordinary results that MEGsim achieves in terms of accuracy, with a maximum relative error for the number of cycles lower than 1.5% with a confidence of 95%.

TABLE IV: Number of frames to be simulated for MEGsim and random sub-sampling to achieve the same accuracy.

Benchmark	Max. relative error	MEGsim frames	Random sub-sampling frames	Reduction factor
asp	1.49	23	1262	54.9×
bbr1	2.53	40	349	8.7×
bbr2	1.91	47	418	8.9×
hcr	0.11	27	1960	72.6×
hwh	1.11	30	1243	41.4×
jjo	0.3	28	3193	114.0×
pvz	0.09	30	4852	161.7×
spd	3.86	37	213	5.8×
Average	1.43	32.8	1686.3	58.5×

VI. Conclusions

This paper introduces a novel methodology to efficiently simulate graphics workloads in GPUs, called MEGsim, that automatically determines representative frames in a video sequence. MEGsim is able to accurately characterize a long sequence of frames by using a tiny subset of them, so the cost in time and storage (for the trace files) to carry out simulations is drastically reduced.

Our experimental results show that MEGsim achieves an average reduction of 126× in the number of frames to be

simulated, allowing to reduce the simulation time from several days to a few hours for a typical benchmark. In terms of accuracy, MEGsim performs extraordinarily well, not only for simple 2D games but also for complex 3D games, obtaining an average relative error of 0.84% in the measured execution time (number of cycles), 0.99% in the number of memory accesses, 1.2% in the number of L2 cache accesses and 0.86% in the number of Tile cache accesses. Furthermore, we have shown that MEGsim outperforms other techniques like random subsampling, which requires to simulate 58.5× more frames to achieve the same accuracy as MEGsim.

Overall, these results confirm that based on a judicious selection of parameters it is possible to accurately characterize the activity carried out along the graphics pipeline to more efficiently perform costly cycle-accurate simulations in GPU architectures.

ACKNOWLEDGMENTS

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, and the ICREA Academia program.

REFERENCES

[1] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: A toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS)*, New York, NY, USA, 2013, pp. 37–46.

[2] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 2002, pp. 45–57.

[3] R. E. Kass and L. Wasserman, "A reference bayesian test for nested hypotheses and its relationship to the schwarz criterion," *Journal of the American Statistical Association*, vol. 90, no. 431, pp. 928–934, 1995.

[4] T. Akenine-Moller and J. Strom, "Graphics processing units for handhelds," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 779–789, 2008.

[5] Imagination Technologies Limited, "PowerVR Hardware. architecture overview for developers," http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf, accessed March 2022.

[6] D. Corbalán-Navarro, J. L. Aragón, M. Anglada, E. De Lucas, J.-M. Parcerisa, and A. González, "Omega-test: A predictive early-z culling to improve the graphics pipeline energy-efficiency," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–14, 2021.

[7] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, p. 26, 2014.

[8] J. Pool, "Energy-precision tradeoffs in the graphics pipeline," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2012.

[9] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/CRC Press, 2019.

[10] J. Y. Joshua, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, no. 11, pp. 22–29, 2006.

[11] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, "A simple model for portable and fast prediction of execution time and power consumption of gpu kernels," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, dec 2021.

[12] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Performance and power prediction for concurrent execution on gpus," *ACM Trans. Archit. Code Optim.*, pp. 1–25, feb 2022.

[13] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[14] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A PIN-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[15] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[16] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 175–184, 2006.

[17] S. Eyerman, L. Eeckhout, and J. E. Smith, "Studying compiler-microarchitecture interactions through interval analysis," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 406–406.

[18] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems*, vol. 27, no. 2, pp. 1–37, 2009.

[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2003, pp. 84–97.

[20] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, 2004.

[21] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, 2019.

[22] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins, "Characterizing and comparing prevailing simulation techniques," in *11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 266–277.

[23] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe, "Simulation sampling with live-points," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 2–12.

[24] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

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

[26] S. Flolid, E. Shriver, Z. Susskind, B. Thorell, and L. K. John, "Simtrace: Capturing over time program phase behavior," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 226–228.

[27] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 724–737.

[28] D. Pelleg, "Extending k-means with efficient estimation of the number of clusters in icml," in *Proceedings of the 17th International Conference on Machine Learning*, 2000, pp. 277–281.

[29] A. Foglia. (2012) Notes on bayesian information criterion calculation for x-means clustering. [Online]. Available: https://github.com/bobhancock/goxmeans/blob/master/doc/BIC_notes.pdf

[30] Arm MALI-450 GPU. [Online]. Available: https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu/

[31] "Android Studio," https://developer.android.com/studio, March 2022.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

[33] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[34] Google play. [Online]. Available: https://play.google.com