Emerald: Graphics Modeling for SoC Systems

Ayub A. Gubran and Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia
{ayoubg,aamodt}@ece.ubc.ca

ABSTRACT

Mobile systems-on-chips (SoCs) have become ubiquitous computing platforms, and, in recent years, they have become increasingly heterogeneous and complex. A typical SoC includes CPUs, graphics processor units (GPUs), image processors, video encoders/decoders, AI engines, digital signal processors (DSPs) and 2D engines among others [33, 70, 71]. One of the most significant SoC units in terms of both off-chip memory bandwidth and SoC die area is the GPU. In this paper, we present Emerald, a simulator that builds on existing tools to provide a unified model for graphics and GPGPU applications. Emerald enables OpenGL (v4.5) and OpenGL ES (v3.2) shaders to run on GPGPU-Sim's timing model and is integrated with gem5 and Android to simulate full SoCs. Emerald thus provides a platform for studying system-level SoC interactions while including the impact of graphics.

We present two case studies using Emerald. First, we use Emerald's full-system mode to highlight the importance of system-wide interactions by studying and analyzing memory organization and scheduling schemes for SoC systems. Second, we use Emerald's standalone mode to evaluate a novel mechanism for balancing the graphics shading work assigned to each GPU core.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; **Graphics processors**; • **Computer systems organization → System on a chip**; *Heterogeneous (hybrid) systems*.

KEYWORDS

GPU, Graphics, SoC, Simulation

1 INTRODUCTION

The end of Dennard scaling has resulted in limits to the percentage of a chip that can switch at full frequency [27]. This has lead to increasing use of specialized accelerator cores in heterogeneous

systems [33] and heterogeneous systems-on-chips (SoCs) are now ubiquitous in energy-constrained mobile devices.

With the increasing prevalence of heterogeneous SoCs there is a commensurate need for architectural performance models that model contemporary SoCs. Although existing performance models such as gem5 [17] and MARSSx86 [55] enable running a full operating system, they lack an ability to capture all hardware due to the absence models for key elements such as graphics rendering [28, 62]. The exclusion of such key components limits researchers' ability to evaluate common use cases (e.g., GUI and graphics-intensive workloads), and may lead researchers to develop solutions that ignore significant system-wide behaviors. A recent study [62] showed that using software rendering instead of a hardware model "can severely misrepresent" the behavior of a system.

To provide a useful model for studying system-wide behavior we believe an architecture model for a contemporary heterogeneous SoC must: (1) support a full operating system software stack (e.g., Linux and Android), (2) model the main hardware components (e.g., CPU, GPUs and the memory hierarchy), (3) provide a flexible platform to model additional special components (e.g., specialized accelerators). A tool that meets these requirements would allow architecture researchers to evaluate workloads capturing the intricate behavior of present-day use-cases, from computer games to artificial intelligence and augmented reality.

The gem5 simulator [17] can evaluate multicore systems including executing operating system code and has been extended to model additional system components. Table 1 compares various frameworks for modeling GPU enabled SoCs including Gem-Droid [20], gem5-gpu [56]. These two simulators extend gem5 to capture memory interactions and incorporate existing GPGPU models, respectively. GemDroid uses multiple single-threaded traces of a modified Android emulator that captures events (e.g., IP core invoked via an API call) and adds markers in the instruction trace. When an event is encountered while replaying an instruction trace memory traffic generated by that event is injected while pausing instruction execution. Multiprogrammed parallelism is modeled by replaying multiple independent single-threaded traces. Using static traces limits the ability to capture intricate system interactions or to evaluate ideas that exploit low-level system details. While gem5-gpu adds a GPGPU model to gem5, it does not support graphics workloads.

In this paper, we introduce Emerald, a GPU simulator. Emerald is integrated with gem5 to provide both GPU-only and full-system performance modeling meeting the three requirements above. gem5-emerald builds on GPGPU-Sim [16] and extends gem5 full-system simulation to support graphics. Emerald's graphics model is able to execute graphics shaders using the same model used by GPGPU-Sim thus enabling the same microarchitecture model to be used for both graphics and GPGPU workloads. Additional models were
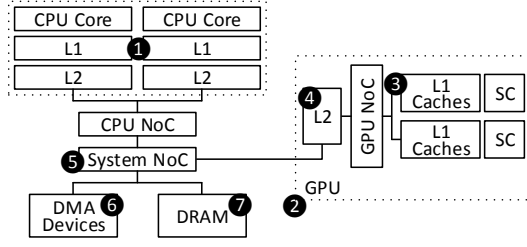
Figure 1: Overview of gem5-emerald HW architecture

added to support graphics-specific functions. In addition, to use Emerald under a full-system SoC model, we modified Android to add a driver-like layer to control our GPU model.

We employ Emerald in two case studies. In the first, we use gem5-emerald full-system to re-evaluate work [20, 74] that used trace-based simulation to evaluate memory scheduling and organization schemes for heterogeneous SoCs. This case study demonstrates issues that are difficult to to detect under trace-based simulation.

The second case study uses the stand-alone GPU to evaluate dynamic fragment shading load-balancing (DFSL), a novel technique that dynamically balances fragment shading across GPU cores. DFSL employs "temporal coherence" [63] – similarity of successively rendered frames – and can reduce execution time by 7.3-19% over static work distribution.

Simulator	Model	GPU Model		FS Simulation
		GPGPU	Graphics	
gem5	Execution driven	No	No	Yes
GemDroid	Trace driven	No	Yes	No
gem5-gpu	Execution driven	Yes	No	Yes
Emerald	Execution driven	Yes	Yes	Yes

Table 1: Simulation platforms

The following are the contributions of this paper:

(1) Emerald[1], a simulation infrastructure that extends GPGPU-Sim [16] to provide a unified simulator for graphics and GPGPU workloads;

(2) An infrastructure to simulate SoC systems using Android and gem5;

(3) A study of the behavior of memory in SoC systems highlighting the importance of detailed modeling incorporating dependencies between components, feedback from the system, and the timing of events;

(4) DFSL, a dynamic load-balancing technique exploiting graphics temporal coherence to improve GPU performance.

2 EMERALD SOC ARCHITECTURE

Figure 1 shows an overview of the SoC architecture that gem5-emerald models. The SoC model consists of a CPU cluster (❶) with either in-order or out-of-order cores with multiple cache levels. The GPU cluster (❷) consists of multiple GPU shader cores (SCs). Each

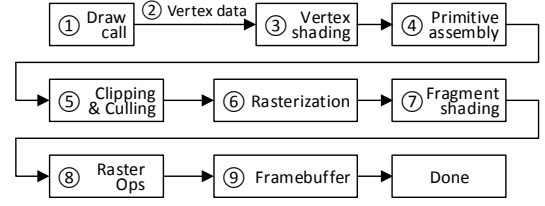[1]The source code for Emerald can be found at https://github.com/gem5-graphics/gem5-graphics

Figure 2: OpenGL pipeline realized by Emerald

SC consists of multiple execution lanes executing multiple GPU threads in a lock-step manner. GPU threads are organized in groups of 32 threads, i.e., warps [53].

In our baseline design, GPU L1 caches (❸) are connected to a non-coherent interconnection network; meanwhile, the GPU's L2 cache (❹) is coherent with CPU caches (following a similar design to [56]). The system network (❺) is a coherent network that connects the CPU cluster, the GPU cluster, DMA devices ❻ (e.g., display controllers) and the main memory ❼.

It is relatively easy to change the connections and network types (gem5 provides several interchangeable network models). We use gem5 classic network models as they provide efficient implementations for faster full-system simulation, which can consume a considerable amount of time with slower models that do not support fast-forwarding (e.g., gem5 Ruby). The number of CPUs, GPU's SCs, the type and number of specialized accelerators and the on-chip network connecting them are widely configurable as well. Additionally, cache hierarchies and coherence are also configurable.

3 GRAPHICS ARCHITECTURE

Below we describe the OpenGL pipeline and graphics hardware modeled in Emerald.

3.1 Graphics Pipelines

Figure 2 shows the OpenGL pipeline realized by Emerald. An OpenGL API draw call (①) sends vertex data (②) to the vertex shading stage (③). Vertex data includes 3D positions and other optional vertex attributes such as color, texture coordinates and normal vectors. The vertex shading stage transforms 3D vertex coordinates to 2D screen-space coordinates by running a vertex shader program. After vertex shading, vertex positions and attributes are are passed to primitive assembly (④), which converts a stream of screen-space vertices to a sequence of base primitives (triangles, triangle fans or strips as per the current OpenGL configuration). Assembled primitives then proceed through clipping and culling stages (⑤). Trivially invisible back-faced and out of rendering volume primitives are discarded. Primitives that fall partially outside the rendering volume are clipped into smaller primitives. The resulting triangles proceed to the rasterization stage (⑥).

Rasterization is the generation of fragments from primitive vertices by interpolating position attributes and any additional optional attributes. Optional *variable* attributes, such as texture or any user-defined *variable* attributes, are only interpolated when enabled. Other user-defined attributes (*uniform* attributes), are not interpolated and hold the same value across vertices/fragments.

The rasterization stage passes the generated fragments and the corresponding attributes to the fragment shading stage.

The fragment shading stage (⑦) is, typically, the most compute intensive stage in the pipeline. User-defined fragment shaders procedurally apply lighting computation to each fragment. Texture lookups and fragment coloring are performed at this stage. Once the associated fragment shader program transforms fragments, they are sent to the raster operations stage (⑧).

The raster operations stage is where procedures like blending, depth testing and stencil operations are applied. For blending, fragments are blended with the corresponding pixels on the framebuffer (⑨). In depth testing, each fragment depth is compared against the corresponding depth value in the depth buffer. If a fragment passes the depth test, the new fragment is written to the framebuffer (overwriting the previous value); otherwise, the fragment is invisible and discarded. Stencil operations work similarly to depth testing, where a stencil buffer is used as a mask to determine which fragments can be written to the framebuffer.

3.2 Contemporary Graphics Architectures

The abstract nature of graphics APIs, such as an OpenGL pipeline like that described in Section 3.1 or Direct3D facilitates a large design space for GPU hardware as evidenced by the wide rage of vendor architectures [4, 10, 18, 21, 23, 60, 72].

Emerald builds on the NVIDIA-like GPU model in GPGPU-Sim [16] version 3.2.2. Like all contemporary GPUs, Emerald implements a unified shader model rather than employing different hardware for different shader types. Although many mobile GPUs employ tile-based rendering [23, 60, 72], Emerald, follows a hybrid design similar to that of Nvidia's [21], which is employed in Nvidia's discrete and mobile GPUs [51, 52]. This hybrid approach combines aspects of immediate-mode rendering (IMR) and tile-based rendering (TBR).

IMR architectures process primitives fully, including fragment shading, according to their draw call order. For each primitive, the GPU performs vertex shading operations on world-space primitives to produce the corresponding screen-space versions along with their attributes. The GPU rasterizes the screen-space primitives to create fragments that are input to the fragment shading stage.

Mobile GPU vendors use TBR architectures [10, 18, 23, 60, 72] to reduce off-chip memory footprint and power consumption. TBR architectures reduce the cost of fragment shading, which often dominates the cost of rendering, by binning geometry across a set of 2D tiles in screen-space then rendering using an on-chip buffer. An initial binning pass performs vertex shading on all primitives in a scene and stores the resulting screen-space primitives in off-chip per tile memory bins. Then a follow-up rasterization pass renders the content of each tile by processing the geometry in the corresponding bin. Main memory accesses are avoided using an on-chip tile buffer for depth testing and blending.

Complex geometry can reduce performance on TBR architectures relative to IMR due to binning overhead. Thus, ARM uses a hierarchical tiling scheme [23] to reduce the number of primitives stored in memory bins in geometry-dense scenes, while Qualcomm can support both IMR and TBR [57].
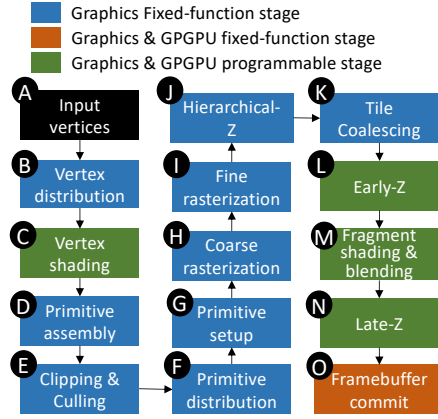

Figure 3: Emerald graphics pipeline

Emerald uses a hybrid rendering approach similar to ones described by AMD and NVIDIA [4, 29, 37, 45]. We we refer to this hybrid approach as immediate tiled rendering (ITR). Similar to TBR, ITR divides the screen into a grid of tiles; however, instead of implementing a separate binning pass where all geometry is processed first, ITR splits primitives into batches, where batch sizes can be adjusted to suit the rendered content [4]. ITR then bins and caches each batch's vertex shading results on-chip before using them immediately for fragment shading [37]. ITR relies on locality between consecutive primitives, which will likely render to the same group of screen-space tiles, to avoid the overhead of storing and reading the entire screen-space geometry to/from off-chip memory as it is the case with TBR; thus, ITR is more efficient than TBR when processing geometry-rich scenes.

3.3 Emerald Architecture

3.3.1 Emerald Graphics Pipeline. Figure 3 shows the hardware pipeline Emerald models. Emerald implements the logical pipeline in Figure 2 by employing hardware stages found in contemporary GPU architectures [2, 5, 6, 26, 29, 30, 44, 47, 64]. A draw call initiates rendering by providing a set of input vertices ⓐ. Vertices are distributed across SIMT cores in *batches* (ⓑ, ⓒ). Once vertices complete vertex shading, they proceed through primitive assembly (ⓓ) and clipping & culling (ⓔ). Primitives that pass clipping and culling are distributed to GPU clusters based on screen-space position (ⓕ). Each cluster performs primitive setup (ⓖ). Then, coarse rasterization (ⓗ) identifies the screen tiles each primitive covers. In fine rasterization (ⓘ) input attributes for individual pixels are generated. After primitives are rasterized, and if depth testing is enabled, fragment tiles are passed to a Hierarchical-Z/stencil stage (ⓙ), where a low-resolution on-chip depth/stencil buffer is used to eliminate invisible fragments. Surviving fragments are assembled in tiles (ⓚ) and shaded by the SIMT cores. Emerald employs programmable, or in-shader, raster operations where depth and blending (stages ⓛ, ⓜ and ⓝ), are performed as part of the shader program as opposed to using atomic units coupled to the memory access schedulers. Hardware prevents races between

fragment tiles targeting the same screen-space position [29] (Section 3.3.5). Depth testing is either performed early in the shader, to eliminate a dead fragment before executing the fragment shader (**L**), or at the end of the shader (**N**). End of shader depth testing is used with fragment shader programs that include instructions that discard fragments or modify their depth values. Finally, fragment data is written to the corresponding pixel position in the framebuffer (**O**).

Name	Description
Execution units (SIMT lanes)	Single instruction multiple data (SIMD) pipeline that executes scalar threads in groups of warps (each warp contains 32 threads).
SIMT Stacks	Stacks are used to manage SIMT threads branch divergence by keeping track of active threads in each branch on the top of the stack.
Register File	A banked register file inspired by an Nvidia's design. Operand collectors are used for communicating between execution units and the register file.
Shared Memory	Per SC banked scratchpad buffer used for intra-thread block communications.
Caches [53][2]	L1I: Instruction cache.
	L1D: For global (GPGPU) and pixel data.
	L1T: Read-only textures cache.
	L1Z: Depth cache.
	L1C: Constant & vertex cache.
	L2: A second level cache coherent with CPU L2 caches (in our baseline model we use L1 caches that are non-coherent with each other or the L2 cache).
Coalescing Logic	Coalesces memory requests from SIMT lanes to caches

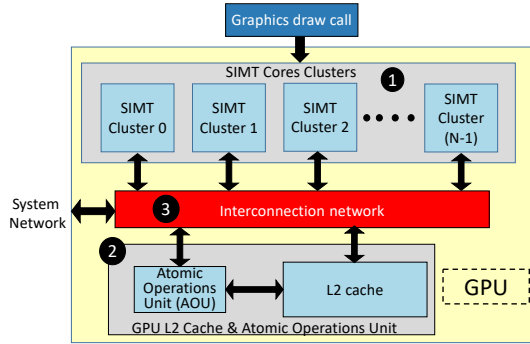**Table 2: SIMT Core Components**

Figure 4: GPU Architecture

3.3.2 Emerald GPU Architecture. As shown in Figure 4, Emerald's hardware architecture consists of a set of single-instruction multiple-thread (SIMT [50, 54]) core clusters (**1**) and L2 cache with atomic operations unit (AOU) (**2**). An interconnection network (**3**) connects GPU clusters, the AUO and the L2 cache to each other and to the rest of the system (i.e., to DRAM or to an SoC NoC).

[2]Baseline cache configurations are shown. Emerald readily allows alternative cache hierarchies to be used.
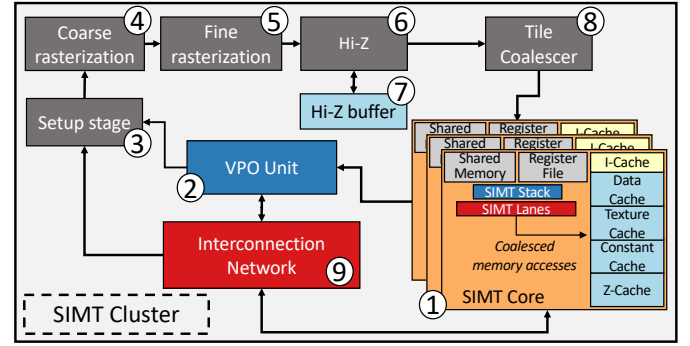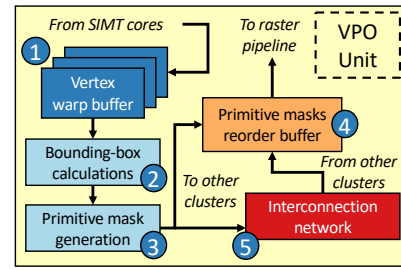

Figure 5: GPU SIMT cluster

Figure 6: VPO Unit

Figure 5 shows the organization of of SIMT cluster. Each SIMT cluster has SIMT cores (**1**) and fixed pipeline stages (**2** to **8**) to implement stages **G** to **K** in Figure 3.

Our SIMT core model builds upon GPGPU-Sim [16] version 3.2.2. SIMT cores execute shader programs for vertices and fragments by grouping them in warps (sets of 32 threads), which execute on SIMT lanes in a lock-step manner. Branch divergence is handled by executing threads of taken and non-taken paths sequentially. A SIMT stack is used to track which path each thread has taken. Memory accesses are handled by a memory coalescing unit, which coalesces spatially local accesses into cache-line-sized chunks. Each SIMT core has a set of caches that handle different types of accesses. Table 2 provides a summary of SIMT core components. Screen-space is divided into tiles (*TC* tiles), where each TC tile position is assigned to a single SIMT core (details in Section 3.3.5).

3.3.3 Vertex Shading. When a draw call is invoked, vertices are assigned for vertex processing, in batches, to SIMT cores in a round-robin fashion. Batches of, sometimes overlapping, warps are used when assigning vertices to SIMT cores. The type of primitive being rendered determines the overlapping degree and, effectively, how vertices are assigned to warps [3]. Without vertex overlapping, primitive processing stages (i.e., VPO **2** and setup **3**) would need to consult vertices from multiple warps when primitive formats with vertex sharing are in use (e.g., *GL_LINE_STRIP* and *GL_TRIANGLE_STRIP*); thus, overlapped vertex warps allow parallel screen-space primitive processing even when using primitive formats with vertex sharing.

Vertex shaders fetch vertex data from memory, transform vertices, and write result vertices, i.e., their position data along with other optional output attributes, to the L2 cache (Figure 4 ②). In addition, SIMT cores (Figure 5 ①) also send vertex position data generated by vertex shaders to the Vertex Processing and Operations (VPO) unit (Figure 5 ②), which distributes primitives among SIMT cores as detailed in the following section.

3.3.4 Primitive Processing. The VPO unit (Figure 6), which is inspired by Nvidia's work distribution crossbar interface [47, 59], assigns primitives to clusters for screen-space processing. SIMT cores write position data of vertex warps into one of the vertex warp buffers (Figure 6 ①). A bounding-box calculation unit (②) consumes position data from each warp in (①) and calculates the bounding-box for each primitive covered by the warp. Since vertices are overlapped between warps, according to the primitive type in use as explained in Section 3.3.3, there is no need to consult vertex position data from other warps that may be rendered on other SIMT clusters.

Bounding-box calculations (②) generate a warp-sized primitive mask for each SIMT cluster. For each vertex warp, a cluster primitive mask conveys if that particular cluster is covered (i.e., mask bit is set to 1), or not covered (i.e., mask bit is set to 0) by each of the primitives in the warp.

The primitive mask generation stage (③) sends each created mask to its corresponding cluster. If the current cluster is the destination, the mask is committed locally to the primitive masks reorder buffer (④), i.e., the PMRB unit. Otherwise, the interconnection network (⑤) is used to communicate primitive masks to other clusters.

On the receiving end, the PMRB unit collects primitive masks from all clusters. Each primitive mask contains an ID for the first primitive covered by the mask which allows the PMRB unit to store masks according to their draw call order. To avoid deadlocks, the vertex shader launcher limits the number of active vertex warps to the available space in PMRB units.

The PMRB unit processes masks according to their draw call order. For each mask, the PMRB unit checks each mask bit to determine if the corresponding primitive covers part of the screen-space area assigned to the current cluster; if not, the primitive is simply ignored. On the other hand, if a primitive covers part of the screen-space area assigned to the current cluster (mask bit is set to 1), the primitive should be processed by the current cluster and the corresponding primitive ID is communicated to the setup stage (Figure 5 ③). The setup stage uses primitive IDs to fetch the corresponding vertex data from the L2 cache. Following that, primitives go through stages ④ to ⑧ which resemble the pipeline described in Section 3.3.1.

3.3.5 The TC Stage. The tile coalescing (TC) stage (⑧) assembles quads of fragments from multiple primitives before assigning them to the corresponding SIMT core for fragment shading. By coalescing fragments from multiple primitives, TC coalescing improves SIMT core utilization when running fragment shading, especially when handling micro-primitives.

Figure 7 shows the TC unit. In ①, the TC unit receives *raster tiles* from the fine-rasterization stage or the Hi-Z stage (if enabled).
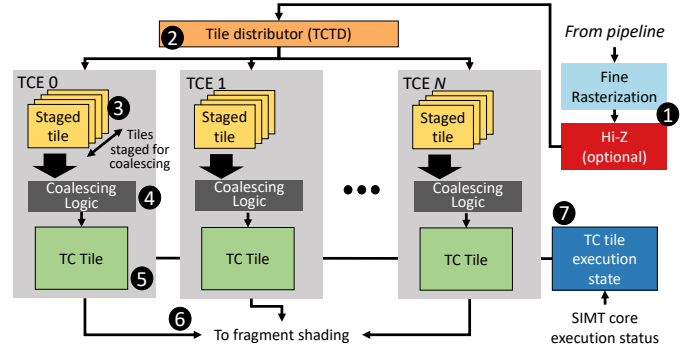

Figure 7: TC Unit

A tile distributor (②) stages incoming primitive fragment tiles to one of the TC engines (TCEs). At any given point, each TCE stages and coalesces (③ and ④) tiles that correspond to a single screen-space TC tile (⑤); this means fragments from a particular TC tile are executed on the same SIMT core. A TC tile can cover multiple raster tiles (e.g., 4×4 raster tiles).

A raster tile is a tile used by the pipeline in Figure 5 (④ to ⑥). When staging a new raster tile, the tile distributor checks if a TCE is already operating on the corresponding TC tile. If this is the case, the new raster tile is staged to the corresponding TCE; otherwise, the tile distributor stages the new raster tile to an empty TCE.

TCE assembles staged raster tiles into one TC tile if there are no conflicts, or to multiple TC tiles if there are overlapping raster tiles. TCEs flush TC tiles to the corresponding SIMT cores when the staging area is full and no new quads can be coalesced. In addition, TCEs set a maximum number of cycles without new raster tiles before flushing the current TC tile.

Before issuing a TC tile to a SIMT core, the TCE checks if a previous TC tile, for the same screen-space location, has not finished shading ⑦. Only a single TC tile is being shaded for a TC location at a given point of time to allow in-shader depth and blending operations to be performed. Once fragment shading for a TC tile is performed, result fragments are committed to the framebuffer.

In case study II in Section 6, we study how the granularity of TC tile mapping to SIMT cores can affect performance, and we evaluate a technique (DFSL) that dynamically adjusts TC mapping granularity.

3.3.6 Out-of-order primitive rendering. In some cases, the VPO unit and the TC stage can process primitives out-of-order. For example, when depth testing is enabled and blending is disabled, primitives can be safely processed in an out-of-order fashion. However, this optimization is unexploited in our model and we plan to add it in our future work.

3.4 Model Accuracy

We have validated our microarchitecture against NVIDIA's Pascal architecture (found in Tegra X2). Microbenchmarks were used to better understand the implementation. For example, for fragment shading, we used NVIDIA's extensions (NV_shader_thread_group) to confirm that screen space is divided into 16x16 screen tiles that

are statically assigned to shader cores using a complex hashing function. In Emerald, we used similar screen tiles that are pre-assigned using a modular hash that takes into account compute cluster and shader core. For vertex shaders, we examined how vertices are assigned to SMs on NVIDIA's hardware (using transform feedback [36]) and found they are assigned to SMs in batches (batch size varies with primitive type). We modified our model accordingly to capture the same behavior. Finally, we profiled Emerald's GPU model against the GPU of Tegra K1 SoC [51] with a set of 14 benchmarks. Our initial results show that draw execution time has a correlation of 98% with 32.2% average absolute relative error (5.2% to 312%), while pixel fill-rate (pixels per cycle) has a correlation of 76.5% with a 33% absolute relative error (9.6% to 77%). Absolute relative error is computed as $\frac{|Hardware-Simulator|}{Hardware}$.

3.5 Model Limitations and future work

Modern APIs specify stages for vertex shading, tessellation, geometry shading, and fragment shading [36]. Currently, Emerald supports the most commonly used stages of vertex and fragment shading. Future work includes adding support for geometry and tessellation.

Emerald covers part of the design spectrum for graphics architectures with submodels that vary in their level of details. Future work also includes elaborating some of the less detailed submodels and adding configuration flexibility to facilitate experimenting with a broader range of architectures. This includes adding detailed texture filtering and caching models, support for compression, adding the option to globally perform depth and blending operations near the L2 cache, and adding support for managed tile buffers.

4 EMERALD SOFTWARE DESIGN

Figure 8 highlights Emerald's software architecture for the two supported modes: standalone and full-system modes. In the standalone mode, only the GPU model is used. On the other hand, in the full-system mode, the GPU operates under Android with other SoC components. In this work, and as detailed below, we chose to utilize a set of existing open-source simulators and tools like APITrace, gem5, Mesa3D, GPGPU-Sim, and Android emulation tools; this makes it easier for Emerald to stay up-to-date with the most recent software tools as systems like Android and graphics standards are consistently changing. We use MESA as our API interface and state handler so that future OpenGL updates can be easily supported by Emerald. Similarly, using Android emulator tools facilitates supporting future Android systems.

4.1 Emerald Standalone Mode

Figure 8a shows Emerald standalone mode. In this mode, we use APITrace [9] to record graphics traces. A trace file ① then is played by a modified version of APITrace through gem5-emerald ② and then to Mesa ③. Emerald can be configured to execute frames of interest (a specific frame, set of frames, or a set of draw calls within a frame). For frames within the specified region-of-interest, Mesa sends all necessary state data to Emerald before execution begins. We also utilized some components from gem5-gpu [56] which connects GPGPU-Sim memory ports to the gem5 interface. Finally,

in ④ our tool, TGSItoPTX, is used to generate PTX shaders that are compatible with GPGPU-Sim (we extended GPGPGPU-Sim's ISA to include several graphics specific instructions). TGSItoPTX consumes Mesa3D TGSI shaders (which are compiled from higher level GLSL) and converts them to their equivalent PTX version.

4.2 Emerald Full-system Mode

In the full-system mode, Emerald runs under the Android OS. Once an Android system has booted, it uses the goldfish-opengl library [7] as a graphics driver to Emerald. We connect the goldfish library to the Android-side gem5-pipe ❶, which captures Android OpenGL calls and sends them to gem5 through pseudo-instructions. In gem5, the gem5-side graphics-pipe captures draw call packets and forwards them to the Android emulator host-side libraries ❷, which process graphics packets and convert them to OpenGL ES draw calls for Mesa ❸. Android host-side emulator libraries are also used to track each OpenGL context of each process running on Android. From Mesa ❸, Emerald follows the same steps in Section 4.1.

For the full-system mode, we also added support for graphics checkpointing ❹. We found this to be crucial to support full-system simulation. Booting Android on gem5 takes many hours and checkpointing the system state, including graphics, is necessary to be able to checkpoint and resume at any point during simulation. Graphics checkpointing works by recording all draw calls sent by the system and storing them along with other gem5 checkpointing data. When a checkpoint is loaded, Emerald checkpointing restores the graphics state of all threads running on Android through Mesa's functional model.

5 CASE STUDY I: MEMORY ORGANIZATION AND SCHEDULING ON MOBILE SOCS

This case study evaluates two proposals for memory scheduling [74] and organization [20] aimed for heterogeneous SoCs. We used these proposals as they represent some of the most relevant work in the area of SoC design. Both proposals relied on trace-based simulation to evaluate system behavior of heterogeneous SoCs.

In this case study, we evaluate the heterogeneous memory controller (HMC) proposed by Nachiappan et al. [20] and the DASH memory scheduler by Usui et al. [74] under execution-driven simulation using Emerald's GPU model and running under the Android OS. We note Usui et al. [74] recognized the shortcomings of approximating memory behavior under a real system using traces.

5.1 Implementation

5.1.1 DASH Scheduler. The DASH scheduler [74] builds on the TCM scheduler [40] and the scheduler proposed by Jeong et al. [34] to define a deadline-aware scheduler. DASH aims to balance access to DRAM by classifying CPU and IP traffic into a set of priority levels that includes: (i) Urgent IPs; (ii) Memory non-intensive CPU applications; (iii) Non-urgent IPs; and (iv) Memory intensive CPU applications.

DASH further categorizes IP deadlines into short and long deadlines. IPs with a unit of work (e.g., frame) that is extremely short ($\leq 10\mu$ seconds) are defined as short deadline IPs. In our system, the frame rate (i.e., 60 FPS, or 16ms per frame) determines both of our
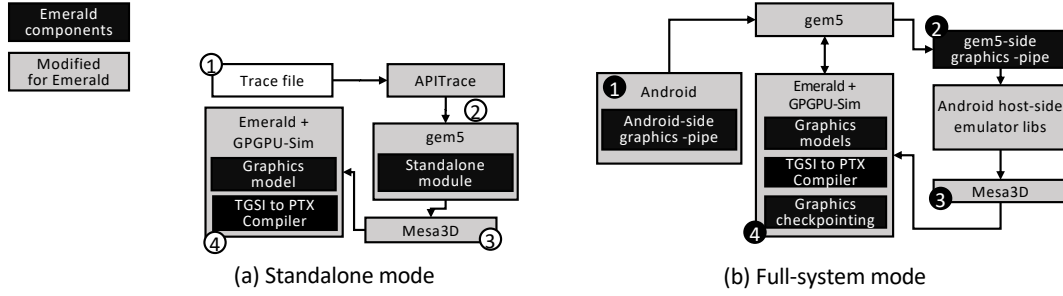
Figure 8: Emerald software architecture

IP deadlines. As a result, both the GPU and the display controller are classified as long-deadline IPs.

In addition to the priority levels listed above, DASH also implements probabilistic scheduling that tries to balance servicing non-urgent IPs and memory-intensive CPU applications. With a probability (P), memory intensive applications are prioritized over non-urgent IPs where P is updated every $SwitchingUnit$ to balance the number of requests serviced to each traffic source.

Defining Clustering Bandwidth. One of the issues we faced in our implementation is the definition of clustering bandwidth as specified by TCM [40] and used by DASH. The dilemma is that TCM-based clustering assumes a homogeneous set of cores, where their total bandwidth $TotalBWusage$ is used to classify CPU threads into memory intensive or memory non-intensive threads based on some clustering threshold $ClusterThresh$. The issue in an SoC system is how to calculate $TotalBWusage$. In our system, which defines a contemporary mobile SoC, the number of CPU threads (i.e., CPU cores) is limited. In addition, there is significant bandwidth demand from non-CPU IPs (e.g., GPUs). Whether $TotalBWusage$ should include non-CPU bandwidth usage is unclear [40, 74]. We found that there are significant consequences for both choices. If non-CPU bandwidth is used to calculate $TotalBWusage$, it is more likely for CPU threads to be classified as memory non-intensive even if some of them are relativity memory intensive. On the other hand, using only CPU threads will likely result in CPU threads classified as memory intensive (relative to total CPU traffic) even if they produce very little bandwidth relative to the rest of the system. This issue extends to heterogeneous CPU cores where memory bandwidth demand vs. latency sensitivity becomes non-trivial as many SoCs deploy a heterogeneous set of CPU cores [11, 52, 61].

In this study, we evaluated DASH using both methods to calculate clustering $TotalBWusage$, i.e., including or excluding non-CPU bandwidth. For other configurations, we used the configurations specified in [74] and [40] which are listed in Table 3.

5.1.2 HMC Controller. HMC [20] proposes implementing a heterogeneous memory controller that aims to support locality/parallelism based on traffic source. HMC defines separate memory regions by using different DRAM channels for handling CPU and IPs accesses. CPU-assigned channels use an address mapping that improves locality by assigning consecutive addresses to the same row buffer (page striped addressing). On the other hand, IPs assigned channels use an address mapping that improves parallelism by assigning

Cycle unit	CPU cycle
Scheduling Unit	1000 cycles
Switching Unit	500 cycles
Shuffling Interval	800 cycles
Quantum Length	1M cycles
Clustering Factor	0.15
Emergent Threshold	0.8 (0.9 for the GPU)
Display Frame Period	16ms (60 FPS)
GPU Frame Period	33ms (30 FPS)

Table 3: DASH configurations

consecutive addresses across DRAM banks (cache-line stripped addressing). HMC targets improving IP memory bandwidth by exploiting sequential accessing to large buffers that benefits from accessing multiple banks in parallel. In our study, we look for the following:

(1) The performance of the system under normal and high loads.
(2) Access locality behavior of CPU cores vs. IP cores.
(3) DRAM access balance (between CPU and IP-assigned channels).

Baseline	
Channels	2
DRAM address mapping	Row:Rank:Bank:Column:Channel
Scheduler	FRFCFS
HMC	
Channels	2 (1 for each source type)
CPU channel address mapping	Row:Rank:Bank:Column:Channel
IP channel address mapping	Row:Column:Rank:Bank:Channel
Scheduler	FRFCFS

Table 4: Baseline and HMC DRAM configurations

5.2 Evaluation

We used Emerald to run Android in the full-system mode using Emerald's GPU model and gem5's CPU and display controller models. For this experiment, an earlier version of Emerald which features a simpler pixel tile launcher and a centralized output vertex buffer and primitive distribution was used. System configurations are listed in Table 5.

To evaluate SoC performance, we profile an Android application that loads and displays a set of 3D models (listed in Table 6). We run our system under normal and high load configurations. We use

the baseline configuration in Table 5 to model a regular load scenario and use a low-frequency DRAM configuration (133 Mb/s/pin) to evaluate the system under a high load scenario. We opted for stressing the DRAM this way as an alternative to setting up different workloads with varying degree of complexity (which can take several weeks under full-system simulation). We compare DASH and HMC to a baseline DRAM configuration (Table 5) with FR-FCFS scheduling and using the baseline address mapping as shown in Table 4.

CPU	
Cores	4 ARM O3 cores [17]
Freq.	2.0GHz
L1	32 kb
L2 (per core)	1 MB
GPU	
# SIMT Cores	4 (128 Cuda Cores)
SIMT Core Freq.	950MHz
Lanes per SIMT Core	32 (warp size)
L1D	16KB, 128B line, 4-way LRU
L1T	64KB, 128B line, 4-way LRU
L1Z	32KB, 128B line, 4-way LRU
Shared L2	128KB, 128B line, 8-way LRU
Output Vertex Buffer	36KB (Max. 9K vertices)
Pixel Tile Size	32×32
Framebuffer	1024×768 32bit RGBA
System	
OS	Android JB 4.2.2.1
DRAM [31]	2-channel 32-bit wide LPDDR3, Data Rate: 1333Mb/s

Table 5: Case Study I system configurations

3D Workloads	
# of frames	5 (1 warm-up frame + 4 profiled frames)
Model	**Name**
M1	Chair
M2	Cube
M3	Mask
M4	Triangles
Configs	
Abbrv.	**Description**
BAS	Baseline configuration
DCB	DASH using CPU bandwidth clustering
DTB	DASH using system bandwidth clustering
HMC	Heterogeneous Memory Controllers
AVG	Average

Table 6: Case Study I workload models and configurations

5.2.1 Regular-load scenario. All configurations produce a similar frame rate under this scenario (less than 1% difference). Both the GPU and the display controller were able to generate frames at 60 FPS (application target frame rate). However, looking further at each stage of rendering, we notice that GPU rendering time differs. Figure 9 shows the normalized execution time (lower is better) of the GPU portion of the frame. Compared to the baseline, the GPU takes 19-20% longer to render a frame with DASH, and with HMC it takes almost twice as long.

First, we investigate why DASH prolongs GPU execution time. We found the reason for the longer GPU execution time is that DASH prioritizes CPU requests over that of the GPU's while frames being rendered. As long as the IP, i.e., the GPU, is consistently
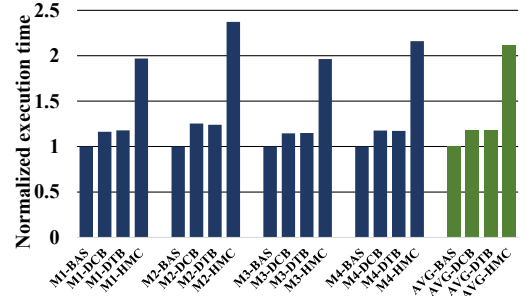

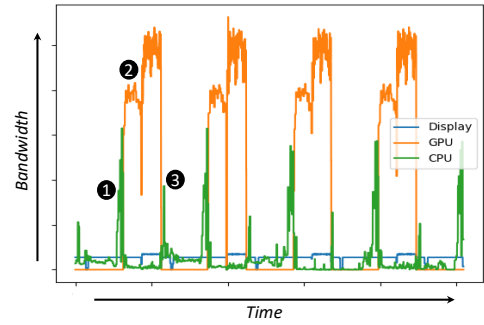**Figure 9: GPU execution time under a regular load**

Figure 10: M3-HMC DRAM bandwidth

meeting the deadline, DASH either fully prioritizes CPU threads (if classified as memory non-intensive) or probabilistically prioritizes CPU threads (if classified as memory intensive) over that of the GPU. The final outcome from the user's perspective has not changed in this case as the application still meets the target frame rate. However, GPU execution taking longer may increase energy consumption, which would be detrimental for mobile SoCs. In addition, this scheduling policy might unintentionally hurt performance as well (Section 5.2.2).

For HMC, we found two reasons for the slower GPU performance. The first reason is that traffic from CPU threads and the GPU was not balanced throughout the frame. When GPU rendering is taking place, CPU-side traffic reduces significantly and the CPU-assigned channels are left underutilized. Figure 10 presents M3-HMC memory bandwidth from each source over time. In ❶ the CPU traffic increases before starting a new frame. Once a frame started, CPU traffic is reduced (❷). This continues until the end of the GPU frame at ❸. As a result, the split-DRAM channel configuration becomes problematic in such cases due to the lack of continuous traffic balance.

The second factor in reduced GPU performance under HMC was the lower row-buffer locality at IP-assigned channels. The issue originates from HMC assumption that all IP traffic will use sequential accesses when accessing DRAM. Although this is true for our display controller, we found that it is not the case for our GPU traffic (we were unable to obtain the traces used in the original HMC study [20] to compare against). As a consequence, IP-assigned channels suffer from lower row-buffer hit rates and reduced number
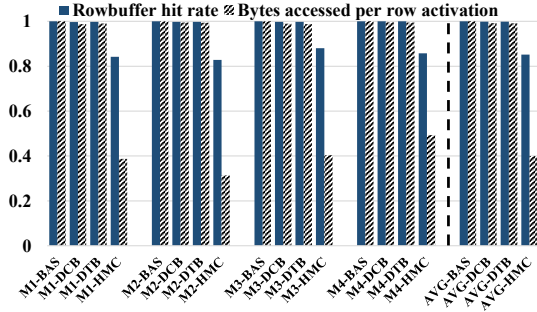
Figure 11: Page hit rate and bytes accessed per row activation normalized to baseline
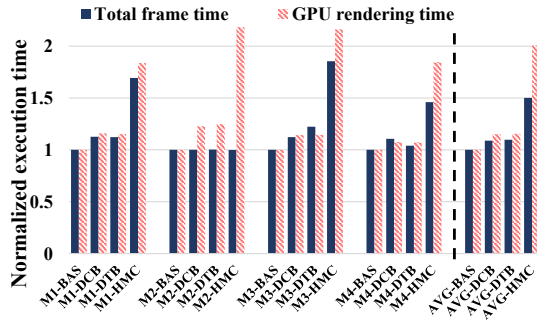

Figure 12: Performance under the high-load scenario

of bytes fetched per row-buffer activation, where the latter indicates higher per byte energy cost. Figure 11 shows HMC DRAM page hit rate and bytes accessed per row activation compared to the baseline. On average, the row buffer hit rate decreases by 15% and the number of bytes accessed per row activation drops by around 60%.

5.2.2 High-load scenario. In this scenario, we evaluate DASH and HMC under lower DRAM bandwidth to analyze system behavior under high memory loads. Figure 12 shows normalized execution times (lower is better).

First, HMC shows similar behavior to that described earlier in Section 5.2.1. With lower page hit rate and reduced locality; it takes on average 45% longer than the baseline to produce a frame.

We found DASH reduces frame rates compared to the baseline by an average of 8.9% and 9.7% for DCB and DTB configurations, respectively. In larger models (M1 & M3) the drop in frame rates was around 12%. The time it takes to render a GPU frame was increased by an average of 15.1% M1-DCB and 15.5% for M1-DTB. Figure 12 also shows that simpler models (M2 & M4) experience lower slowdowns as the DRAM still manages to provide a frame rate close to target even with slower GPU performance.

In addition to GPU performance, we also looked into the display controller performance. The application rendering process is independent of screen refreshing by the display controller, where the display controller will simply re-use the last complete frame if no new frame is provided. As a result of high-load, and unlike the low-load scenario, the display controller suffered from below target
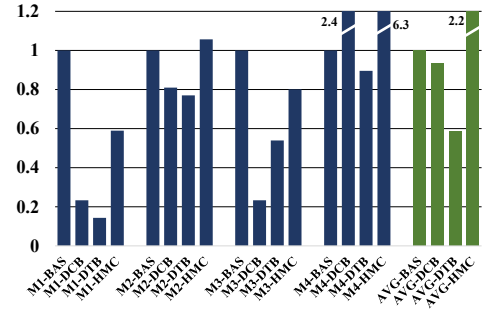
Figure 13: Number of display requests serviced relative to BAS

frame rate. Figure 13 shows the normalized display traffic serviced in each configuration. First, we notice that HMC outperforms BAS, DCB and DTB with the smaller models (M2 & M4). What we found is that since the GPU load is smaller in these two cases (as shown in Figure 12), the IP-assigned channel was available for longer periods of times to service display requests without interference from CPU threads. However, as we can see in Figure 12, the overall performance of the application does not improve over BAS and DASH configurations.
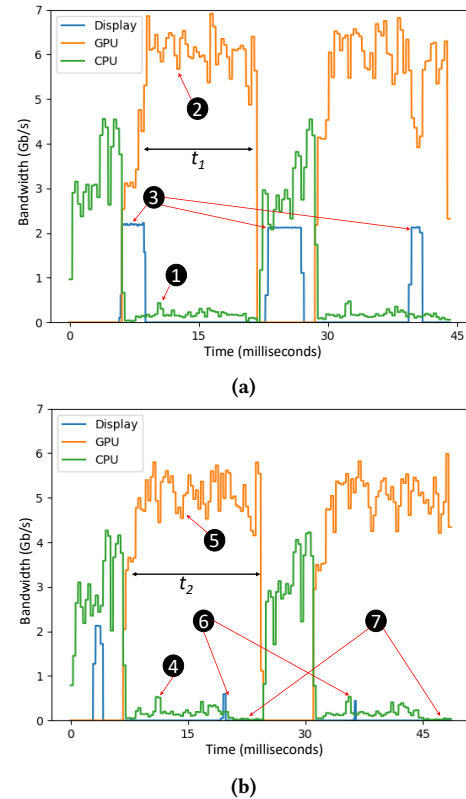

(a)

(b)

Figure 14: M1 Rendering by BAS (a) and DTB (b)

For larger models (M1 & M3) DCB and DTB both deliver lower display traffic and lower overall application frame rate. We can

see the reason for DASH's lower overall performance using the M1 traffic examples in Figure 14 and by discerning the behavior of BAS and DCB traffic. Comparing BAS CPU and GPU traffic to that of DASH, we see that DASH provides higher priority to CPU threads (❹) compared to the FRFCFS baseline (❶). This is because GPU timing is still meeting the set deadline and, as a consequence, classified as non-urgent. CPU threads either have absolute priority (if memory non-intensive) or a probabilistic priority (if memory intensive). By looking at the periods t_1 in Figure 14a and t_2 in Figure 14b we found that GPU read requests latencies are higher by 16.2%, on average, in t_2 compared to t_1, which leads to lower GPU bandwidth in ❺ compared to ❷.

The prioritization of CPU requests during t_2, however, does not improve overall application performance. Looking at ❼ we can see that CPU threads are almost idle at the end of the frame waiting for the GPU frame to finish before moving to the next job. This dependency is not captured by DASH's memory access scheduling algorithm and leads to over-prioritization of CPU threads during t_2; consequently, DASH can reduce DRAM performance compared to FR-FCFS.

Examining display controller performance we find DASH DTB services 85% less display bandwidth than BAS (as shown in Figure 13 and Figure 14 ❸ vs. ❻). Looking at Figure 14b ❻, we can see the reason for DASH's lower display performance. The display controller starts a new frame in ❻, the frame is considered non-urgent because it just started and had not missed the expected progress yet. In addition, the CPU is consuming additional bandwidth as discussed earlier. Both factors lead to fewer display requests to be serviced early on. Eventually, because of the below-expected bandwidth, the display controller aborts the frame and re-try a new frame later where the same sequence of events reoccurs (as highlighted in ❻).

5.2.3 Summary and discussion. In this case study, we evaluated two proposals for memory organization and scheduling for heterogeneous SoCs. We aimed to test these proposals under execution-driven simulation compared to trace-based simulations originally used to evaluate these proposals. We found some issues that trace-based simulation could not capture fully, including incorporating inter-IP dependencies, responding to system feedback, or lacking the details of IP access patterns.

For inter-IP dependencies, traces can be created including inter-IP dependencies. However, adding dependency information is non-trivial for complex systems that feature several IPs and multi-threaded applications. This complexity requires an adequate understanding of workloads and reliable sources of traces. GemDroid collects traces using a single-threaded software emulator and incorporates rudimentary inter-IP dependencies which are used to define mutually exclusive execution intervals, e.g., CPU vs. other IPs. The evaluation of DASH performed by Usui et al. [74] used a mix of traces from unrelated processes "that execute independently" and "does not model feedback from missed deadlines". In contrast, we find traffic in real systems is more complex as it is generated by inter-dependent IP blocks, sometimes working concurrently, as shown in the examples in Figure 10 and Figure 14.

We observed the display controller dropping frames when failing to meet a deadline. Thus, modeling system feedback between IP
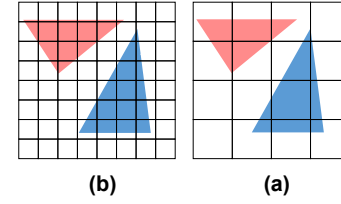

Figure 15: Fine (a) vs. Coarse (b) screen-space division for fragment shading.

blocks appears important, at least when similar scenarios are relevant to design decisions. Another issue highlighted by our study is the impact of simulating SoCs without a detailed GPU model. HMC was designed assuming IP traffic that predominantly contains sequential accesses which is not true for graphics workloads. A related key hurdle for academic researchers is difficulty obtaining real traces from existing (proprietary) IP blocks; our results suggest it the value of creating detailed IP models that produce representative behavior. Detailed IP models provide the ability to evaluate intra-IP architectural changes and their system level impact.

6 CASE STUDY II: DYNAMIC FRAGMENT SHADING LOAD-BALANCING (DFSL)

In this section, we propose and evaluate a method for dynamically load-balancing the fragment shading stage on the GPU. This is achieved by controlling the granularity of the work assigned to each GPU core.

The example in Figure 15 shows two granularities to distribute work amongst GPU cores. The screen-space is divided into tiles, where each tile is assigned to a GPU cluster/core as described in Section 3.3.2. In Figure 15a, smaller tiles are used to distribute load amongst GPU cores. When assigning these tiles to the GPU cores (e.g., round-robin), we improve load-balance across the cores. On the other hand, using larger tiles, as in Figure 15b, load-balance is reduced but locality improves. Locality might boost performance by sharing data across fragments as in texture filtering, where the same texels can be re-used by multiple fragments. Also, as we will see later, load-balance can change from one scene to another based on what is being rendered.

6.1 Experimental Setup

For this case study, we evaluate GPU performance using Emerald standalone mode (Section 4.1). Since this case-study focuses on the fragment shading stage, we only show performance results for fragment shading. We used a GPU configuration that resembles a high-end mobile GPU [51, 52] (Table 7). We selected a set of relatively simple 3D models frequently used in graphics research, which are listed in Table 8 and shown in Figure 16 (figures shown were rendered with Emerald); using more complex workloads, e.g., game frames, is possible but requires much longer simulation times with limited additional benefit for most architecture tradeoff studies. For work granularity we use *work tile* (***WT***) to define round-robin work granularity when assigning work to GPU cores (as explained in Figure 15). A WT of size N is N×N TC tiles, where N≥1. As
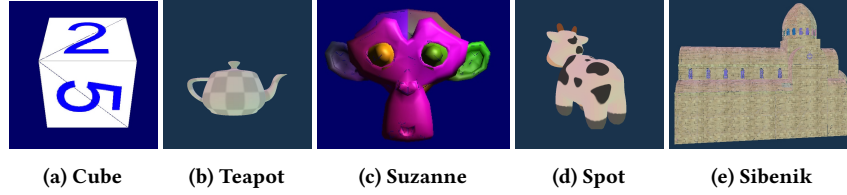
| (a) Cube | (b) Teapot | (c) Suzanne | (d) Spot | (e) Sibenik |

Figure 16: Case study II workloads

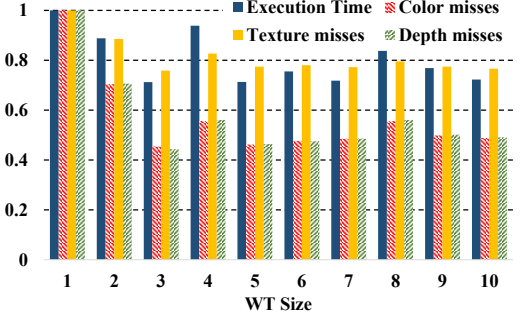| # SIMT Clusters	6 (192 CUDA Cores)
SIMT Core Freq.	1GHz
Max Threads per core	2048
Registers per core	65536
Lanes per SIMT Core	32 (warp size)
L1D	32KB, 128B line, 8-way LRU
L1T	48KB, 128B line, 24-way LRU
L1Z	32KB, 128B line, 8-way LRU
Shared L2	2MB, 128B line, 32-way LRU
Raster tile	4×4 pixels
TC tile size	2×2
TC engines per cluster	2
TC bins per engine	4
Coarse & fine raster throughput	1 raster tile/cycle
Hi-Z throughput	1 raster tile/cycle
Memory	4 channel LPDDR-3 1600Mb/s

Table 7: Case Study II GPU configuration

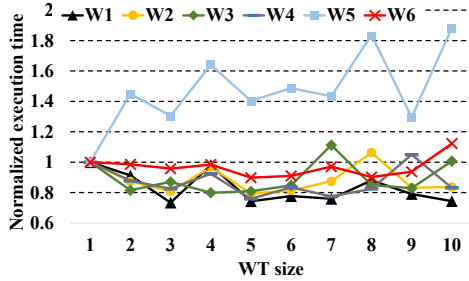| Model Abbrv.	Name	Textured?	Translucent?
W1	Sibenik [46]	Yes	No
W2	Spot [22]	Yes	No
W3	Cube [46]	Yes	No
W4	Blender's Suzanne	Yes	No
W5	Suzanne transparent	Yes	Yes
W6	Utah's Teapot [46]	Yes	No

Table 8: Case Study II workloads

Figure 17: Frame execution time for WT sizes of 1-10 normalized to WT of 1

discussed in Section 3.3.2, the minimum work unit that can be assigned to a core is a TC tile.

6.2 Load-Balance vs. Locality

Figure 17 shows the variation in frame execution time for WT sizes 1 to 10. For WT sizes larger than 10 the GPU is more prone to load-imbalance. As we can see in Figure 17, frame execution time can vary by 25% in W6 to as much as 88% in W5. The WT size

Figure 18: Normalized (to WT1) execution times and the total L1 cache misses of various caches for W1

that achieves the optimal performance varies from one workload to another; for W5, the best-performing WT size is 1, and for W2 and W4 the best performing WT size is 5.

We looked at different factors that may contribute to the variation of execution time with WT size. First, we noticed that L2 misses/DRAM traffic are very similar across WT sizes. However, we found that L1 cache miss rates significantly change with WT size. Figure 18 shows execution times and L1 misses for a W1 frame vs. WT sizes. The figure shows that L1 cache locality is a significant factor in performance. Measuring correlation, we found that execution time correlates by 78% with L1 misses, 79% with L1 depth misses and 82% with texture misses.

6.3 Dynamic Fragment Shading Load-Balancing (DFSL)

In this section, we evaluate our proposal of dynamic fragment shading load-balancing (DFSL). DFSL exploits temporal coherence in graphics [63], where applications exhibit minor changes between frames.

DFSL exploits graphics temporal coherence in a novel way – by utilizing it to dynamically adjust work distribution across GPU cores so as to reduce rendering time. The goal of DFSL is not to achieve consistently higher frame rates but rather to lower GPU energy consumption by reducing average rendering time per frame assuming the GPU can be put into a low power state between frames when it is meeting its frame rate target.

DFSL works by running two phases, an evaluation phase and a run phase. Algorithm 1 shows DFSL's evaluation and run phases. For a number of frames equal to possible WT sizes, the evaluation phase (lines 13-25), renders frames under each possible WT value. At the end of the evaluation phase, WT with the best performance

---

Algorithm 1 Find and render with best WT sizes

```
 1: Parameters
 2:   RunFrames: number of run (non-evaluation) frames
 3:   MinWT: minimum WT size
 4:   MaxWT: maximum WT size
 5:   MAX_TIME: maximum execution time
 6:
 7: Initialization
 8:   -CurrFrame ← 0
 9:   -EvalFrames ← MaxWT − MinWT
10:
11: procedure DFSL RUN
12:   while there is a new frame:
13:     if CurrFrame%(EvalFrames+RunFrames) == 0 then
14:       MinExecTime ← MAX_TIME
15:       WTSize ← MinWT
16:       WTBest ← MinWT
17:     end if
18:
19:     if CurrFrame%(EvalFrames+RunFrames) < EvalFrames then
20:       ExecTime ←execution time with WTSize
21:       if ExecTime < MinExecTime then
22:         MinExecTime ← ExecTime
23:         WTBest ← WTSize
24:       end if
25:       WTSize ← WTSize + 1
26:     else
27:       Render frame using WTBest
28:     end if
29:     CurrFrame ← CurrFrame + 1
30: end procedure
```
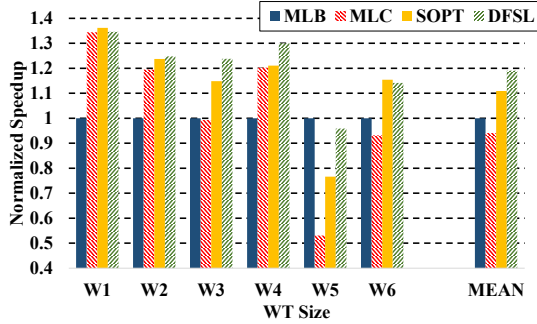
Figure 19: Average frame speedup normalized to MLB

($WTBest$) is then used in the run phase for a set of frames, i.e., *RunFrames*. At the end of the run phase, DFSL starts another evaluation phase and so on. By changing *RunFrames*, we can control how often $WTBest$ is updated.

Implementation. DFSL can be implemented as part of the graphics driver, where DFSL can be added to other context information tracked by the driver. DFSL Algorithm 1 will only execute a few times per second (i.e., at the FPS rate). For each application, the GPU tracks the execution time for each frame and $WTBest$. In our experiment, each workload used 1-2 draw calls and we tracked $WTBest$ on per frame basis. DFSL can be extended to also track $WTBest$ at the draw call level or for a set of draw calls in more complex workloads.

In Figure 19, DFSL performance is compared against four static configurations. **MLB** for maximum load-balance using WT size of 1, **MLC** for maximum locality using WT size of 10, and **SOPT**. To find **SOPT**, we ran all the frames across all configs and found

the best WT, on average, across all workloads. For DFSL, we used an evaluation period of 10 frames and a run period of 100 frames. Results are shown in Figure 19, where it shows that DFSL is able to speed up frame rendering by an average of 19% compared to **MLB** and by 7.3% compared to **SOPT**.

7 RELATED WORK

System simulation. Researchers created several tools to simulate multi-core and heterogeneous systems [17, 19, 55](see Table 1). However, these simulators have focused on simulating heterogeneous CPU cores or lack the support for specialized cores like GPUs and DSPs. Other work focused on CPU-GPGPU simulation [56, 73], while gemDroid provided an SoC simulation tool that combines software-model traces using gem5 DRAM model [20, 48, 49, 77]. Another gem5 based simulator, gem5-aladdin [66], provides a convenient way to model specialized accelerators using dynamic traces. Aladdin and Emerald can be integrated to provide a more comprehensive simulation infrastructure.

GPU Simulators. Attila [25] is a popular graphics GPU simulator that models an IMR architecture with unified shaders. The Attila project has been inactive for a few years and their custom graphics driver is limited to OpenGL 2.0 and D3D 9. Another tool, the Teapot simulator [13], has been used for graphics research using a TBR architecture and OpenGL ES [8, 14]. Teapot, however, is not publicly available. In addition, Teapot models a pipeline with an older architecture with non-unified shader cores. Qsliver [67] is an older simulation infrastructure that also uses non-unified shaders. Finally, a more recent work, GLTraceSim [65], looks at the behavior of graphics-enabled systems. GLTraceSim does not model a particular GPU architecture, but it approximates GPU behavior using the memory traces generated by the functional model provided by Mesa 3D [1].

Besides graphics GPU simulators, other GPU simulators focus on GPGPU applications; this includes simulators like Macsim [38] and GPGPU-Sim [16], where Emerald builds on the latter.

SoC memory scheduling. Several techniques were developed for memory scheduling in multi-core CPU systems [15, 39, 40, 43, 69]. For GPUs, the work by Jog et al. focused on GPU inter-core memory scheduling in GPGPU workloads [35]. Another body of work proposed techniques for heterogeneous systems [34, 58, 74]. Previous work on heterogeneous systems, however, relied on using a mix of independent CPU and GPU workloads, rather than using workloads that utilize both processors. In our work, we introduce an extensive infrastructure to enable the simulation of realistic heterogeneous SoC workloads.

Load-Balancing on GPUs. Several proposals tackled load-balancing for GPGPU workloads. Son et al. [68] and Wang et al. [76] worked on scheduling GPGPU kernels. Other techniques use software methods to exploit GPGPU thread organization to improve locality [42, 75, 76]. On the other hand, DFSL tries to find a solution for workload execution granularity, where the balance between locality and performance is reached through exploiting graphics temporal coherence. Other GPGPU scheduling techniques, e.g., the work by Lee et al. [41], can be adopted on top of DFSL to control the flow of execution on the GPU.

Temporal coherence in graphics. Temporal coherence has been exploited in many graphics software techniques [63]. For hardware, temporal coherence has been used to reduce rendering cost by batch rendering every two frames [12], use visibility data to reduce redundant work [24], and in tile-based rendering to eliminate redundant tiles [32]. In this work, we present a new technique for exploiting temporal coherence to balance fragment shading on GPU cores.

8 CONCLUSION AND FUTURE WORK

This work introduces Emerald, a simulation infrastructure that is capable of simulating graphics and GPGPU applications. Emerald is integrated with gem5 and Android to provide the ability to simulate mobile SoC systems. We present two case studies that use Emerald. The first case study evaluates previous proposals for SoC memory scheduling/organization and shows that different results can be obtained when using detailed simulation. The case study highlights the importance of incorporating dependencies between components, feedback from the system, and the timing of events when evaluating SoC behavior. In the second case study, we propose a technique (DFSL) to dynamically balance the fragment shading stage on GPU cores. DFSL exploits graphics temporal coherence to dynamically update work distribution granularity. DFSL improves execution times by 7-19% over static work distribution.

For future work, and in addition to improving graphics modeling (as highlighted in subsection 3.5) and developing Emerald compatible GPUWattch configurations for mobile GPUs, we plan to create a set of mobile SoC benchmarks for Emerald that represent essential mobile uses-cases running commonly used Android applications.

ACKNOWLEDGMENTS

REFERENCES

[1] Mesa 3D. [n. d.]. The Mesa 3D Graphics Library. http://www.mesa3d.org/ [Online; accessed 25-Apr-2019].

[2] Michael Abrash. 2009. Rasterization on larrabee. *Dr. Dobbs Journal* (2009). http://www.drdobbs.com/parallel/rasterization-on-larrabee/217200602

[3] Niket Agrawal, Amit Jain, Dale Kirkland, Karim Abdalla, Ziyad Hakura, and Haren Kethareswaran. 2017. Distributed index fetch, primitive assembly, and primitive batching. US Patent App. 14/979,342.

[4] AMD Radeon Technologies Group. 2017. Radeon's next-generation Vega architecture. https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf [Online; accessed April 25, 2019].

[5] Michael Anderson, Ann Irvine, Nidish Kamath, Chun Yu, Dan Chuang, Yushi Tian, and Yingyong Qi. 2005. Graphics pipeline and method having early depth detection. US Patent App. 10/949,012.

[6] Anderson, Michael and Irvine, Ann and Kamath, Nidish and Yu, Chun and Chuang, Dan and Tian, Yushi and Qi, Yingyong and others. 2004. Graphics pipeline and method having early depth detection. US Patent 8184118.

[7] Android. [n. d.]. Android Goldfish OpenGL. https://android.googlesource.com/device/generic/goldfish-opengl [Online; accessed April 25, 2019].

[8] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L Aragón, and Antonio González. 2019. Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 635–646.

[9] APITrace. [n. d.]. APITrace. http://apitrace.github.io [Online; accessed April 25, 2019].

[10] Apple. 2019. About GPU Family 4. https://developer.apple.com/documentation/metal/mtldevice/ios_and_tvos_devices/about_gpu_family_4 [Online; accessed April 25, 2019].

[11] ARM. 2013. big.LITTLE Technology: The Future of Mobile. *ARM whitepaper* (2013). https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[12] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE Press, 83–92.

[13] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2013. TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. ACM, 37–46.

[14] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2014. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE Press, 529–540.

[15] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 416–427.

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

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[18] Broadcom. 2013. VideoCore IV 3D Architecture Reference Guide. https://docs.broadcom.com/docs/12358545 [Online; accessed April 25, 2019].

[19] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages.

[20] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2014. GemDroid: A Framework to Evaluate Mobile Platforms. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Science*. ACM, 355–366.

[21] Christoph Kubisch. 2015. Life of a triangle - NVIDIA's logical pipeline. https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline [Online; accessed April 25, 2019].

[22] Keenan Crane. [n. d.]. Keenan's 3D Model Repository. https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/ [Online; accessed April 25, 2019].

[23] J. Davies. 2016. The bifrost GPU architecture and the ARM Mali-G71 GPU. In *IEEE Hot Chips Symposium (HCS)*. 1–31.

[24] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio Gonzalez. 2019. Visibility Rendering Order: Improving Energy Efficiency on Mobile GPUs through Frame Coherence. *IEEE Transactions on Parallel and Distributed Systems* 30, 2 (Feb 2019), 473–485.

[25] Victor Moya Del Barrio, Carlos González, Jordi Roca, Agustín Fernández, and E Espasa. 2006. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 231–241.

[26] LUM Eric, Walter R Steiner, and Justin Cobb. 2016. Early sample evaluation during coarse rasterization. US Patent 9,495,781.

[27] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 365–376.

[28] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.

[29] Ziyad Hakura, LUM Eric, Dale Kirkland, Jack Choquette, Patrick R Brown, Yury Y Uralsky, and Jeffrey Bolz. 2018. Techniques for maintaining atomicity and ordering for pixel shader operations. US Patent App. 10/019,776.

[30] Ziyad S Hakura, Michael Brian Cox, Brian K Langendorf, and Brad W Simeral. 2010. Apparatus, system, and method for Z-culling. US Patent 7,755,624.

[31] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 201–210.

[32] Peter Harris. 2014. The Mali GPU: An Abstract Machine, Part 2 - Tile-based Rendering. https://community.arm.com/graphics/b/blog/posts/

the-mali-gpu-an-abstract-machine-part-2---tile-based-rendering [Online; accessed April 29, 2019].

[33] Harvard Architecture, Circuits, and Compilers Group. [n. d.]. Die Photo Analysis. http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis [Online; accessed April 25, 2019].

[34] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the Annual Design Automation Conference (DAC)*. ACM, 850–855.

[35] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Science*. ACM, 351–363.

[36] Khronos Group. 2017. The OpenGL Graphics System: A Specification (Version 4.5 Core Profile). https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf [Online; accessed April 29, 2019].

[37] Emmett M Kilgariff, Steven E Molnar, Sean J Treichler, Johnny S Rhoades, Gernot Schaufler, Dale L Kirkland, Cynthia Ann Edgeworth Allison, Karl M Wurstner, and Timothy John Purcell. 2014. Hardware-managed virtual buffers using a shared memory for load distribution. US Patent 8,760,460.

[38] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012). http://comparch.gatech.edu/hparch/macsim/macsim.pdf [Online; accessed April 29, 2019].

[39] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–12.

[40] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 65–76.

[41] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 260–271.

[42] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 297–311.

[43] Zhonghai Lu and Yuan Yao. 2016. Aggregate Flow-Based Performance Fairness in CMPs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4, Article 53 (Dec. 2016), 27 pages.

[44] Eric B Lum, Justin Cobb, and Barry N Rodgers. 2017. Pixel serialization to improve conservative depth estimation. US Patent 9,684,998.

[45] Michael Mantor, Laurent Lefebvre, Mikko Alho, Mika Tuomi, and Kiia Kallio. 2016. Hybrid render with preferred primitive batch binning and sorting. US Patent App. 15/250,357.

[46] Morgan McGuire. 2017. Computer Graphics Archive. https://casual-effects.com/data [Online; accessed April 25, 2019].

[47] Steven E Molnar, Emmett M Kilgariff, Johnny S Rhoades, Timothy John Purcell, Sean J Treichler, Ziyad S Hakura, Franklin C Crow, and James C Bowman. 2013. Order-preserving distributed rasterizer. US Patent 8,587,581.

[48] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, Ravi Iyer, and Chita R Das. 2015. Domain knowledge based energy management in handhelds. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 150–160.

[49] Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2015. VIP: Virtualizing IP Chains on Handheld Platforms. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA) (ISCA '15)*. ACM, 655–667.

[50] Nvidia. 2009. NVIDIA's Next Generation CUDA Compute Architecture. (2009). https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf [Online; accessed 25-Apr-2019].

[51] NVIDIA. 2014. K1: A new era in mobile computing. *Nvidia, Corp., White Paper* (2014). https://www.nvidia.com/content/PDF/tegra_white_papers/Tegra_K1_whitepaper_v1.0.pdf [Online; accessed April 25, 2019].

[52] Nvidia. 2015. NVIDIA Tegra X1. *Nvidia whitepaper* (2015). https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf [Online; accessed April 25, 2019].

[53] Nvidia. 2019. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html [Online; accessed April 25, 2019].

[54] NVIDIA, Tesla. 2008. A Unified Graphics and Computing Architecture. *IEEE Computer Society* (2008), 0272–1732.

[55] Avadh Patel, Furat Afram, and Kanad Ghose. 2011. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st

International Qemu Users' Forum*. 29–30.

[56] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. 2015. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letter* 14, 1 (Jan 2015), 34–36.

[57] Qualcomm. 2013. FlexRender. https://www.qualcomm.com/videos/flexrender [Online; accessed April 25, 2019].

[58] Siddharth Rai and Mainak Chaudhuri. 2017. Improving CPU Performance through Dynamic GPU Access Throttling in CPU-GPU Heterogeneous Processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 18–29.

[59] Johnny S Rhoades, Steven E Molnar, Emmett M Kilgariff, Michael C Shebanow, Ziyad S Hakura, Dale L Kirkland, and James Daniel Kelly. 2014. Distributing primitives to multiple rasterizers. US Patent 8,704,836.

[60] Rys Sommefeldt. 2015. A look at the PowerVR graphics architecture: Tile-based rendering. https://www.imgtec.com/blog/a-look-at-the-powervr-graphics-architecture-tile-based-rendering

[61] Samsung. 2013. Samsung Exynos 5410. *Samsung whitepaper* (2013). https://pdfs.semanticscholar.org/54c4/6e6cd3ac84ab5c8586760d9b7cb62cd3427b.pdf

[62] Andreas Sandberg et al. 2016. NoMali: Simulating a realistic graphics driver stack using a stub GPU. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 255–262.

[63] Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V. Sander, Michael Wimmer, and Elmar Eisemann. 2012. Temporal Coherence Methods in Real-Time Rendering. *Computer Graphics Forum* 31, 8 (2012), 2378–2408.

[64] Larry D Seiler and Stephen L Morein. 2011. Method and apparatus for hierarchical Z buffering and stenciling. US Patent 7,978,194.

[65] Andreas Sembrant, Trevor E Carlson, Erik Hagersten, and David Black-Schaffer. 2017. A graphics tracing framework for exploring CPU+ GPU memory systems. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 54–65.

[66] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and soc interfaces using gem5-aladdin. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[67] Jeremy W Sheaffer, David Luebke, and Kevin Skadron. 2004. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. ACM, 85–94.

[68] Dong Oh Son, Cong Thuan Do, Hong Jun Choi, Jiseung Nam, and Cheol Hong Kim. 2017. A Dynamic CTA Scheduling Scheme for Massive Parallel Computing. *Cluster Computing* 20, 1 (March 2017), 781–787.

[69] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2014. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*. IEEE, 8–15.

[70] TechInsights. 2017. Huawei Mate 10 Teardown. https://www.techinsights.com/about-techinsights/overview/blog/huawei-mate-10-teardown

[71] TechInsights. 2018. Samsung Galaxy S9 Teardown. https://www.techinsights.com/about-techinsights/overview/blog/samsung-galaxy-s9-teardown [Online; accessed April 25, 2019].

[72] Qualcomm Technologies. 2014. THE RISE OF MOBILE GAMING ON ANDROID: QUALCOMM SNAPDRAGON TECHNOLOGY LEADERSHIP. https://developer.qualcomm.com/qfile/27978/rise-of-mobile-gaming.pdf [Online; accessed April 25, 2019].

[73] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: a simulation framework for CPU-GPU computing. In *Proceedings of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 335–344.

[74] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4, Article 65 (Jan. 2016), 28 pages.

[75] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 829–842.

[76] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 358–369.

[77] Praveen Yedlapalli, Nachiappan Chidambaram Nachiappan, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. 2014. Short-circuiting memory traffic in handheld platforms. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 166–177.