Microarchitectural Simulator for Shader Cores in a Modern GPU Simulation Infrastructure

Diya Joseph

July 2019

MASTER THESIS
MIRI - HPC
Universitat Politècnica de Catalunya, Spain

Supervisors:
Martí Anglada and Antonio González

Acknowledgement

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. Firstly, Prof. Antonio González for his guidance and patience during this project and the writing of this thesis. I would also like to thank my committee members, for their patience and understanding. Additionally, I would like to thank Martí Anglada without whose constant guidance, heartfelt involvement and timely cheerleading, this project would not have been fruitful.

Abstract

Mobile devices have emerged as one of the most rapidly spread technologies, with users spending a significant amount of time playing games on these devices. The evolution of smartphone games along with a supplementary increase in resolution has lead to a growing demand for more visually compelling graphics on mobile devices, which require significant energy consumption to maintain. However, the battery capacity of these devices does not grow at the same time as the computing capabilities, creating an ever-increasing gap. Therefore, it has become increasingly important to facilitate the study of energy-efficient architectures for GPUs in mobile devices. Inspite of this, accurate full-system simulators for mobile graphics systems are rare.

TEAPOT is a cycle-accurate simulator for mobile-GPU systems and is the state of the art in this area. There were certain aspects of the shader core in TEAPOT that were identified to be capable for improvement. The objective of this project is to redesign the shader cores in TEAPOT to match the contemporary microarchitecture of shader cores. The register file was changed and a banking mechanism used. A new stage named 'Operand Collector' was added in order to buffer instructions with register bank conflicts. A new Issue Scheduling mechanism was implemented. The width of the pipeline was changed to two. An I-Buffer was included in the Fetch stage. The Execute unit was pipelined and more write ports were added to the Writeback stage. After this, the functionality and accuracy of the new features of the implemented model were validated using various tests. The tests involved observing different metrics of the shader core, like IPC, while some parameters of the newly implemented model were varied.

Lastly, three experiments were conducted in order to explore some microarchitectural designs with a representative set of current mobile graphics applications. With these, we first studied the improvement in performance using the register allocation scheme used in TEAPOT. We also found the optimum range of the total number of warps and the optimum register file size for the benchmarks used.

Contents

Chapter 1

Introduction

Mobile technology has spread rapidly around the globe. Nowadays, more than 5 billion people have mobile devices, and over half of these connections are smartphones [1]. Figure 1.1 shows the number of smartphones sold to end users worldwide from 2007 to 2018. For 2018, we see that this number is around 1.56 billion [2]. This is a significant increase from the 680 million units sold in 2012. It is estimated that over 28 percent of the world's total population owned a smart device in 2016, a figure that is expected to increase to 37 percent by 2020. In the same year smartphone penetration is set to reach 60.5 percent in North America as well as in Western Europe. This is a large rise in the 29.3 percent of people in North America and 22.7 percent of Western Europeans who had smartphones in 2011 [1].

It is evident that mobile devices have emerged as one of the most rapidly spread technology [3]. At the same time, mobile user interfaces have evolved from simple text-based displays to interactive high definition 3D graphics. Smartphones cater to a variety of uses these days such as mobile web browsing, gaming, voice calls and multimedia capabilities. It is noteworthy that users spend 43% of their "smartphone time" playing games [4]. As of 2019, there are 2.2 billion mobile gamers worldwide [4]. As the interest to play videos in smartphones has grown, so has their screen size and resolution. In order to provide the necessary frame rates, mobile phones had to deviate from the early software rendering approaches and embrace a hardware-based accelerator, a Graphics Processing Unit (GPU). With greater rendering capabilities, the multimedia and gaming demands increased, and bigger phones with Full HD resolution and more powerful GPUs to sustain the frame rates

were released. According to various studies the GPU and the screen have been seen to consume the most battery power [5, 6].

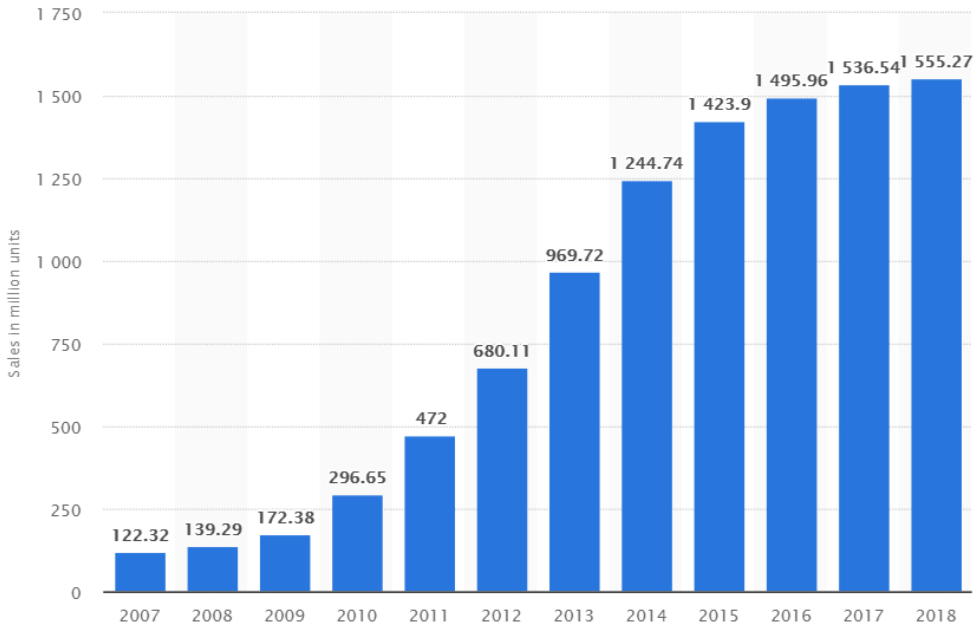


Figure 1.1: Number of smartphones sold to end users worldwide from 2007 to 2018 (in million units)[1]

A growing demand for more visually compelling graphics on mobile requires more computing capabilities, which in turn consumes more energy. Energy is a scarce resource in mobiles as all of the mobile segment devices are battery-operated and thus power consumption is a key design aspect. Therefore, it has become increasingly important to facilitate the study of energy-efficient architectures for GPUs in mobile devices.

Prior work on GPU simulation tools [7, 8, 9, 10] focuses mainly on modelling desktop-like GPUs. These simulators do not represent the mobile segment. In fact, none of them have support for the OpenGL ES API [11], which means that they cannot run smartphone applications. Furthermore, most of them do not provide a power model. It is also noteworthy that none of the existing simulators provide energy estimations for the OLED screens. This is an important aspect of graphics simulation as the screen is one of the

main battery consumers and its energy depends on the output generated by the GPU [12]. Full system GPU simulation is another important benefit provided by our simulator that is not available in any existing GPU simulation infrastructure. This means that frequent GPU-related tasks such as image composition, or rendering of background applications (e.g., advertisements), are usually not taken into consideration. It has been shown [13] that the OS GPU usage is non-negligible, representing up to 52% of GPU time and up to 48% of GPU energy for commercial Android games. TEAPOT, is one such toolset for evaluating the performance, energy and image quality of a mobile graphics subsystem, that facilitates to this need [13]. It is developed in the ARCO lab.

1.1 Motivation

Although TEAPOT is the state-of-the-art simulator for mobile Graphics Processors, it has components that are not representative of contemporary architectures. The main one that we identified was the shader core, which requires accurate modelling as it is the part that contributes most towards energy consumption [6, 14, 15, 16]. The shader core presently is a very simple in-order, SIMT processor with four pipeline stages and non-pipelined functional units. It has a huge register file with more than sufficient number of registers for each warp. In order to conduct accurate microarchitectural research on the shader cores of GPUs, this part of the simulator needed to be redesigned to match contemporary architectures.

1.2 Objectives

- Build a cycle-accurate simulator of the shader core of GPUs close to a contemporary GPU and integrate it into the already existing simulation framework of TEAPOT.
- Validate the model, built and embedded into TEAPOT, by performing tests and analyzing the results.
- Test alternative microarchitectural configurations for the new shader core and analyze differences in performance and resource utilization.

Chapter 2

Background

This chapter lays some groundwork to facilitate understanding the details of this project. The Graphics pipeline is first explained to give context to the purpose of a shader core. The second gives an overview of how TEAPOT works and the architecture it uses. The last section is about related work in the area of GPU simulators.

2.1 The Graphics Pipeline

Rendering or image synthesis is the automatic process of generating a two dimensional image from a two-dimensional or three-dimensional model, given a virtual camera, light sources, texture information or more inputs [17]. This process is performed in a pipeline which is shown in Figure 2.1.

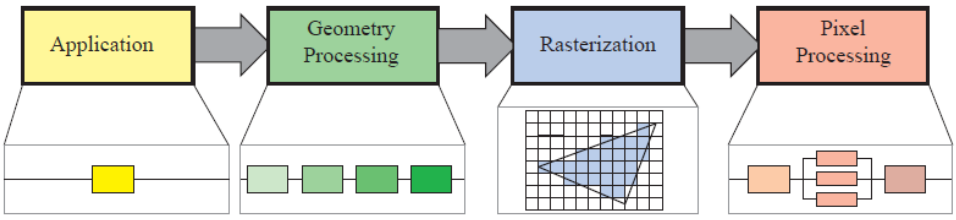


Figure 2.1: The basic construction of the Graphics Pipeline [18]

The application stage is typically implemented in software running on general-purpose CPUs, which performs some tasks such as collision detec-

tion, global acceleration algorithms, animation, physics simulation. Since CPUs commonly include multiple cores that are capable of processing multiple threads of execution in parallel, CPUs are able to efficiently run the large variety of tasks that are the responsibility of the application stage. The application stage also communicates with a Graphics Processing Unit (GPU), an accelerator for rendering.

The next main stage is geometry processing, which deals with transforms, projections, and all other types of geometry handling. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a GPU that contains many programmable cores as well as fixed-operation hardware. Here the pipeline receives a list of vertices as its input, and executes the vertex shader: a program that applies lighting operations and unifies all the coordinate systems of the different objects of the scene. This program is run on a vertex shader core. These shaded vertices are then assembled into triangles and projected onto the screen plane, which allows to discard the triangles that would appear off-screen.

The rasterization stage typically takes as input three vertices that form a triangle, and generates the list of fragments that cover it. Fragments are pixel-sized elements with the necessary information to compute the colour of a pixel, mainly interpolated data from the shaded vertices.

Finally, in the the pixel processing stage, these fragments are processed by the fragment shader cores, processors that compute the final colour of the visible pixels based on user-defined programs that consider texture and light information. This stage may involve depth testing to see whether the computed colour of the pixel is visible or not. It may also perform per-pixel operations such as blending the newly computed color with a previous color.

2.2 TEAPOT Description

TEAPOT [13] is a toolset for evaluating the performance, energy and image quality of a mobile graphics subsystem. It provides full-system simulation of unmodified commercial Android applications. It includes a cycle-accurate GPU simulator, a power model for mobile GPUs based on McPAT [19],

and a power model for OLED screens [12]. Furthermore, it is able to provide image quality assessment using the models presented in the work of Wang et al. [20]. In terms of the GPU microarchitecture, TEAPOT models Tile-Based Deferred Rendering (TBDR) [21, 22]. While Immediate Mode Rendering(IMR) is more popular for desktop GPUs, TBDR seems to be the design of choice for GPUs targeting energy efficiency, like the ARM Mali [22] and PowerVR [23].

In TBDR, shown in Figure 2.2, the screen is divided into tiles, rectangular blocks of pixels. Transformed triangles are not immediately sent to the Raster Unit. Instead, the Tiling Engine stores the triangles in memory and sorts them into tiles, so that for each tile that a triangle overlaps, a pointer to that triangle is stored. Once all the geometry for the frame has been fetched, transformed and sorted, the rasterization starts. Just one tile is processed at a time in each Raster Unit, so all the color and depth information of the pixels of the tile can be stored in local on-chip memory. The final colors of the tile are transferred just once to the off-chip color buffer when the tile is ready, which saves a lot of traffic with main memory. However, transformed triangles have to be stored in memory and fetched back for their processing, so there is a trade-off between memory traffic for geometry and memory traffic for pixels.

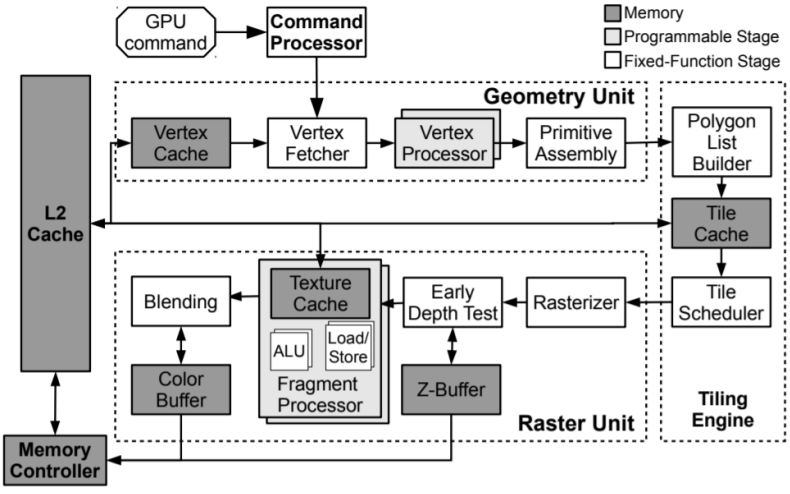


Figure 2.2: Tile-Based Deferred Renderer [13]

The two main strengths of TEAPOT are that firstly TEAPOT allows full-system mobile GPU simulation, which is crucial to achieve accurate results, since common OS tasks such as image composition consume a significant amount of GPU time and energy [13]. Moreover, TEAPOT can be employed to evaluate energy saving techniques that trade quality for energy, which are a popular research area in low-power graphics [16, 24].

2.2.1 Tools

Figure 2.3 illustrates the overall infrastructure of TEAPOT. TEAPOT leverages existing tools, such McPAT [19] or Gallium3D [25], that have been coupled with its GPU models and adapted for the low-power segment. The Gallium3D driver has been modified in order to profile OpenGL ES commands and collect a complete GPU instruction and memory trace. This trace is then fed to a cycle-accurate GPU simulator with which the power and performance for the given application are estimated. As mentioned earlier, TEAPOT also includes a power model for OLED screens. Thus, when analyzing an energy saving technique for mobile GPUs it must be ensured that possible energy savings in the GPU are not compensated by an increment in screen energy and vice versa. The workflow of the simulation infrastructure of TEAPOT is explained below.

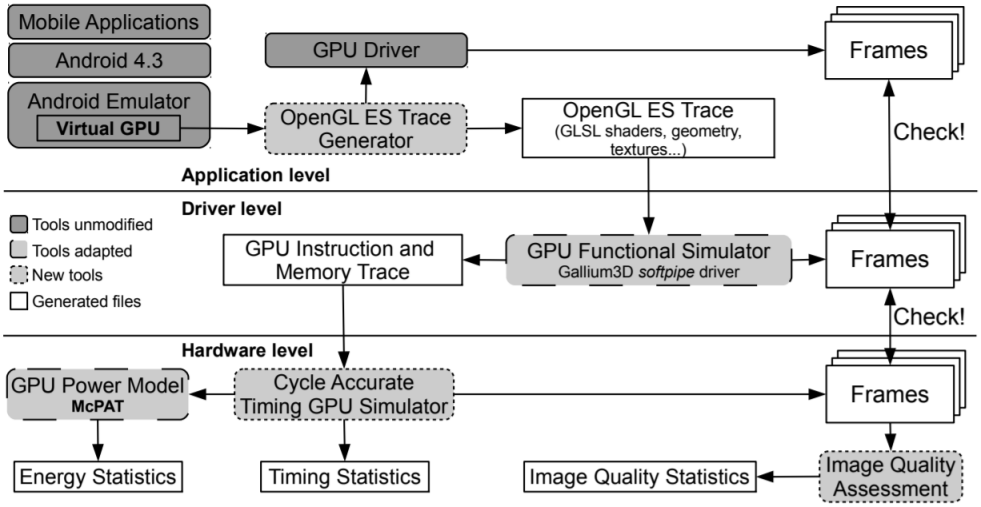


Figure 2.3: TEAPOT: Mobile GPU simulation infrastructure [13]

Application Level

TEAPOT uses the Android Emulator available in the last version of the Android SDK for running mobile applications on a desktop computer. The *OpenGL ES Trace Generator* component captures the OpenGL ES command stream generated by the Android applications and redirects it to the appropriate GPU driver. Therefore, OpenGL ES commands are not processed inside the emulator but in the desktop GPU, which makes them completely visible to the host system. The GPU commands are saved in a trace file, which contains the GLSL vertex and fragment shaders [26], i.e. the code executed by the GPU, and all the data employed for rendering including textures, geometry and state information.

Driver Level

The Gallium3D driver provides GPU functional emulation in TEAPOT. Gallium3D is an infrastructure for developing GPU drivers. It includes several front-ends for different graphics APIs, including OpenGL ES, and multiple back-ends for distinct GPU architectures. An *Instrumented version of Gallium3D* is employed for executing the commands stored in the OpenGL ES trace file. A software-based backend is selected for rendering since it can be easily instrumented in order to get a complete GPU instruction and memory trace. Off-screen rendering is employed so the frames are written into an image file instead of being displayed on the screen.

Hardware Level

A *Cycle-accurate GPU simulator* is employed for estimating the GPU execution time taking as input the instruction and memory traces generated by Gallium3D. GPU energy estimations are also provided by using a modified version of McPAT and the GPU activity factors obtained by the simulation. The output frames generated by Gallium3D are then used for computing the energy consumed by the OLED screen. Finally, the image quality assessment module estimates the visual quality of the output images.

- **Cycle-Accurate GPU Simulator** TEAPOT models the TBDR architecture shown in Figure 2.2. The mobile GPU employs several first level caches for storing vertices, instructions and textures. These caches are connected through a bus to a second level shared cache. In TBDR,

local on-chip memories are employed for storing all the color and depth of the pixels within a tile.
As shown in Figure 2.2, there are more than one vertex and fragment processors. TEAPOT has made the number of these processor parameterised. These processors (also known as shader cores) are the programmable part of the Graphics Pipeline. The architecture of the shader core is explained in detail in Chapter 3. Vertex Processors are similar to Fragment Processors, but they do not have to handle memory or texture instructions so they do not include Memory units, Textures Units or Pixel/Texture caches. We assume a non-unified architecture as opposed to a unified architecture where all the processors can handle both vertices and fragments. Usually, unified architectures offer better workload balance, whereas non-unified architectures can exploit the difference between vertex and fragment processing to build more specialized and optimized processors. For instance, the results obtained by using McPAT indicate that a Vertex Processor has just 64% of the area of a Fragment Processor.

- **Power Model** A modified version of McPAT [19] is used for estimating GPU energy consumption. During start-up, the GPU simulator calls to McPAT passing all the microarchitectural parameters, such as the number of processors or the cache sizes, so it can build the internal chip representation. McPAT estimates the dynamic energy required to access each one of the hardware structures and the leakage power. During simulation, the cycle-accurate GPU simulator collects statistics for each unit and, at the end of every frame, it submits all the activity factors to McPAT. The dynamic energy is computed by accounting for events in the GPU simulator and then multiplying these events by a given energy cost estimated by McPAT. The static energy is obtained by multiplying the total GPU leakage by the execution time.

- **Image Quality Assessment** Image quality is evaluated by comparing a reference image, usually as a result of a high quality rendering, with a distorted image. TEAPOT implements the two types of metrics typically employed for image quality assessment: metrics based on per-pixel errors (Mean Squared Error and Peak Signal-to-Noise Ratio) and metrics based the human visual perception system (Mean Structural Similarity).

2.3 Related Work

Several simulators for evaluating GPU workloads exist, such as GPGPUSim [7], which models General Purpose GPU (GPGPU) architectures. This tools support CUDA [27] or OpenCL [28], but it does not support graphics APIs such as OpenGL up until the conception of this project. GPGPUSim includes a power model, GPUWattch [29], which is also based on McPAT as in TEAPOT. Both power models are similar, but GPUWattch focuses on GPGPU specific features whereas TEAPOT models more specialized graphics hardware. For instance, GPGPUSim models FP units that can be combined to execute 1 double-precision (DP) or 2 single-precision (SP) operations, but TEAPOT relies on SP units since DP is common in scientific workloads but not in games. On the contrary, TEAPOT models specialized Texture Sampling units since texture fetching instructions are frequent in graphical workloads.

ATTILA [9], though now an inactive project provided an OpenGL framework for collecting traces of desktop games and a cycle-accurate GPU simulator. Although ATTILA provided full support for desktop games, it cannot run applications for smartphones. Furthermore, its GPU simulator models a desktop-like immediate-mode renderer. Finally, ATTILA does not include a power model. Other simulators like the GLTraceSim [30] is a graphics tracing framework to explore system-level effects on heterogeneous CPU+GPU memory systems. Qsilver [10] can also collect and simulate traces from desktop OpenGL games, and it includes a power model. GRAAL [31] also provides OpenGL support and a power model for GPUs. Furthermore, it models a low-power GPU based on TBDR. However, OpenGL ES support is not available in any of these simulators so they cannot run mobile applications for smartphones and tablets.

Unlike the aforementioned tools, TEAPOT supports full-system GPU simulation, being able to profile multiple applications accessing the GPU concurrently. TEAPOT additionally provides image quality metrics for automatic image quality assessment and includes a power model for OLED screens, since it has been designed for analyzing graphical workloads in the low power segment.

Chapter 3

Description of the Microarchitecure

This chapter details the architecture of the baseline shade core and all the modifications added to it.

3.1 Initial Shader Core Architecture

The smallest unit of work in a shader core is a warp. A warp is a group of threads executed in lockstep mode. This implies that the same instruction is executed by all the threads but each thread operates on a different set of data. In graphics workloads, threads correspond to fragments and vertices, which are grouped in warps of four threads by the control logic in the Fragment Stage and the Vertex Stage, respectively. A Warp-Entry scheduler issues a warp to a shader core if this does not exceed the total number of warps allowed. The warp is then assigned a warp ID by the core. This ID is the lowest Warp ID that is free. The warp is also assigned a total of 80 registers regardless of the number of registers the program of the warp will use. Another point to note is that there are two L1 caches connected to the shader core: an Instruction Cache and a Texture Cache. Both caches have request queues that let only one request to be sent and served per cycle. The shader core has four pipeline stages as shown in Figure 3.1. Each pipeline stage will be explained in the next subsection.

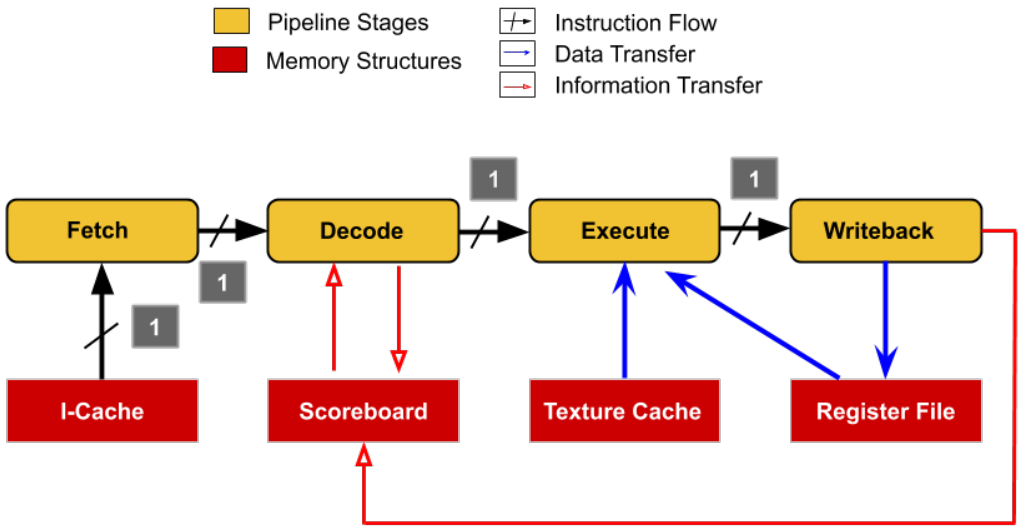


Figure 3.1: The baseline version of TEAPOT's shader core. The numbers next to the *Instruction Flow* arrows represent the number of instructions that can be transferred.

3.1.1 Pipeline Stages

Fetch

A fetch warp scheduler implementing a Round Robin scheme decides which warp gets to fetch every cycle. A single warp is selected, which sends only one request to the I-Cache each cycle.

Decode

The Decode operands checks for dependencies and reads operands from the Register File. An instruction does not advance to the Execute stage until their dependencies with previous instructions from the same warp are resolved. This is done using a scoreboard. Scoreboarding is a mechanism implemented to reserve the registers of the instructions being executed and free them after the Writeback stage. If a warp is stuck at Decode because it is executing an older instruction in the Execute Stage, then the Decode and Fetch stages get stalled. There is no buffer between the Fetch and Decode stages, which limits the parallelism of the front-end and back-end because a stall in the back-end leads to a stall up until the Fetch stage.

Execute

An unbounded limit of funcional units is assumed: as many instructions as in-flight warps can be executed, regardless of their type. Additionally, the functional units are not pipelined.

Writeback

The Register File stage has just one write port, so only the result of one warp can be written in each cycle. This stage frees the register that is being written back by changing its state in the scoreboard.

3.2 Implemented Shader Core Architecture

The register file in the baseline shader core was unbounded, able to assign 80 registers to every warp that entered the core. This number was dimensioned to match the requirements of the OpenGL specification [32], which is a worst-case scenario that rarely occurs. Not only is the number 80 too high for mobile graphics workloads, contemporary GPU architectures dynamically assign registers to warps rather than a static set. The register file has thus been modified to have an absolute bounded number of registers which is not a function of the number of active warps in the core. The Warp-Entry Scheduler has also been modified to check if there are enough physical registers in the core for the new warp that is requesting entry. The baseline shader would assign the lowest Warp ID available to the warp that is entering. This has been modified to a Round Robin scheme to allocate a free Warp ID to the warp. In order to increase the throughput in such a huge register file, register banks were employed. Even though there are now as many read ports as there are number of banks, one cannot ensure that at a particular time, all instructions ready to read from the register file have source operands belonging to different register banks. This called for a mechanism to buffer instructions while they waited for all their source operands to be read.

A separate pipeline stage was created called *Operand Collector* which has buffer units that store information about instructions. Each instruction in the Operand Collector has a separate buffer, called *Collector Unit* (CU). Once an instruction has a CU allocated, its source operands get sent to separate request queues depending on their bank ID. The requests to each bank

are resolved using an Arbiter. Instructions take a variable number of cycles in the Operand Collector, depending on their number of operands and the occupancy of the read queues of each bank.

The baseline shader has a pipeline width of one instruction whereas modern GPUs have wider pipelines. Therefore, the pipeline width has been modified to allow 2 fetches per cycle and 2 issues per cycle. The Execute stage has now been modified to have a parameterised number of functional units which are pipelined. Unlike desktop GPUs such NVIDIA's Volta architecture, which have close to a hundred functional units per core [33], mobile GPUs have a much modest number, ranging from 4 ot 16 functional units per core [34]. The baseline Register File had only one write port, which limited the Write-back stage throughput. The new architecture has now one write port for each bank in the Register File and one write port specifically allocated for memory instructions. The next section describes the various stages of the pipeline in detail.

3.2.1 Pipeline Stages

Figure 3.2 shows the implemented model of the shader core.

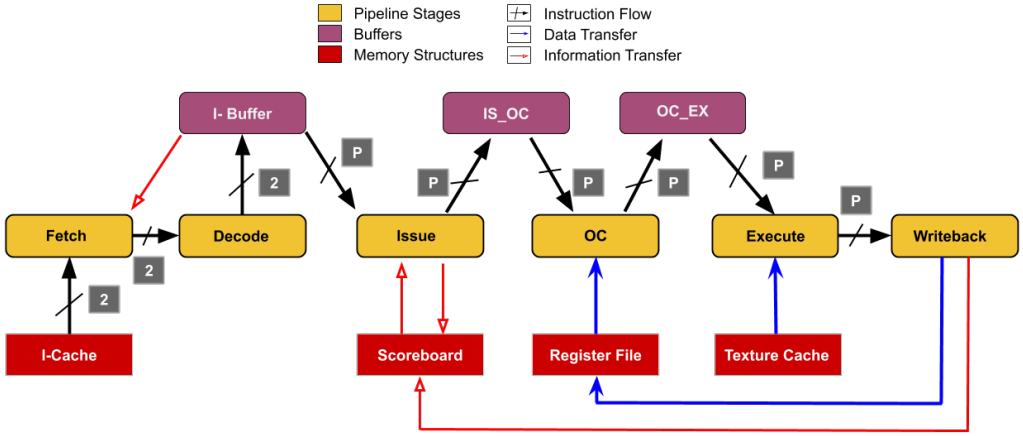


Figure 3.2: The implemented version of TEAPOT's shader core. The numbers next to the *Instruction Flow* arrows represent the number of instructions that can be transferred. *P* indicates parameterised.

Fetch

The Simulator allows two consecutive instructions from the same warp to be fetched in one cycle. The FetchScheduler decides using a Loose Round Robin scheme which warp is going to fetch. The looseness refers to the exceptions made in Round Robin such as not choosing a warp that has been blocked and not choosing a warp that has an instruction still waiting to be issued in the I-Buffer (explained in Section.3.2.1). If the chosen warp encounters a miss, then it does not forward anything to Decode until it receives the instruction.

Decode

This stage reads the instructions present in the I-Buffer and stores them back decoded.

The I-Buffer is used to store instructions that have been fetched but not issued. It is crucial to this design of a multi-warp processor, as it allows the front end to keep working even if some instruction could not get issued right after the Decode stage. The I-Buffer has a two instruction wide slot for every warp. Each of the instruction slots also has a valid bit.

Issue

This stage allows a parameterised number of warps to be issued in a cycle. The warps to be issued are chosen by another scheduler called the Issue Scheduler, which also employs a Loose Round Robin Scheme to select warps. The looseness here refers to exceptions made in Round Robin such as not choosing warps with instructions that have Read After Write and Write After Write dependencies with the instructions from the same warp that have been issued but have not finished. The looseness also refers to the exception for end of program instructions which do not get issued until all the instructions from that warp have retired.

The dependencies are checked through scoreboarding, as mentioned before. Since instructions spend a variable amount of time in the Operand Collector and the Execute stages, an instruction might finish before an older instruction of the same warp. This has a potential to cause Read after Write, Write after Write and Write after Read hazards. This possibility is removed by using a scoreboard which does not issue instructions that might cause one

of these hazards.The instructions issued from each warp are then put into a buffer called *IS_OC Buffer*.

The *IS_OC Buffer* is a FIFO structure of parameterised size that holds the instructions that have been issued until there is a free CU in the Operand Collector. When the buffer gets filled, no more instructions are allowed to be issued. The width of the input port and the output port is parameterised. The instructions sit in the buffer until the Operand Collector finds free CUs for them.

Operand Collector (OC)

This stage has three sub-stages: Allocate Collector Unit, Read Operands and Dispatch.

- Allocate - This stage takes a parameterised number of instructions from the *IS_OC* Buffer and allots them free Collector Units. Collector Units are buffers specially built for storing instruction and operand information. They have fields for warp ID, Opcode and three operands. The fields for operands contain a valid bit, a register index field, a field to store the value of the register after being read and a ready bit to indicate that this operand has been read successfully.

- Read Operands - This stage takes all the requests from the Collector Units and sends them to their respective read request queue depending on their Bank ID. The scheme for the allocation of registers to register banks follows the *Warp_shift* scheme shown in Equation.3.1.

$$RegisterBankID = (WarpID + RegisterIndex)\%(NumberofRegisterBanks) \tag{3.1}$$

  Each register bank has a read request queue from which only one request is served per cycle. After the requests have been serviced, the operand values are filled in respective CUs, and the corresponding ready bits are set. Reading from multiple CUs to the same bank in a single cycle would require important area overheads in read ports and crossbar communication. Reading is, therefore, limited to one request per bank and one request per CU, which calls for an arbitration mechanism to decide the requests that will be serviced in a particular cycle. The

mechanism used is a well known algorithm called *Wavefront Allocator* [35, 36]. This allocation scheme works on the principle of an arbitration "wave" across an array of arbitration cells. Studies done on Allocators show that the Wavefront Allocator works best for crossbars that have a small number of input and output ports.

- Dispatch - This stage uses a Loose Round Robin mechanism to check which CUs are ready with all operands and are ready to be dispatched to the next stage. The looseness here refers to exceptions made when the CU is not ready for dispatch. The number of instructions that can be dispatched in one cycle is parameterised. These instructions are then moved onto the *OC_EX Buffer* which sit there to be taken up by the Execute stage.

  The *OC_EX Buffer* holds the instructions that have been dispatched from the Operand Collector. Its size is parameterised. Each slot in the buffer can hold one instruction. This buffer is emptied giving priority to the oldest but it can move on to another instruction if the first one cannot be executed as per restrictions posed by the execute unit. The width of the input port and the output port is parameterised.

Execute

This stage takes in a parameterised number (N) of instructions from the *OC_EX* Buffer depending on the availability of the functional unit that the instruction is requesting. A maximum of N number of memory operations are allowed in one cycle. As mentioned before, the cache has request queues but only allows one request to be sent and serviced in a single cycle. A maximum of N End of Program Instructions are allowed in one cycle. And all the other instructions use a general purpose ALU which has a parameterised number of replicas. As different instructions take varying number of cycles to finish execution, structural hazards may occur during Writeback. Such conflicts between instructions are resolved before being picked for execution.

Writeback

This stage frees the register being written back by changing its state in the scoreboard. All the functional units together have as many Writeback ports

as the number of register banks. The memory operations have a separate Writeback port.

Chapter 4

Validation

This section characterizes the benchmarks and the experiments used in order to validate the functionality and accuracy of the new features of the model. For the purpose of all tests in validation, the parameters in the shader core have been kept constant with the values shown in Table 4.1.

Table 4.1: Shader Core Simulation Parameters

Parameter	**Value**
Number of Functional Units	4
Issue Width	2
IS_OC Input Port Width	16
IS_OC Output Port	16
IS_OC Buffer Size	25
OC_EX Input Port Width	16
OC_EX Output Port	16
OC_EX Buffer Size	25
Total Number of Warps	32
Total Number of Registers	8000

4.1 Benchmarks Used

Table 4.2 lists the benchmarks that have been used in the validation process, which correspond to commercial, unmodified Android applications. The selection of these application has been made with the intent that the benchmark

suite illustrates the mobile gaming landscape: it includes widely popular applications (with hundreds of thousands of downloads in the Google Play Store [37]), in a variety of genres and a mix of 2D and 3D graphics. The analysis of the different studies has been performed with a single frame corresponding to a typical use case from each benchmark, i.e., not loading or menu screens. A single frame is representative of the whole benchmark since all frames in the same scene have almost identical geometry and shader programs.

Table 4.2: Benchmark Suite

Benchmark	**Alias**	**Genre**	**Type**
Candy Crush Saga	CCS	Puzzle	2D
Shoot War: Professional Striker	SWa	First-Person Shooter	3D
TempleRun	TRu	Endless runner	3D
City Racing	CRa	Arcade	3D
Rise of Kingdoms: Lost Crusade	RoK	Strategy	2D
Derby Destruction Simulator	DDS	Racing	3D

4.1.1 Characterization

The following plots depict some basic characterization of the benchmarks, both to show the main differences between traditional CPU/GPGPU workloads and mobile graphics workloads and as a means for better understanding the results obtained by the validation process.

Average number of registers per shader program

Figure 4.1 shows the average number of registers used per shader program. This metric is an indicator of the resources that each warp will use in its lifetime inside the shader core. This is also an indicator of how many register banks are appropriate for graphics workloads. It can be seen that four of the benchmarks demand around 10-15 registers per program whereas two of the benchmarks have a relatively lower demand for registers. The average number of registers is around 11. We can thus also predict that the improvement in performance will quickly plateau as the number of register banks increase.

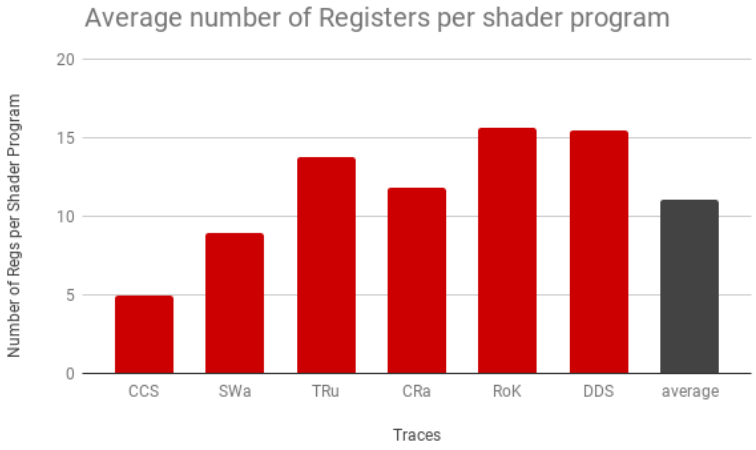


Figure 4.1: Average number of registers per shader program for different benchmarks

Average number of instructions per shader program

Figure 4.2 plots the average number of instructions per shader program, a metric indicative of each program's complexity and its average stay within a core. *CCS* has an average of only four instructions per program whereas *DDS* has an average of twenty instructions. The simple models and textures used in *CCS* require very short programs to be rendered, while the realism and detailed effects in *DDS* demand, comparatively, programs that are five times bigger. The other benchmarks lie somewhere in the middle of both these values and the average value is about 10 instructions.

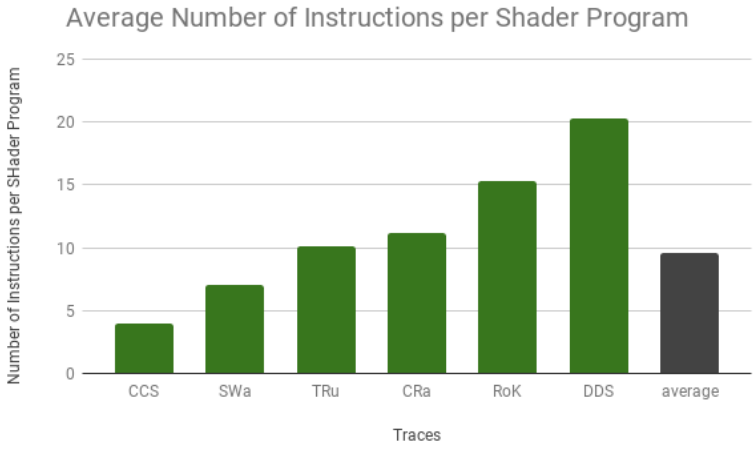


Figure 4.2: Average number of instructions per shader program for different benchmarks

Total number of instructions for one frame

Figure 4.3 is a plot of the total number of instructions run in the fragment processor for one frame of the benchmarks. Since every pixel in the screen must have a color assigned, the lower bound of the total instructions executed corresponds to the resolution of the frame and the average number of instructions per shader program. However, since hidden-surface determination is done using either the painter's algorithm or depth buffering [38], it is normal that the color of two or more fragments are computed for the same position in the screen. This means that the total number of instructions might exceed the average number of instructions per shader program multiplied by the total number of quads(a group of four fragments). Each shader program works on a quad and the total number of quads are determined by the resolution of the frame. As expected, it is seen that the values for total number of instructions range from 2 million instructions to 10 million instructions for the different benchmarks. The lower value being for *CCS* which has less instructions per shader core and the higher for *DDS* which has a higher number of instructions per shader. On an average every benchmark has about 4 million instructions per frame.

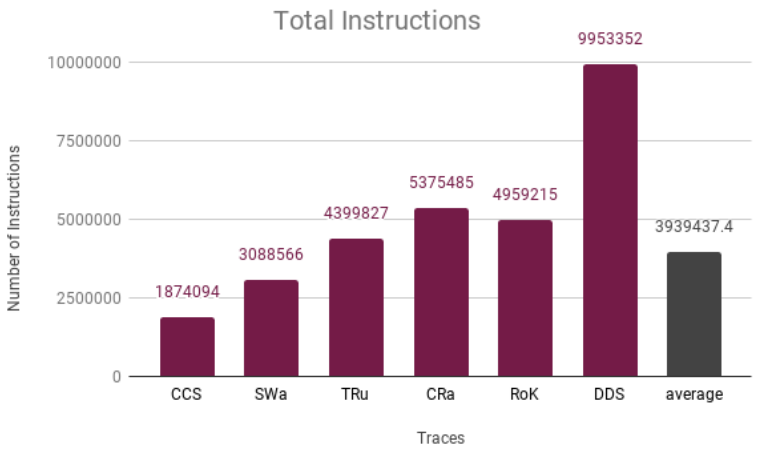


Figure 4.3: Total number of instructions executed in the Fragment Cores

Average distribution of number of operands per instruction

Figure 4.4 is a plot of the distribution of the number of operands per instruction on an average. Since the TGSI ISA allows upto three operands (like in the case of 'Multiply and Add') and no operands (like in the case of 'End of Program'), the distribution is between the numbers zero and three. It is an important characteristic since the number of operands is an indicator for how long an instruction might spend in the Operand Collector. Since only one operand per instruction is allowed to be read per cycle, the instruction would have to spend at least that amount of cycles just reading in the Operand Collector. This characteristic allows us to estimate the minimum average number of cycles that an instruction must spend in the Operand Collector.

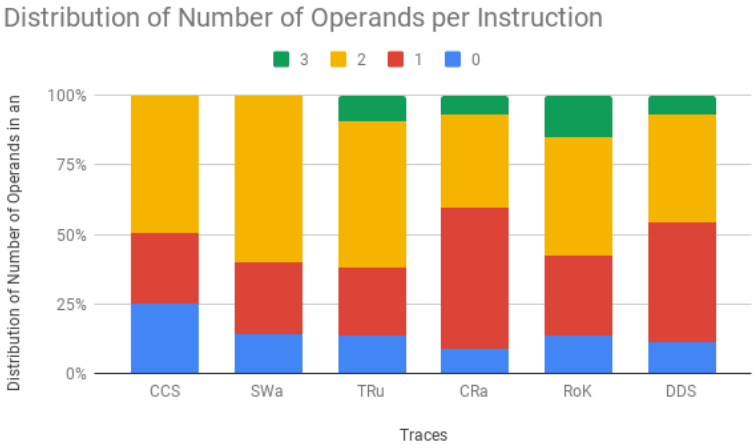


Figure 4.4: Average Distribution of Number of Operands per Instruction in different benchmarks

Average number of operands per instruction

Figure 4.4 shows the average number of operands per instruction. This has been derived from the previous distribution plot. This is also an important plot to estimate the minimum time spent by instructions in the Operand Collector on an average. We see that for all the values this average lies near 1.5. This implies that on an average each instruction must at least spend 3.5 cycles in the Operand Collector including one cycle each for the *Allocate* and *Dispatch* stages.

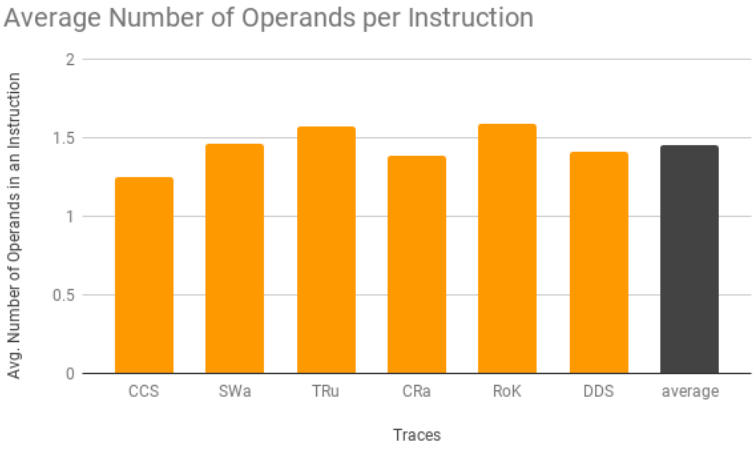


Figure 4.5: Average number of operands per instruction in different benchmarks

4.2 Tests

4.2.1 Cycle by Cycle

The context of each instruction in each stage and buffer was printed out for the cycles that the shader core was active with at least one warp. This was done only to ensure the basic working of the pipeline stages. This is a very crude test but it weeds out the very basic bugs or even design faults. Some operations like the conversion of logical register IDs to physical register IDs, the Round Robin schemes of the two schedulers, the Round Robin assignment of warp IDs when a warp enters a shader core can be validated. Operations like the freeing of physical registers when the warp exits the core, the proper functioning of the pipelined Execute Units and various other basic things can also be validated by this method. It is a crude method as not a lot of cycles can be checked. But it weeds out trivial errors.

4.2.2 Count Instructions

A measure of instructions entering and leaving the pipeline is a good way of ensuring all the instructions went through all the stages and finished correctly. Since there are various buffers in the model, there is a chance that

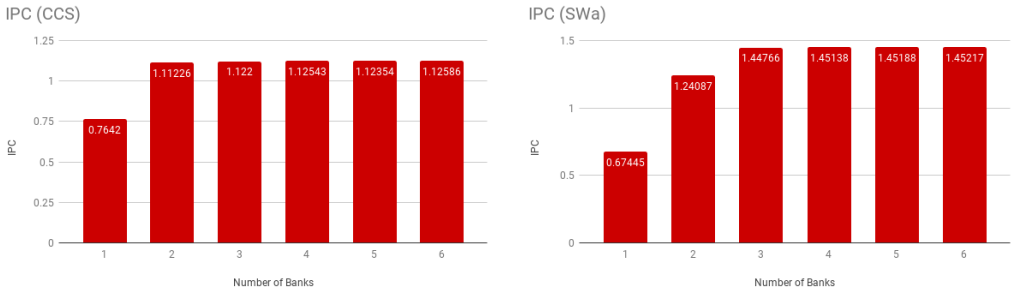
if coded carelessly, the simulator might allow the overwriting of instructions in the buffers. Counters were kept in the stage where the warp enters the pipeline, in the Operand Collector and after the Execute Unit. The numbers were matched and corroborated.

4.2.3 IPC

Firstly, the absolute value of the IPC of a shader core was checked for different benchmarks to ensure that it was in the correct range. Since the pipeline was designed to be two instruction wide, the upper limit of the IPC is 2. There is technically no hard lower limit as a lot of things may vary with different benchmarks. But skepticism can arise when the IPC is too low which may lead to a closer investigation of the model. The IPC should tend to increase as the number of register banks increases. This is because an increase in banks implies more ports in the register file and that implies a potential to service more register reads every cycle.

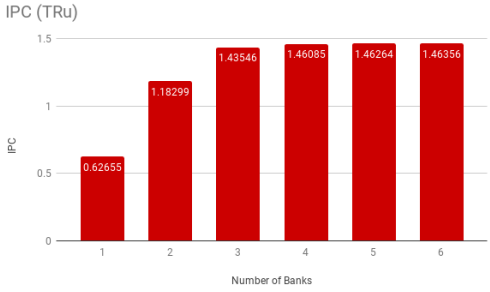
The IPC in all the cases is seen to be below 2. The IPC shows a monotonic increase as the number of banks increase. The increase is not linear. For most of the benchmarks, the plot plateaus after 4 banks. Since there is a monotonic increase in the IPC with the increase in the number of banks, it is an indicator to the correctness of the implementation.

In case of the benchmark *CCS*, it can be seen that the plot plateaus at two banks. It shows that this benchmark does not benefit from having more than two banks. It must be noted that the average number of registers in a shader program for *CCS* is just about five registers (see Figure 4.1) and the average number of instructions per program is just four (see Figure 4.2). This is one of the reasons for a need for lesser number of register banks. On the other hand the benchmark *RoK* shows an increase in IPC from four banks to five banks. The registers in this case is around fifteen and the number of instructions per program is also fifteen and hence the improvement seen in the IPC.

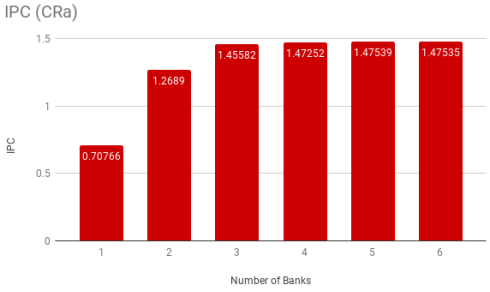


(a) CCS

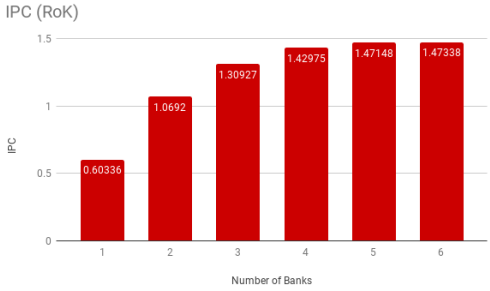
(b) SWa



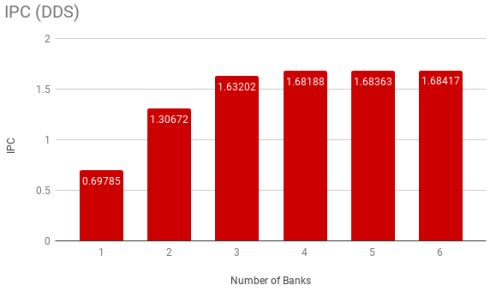
(c) TRu



(d) CRa



(e) RoK



(f) DDS

Figure 4.6: IPC of different benchmarks as the number of banks increases

4.2.4 Buffer Occupancies

- **IS_OC Buffer** - As the input and output ports of this buffer are 16 instructions wide, which is much higher than the width of the pipeline, it is expected that the average number of slots occupied in the buffer per cycle is close to the width of the pipeline which is 2 in this case. But

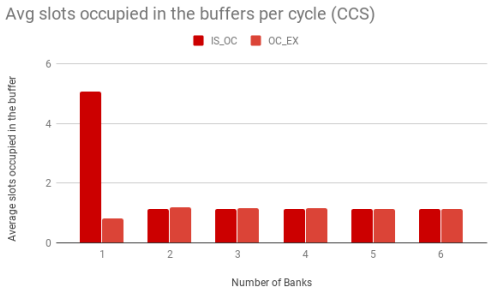
if the Operand Collector is filled for most of the time then instructions get accumulated in the *IS_OC* Buffer. This is the case when the number of banks is very low.
It can be observed that the buffer is occupied by around 2 instructions every cycle for number of banks higher than two. But for the cases with 1 bank and 2 banks, there is a larger average occupation for this buffer.

- **OC_EX Buffer** - As in the previous case, the input and output ports of this buffer are 16 instructions wide. The average slots occupied in this buffer follow the rate at which the execute takes instructions into the Execute stage. It can be observed that the buffer behaves as expected.

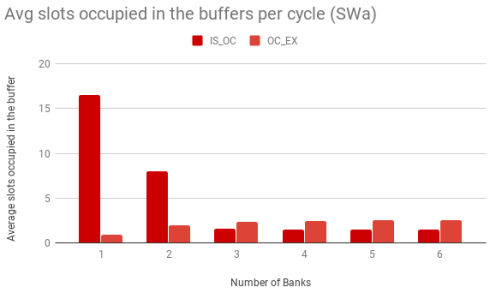
4.2.5 CU occupancies

The average number of CUs occupied in a cycle (when the shader core is active) should be somewhere around the average number of cycles spent by an instruction in the Operand Collector. This occupancy should decrease as the number of banks increases. The upper bound is the total number of CUs in the shader core, which in this case is sixteen.

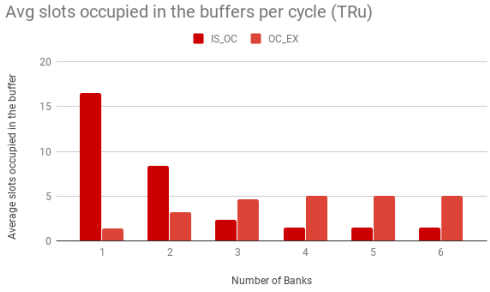
Figure 4.8 shows that the average number of CUs occupied is around sixteen in the case of one bank and this average occupancy decreases significantly as the number of banks increase. With lower number of banks you see a really high occupancy and as you go higher the reads happen faster and instructions do not need to sit in the Operand Collector for long.



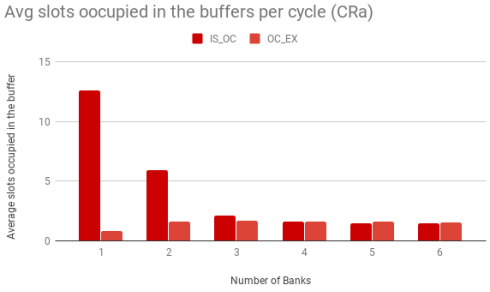
(a) CCS



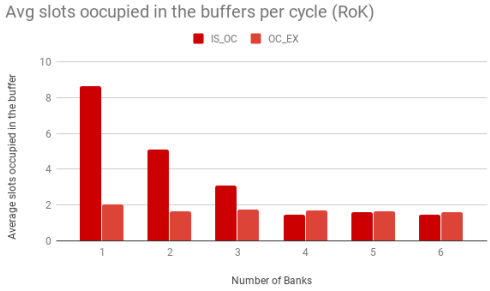
(b) SWa



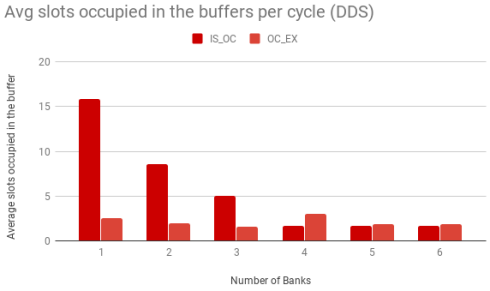
(c) TRu



(d) CRa

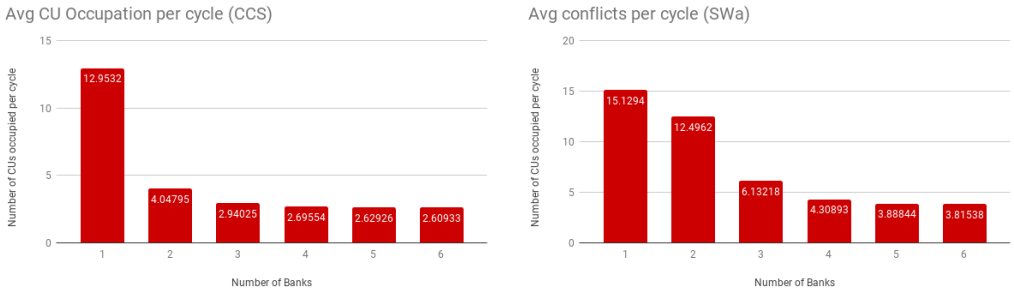


(e) RoK



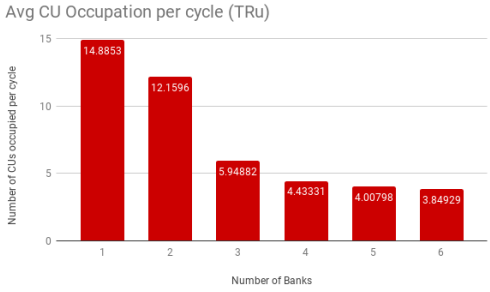
(f) DDS

Figure 4.7: *IS_OC* and *OC_EX* buffer occupancy of different benchmarks as the number of banks increases

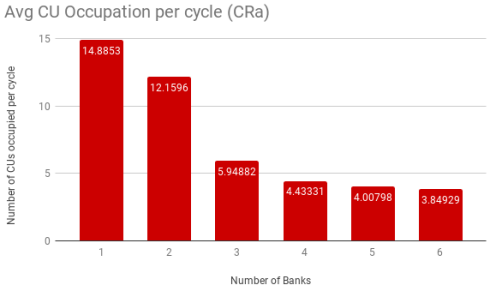


(a) CCS

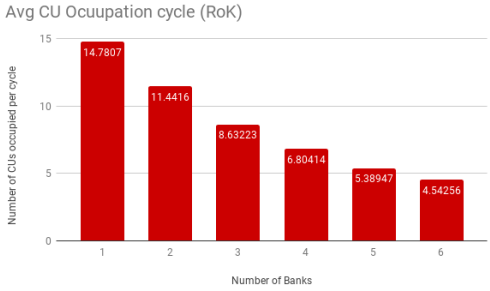
(b) SWa



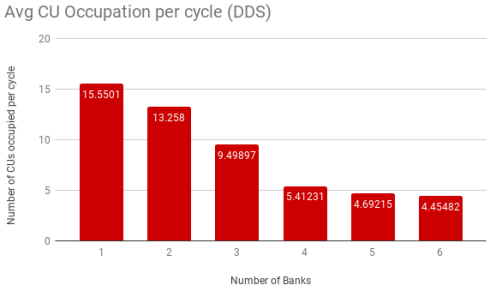
(c) TRu



(d) CRa



(e) RoK



(f) DDS

Figure 4.8: CU occupancy of different benchmarks as the number of banks increases

4.2.6 Conflicts

A conflict for the purpose of our test is defined as the number of requests that could have been served, but were not served, if there were boundless read ports in each register file. Note that this does not include the read

requests that are not served simultaneously because they belong to the same instruction. The metric of the number of conflicts in each cycle is calculated by dividing the total conflicts by the number of cycles that there was at least one request in any of the register banks. This average number of conflicts is expected to decrease as the number of register banks increase. The simple reasoning to this is that as the number of banks increase, the number of ports increase and the potential for more reads per cycle increases.

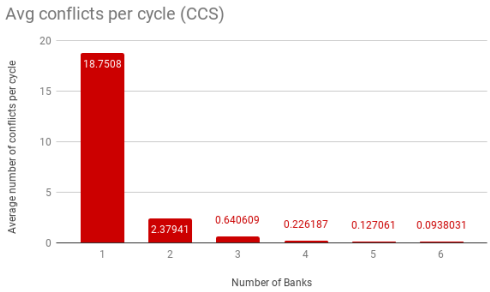
It can be seen that the benchmarks all behave as expected. The conflicts grow close to zero as the number of register banks increases. This number does not reach a zero and yet we observed before that the IPC plot plateaus at four banks. This indicates that the key to better performance now lies outside the Operand Collector.

4.2.7 Average Time in the Operand Collector

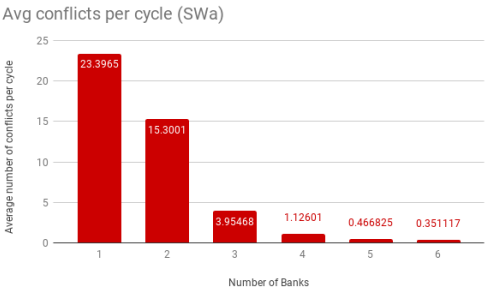
The average number of cycles spent in the Operand Collector by an instruction must decrease as the number of register banks increase. The reason is that as each bank serves one request, more banks mean more requests served in parallel. This can also be seen in the previous section where conflicts decrease as the number of banks increase. Decrease in conflicts indicates that lesser number of requests are waiting. This has a correlation with reduced waiting times for an instruction itself. The lower limit to the average time should be:

$$Avg\_number\_of\_operands\_per\_instruction + 2(cycles\_in\_allocate\_and\_dispatch) \tag{4.1}$$

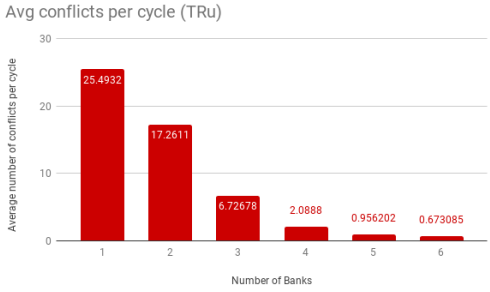
Figure 4.10 shows that as the number of banks increase, the average cycles spent in the Operand Collector decreases. The last bar (in black) is the lower limit that has been calculated using Equation 4.1. It can be seen that the values tend to decrease to reach that value.



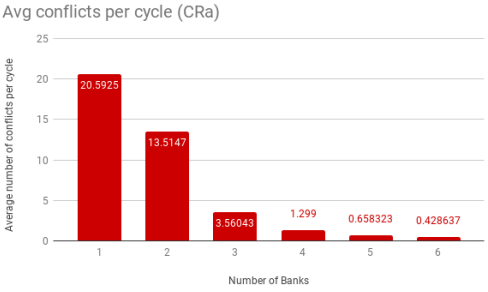
(a) CCS



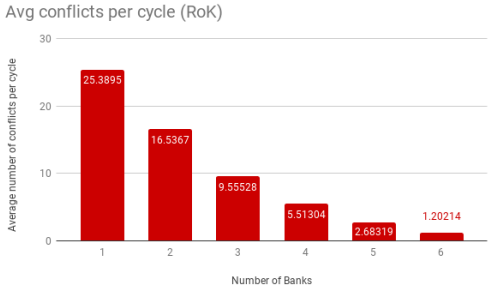
(b) SWa



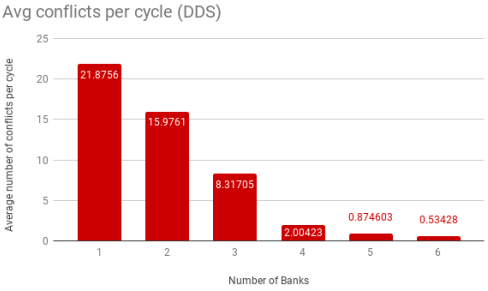
(c) TRu



(d) CRa

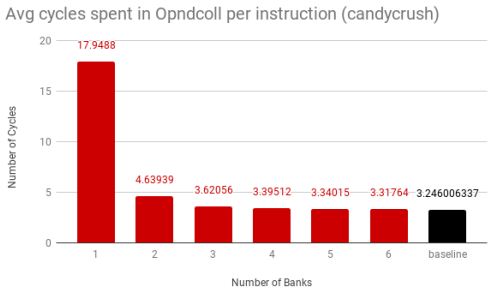


(e) RoK

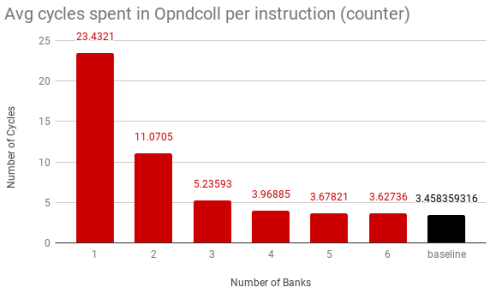


(f) DDS

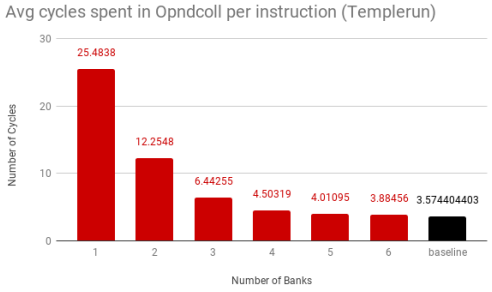
Figure 4.9: Average Conflicts per cycle in the Operand Collector, of different benchmarks, as the number of banks increase



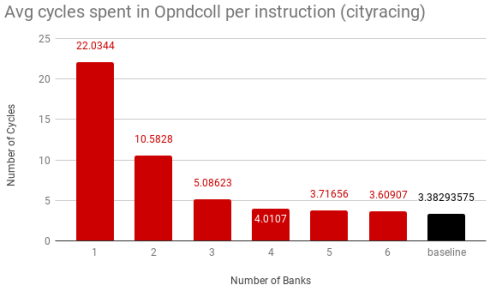
(a) CCS



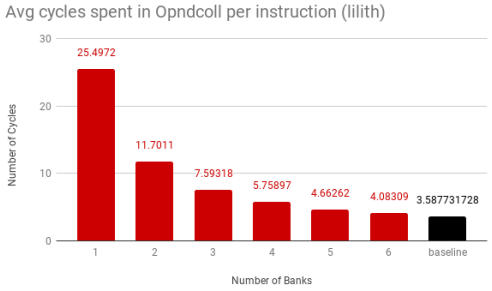
(b) SWa



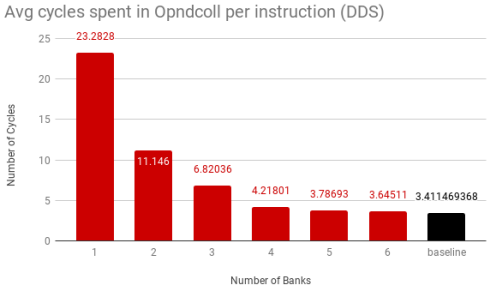
(c) TRu



(d) CRa



(e) RoK



(f) DDS

Figure 4.10: Average number of cycles spent by an instruction in the Operand Collector, of different benchmarks, as the number of banks increase

Chapter 5

Microarchitectural Design Exploration

With all the additions to the simulator, the model is now more accurate and allows for additional microarchitectural design exploration which was previously not possible. We conducted three experiments to further use the implemented model of the shader core to check which design values are suited for mobile graphics workloads. The first experiment studies the benefits of the Register Allocation Scheme that has been implemented. The second experiment is a study of the benefits and costs of increasing the number of warps in the shader core. The third experiment explores the effects of limiting the total number of physical registers in a shader core. The parameters for these experiments are the same as the ones utilized in the Validation Section (Table 4.1) except for the ones that are an object of study.

5.1 Register Allocation Hash Function

As mentioned in Section 3.2.1, the bank ID of the register is assigned using a Warp_shift scheme, a hash function that uses the physical register index and the Warp ID. The mechanism has been repeated below in Equation 5.1.

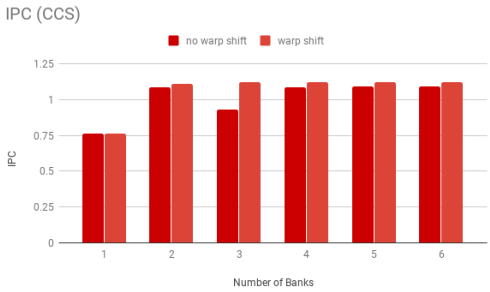
$$RegisterBankID = (WarpID + RegisterIndex)\%(NumberofRegisterBanks) \tag{5.1}$$

The registers in a warp are assigned sequential indices starting from zero, when they enter the shader core. A simple way to have gone about Register

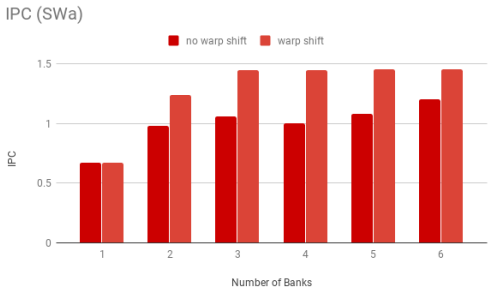
to Bank Allocation is to have Equation 5.2 as the hash function.

$$RegisterBankID = (RegisterIndex)\%(NumberofRegisterBanks) \quad (5.2)$$

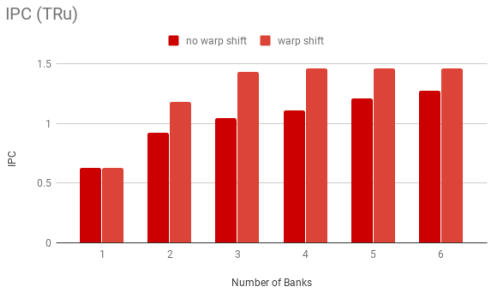
An experiment has been conducted to analyze the difference in performance when using both schemes. All the benchmarks were run with 32 warps, with changing number of register banks, first without the Warp-shift scheme as in Equation 5.2 and then with the Warp-shift scheme as in Equation 5.1. Figure 5.1 shows the IPCs of the simulation with and without the Warp-shift scheme. We calculated that there is a 23.8% increase in IPC when we use the Warp-shift scheme. This increase in IPC is because of the fact that when the registers in a program do not have a uniform rate of read requests or the number of registers in a program is not a multiple of the number of register banks, the scheme of assigning the banks as a function of only the register index, puts a non-uniform pressure on different register banks. This leads to lower performance as the resources are not being used uniformly. That can clearly be seen in Figure 5.1. Figure 5.2 also shows that the reads in each bank are significantly unbalanced when not using the Warp-shift scheme, whereas the read distribution in the case of the Warp-shift scheme is almost a 100% balanced. This emphasizes the improvement that this scheme brings to the overall performance of the core. Thus we see that this is a design decision that can significantly affect performance, which indicates that there is room for research on even better mechanisms.



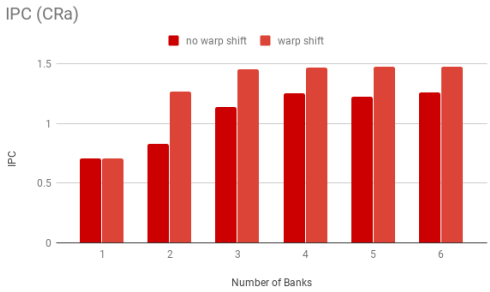
(a) CCS



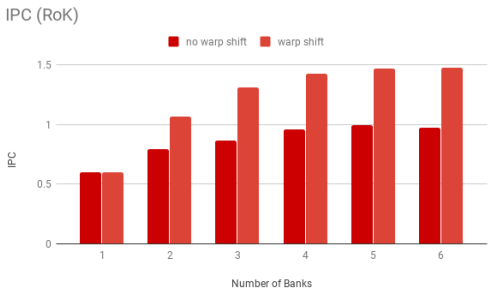
(b) SWa



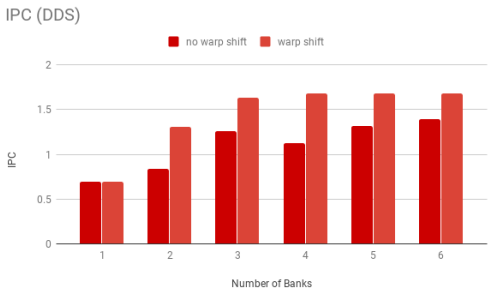
(c) TRu



(d) CRa

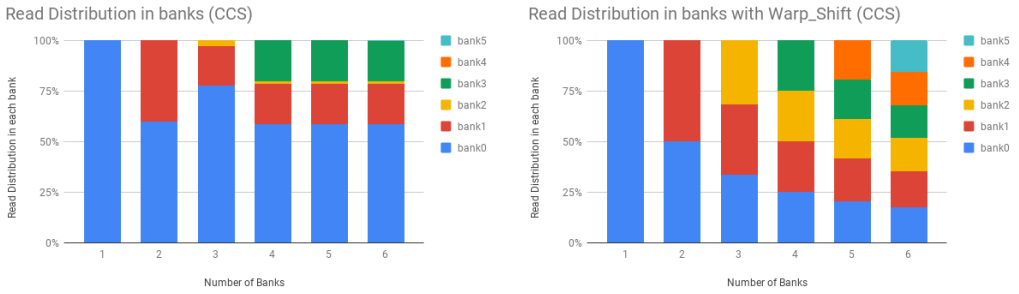


(e) RoK



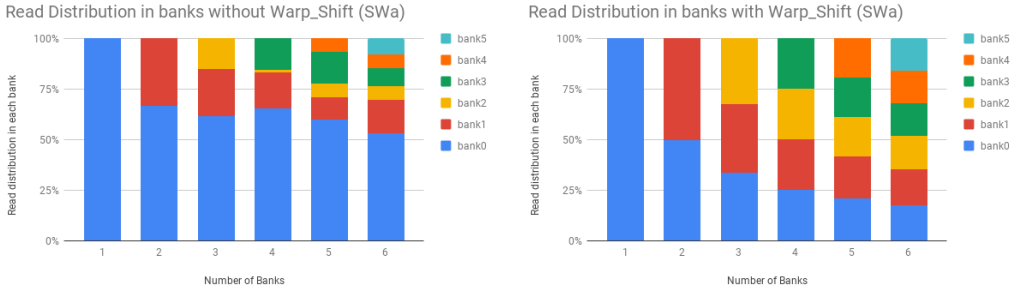
(f) DDS

Figure 5.1: IPC of different benchmarks as the number of banks increase with the no-warp shift scheme compared with the warp-shift scheme



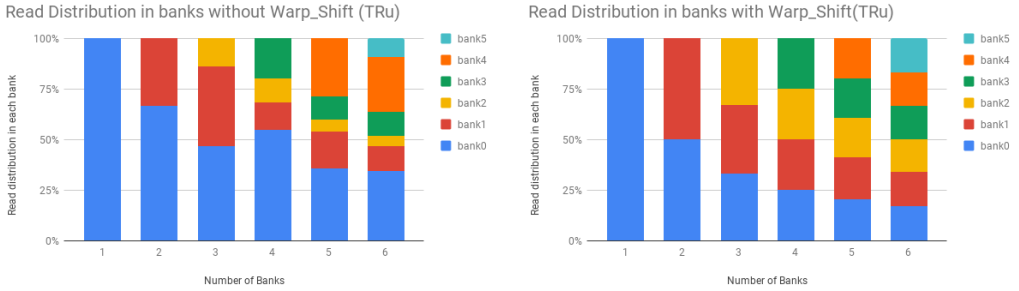
(a) CCS without Warp-shift

(b) CCS with Warp-Shift



(c) SWa without Warp-shift

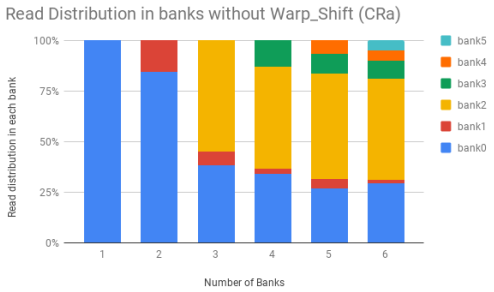
(d) SWa with Warp-Shift



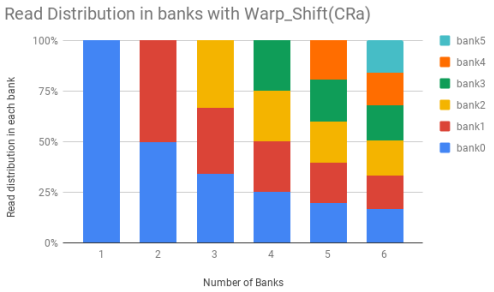
(e) TRu without Warp-shift

(f) TRu with Warp-Shift

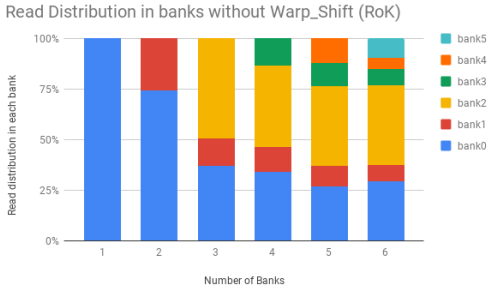
Figure 5.2: Comparison of read distribution in banks without and with the Warp-Shift register assignment



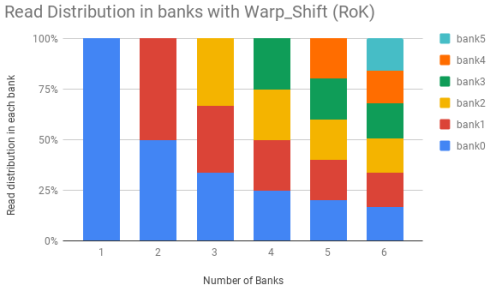
(g) CRa without Warp-shift



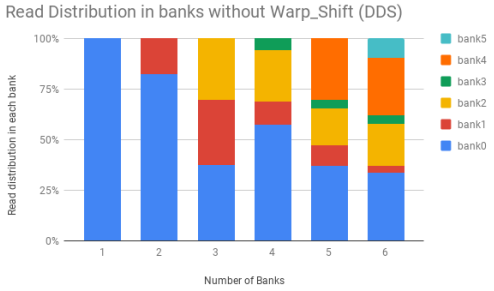
(h) CRa with Warp-Shift



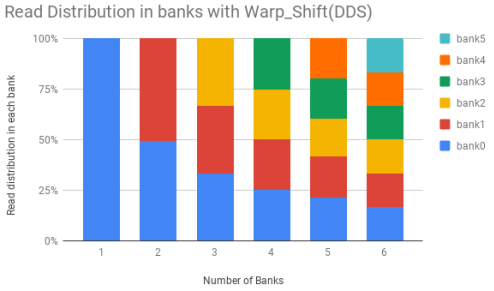
(i) RoK without Warp-shift



(j) RoK with Warp-Shift



(k) DDS without Warp-shift



(l) DDS with Warp-Shift

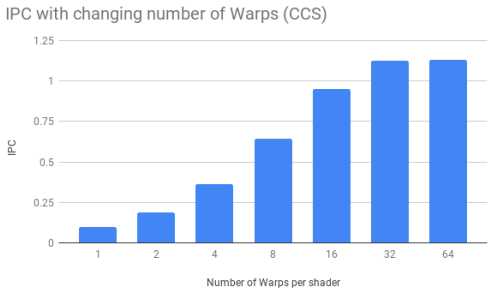
Figure 5.2: Comparison of read distribution in banks without and with the Warp-Shift register assignment

5.2 Total number of warps

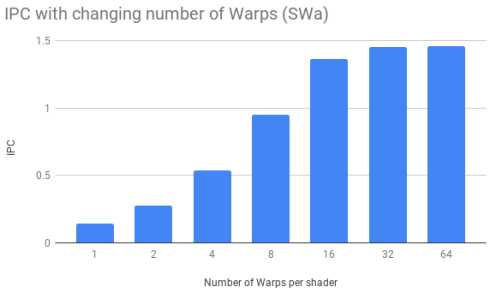
An experiment has been conducted to analyze the improvement in performance when changing the number of warps allowed in a shader core. All the

benchmarks were run with changing number of warps starting from 1 and going up to 64. The number of register banks was kept constant at 4. Figure 5.3 shows the increase in IPC of the benchmarks as the number of warps increase. It can be seen that the plot plateaus at 32 warps. This shows us that bringing in more warps will not improve the IPC beyond that point. There have been various studies conducted to find the optimum number of warps in a GPU for GPGPU workloads [39, 40]. These studies indicate that while increasing the number of warps in a shader could lead to an increase in performance, sometimes it could also lead to a decrease in performance due to cache pollution. Having more warps may lead to untimely evictions of one warp's data by the other. This leads to an increase in the number of misses in the cache. On checking if this was the case for our workloads, we found that there was no significant change in the number of misses.

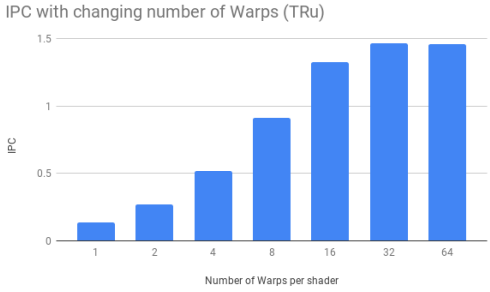
We also investigated the cost of increasing the number of warps allowed in the shader core. Figure 5.4 shows that the average number of registers used in a cycle increases linearly as the number of warps is increased. This means that the cost of improvement in performance has to be paid by increasing the number of registers in the shader core. The slow increase in performance and a linear increase in the cost associated with it, lead to a trade-off in deciding the optimum value for the total number of warps. We conclude that for the given graphics benchmarks used in this project, the number lies somewhere in between 16 and 32, depending on the cost of adding more registers to the core.



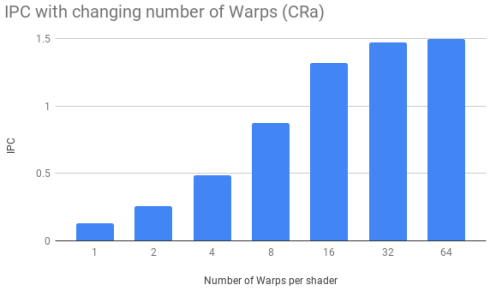
(a) CCS



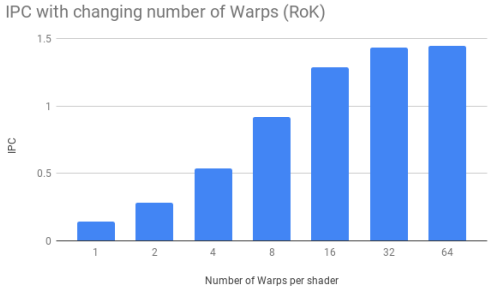
(b) SWa



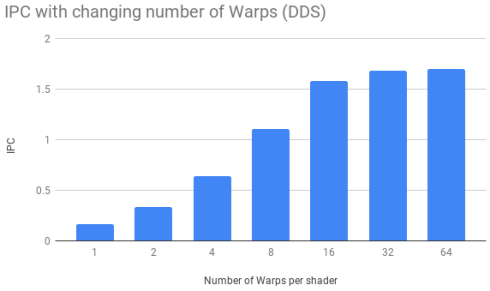
(c) TRu



(d) CRa

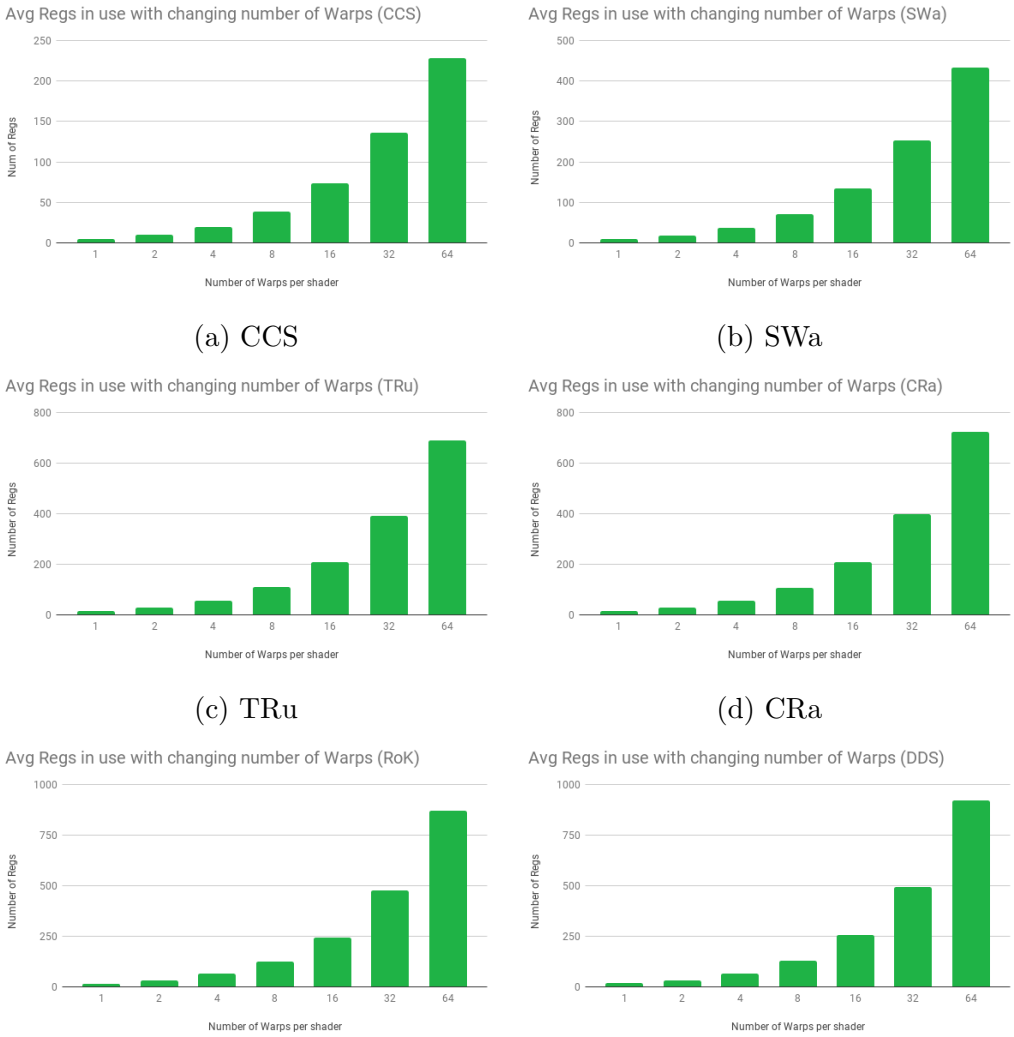


(e) RoK



(f) DDS

Figure 5.3: IPC of different benchmarks as the total number of warps vary



(a) CCS

(b) SWa

(c) TRu

(d) CRa

(e) RoK

(f) DDS

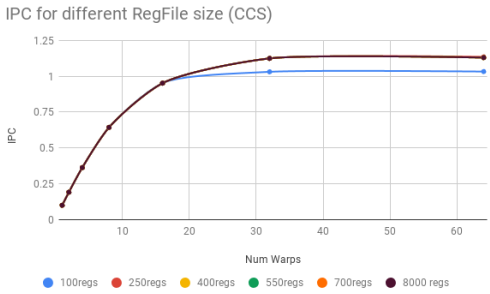
Figure 5.4: Average number of registers used per cycle as the total number of warps vary

5.3 Total Number of Registers

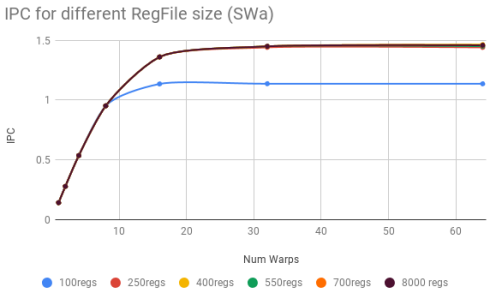
An experiment has been conducted to analyze the improvement in performance when the total number of registers in the register file is varied. The

previous experiment with changing 'Number of warps' was conducted for different values of 'Total number of registers'. Starting from a hundred registers to seven hundred registers and an additional plot with eight thousand registers (an arbitrary large number to set the upper limit for performance). The number of register banks was kept constant at 4. Since we have seen in Figure 5.4 that the average number of registers used per cycle vary with benchmarks and 'Number of warps', it is expected that the performance will be affected if the 'Total number of registers' are lower or very near the average values found previously. It is expected that for the lower values of the variable 'Number of warps', the performance will match that of the best case (8000 registers), as average in those cases are below a hundred. But for other cases, it is expected that the IPC will be lower than the best case, depending on the benchmarks.

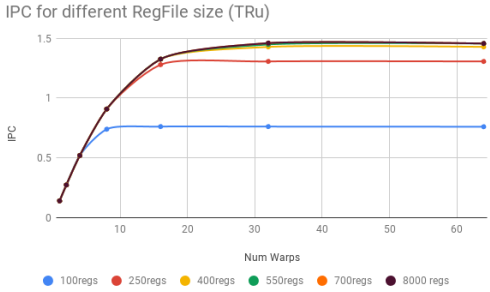
We see that for lower values of 'Total number of registers' the IPC plateaus after a point. For most benchmarks, this point is at 16 warps. It can be seen that the IPC matches the upper limit for 'Number of Warps' 1, 2, 4 and 8, in most cases. We also see that the IPC is significantly lower than the best case in the case of lower values like 100 registers or 250 registers. *CCS* is an exception here as the IPC matches quite close to the best case even for 250 registers. This can be explained using Figure 5.4a which conveys that the average number of registers used is less than 250 even in the case of 64 warps. Analyzing all the values, we conclude that a value of around 550 registers per shader core, is enough to achieve a near-optimum IPC for all the benchmarks. In our model of vectorised registers, this value would correspond to a size of around 8.8 KB. This value is almost 60 times smaller than the 256 KB sized register files that NVIDIA uses in their latest desktop GPU architecture (Volta)[33].



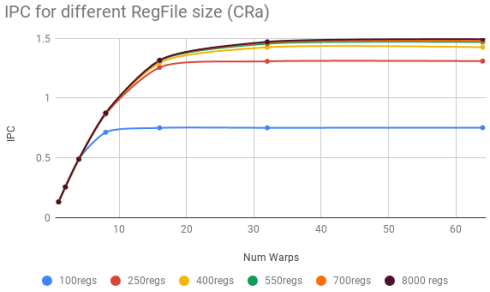
(a) CCS



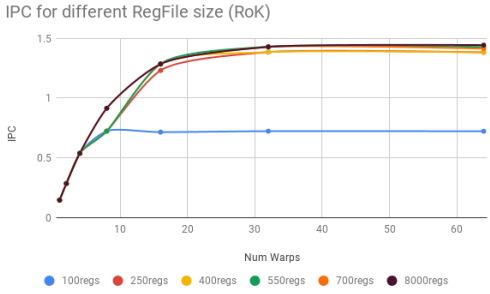
(b) SWa



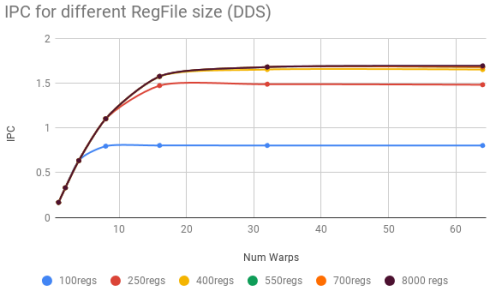
(c) TRu



(d) CRa



(e) RoK



(f) DDS

Figure 5.5: IPC of different benchmarks as the total number of warps vary, plotted for different values of Register File size. 8000 registers is used as the best case.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Graphics Processing Units are an important part of the smartphone industry and yet accurate modelling tools for such systems are not abundant. TEAPOT, a cycle-accurate simulator for mobile-GPU workloads, is the state of the art in this area. However, there were certain aspects of the shader core that were identified to be in need of an update. In order to facilitate this improvement, during the course of the project we have studied the microarchitecture of the shader cores of Graphics Processing Units. Previous work in this area has been focused on general purpose workloads for the GPU. We have specifically studied different microarchitectural design choices for the shader cores of a GPU for graphic workloads and then chosen to implement some of those in TEAPOT. Some of the changes made, were to make the simulator's shader core more practical and detailed as in the case of making a bounded register file and pipelining the Execute Units. While other changes like introducing an Operand Collector and using register banks, were done to imitate contemporary microarchitectures for shader cores in the industry. Some of the designs introduced had characteristics that were parameterised in order to study the optimum design point.

We also validated the implementation through various studies. We hypothesized how the behaviour of the core would change while keeping most design parameters constant and varying one of them. We found all the results to be coherent with the hypotheses.

We also conducted three experiments to further use the implemented model of the shader core to check which design values are suited for graphics workloads.

The first Experiment was to see if the hash function used to assign registers to register banks was a good design choice. We found that the function that we used (Warp-shift) showed a significant improvement in performance compared to sequentially assigning the banks. We also checked the load balance across register banks and found that the distribution is extremely close to a 100% balanced.

The second experiment was to check the optimum number of warps that should be allowed in a shader core. We found that the performance of the core improves but plateaus at around 32 warps. The cost of having more warps is an increase in the average number of registers used. This slow increase in performance and a linear increase in the cost associated with it leads to a trade-off in deciding the optimum value for the total number of warps. We conclude that for the given graphics benchmarks used in this project, the number lies somewhere in between sixteen and thirty two, depending on the cost of adding more registers to the core.

The third experiment was to check how the shader core behaved when the total number of registers were restricted. We found that for low values of 'Total number of registers', like 100 or 250, the performance plateaus as the number of warps increases.

6.2 Future Work

In this project, the shader cores in TEAPOT have received a significant update, but there is still room for improvement. The scheduler for the warps to be issued uses a Loose Round Robin scheme as of now. Other popular schedulers like the Greedy First Scheduler or the Oldest First Scheduler can be implemented. This will greatly help in studying which schedulers suit graphics workloads the best. The more parameterized the simulator is, the wider the studies you can conduct on it. As an example, the front-end and the back-end of the shader core, which is fixed at a width of two instructions

right now, can be parameterised to have different widths so as to study the effects of widening the pipeline for graphics workloads. The number of input ports and output ports and the size of all the buffers in the shader core are parameterised.

Another way is to study the new core and optimize it for graphics workloads. The effects that the implemented design has on power can be studied. The hypothesis is that the buffers and logic that has been introduced might not have a significant impact but the drastic change in the read and write accesses into the register file may have an effect on the dynamic power. This has potential research value. Deterministic allocation of warps to the shader core for better load balance has the potential to improve the overall GPU performance. Another possibility is to compare the techniques used for GPGPU workloads with that of graphics workloads. Further studies can be done to find the optimum values of different parameters in the shader core, depending on different trade-offs.

Bibliography

[1] P. R. Centre, "Pew research centre." https://www.pewresearch.org/global/2019/02/05/smartphone-ownership-is-growing-rapidly-around-the-world-but-not-always-equal Accessed on 2019-06-17.

[2] Gartner, "Gartner." https://www.gartner.com/en/newsroom/press-releases/2019-02-21-gartner-says-global-smartphone-sales-stalled-in-the Accessed on 2019-06-25.

[3] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 168–178, ACM, 2009.

[4] Mediakix, "Mediakix." https://techjury.net/stats-about/mobile-gaming/. Accessed on 2019-06-23.

[5] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 84–93, IEEE Computer Society, 2012.

[6] A. Carroll, G. Heiser, *et al.*, "An analysis of power consumption in a smartphone.," in *USENIX annual technical conference*, vol. 14, pp. 21–21, Boston, MA, 2010.

[7] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, IEEE, 2009.

[8] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 351–360, IEEE, 2010.

[9] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 231–241, IEEE, 2006.

[10] J. W. Sheaffer, D. Luebke, and K. Skadron, "A flexible simulation framework for graphics architectures," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 85–94, ACM, 2004.

[11] khronos, "Opengl." `https://www.khronos.org/opengles/`. Accessed on 2019-06-17.

[12] M. Dong, Y.-S. K. Choi, and L. Zhong, "Power modeling of graphical user interfaces on oled displays," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 652–657, ACM, 2009.

[13] 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*, pp. 37–46, ACM, 2013.

[14] T. Akenine-Möller and J. Ström, "Graphics for the masses: a hardware rasterization architecture for mobile phones," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 801–808, 2003.

[15] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pp. 105–114, Eurographics Association, 2012.

[16] J. Pool, *Energy-precision tradeoffs in the graphics pipeline*. PhD thesis, The University of North Carolina at Chapel Hill, 2012.

[17] Wikipedia, "Wikipedia graphics." `https://en.wikipedia.org/wiki/Rendering_(computer_graphics)`. Accessed on 2019-06-17.

[18] E. H. N. H. Tomas Möller, Duane C. Abbey, *Real Time Rendering*. CRC Press, 2018.

[19] 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*, pp. 469–480, ACM, 2009.

[20] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, *et al.*, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[21] Wikipedia, "Wikipedia tile based rendering." `https://en.wikipedia.org/wiki/Tiled_rendering`. Accessed on 2019-06-17.

[22] Arm, "Arm." `https://developer.arm.com/solutions/graphics/developer-guides/tile-based-rendering`. Accessed on 2019-06-17.

[23] Imagination, "Imagination." `https://www.imgtec.com/developers/powervr-sdk-tools/documentation/`. Accessed on 2019-06-17.

[24] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 529–540, IEEE Press, 2014.

[25] Wikipedia, "Wikipedia gallium." `https://en.wikipedia.org/wiki/Mesa_(computer_graphics)#Gallium3D`. Accessed on 2019-06-17.

[26] Wikipedia, "Wikipedia glsl." `https://en.wikipedia.org/wiki/GLSL`. Accessed on 2019-06-17.

[27] NVIDIA, "Cuda." `https://docs.nvidia.com/cuda/`. Accessed on 2019-06-17.

[28] khronos, "Opengl." `https://www.khronos.org/opencl/`. Accessed on 2019-06-17.

[29] A. The University of Texas, "Gpuwattch." `http://www.gpgpu-sim.org/gpuwattch/`. Accessed on 2019-06-17.

[30] A. Sembrant, T. E. Carlson, E. Hagersten, and D. Black-Schaffer, "A graphics tracing framework for exploring cpu+ gpu memory systems," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 54–65, IEEE, 2017.

[31] B. Juurlink, I. Antochi, D. Crisu, S. Cotofana, and S. Vassiliadis, "Graal: A framework for low-power 3d graphics accelerators," *IEEE computer graphics and applications*, vol. 28, no. 4, pp. 63–73, 2008.

[32] K. Group, "Arb specification." `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_fragment_program.txt`. Accessed on 2019-06-17.

[33] NVIDIA, "Volta whitepaper." `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Accessed on 2019-06-17.

[34] ARM, "Arm mali midgard." `https://developer.arm.com/solutions/graphics/developer-guides/the-midgard-shader-core/single-page`. Accessed on 2019-06-17.

[35] D. U. Becker and W. J. Dally, "Allocator implementations for network-on-chip routers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, IEEE, 2009.

[36] Y. Tamir and H.-C. Chi, "Symmetric crossbar arbiters for vlsi communication switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 13–27, 1993.

[37] Google, "Google play stored." `https://play.google.com/store`. Accessed on 2019-06-17.

[38] tutorialspoint, "Hidden surface removal." `https://www.tutorialspoint.com/computer_graphics/visible_surface_detection.htm`. Accessed on 2019-06-17.

[39] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 72–83, IEEE Computer Society, 2012.

[40] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 99–110, ACM, 2013.