UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

MASTER'S THESIS

Microarchitectural-level simulator for parallel tile rendering on mobile GPUs

Aurora Tomás Berjaga

Advisors:

Diya Joseph
Juan Luis Aragón
Antonio González

In partial fulfillment of the requirements for the master's degree in:

Master in Innovation and Research in Informatics (MIRI)
High Performance Computing

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
Facultat d'Informàtica de Barcelona (FIB)

July 1, 2022

Para vosotros: papa, mama y hermana

Abstract

Mobile devices have led the boom in the technological segment in the recent years. They have witnessed a tremendous improvement in screen resolution and high-quality graphics because of the growing demand for playing games and other animated graphics applications. However, the demand for rendering more realistic scenes brings with it a significant increase in computation and memory bandwidth. This inevitably translates to an increase in energy consumption. Since GPUs are battery operated, energy-efficiency is an important design factor as it dictates their autonomy.

In this work, we present a novel technique which we term *Parallel Tile Rendering* (PTR), which aims to exploit new sources of parallelism in a GPU. Under PTR, we rasterize multiple tiles in parallel using two different rasterization lanes, called Raster Units, in architectures for mobile GPUs. In this way, we dramatically reduce the required cycles for rasterization, which has been seen to be the most time-demanding process when rendering images. Experimental results show that PTR can achieve an average speedup of 83% for a wide range of different benchmarks, each of them with different characteristics. In fact, it is much more effective than having the same amount of computing resources but in a single Raster Unit, with an increase in performance of 8.3% on average. Moreover, PTR provides significant energy savings with an average decrease of 9.86%.

Keywords – Mobile GPU, Graphics Pipeline, Tile-Based Rendering, Energy efficiency.

Acknowledgments

End of June 2022, another step in my life is over. Unfortunately, in recent times I have learned that life is so capricious that it has not allowed all the people I wanted to see this work. Despite this, I want to thank all those people who have accompanied me.

First of all, I want to thank Prof. Antonio González for having trusted me and giving me the opportunity to be part of the ARCO research group, in addition to his advice. Thank Diya Joseph and Prof. Juan Luis Aragón for their guidance and dedication throughout the project. To Prof. Joan Manuel Parcerisa for his valuable contributions. To David Corbalán for clarifying concepts.

On the other hand, I want to thank my family for always supporting and accompanying me. To my friends with whom I have lived moments that I will always remember and we have supported each other. Last but not least, to the professors for contributing their knowledge, despite the difficulties caused by the pandemic.

Contents

1 Introduction **13**

1.1 Motivation 13

1.2 Objectives 15

1.3 Thesis organization 15

2 Background **17**

2.1 Graphics Pipeline 17

2.2 Tile-Based Rendering 18

2.2.1 Geometry stage 19

2.2.2 Raster stage 20

3 Experimental framework **21**

3.1 Simulation infrastructure 21

3.2 Benchmarks 23

4 Baseline improvement **27**

4.1 Throughput increase 27

4.1.1 Raster Unit 27

4.1.2 Tiling Engine 30

4.2 DRAM memory model upgrade 33

4.3 GPU configuration 34

5 Parallel Tile Rendering **37**

5.1 Baseline architecture 37

5.2 Better than just doubling resources 40

5.3 Opportunities for texture sharing 43

6 Inter-tile texture sharing **45**

6.1 Motivation 45

6.2 Baseline architecture 46

6.3 Quad mapping distributions 47

6.4 Experimental results 49

7 Related work **53**

8 Conclusions and future work **55**

List of Figures

1.1 Smartphone users worldwide (in millions) [33]. 14
1.2 Gaming market revenue worldwide in 2021, by segment (in billion U.S. dollars) [34]. 15

2.1 Overview of the Graphics Pipeline stages. [5] 18
2.2 Overview of a Tile-Based Rendering (TBR) architecture. 19

3.1 Overview of the TEAPOT simulation infrastructure. 22

4.1 Average attributes per primitive. 28
4.2 Average primitives per tile. 29
4.3 Baseline scheme. 29
4.4 Structure of the Parameter Buffer. 30
4.5 New hardware design for the Tile Fetcher. 31
4.6 Impact on energy consumption by the Primitive Table. 33

5.1 Distribution of time in the GPU. 37
5.2 Raster Pipeline architecture employed for Parallel Tile Rendering. Each Raster Unit has its own private resources. 38
5.3 Speedup comparison in rasterization when applying different tile scheduling policies. 39
5.4 Active cycles in Fragment Stage with a static tile scheduling. Both Raster Units are quite well workload balanced. 39
5.5 Comparison of the overall energy consumption between the baseline and PTR, presenting the energy reduction obtained with PTR. 40
5.6 Speedup when rasterizing with the double of resources in a single Raster Unit. . 41
5.7 Benefits of PTR over just doubling resources. 41
5.8 Breakdown of the execution time between compute and memory tasks for PTR. . 42
5.9 Misses per 1000 instructions (MPKI) for the L1 texture caches and L2 cache due to texture accesses for PTR. 42
5.10 Average fragments per tile. 43
5.11 Memory bandwidth usage. 43
5.12 Fraction of shared texture blocks among tiles in a frame. On average, 50% of them are reused inside a frame. 44

6.1 Different assignments of quads to fragment processors within a tile. 45
6.2 Raster Pipeline architecture employed for PTR's extension. Each processor shares the texture cache with a processor from another Raster Unit. 46
6.3 Evaluated quad mapping distributions. 48
6.4 Average speedup for all the games for each configuration with respect to the baseline PTR, which owns 8 private texture caches. 49

6.5 Example of processor workload balance using different combinations in a given frame. 50
6.6 Average energy savings for all the games for each configuration respect to the baseline PTR, which owns 8 private texture caches. 50
6.7 Speedup when employing *11-blocking* combination with respect to the baseline PTR, which owns 8 private caches. 51
6.8 Misses per 1000 instructions (MPKI) for the L1 texture caches when employing PTR with private caches compared to shared caches with the *11-blocking* combination. 51

1 Introduction

This chapter introduces the motivation in the computer architecture community for which there is a great deal of interest behind research in mobile GPUs. Then, the main objectives of this work are presented and, finally, it is shown how the rest of the thesis is structured.

1.1 Motivation

It is indisputable that mobile technology has evolved in recent years and has spread worldwide practically at the speed of light. This boom in the technological segment is led by the emergence of the well-known smartphones, which nowadays are essential in our daily life. Thanks to important advances in hardware and software, these mobile devices are no longer just used for phone calls and text messages but also allow a wide variety of functionalities such as connectivity, gaming and multimedia, among others.

The app stores are digital distribution platforms for applications (e.g., Google Play [18] and App Store [6] for Android and Apple devices, respectively) and play a crucial role in this mobile device revolution. These platforms allow users to have a wide range of new features within their reach. The accumulation of improvements in both the technological and software sides has made mobile devices widely accepted by the population. Figure 1.1 shows the evolution of the number of smartphone users worldwide, where it is revealed that the current number of smartphone users in the world today is about 6.5 billion. Therefore, as the current world population is 7.9 billion as of May 2022 [40], it means that 82% of the world's population owns a smartphone. Note also the rapid increase, as in only 6 years, the percentage of the world's population owning a smartphone has almost doubled (in 2016 only 48% of the world's population owned a smartphone).

The video game industry has been one of the markets most benefited by the adoption of mobile devices in our lives. The main reason for this is the convenience of being able to play mobile games anywhere, in contrast to console and PC games, which has caused a boom in the number of people that play games frequently (known as mobile gamers). In fact, recent studies [32, 35] show that gaming apps are the most popular app category in Google Play, accounting

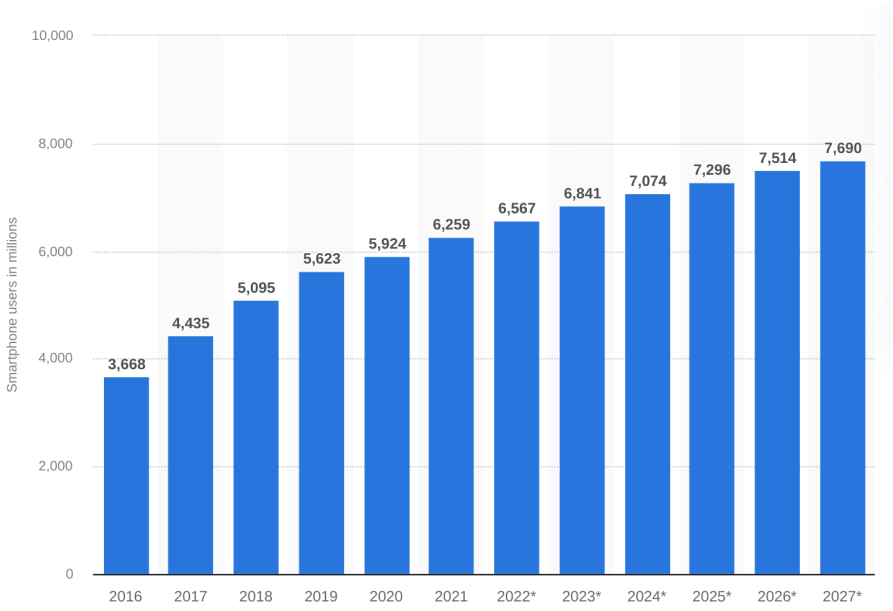


Figure 1.1: Smartphone users worldwide (in millions) [33]. * Projected data.

for 13.63% of available apps worldwide, and 21% of the apps downloaded are games. In addition, 62% of smartphone owners install a game within a week of owning their phone, and today's number of active mobile gamers worldwide is over 2.2 billion, which represents more than a third of the world's population. But this is not all; 43% of smartphone use is invested in playing games.

In summary, mobile games have a strong presence in our daily lives, which makes it an important sector for the economy. As can be seen in Figure 1.2, games on mobile devices dominate the market, accounting for 52% of global revenues, followed by console games (28%) and PC games (20%).

This growing interest in mobile games has led to better screens with a higher resolution and increased refresh rates, bringing important visual improvements to enhance the user experience. In addition, the increase in performance of smartphones has also allowed the processing of more data, thus, reaching more detailed object models. In order to provide high-quality graphics in mobile devices, a Graphical Processing Unit (GPU), a dedicated hardware accelerator for real-time graphics rendering, is required to satisfy the high computing demands required by games.

However, it is not all plain sailing. This demand for more realistic and detailed models requires a significant growth in computational power and memory bandwidth, which translates to an increased energy consumption. For example, a very common screen resolution nowadays in smartphones is Full HD, composed by 1920 pixels horizontally and 1080 pixels vertically, which implies more than 2 million pixels being rendered per frame. Note however, that in order to provide a satisfactory user experience, several frames have to be displayed per second (typically 60), known as frames per second (FPS), to create the illusion of movement.

As a result, energy-efficient GPU designs for mobile devices are a critical design aspect for computer architects since these devices are battery-operated and this determines their autonomy, making it challenging. In fact, recent reports claim that battery life is the most important feature

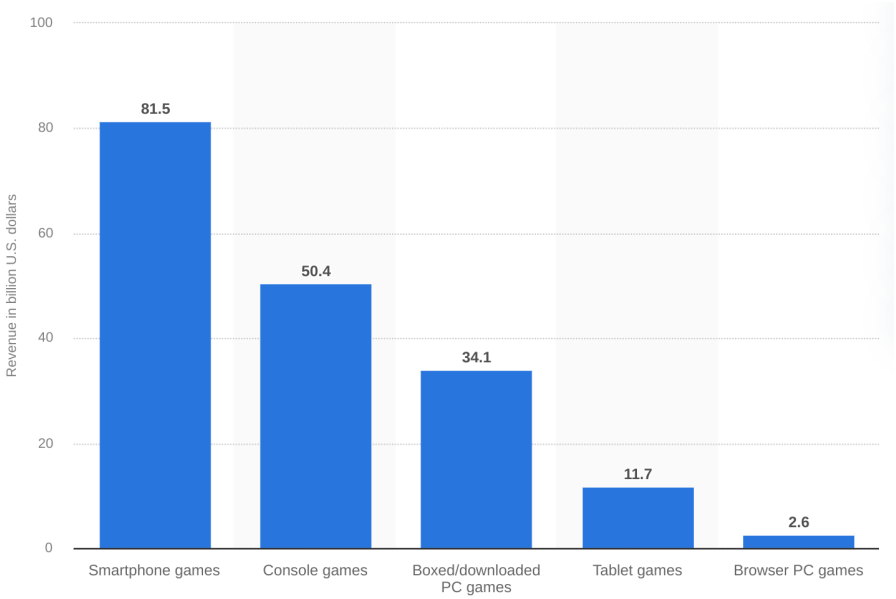


Figure 1.2: Gaming market revenue worldwide in 2021, by segment (in billion U.S. dollars) [34].

when purchasing a smartphone over other characteristics such as camera quality [42]. However, in addition to battery life, smartphones and tablets are designed to be carried everywhere, so they are very lightweight. This feature makes them have a tight thermal limit and, as they are extremely thin to be portable, it adds more complexity as they cannot be equipped with sophisticated cooling systems to maintain the temperature of the internal components below a thermal threshold, as done in desktop devices.

1.2 Objectives

The objective of this thesis is to improve the performance of mobile GPUs by processing data in parallel through independent stages while providing full throughput, which in turn, will lead to an improvement in its energy efficiency. In summary, this work has a twofold objective:

- Develop a cycle-accurate simulator.
- Investigate the potential of a new GPU architecture based on rendering in parallel multiple regions (called tiles) of the frames.

1.3 Thesis organization

The remainder of this thesis is structured as follows:

- Chapter 2 provides a basic background on the graphics pipeline and an overview of the mobile GPU architecture.
- Chapter 3 describes the evaluation methodology used to perform all the experimentation.
- Chapter 4 presents the main improvements performed to construct a robust and more up-to-date baseline.

- Chapter 5 presents and evaluates our approach to rendering multiple tiles in parallel.
- Chapter 6 extends the approach presented in the previous chapter by taking advantage of the texture sharing potential between tiles, leading to an opportunity to save memory bandwidth.
- Chapter 7 reviews some related work.
- Chapter 8 summarizes the main conclusions of this work.

2 Background

This chapter introduces important concepts and terminology required to later describe the technical aspects of this thesis. Therefore, it first provides an overview on the graphics pipeline to understand how data is processed along its different stages and then, it presents a GPU architecture for mobile devices that targets a low-power design, known as Tile-Based Rendering (TBR), which will be used as the baseline in the experiments that will be carried out in this thesis.

2.1 Graphics Pipeline

The graphics pipeline, also known as the rendering pipeline, is a process that consists in rendering two-dimensional images, called *frames*, from the geometrical description of the scene, including three-dimensional objects, light sources, and camera position and orientation [1]. This rendering process is abstracted into three main stages: Application, Geometry, and Raster, as illustrated in Figure 2.1.

The Application stage is a software phase running in the CPU, and is responsible for preparing the geometry models that describe the scene, by means of polygons, and all its associated information that is needed for rendering the polygons in the subsequent stages. The Application stage is also responsible for setting the state of the pipeline (by enabling or disabling capabilities such as depth test, blending, etc.), and loading the scene objects which are then transferred to the GPU for further processing.

The objects are comprised of *vertices*, which are points in a three-dimensional space with additional information associated, referred to as *attributes*, which describe the characteristics of the vertices (e.g., color, normal vectors, coordinates, texture, etc.). In addition, objects have *shader programs* associated, which is the customizable part of the pipeline and where user-defined code is executed. Also *textures* are attached to objects, which are typically 2D images that are mapped onto polygon surfaces to add high-frequency detail.

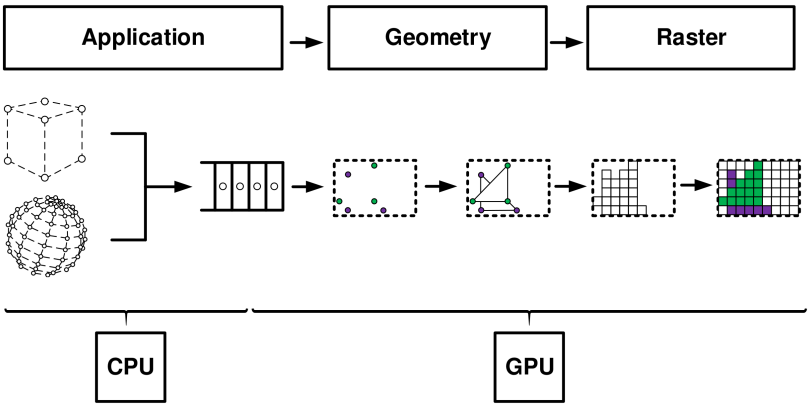


Figure 2.1: Overview of the Graphics Pipeline stages. [5]

Thus, in this stage is where the CPU-GPU interaction occurs. For it, the application sends the information to the GPU through *commands*, where some of them are in charge of rendering a batch of vertices, which are called *draw calls* (e.g., glDrawArrays, glDrawElements in OpenGL). In summary, it can be said that this all-CPU stage manages the I/O data interaction between the CPU and the GPU, and sends the commands to the GPU to tell it how to render the scene.

The next stage is the Geometry stage, which is triggered when a *draw call* is received and is in charge of performing all the geometry-related operations, such as transforms, projections, and all other types of geometry handling. Basically, this stage determines what, how and where an object should be drawn. For it, it receives a stream of vertices as input, which are converted into a set of transformed 2D primitives. Finally, these primitives are received in the Raster stage, where the color for each pixel inside them is computed. Because both Geometry and Raster stages are entirely processed in the GPU, they are discussed in more detail in the following section, with an emphasis on the baseline architecture chosen for this work (Tile-Based Rendering architecture) while reviewing particular architectural details. In case the reader is drawn to further details on the graphics pipeline, there are interesting books available [1, 17].

2.2 Tile-Based Rendering

Mobile GPUs typically implement a Tile-Based Rendering (TBR) architecture in order to reduce accesses to main memory. This rendering approach is also commonly classified in the literature as sort-middle [1], and is very popular in low-power graphics and memory-bandwidth-limited systems. TBR is characterized by dividing the screen space into a grid of smaller regions, called *tiles*. Tiles are generally sized as 32x32 pixels, which is small enough to perform many operations on small tile-sized on-chip buffers, which exploit locality and significantly reduce power-hungry accesses to off-chip main memory, thus, saving memory bandwidth. There are a plethora of commercial examples that employ TBR, such as ARM Mali [8], Qualcomm Adreno [29], and Apple GPUs [7].

Figure 2.2 shows a block diagram of the graphics pipeline and the memory hierarchy organization of a TBR architecture. As it can be seen, the graphics pipeline is split into the Geometry Unit and the Raster Unit, also commonly referred to as Geometry Pipeline and Raster Pipeline, respectively. In addition, there is a new intermediate pipeline stage called Tiling Engine, which

is characteristic of TBR architectures, and is where the tiling process is carried out. In fact, this new stage acts as a serialization point between both pipelines, since tile-based processing requires all the geometry to be previously processed to determine which primitives belong to each tile to later proceed with the raster process. The following subsections overview these stages.

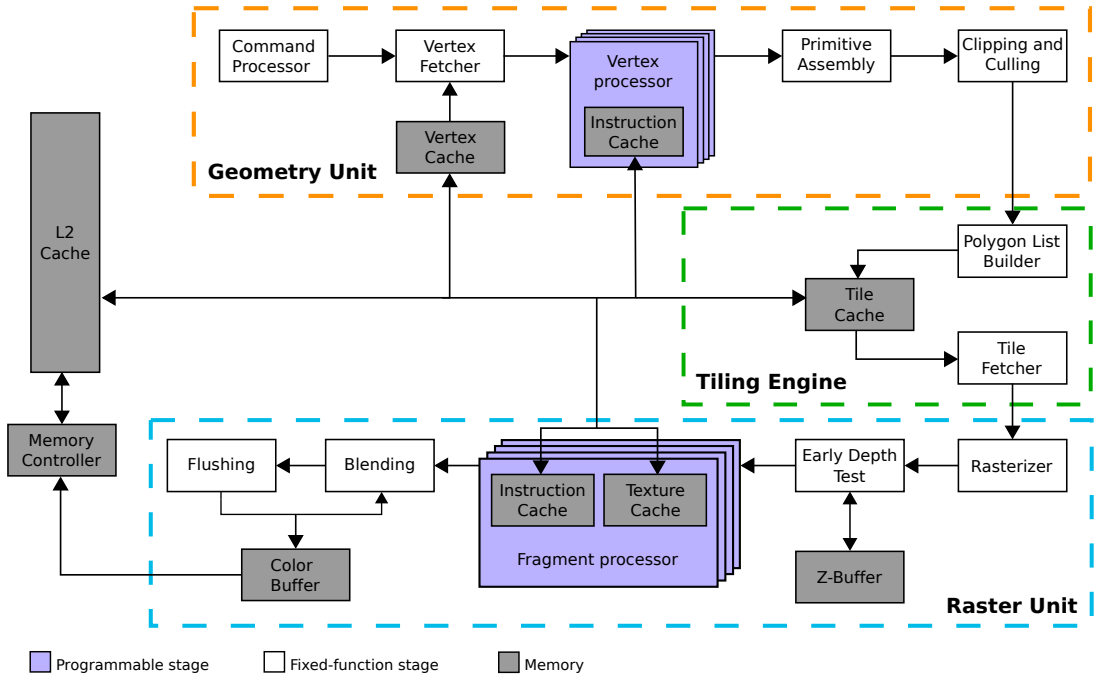


Figure 2.2: Overview of a Tile-Based Rendering (TBR) architecture.

2.2.1 Geometry stage

This initial stage performs all the geometry-related operations in the graphics pipeline in which an input stream of vertices are converted into a set of transformed 2D primitives. First, the Command Processor reads the commands issued by the CPU. When a draw call is received, the Vertex Fetcher is triggered, which fetches the requested vertices by the draw call from memory. Then, the Vertex Processors are responsible for transforming these vertices by executing user-defined programs, known as *vertex shaders*. Once processed, these vertices are assembled to generate polygons, called *primitives*, in the corresponding topology selected for describing the geometry. The most typical mode employed for defining primitives is *triangles*, in which every group of three vertices forms the polygon. Afterward, for each primitive, it must be determined whether the primitive lies within the *view frustum*, which determines a 3D volume corresponding to the visible region on the screen. Therefore, the Culling process discards the triangles that are detected to be entirely outside of this viewing volume. However, in case a triangle is partially visible, a Clipping operation is applied, in which the primitive is split into smaller triangles that entirely fall inside this volume. Since this is a tile-based architecture, before proceeding to the rasterization process, the Polygon List Builder is in charge of sorting the primitives into tiles, i.e., to produce a list for each tile with all the primitives that totally (or partially) fall inside it. Finally, the Polygon List Builder is responsible for storing these per-tile primitive lists in the

Parameter Buffer, a particular region kept in main memory, which are the input data to the Raster Unit. Subsection 4.1.2 provides further details regarding the Parameter Buffer.

2.2.2 Raster stage

Once all the geometry has been processed and binned, the Raster stage is executed and it is determined the colors for each pixel inside a primitive. To do this, first, the Tile Fetcher fetches the primitives and its attributes for a given tile in the frame in a tile-by-tile fashion, which are served as inputs to the Raster Unit. Refer to subsection 4.1.2 for further details on the Tile Fetcher.

Then, the Rasterizer converts primitives into a set of two-dimensional *fragments*, which represents the pixels that can be drawn on the screen. In addition, for the pixels that are covered by a primitive, the values of the primitive's attributes are interpolated at the pixel's position. Fragments are assembled into groups of 2x2 adjacent fragments, known as *quad fragments*, that are sent to the next pipeline stage, the Early Depth Test. This stage aims to eliminate the fragments that are known to be occluded by a previously processed quad, thus allowing to save time and energy from processing useless data that will not contribute to the final image. To that end, a buffer called *Z-Buffer* is employed to store the depth values for each fragment. So, the depth value from an incoming fragment is compared with the one stored in the same position in the buffer. If the quad is closer to the camera than a previously visible quad, the depth value will be overwritten by this new quad. As the depth value of a fragment is the interpolated z component, this stage is also commonly known as the Early Z-Test. A fragment that passes the depth test proceeds to be processed by the Fragment Processors in the next stage. Otherwise, the fragment is invisible and, thus, is discarded.

Next, the *presumably* visible quad fragments proceed to the Fragment Stage, where the Fragment Processors execute user-defined programs, called *fragment shaders*, to compute the color for each fragment. Moreover, it is very common to access textures in this stage, as they enrich the appearance of the models by providing a higher level of detail. Then, these colors are later processed by the Blending Unit, where they are combined with the ones already in their same position in a buffer called *Color Buffer*. In addition, this stage also allows to achieve transparency effects. This is performed using the *alpha* channel, which is an additional value besides the RGB color that describes a fragment's opacity range.

Finally, right after all the primitives in a tile have been completely rendered, the contents stored in the Color Buffer are flushed to the *Frame Buffer*, which is a region in main memory used to hold the data that is going to be displayed in the screen. This way, the Color Buffer is written into main memory once for each tile. After all the tiles of a frame have been processed, the frame is ready to be displayed. Notice that both the Z-Buffer and Color Buffer are tile-sized, which means that they can be stored in on-chip memory, thus, reducing accesses to DRAM and saving bandwidth.

3

Experimental framework

This chapter presents the methodology employed for developing this thesis. First, it is introduced the simulation infrastructure used for the performance and energy consumption evaluation for a mobile GPU. Then, the benchmarks used for carrying out the experiments are described.

3.1 Simulation infrastructure

For this thesis, we have employed the TEAPOT toolset [10], a simulation framework developed in the ARCO research group at UPC, which evaluates the performance, energy consumption, and image quality of a mobile graphics subsystem. This set of tools targets mobile graphical applications, as it provides support for the OpenGL ES API [28] and models a cycle-accurate Tile-Based Rendering architecture, popular in low-power designs, as shown in Figure 2.2. Figure 3.1 shows the flow and the different tools that comprise TEAPOT.

Nevertheless, other widely used GPU simulators can be found in the literature, but they do not fit our needs. For instance, GPGPUSim [11] and Barra [12] are designed to model General-Purpose GPU (GPGPU) architectures, providing support for OpenCL [27] and CUDA [14]. However, despite their acceptance in the GPU research community, they do not provide support for graphics APIs such as OpenGL. Thus, they target scientific-like workloads instead of graphics workloads. Along the same lines is MacSim [21], which also only supports CUDA. Besides that, ATTILA [25] and Multi2Sim [37] provide support for OpenGL graphics applications but are not capable of running graphics workloads on embedded systems as they do not support OpenGL ES API, which is the variant needed for these devices. In addition, they model architectures that are popular in desktop GPUs instead of architectures employed in the low-power segment.

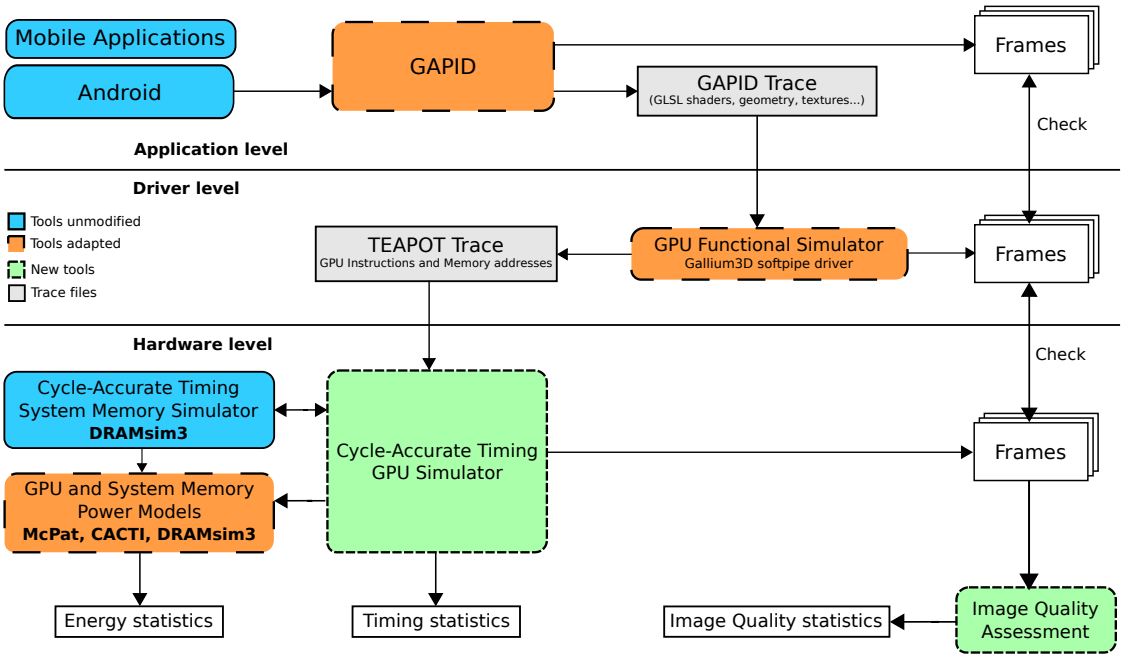


Figure 3.1: Overview of the TEAPOT simulation infrastructure.

Application Level:

The first step is to inspect and record the OpenGL ES API calls made by an application to the GPU. To that end, TEAPOT employs a modified version of GAPID [16], a Graphics API Debugger developed by Google. This tool intercepts the graphics commands of animated applications to a mobile device while running, which can be either a real smartphone or an Android Virtual Device (AVD) [2]. These captured OpenGL commands issued by the applications are stored in a trace file, which we refer to as a GAPID trace, containing all the necessary data to be able to reproduce the original OpenGL ES command stream, such as GLSL shaders, textures, geometry, and state information.

Driver Level:

In order to perform a cycle-accurate GPU simulation, TEAPOT employs a GPU functional simulator that generates a trace with all the necessary information to carry it out. For it, the GAPID trace generated in the previous level is executed with the GAPID replay (*gapir*) service over Gallium3D [15], an infrastructure for developing GPU drivers, configured with OpenGL ES as its frontend and an instrumented version of a software renderer called *softpipe* as its backend. This modified software renderer executes each of the OpenGL commands stored in the GAPID trace while gathering useful information in any of the rendering stages, such as vertices, fragments, texels, and shader programs to generate the TEAPOT trace. Actually, Gallium3D translates the vertex and fragment shader programs into TGSI instructions [36], an intermediate assembly language, so this resulting trace contains all the executed instructions by the GPU and referenced memory accesses.

Hardware Level:

The TEAPOT trace generated in the previous level is fed to a cycle-accurate GPU simulator, which gathers the activity factors from all the stages in the rendering process and outputs timing and energy statistics. Several architectures are modeled by the cycle-accurate simulator, among them, a Tile-Based GPU closely resembling any typical low-power GPU model available in the market. In any case, the architecture is highly configurable and the details and the configuration used for the experiments are explained in chapter 2 and chapter 4.

TEAPOT relies on DRAMsim3 [24] to model timing but also energy consumption of the system memory. DRAMsim3 is a popular cycle-accurate simulator that models a wide range of memory models, among them LPDDR4, which is very common in the mobile segment, including a DRAM memory controller, DRAM banks, and the required interfaces to interact with the GPU. Previously to this work, an older version was used, but as explained in section 4.2 it has been updated to the most recent version to be more accurate and representative of current memory technologies.

Regarding the GPU power statistics, the well-known McPAT [23] power framework is used for estimating the GPU's energy consumption. This tool estimates the static power for each of the hardware structures in the GPU, such as queues, caches, processors, etc., and the dynamic power derived from their activities. When the simulation ends, the activity factors gathered by all of these components are combined with their individual activation costs to obtain the overall dynamic GPU power. Finally, the energy consumption is obtained by multiplying the total power statistics by the execution time.

Last but not least, TEAPOT also includes some statistics related to image quality in order to quantify the quality of the frames generated, such as VSSIM [38], which evaluates a sequence of frames based on the human visual perception system. These metrics are useful when evaluating energy-saving techniques that trade off image quality for energy. However, since it is not the case for this work, they have not been used.

3.2 Benchmarks

The benchmark suite used to evaluate this work consists of a wide number of commercial Android applications, which are listed in Table 3.1, along with the alias names used to identify them along the experimental results. The selection criteria for these games lies in choosing the ones that are representative of the diversity that can be found in the app stores, considering variety as the main criteria, which is determined by the genre and type of the graphics, but also by its own graphical characteristics. Nevertheless, popularity has also been another important factor when choosing the games to evaluate, which is defined by the number of downloads. In fact, some of the selected games, such as Subway Surfers and Candy Crush Saga, are leaders in the download ranking chart by the time of this writing and have reached a billion downloads.

Next, we present a brief analysis of relevant aspects so we can provide a general view of the graphical characteristics for our set of applications for both Geometry and Raster pipelines. Table 3.2 shows information regarding the geometry complexity of each benchmark. The second column accounts for the average number of draw calls per frame, which is an interesting statistic that shows the impact on the number of objects. The third column shows the average number of instructions executed by the Vertex Shaders per vertex, which gives some hints on the activity in those shaders. The fourth column reports the average number of primitives per draw call, which

Benchmark	Alias	Genre	Type	Installs (millions)
Beach Buggy Racing 2	BBR	Racing	3D	10
Candy Crush Saga	CCS	Casual	2D	1000
Captain America: Sentinel of Liberty	CAm	Action	2.5D	5
City Racing 3D	CRa	Racing	3D	50
Clash of Clans	CoC	Strategy	2.5D	500
Counter Strike	CoS	Action	3D	50
Crazy Snowboard	CrS	Sports	3D	15
Derby Destruction Simulator	DDS	Racing	3D	10
Forest 2	Fo2	Adventure	3D	1
Golf Battle	GoB	Sports	3D	50
Gravity: Don't Let Go	Gra	Arcade	3D	1
Hot Wheels: Race Off	HoW	Racing	2.5D	50
3D Maze	Maz	Adventure	3D	10
Miami Crime Simulator	MCS	Action	3D	10
Real Steel World Robot Boxing	RSt	Action	3D	50
Rise of Kingdoms: Lost Crusade	RoK	Strategy	2.5D	50
Sniper 3D: Gun Shooting Games	S3D	Action	3D	500
Sonic Dash	SoD	Arcade	3D	100
Subway Surfers	SuS	Arcade	3D	1000

Table 3.1: Benchmark set.

reflects an application to be CPU limited when is too low, as it means that the CPU is unable to feed the GPU fast enough, so the GPU is mostly idle. This is because the GPU can transform and render triangles much faster than the CPU is able to submit them [39]. The last column shows the average number of assembled primitives binned per tile, which allows us to measure how much they stress the Tile Cache.

It can be seen that the benchmark set covers a rich variety of geometric complexity, from games such as *CCS* sending 16 draw calls per frame on average, to games as *Fo2* that generate 257 draw calls per frame. There is also a great diversity in the complexity in the Vertex Shader, where we can identify that *CoC* has few instructions on average per vertex, about 6, to games such as *RSt* close to 80, which is one order of magnitude larger. Finally, the footprint differences in the tiling overhead among them can be observed. For instance, *Fo2* requires storage and fetching for 75 primitives per tile on average, whereas other games such as *RoK* have a slight impact, requiring storage for an average of 3 primitives per tile. Thus, it can be seen that our set of benchmarks stresses the geometry pipeline in multiple ways.

Once the geometry stage has finished, the next step is to perform the rasterization process, where the primitives are discretized into fragments containing interpolated values of the per-vertex attributes and then compute the color of each fragment in the Fragment Shaders. Table 3.3 shows the information obtained regarding the complexity of the fragment stage. The second column shows the average number of fragments per primitive, which gives an idea of the size of the primitives. The third column reports the average number of attributes per primitive, which measures the work for the interpolation of the attributes of the vertices of a triangle. The fourth column shows the average number of instructions executed by the fragment shaders per fragment to evaluate the shaders' activity. The fifth column shows the average number of ALU instructions

per texture instruction. Considering that the texture instructions are the only ones that allow the programmer to access memory within the fragment shaders, thus, the higher the ratio the less memory-intensive is the workload. The sixth column shows the average number of *texels* (texture elements) fetched per fragment, which depends on the texture filtering type employed by the game. There are multiple filtering methods, e.g., nearest-neighbor, bilinear, and trilinear that fetch 1, 4, and 8 texels respectively for one texture. The last column reports the overdraw factor, computed as the average number of fragments rendered per pixel position.

Benchmark	**Draw calls per frame**	**VS insns per vertex**	**Primitives per draw call**	**Primitives per tile**
BBR	245.13	43.99	95.23	25.37
CCS	16.07	12.77	316.81	5.58
CAm	22.37	15.35	273.14	6.70
CRa	64.58	12.91	227.20	15.95
CoC	87.38	6.40	116.50	11.16
CoS	21.31	8.59	494.31	11.55
CrS	20.07	13.45	212.62	4.68
DDS	77.83	55.30	246.53	20.85
Fo2	257.24	45.26	266.87	75.27
GoB	146.93	25.33	196.80	31.43
Gra	48.05	49.26	302.84	15.96
HoW	53.25	65.57	1003.37	58.08
Maz	63.35	41.23	654.92	45.09
MCS	152.06	39.86	78.71	11.97
RSt	31.00	77.56	414.00	14.07
RoK	61.73	33.18	53.49	3.59
S3D	62.53	40.38	345.03	23.45
SoD	64.58	64.52	250.68	12.65
SuS	65.60	33.21	782.95	25.18

Table 3.2: Geometry workload characterization.

This reported information shows the diversity covered by the workload at the fragment level too. Notice that the number of fragments is about two orders of magnitude greater than the number of primitives. In addition, the heterogeneity between the primitive sizes can be seen, where *HoW* has little primitives whereas others as *CrS* has big primitives, with on average of 91 and 491 fragments per primitive, respectively. It can also be observed the variation in the number of attributes per primitive that the Rasterizer must interpolate, ranging from *SuS* with less than 2 to *BBR* with slightly more than 4. However, we can see that there is not much variation among them, being the average around 3. We can also see the workload diversity regarding the fragment shaders. For instance, *CoC* uses fragment shaders with 3.45 instructions per fragment, whereas *BBR* is significantly more complex with 26.04 instructions per fragment on average. Likewise, the ratio between ALU per texture instructions is substantially different, from *CoC* providing a ratio of 2.33, whereas *SoD* provides a ratio of 18.73. Finally, our benchmarks exhibit different levels of overdraw, ranging from modest degrees such as 1.21 for *SoD* to significant ones as 4.50 for *Fo2*. This is an important metric, as it exposes that an important amount of energy, time, and memory bandwidth is wasted on processing occluded fragments for a given screen position.

Benchmark	Fragments per primitive	Attributes per primitive	FS insns per fragment	ALU insns per TEX insns	Texels per fragment	Overdraw
BBR	133.39	4.38	26.04	16.31	16.57	1.96
CCS	407.37	2.99	4.00	4.25	3.97	2.26
CAm	326.21	2.70	5.60	4.64	5.88	1.96
CRa	223.46	2.75	10.94	7.67	8.43	2.13
CoC	350.26	2.69	3.45	2.33	4.00	3.88
CoS	314.19	2.41	7.42	5.76	7.47	2.24
CrS	491.68	2.71	6.31	8.10	4.00	1.22
DDS	202.70	3.44	19.49	15.94	6.16	2.09
Fo2	178.85	3.05	17.36	7.53	14.14	4.50
GoB	244.30	3.27	14.14	17.99	5.79	3.44
Gra	105.91	3.98	17.43	9.04	10.17	1.35
HoW	91.06	3.60	25.62	11.78	17.53	2.55
Maz	117.94	2.65	17.65	12.44	7.59	2.91
MCS	457.17	3.00	10.68	13.49	4.67	3.75
RSt	346.96	3.21	8.65	11.26	4.94	2.75
RoK	387.33	2.65	14.91	18.49	4.46	1.37
S3D	243.43	3.87	18.46	11.11	10.30	2.96
SoD	215.96	3.37	17.55	18.73	6.85	1.21
SuS	229.50	1.92	9.65	11.03	4.07	2.68

Table 3.3: Fragment workload characterization.

4

Baseline improvement

This chapter presents the main improvements performed to the TEAPOT simulator in order to achieve a more realistic architecture model, more closely resembling the graphics pipeline of a modern GPU as those currently available in the market. It is important to note that it is crucial to work with a robust and trustworthy baseline model since it is the starting point from which new ideas arise and determine whether or not they make sense.

4.1 Throughput increase

The initial baseline model in TEAPOT had a relatively low throughput in the Raster Pipeline, about 1 quad per cycle, which was sufficient for previous works with less flexible configurations. However, for the proposal that will be presented in chapter 5, a more flexible baseline model design is needed to remove some of its limitations and to avoid any potential bottleneck that may limit the benefits of our proposal.

For this reason, on the one hand, the Raster Unit has been modified in all the stages except for the Fragment stage, which is meant to be the bottleneck in the GPU pipeline. On the other hand, regarding the Tiling Engine, the Tile Fetcher has been completely re-designed.

4.1.1 Raster Unit

As fragment shader programs typically include complex operations, such as texture fetching, the Fragment stage is usually known to be the congesting zone in the pipeline. For this reason, the throughput for most of the stages in the Raster Unit has been increased, so they do not become a bottleneck. Thus, the throughput for the Rasterizer, Early Depth stage and Blending stage has been increased so they can process up to 4 quads per cycle.

Recall that the Rasterizer interpolates the value of the primitive's attributes at the pixel's position and then generates groups of 2x2 adjacent fragments (a quad fragment) that are sent together to the next stage of the pipeline. In Figure 4.1, it is shown the average number of

attributes per primitive, where it can be seen that the average is around 3.1. Therefore, setting it to process up to 4 attributes per quad prevents that it can become a bottleneck.

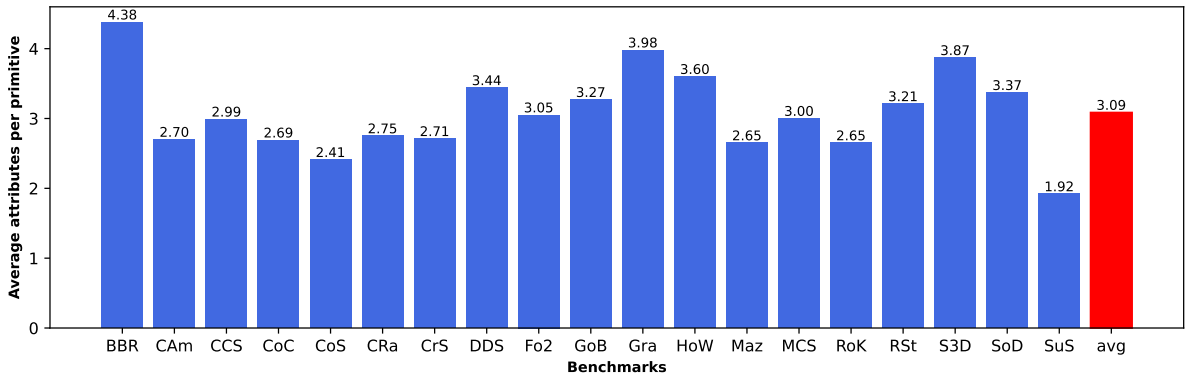


Figure 4.1: Average attributes per primitive.

Regarding the Early Depth stage, it is not bottlenecking the system as it has no resource limitation since it only needs to compare a pixel depth value with the one previously read from the Z-Buffer (an on-chip buffer) in the same tile position and, depending on the result, make an update of the pixel depth value. These on-chip buffers are fixed-latency memories, in our case 1 cycle, where no misses occur. In addition, there are separate ports for reading and writing, so there is no conflict. Thus, there is no reason for this stage to be the bottleneck. In any case, the throughput has been increased to 4 quads/cycle in order to allow for a continuous flow of quads to be tested and not become a congestion point in the graphics pipeline.

As for the Blending stage, the behavior is similar to the Early Depth stage, where there is a read/write operation over an on-chip buffer, known as the Color Buffer. Therefore, for the same reason, the throughput for this stage has been increased to 4 quads/cycle in order not to become an obstacle.

The remaining stage in the Raster Unit is the Fragment stage. However, as it is meant to be the bottleneck since it is the one with the highest hardware requirements, it has not been improved. Nevertheless, the input for this stage has. In the original baseline, there was a single input queue for the Fragment stage, but that is a bad design choice as having one single input queue in the Fragment stage can become a significant source of delays. The reason for this is that each quad fragment is statically assigned to a particular Fragment Processor according to its tile coordinates, so in case the quad sitting at the head of the queue is assigned to a busy Fragment Processor, it blocks the queue and prevents other processors that may be idle from doing useful work (note the input queue works in a FIFO fashion). Therefore, one first architectural change has been to increase the number of input queues so that each shader core has its own one, in order to minimize starvation issues.

Other important structures that need to be considered are the inter-stage queues, which have also been re-sized, so they do not become a bottleneck. In the first place, the Tile Queue has been sized to 32, so it can store up to 32 primitives. The reason for this particular value is because, as seen in Figure 4.2, the average number of primitives per tile is 22.03. Thus, for most of the games, it will be able to hold a whole tile and also leave some extra margin. Regarding the Post-Rasterization Queue, it has been re-sized to hold up to 512 quads. Since our tile size is 32x32 pixels, it means that each tile has 16x16 = 256 quads. However, it has been

provisioned to hold 2x that number of quads since *overdraw* is a very common effect in most of the games. Thus, the chosen size will allow, in general, a continuous flow in the pipeline. As for the Pre-Fragment Processing Queues, the reasoning is the same as for the latter, but the overall size has been distributed among the four different queues. Thus, each of them holds up to 128 quads. Finally, the Color Queue has been set to hold up to 64 quads, which is enough as the processing throughput drops at the Fragment processors, right before reaching the Blending stage.

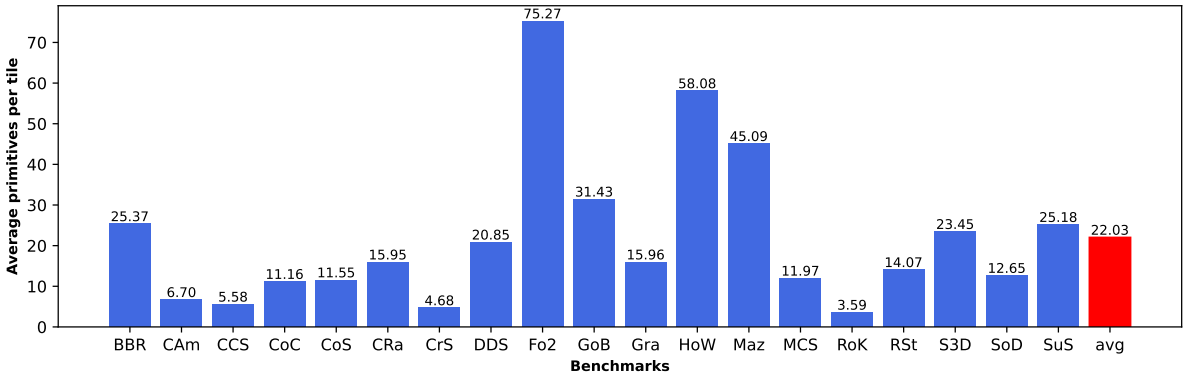


Figure 4.2: Average primitives per tile.

Figure 4.3 depicts the Raster Pipeline's diagram for the baseline design. Note that in section 4.3, we will report all the configuration parameters used for the GPU simulation.

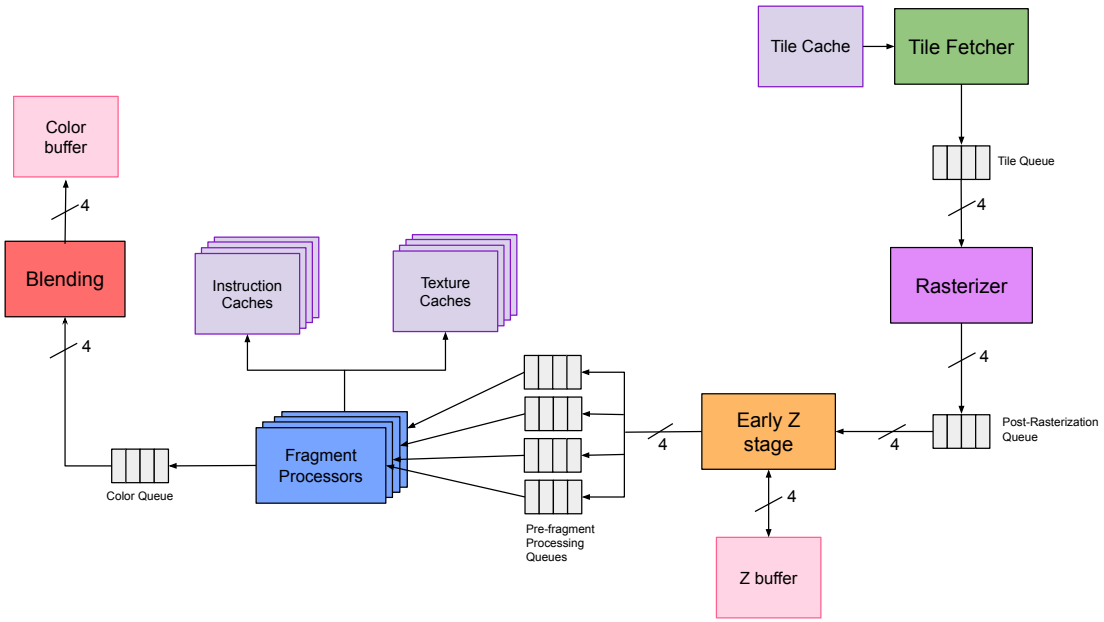


Figure 4.3: Baseline scheme.

4.1.2 Tiling Engine

The Tile Fetcher is a key component, as its performance is crucial for exploring and evaluating the benefits of our proposal. Remember that this unit retrieves the tile's primitives that the Polygon List Builder previously stored in the main memory region referred to as Parameter Buffer. Therefore, as our approach improves the Raster Pipeline performance, the Tile Fetcher has to be able to provide an acceptable throughput to multiple Raster Units, so its performance is not limited because of the Tiling Engine.

The Tile Fetcher works as follows. First, it is determined which is the next tile to be scheduled. The order in which tiles are processed is specified by the Tiling Engine, which in the baseline setup is the scanline order. The next step is to compute the address of the pointer to the next primitive to fetch (which is derived from the IDs for both the primitive and the tile) and is sent as a request to the Tile Cache. Once this primitive pointer has been received, it is calculated the address of the pointer to the primitive's first attribute (which is derived from the primitive pointer previously received) and is sent as a request to the Tile Cache. Recall that attributes describe the characteristics of the vertices of a primitive, such as color, texture, normal vectors, etc. After receiving an attribute, it is checked if the primitive has more attributes. If so, the same steps will be followed: calculate the address for the new attribute, send the request to the Tile Cache, and wait until receiving it. This procedure is repeated until all the attributes belonging to that primitive have been received. When so, the primitive will be pushed into a FIFO queue so it can be consumed by the Raster Unit. To better clarify the explanation, Figure 4.4 shows the structure of the Parameter Buffer graphically.

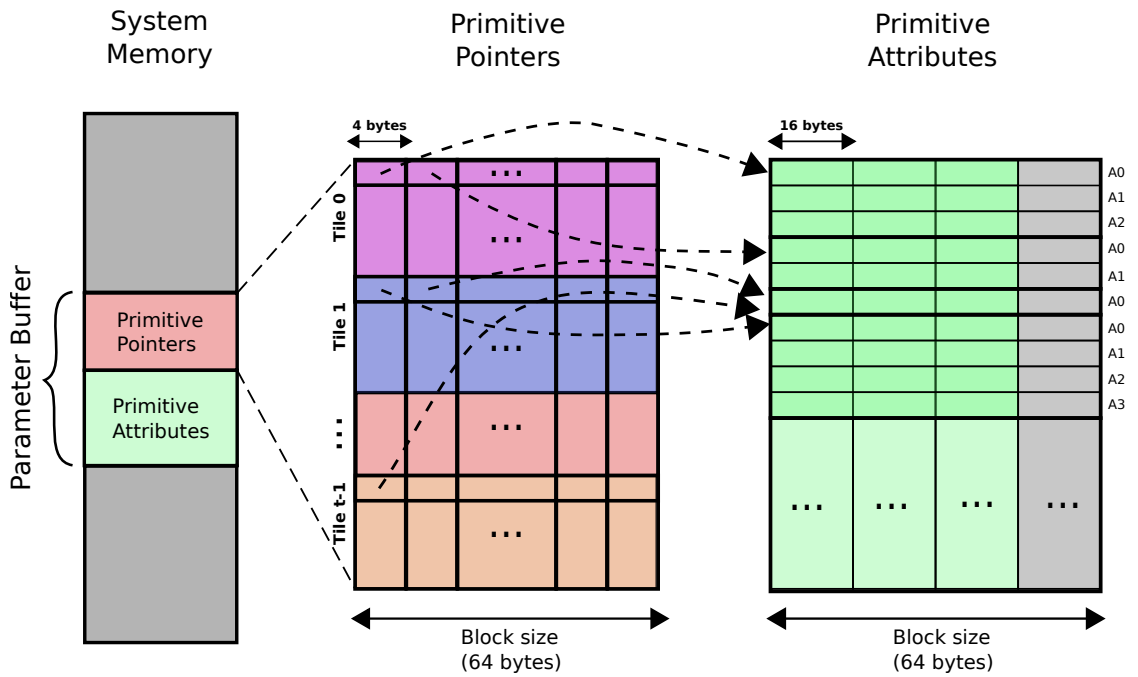


Figure 4.4: Structure of the Parameter Buffer.

Nevertheless, in this baseline design, we have detected two deficiencies that have been fixed as they were producing unnecessary stalls in the pipeline. On the one hand, it was observed

that the Tile Fetcher remains stalled during many cycles due to the memory latency of cache misses that may occur both in the Tile Cache and in the L2 Cache. On the other hand, the baseline design has another issue: it does not start processing the next tile until the current one has totally finished its rendering process. Thus, a new design for the Tile Fetcher had to be done.

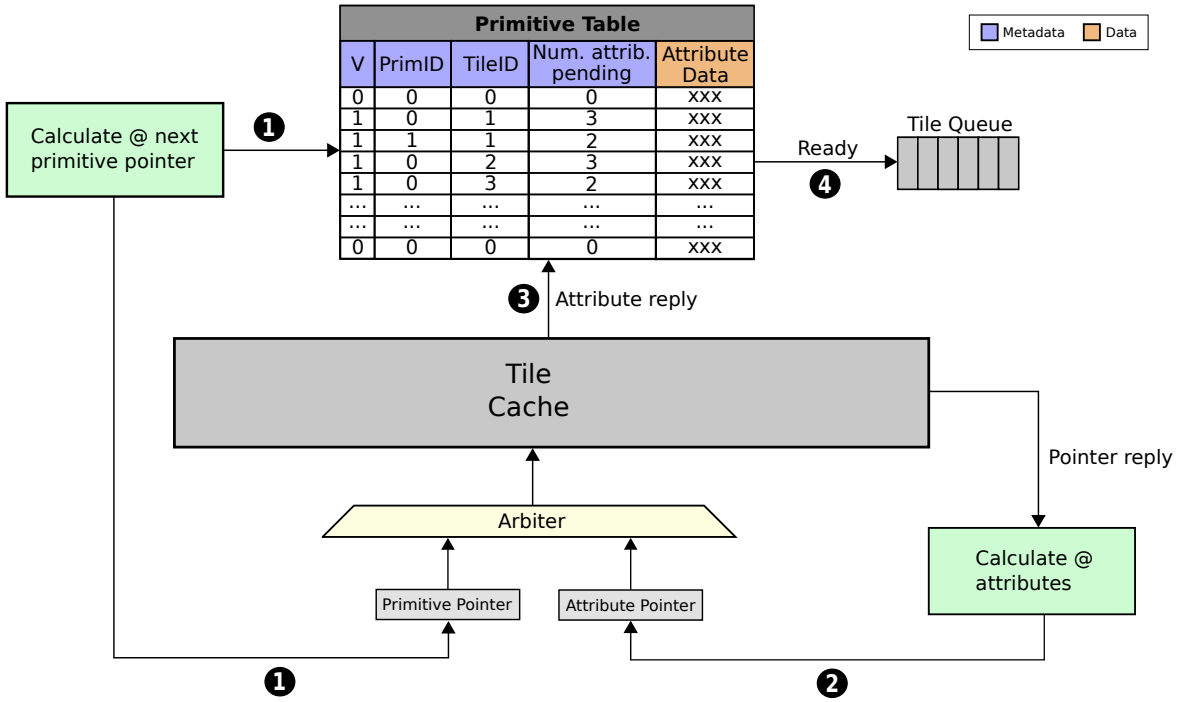


Figure 4.5: New hardware design for the Tile Fetcher.

The first objective is to **hide memory latency**. To do that, the Tile Fetcher has been pipelined among four stages, as illustrated in Figure 4.5. The new Tile Fetcher works as follows. For a given tile, the first stage (❶) computes the address for the next primitive pointer within that tile. Notice that the memory request for that address is pushed into a register, referred to in the drawing as a Primitive Pointer, in which the request for the next primitive pointer is stored. In addition, this stage also reserves an entry in a new hardware structure, which we refer to as the Primitive Table.

As several primitives have to be fetched in parallel, a structure to organize the status of each of them is needed. Each entry in the Primitive Table contains a "valid" field to indicate if the entry is in use or not; the IDs for both the primitive and tile (PrimID and TileID, respectively) to identify the owner of the data on that entry; the number of pending attributes to be received and the attribute data itself, which are needed for further processing in the next Raster Pipeline stage. Therefore, in case the Primitive Table is not full, a new entry in this table is reserved and marked as valid (V), while also updating the table with the IDs for the primitive and tile it belongs to.

Once the pointer for a primitive has been received, it proceeds to the second stage (❷). In this stage, the address of the next attribute of a primitive is computed and pushed to a register that we refer to as Attribute Pointer, which holds the memory address for the next primitive attribute request. In addition, when a primitive is inserted in the Primitive Table, the field that

tracks the number of pending attributes is set to the total number of attributes.

When the Tile Cache serves an attribute, it is passed over to the third stage (❸). In this stage, it is updated the Primitive Table by storing the attribute data received and decreasing the number of pending attributes. It also compares if that particular primitive has received all of its attributes, or there are more to be requested. In case it has finished, the next cycle will execute the last stage (❹), which pushes the primitive and its attributes data to the output FIFO queue referred to as Tile Queue, from which the Raster Pipeline is fed. Otherwise, it will go back and compute the address for a new attribute for that primitive (❷). However, note that in order to maintain the primitive order, it will be pushed in case it is the oldest primitive in the Primitive Table. Contrarily, it will wait until it becomes.

Notice that, before reaching the Tile Cache, there is an arbiter for both Primitive Pointer and Attribute Pointer registers, which sends up to one memory request per cycle. This arbiter has been added in order to orchestrate which memory request is sent before another, as the Tile Cache has only one port (thus, the bottleneck in the Tile Fetcher is the Tile Cache). The arbiter follows an **attributes-first policy**, giving priority to the attribute requests rather than pointer requests. The reason for this is that the attribute requests in the Attribute Pointer register belong to a previous primitive than the one being requested in the Primitive Pointer register. This is an important factor to take into account, as we process the tiles in-order. Thus, prioritizing attributes over primitives avoids stalling, as data is not pushed to the Tile Queue unless all the attribute data belonging to a primitive has been received.

On the other hand, the other objective for improving the Tile Fetcher's efficiency is to **pipeline the tiles**. As mentioned before, we identified an issue with the Tile Fetcher which does not start fetching the next tile until the current one has finished all the rendering process. This is a significant source of inefficiency since tiles are totally independent of each other. Thus, this characteristic of independence has a lot of potential, since the Tile Fetcher can proceed to process the next tile right after finishing the previous one, avoiding unnecessary stall cycles that cause a serious performance bottleneck in the pipeline. However, as tiles cannot be mixed between them, some extra logic has been added between the Raster Pipeline stages to act as barriers. This way, a stage will not start processing a new tile until it has completely finished the current one.

Another important aspect to consider is the impact of this new hardware structure on the GPU energy consumption instead of just focusing on the performance benefits. The Primitive Table has been sized to 16 entries, which is a small number that provides a good trade-off between performance and table size. The size of each entry is a bit big, as it needs to store all the attribute data, which occupy significant space. So, an entry of the Primitive Table is composed of metadata (marked in purple in the Figure) and attribute data (marked in orange). As for the metadata side, it is required 1 bit to mark an entry as in-use or not. Subsequently, given that the maximum number of primitives within a tile has been set to be 1264 as some games were requiring a close resource value, 11 bits are enough to identify these primitives in the primID field. As for the tileID, given that the smartphone employed for the experiments is equipped with a 1920x1080 display (which is a common display resolution nowadays) and that the tile size is set to 32x32, the maximum tiles per frame that we can obtain is $\lceil \frac{1080}{32} \rceil \times \lceil \frac{1920}{32} \rceil = 34 \times 60 = 2040$. As a result, 11 bits are sufficient to be able to identify all the tiles present on each frame. Finally, 4 bits are needed for accounting for the number of pending attributes to be received, as OpenGL ES 3.0 guarantees support for up to 16 attributes per vertex [17]. We have not added more bits for it as we have seen that none of our evaluated benchmarks has even come close to it. Summarizing, for storing the metadata it is needed: 1 bit (V) + 11 bits (primID) + 11 bits

(tileID) + 4 bits (number of pending attributes) = 27 bits.

As for the attribute data part (where the attributes of a primitive are stored), given that it is guaranteed hardware support for at least 16 attributes per vertex, and each primitive has three vertices, this means that we have up to 48 attributes per primitive. As each attribute is 16-byte long, the data field for each entry requires 48 attributes × 16 bytes per attribute = 768 bytes (or 6144 bits).

With all this, each entry of the Primitive Table is 27 bits + 6144 bits = 6171 bits. As a result, even though some fields require a significant amount of space, the full table remains relatively small, requiring about 12KB (16 entries × 6171 bits/entry = 98736 bits) of space in total.

Last but not least, Figure 4.6 shows the relative energy consumption of the Primitive Table over the total energy consumed by the overall GPU and main memory, where it can be seen that this table has a negligible effect.

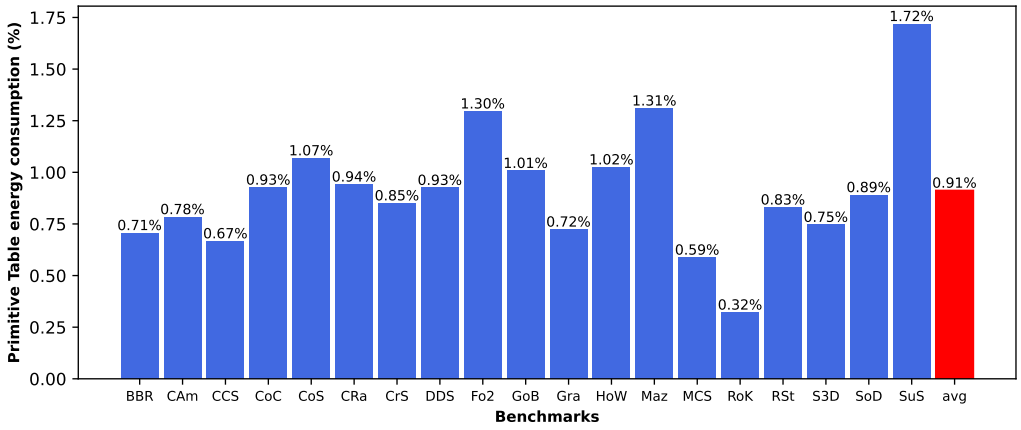


Figure 4.6: Impact on energy consumption by the Primitive Table.

4.2 DRAM memory model upgrade

Previous to this work, TEAPOT used DRAMSim2 [31] to accurately model the timing of main memory, and McPAT power models [23] to estimate its energy consumption. Given that DRAM technology has evolved over the last few years, it became outdated as newer protocols have been introduced into the market.

In order to update the main memory model incorporated in TEAPOT, we have relied on DRAMsim3 [24], a successor to DRAMSim2. DRAMsim3 is a cycle-accurate system memory simulator that models major DRAM technologies: in addition to DDR, it also includes GDDR, LPDDR and stacked memories. In our case, we employ the LPDDR4 memory model as it is a low-power memory that is widely used in embedded devices.

Furthermore, this tool also models the DRAM memory controller, DRAM banks, and provides interfaces that permit interaction with the GPU. Another interesting point compared to other similar works is that this timing simulator has been hardware validated to check its simulation accuracy.

Last but not least, one of the new things it presents is that it provides a power model. Hence, the power model employed for system memory has also been updated, so now we use the one offered by DRAMsim3 instead of using McPAT. We believe this is a reasonable choice since it is a much newer tool, and we expect it to be more realistic. The energy consumption is computed on-the-fly where some activity factors are gathered during the simulation, which are eventually combined with the corresponding energy increments. In addition, it also provides power consumption statistics by dividing by the execution time.

4.3 GPU configuration

Table 4.1 provides the architectural parameters employed to perform the experimentation, which have been configured so it resembles a typical low-power GPU model.

	Baseline GPU Parameters
Tech Specs	800 MHz, 1V, 22nm
Screen Resolution	1920x1080 (Full HD)
Tile Size	32x32 pixels
	System Memory
Tech Specs	1.2 GHz, 1.2V
Latency	50-100 cycles
Bandwidth	4 bytes/cycle (dual-channel LPDDR4)
Size	8 GB
	Queues
Vertex (input and output)	16 entries
Primitive and Tile	32 entries
Post-Rasterization	512 entries
Pre-Fragment processing (x4)	128 entries
Color	64 entries
	Caches All of 64 bytes/line
Vertex Cache	4 KB, 2-way associative, 1 bank, 1 cycle
Tile Cache	32 KB, 4-way associative, 1 bank, 2 cycles
Texture Caches (x4)	8 KB, 2-way associative, 1 bank, 2 cycles
Instruction Caches (x2)	16 KB, 2-way associative, 2 banks, 2 cycles
L2 Cache	2 MB, 8-way associative, 8 banks, 18 cycles
	On-chip memories
Depth Buffer	4 KB, 1 bank, 1 cycle
Color Buffer	4 KB, 1 bank, 1 cycle
	Non-programmable stages
Primitive Assembly	1 triangle/cycle
Rasterizer	16 attributes/cycle
Early Z stage	20 in-flight quad fragments
Blending stage	20 in-flight quad fragments
	Programmable stages
Vertex stage	4 vertex processors
Fragment stage	4 fragment processors
	Tile Fetcher hardware
Primitive Table	16 entries

Table 4.1: GPU Simulation Parameters.

5

Parallel Tile Rendering

In this chapter, *Parallel Tile Rendering* (PTR), a novel approach to boost GPU performance, is proposed. The motivation for this is that it is observed that most of the rendering time is invested in the Raster Pipeline, as shown in Figure 5.1. Furthermore, as previous work [5] has found this is the most energy-consuming task, we seek to reduce it.

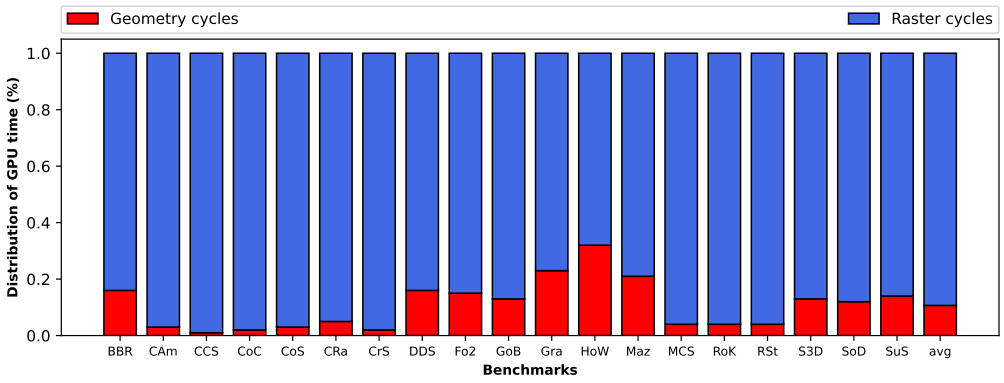


Figure 5.1: Distribution of time in the GPU.

5.1 Baseline architecture

The idea behind Parallel Tile Rendering (PTR) is to rasterize multiple tiles in parallel, maintaining the barrier of processing one tile per stage, with the purpose of reducing the cycles required for rasterizing the whole frame. This makes sense since it is observed that the Tile Fetcher now has enough throughput for feeding two Raster Units at once. Having two independent Raster Units will allow us, ideally, to perform the rasterization process in half of the time.

The assumed GPU baseline implements a Tile-Based Rendering (TBR) architecture, as explained in section 2.2. To rasterize multiple tiles in parallel, the Raster Pipeline now contains

two Raster Units, Raster Unit 0 and Raster Unit 1, each containing the same amount of resources as in the baseline, as illustrated in Figure 5.2. Note, however, that one Tile Queue is required for each Raster Unit as having a single FIFO queue would become a big source of delays because one Raster Unit would not make progress until the other one completely consumes its data.

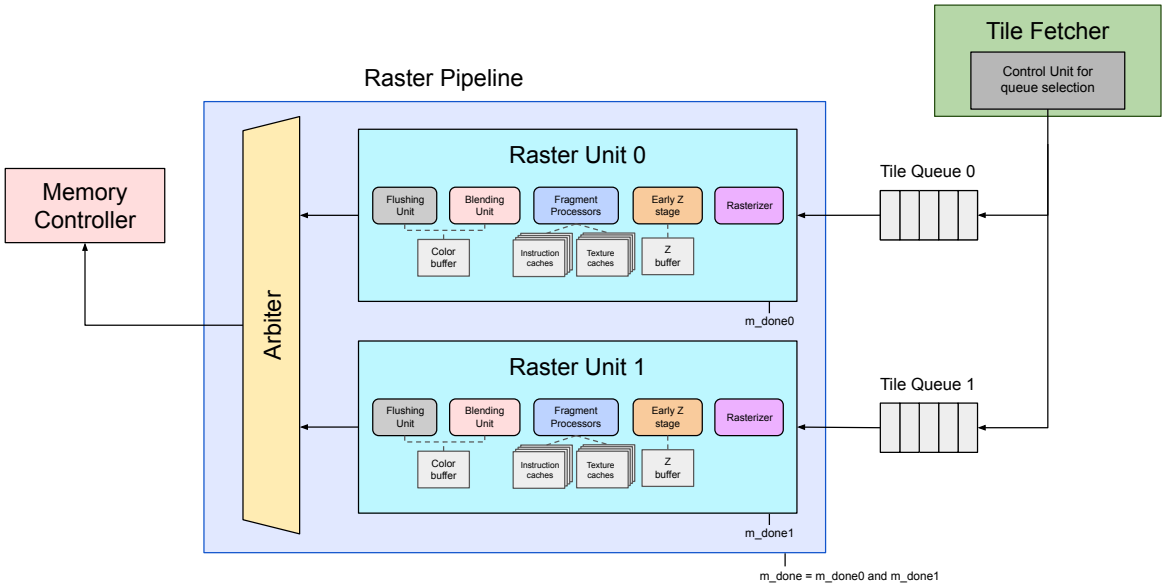


Figure 5.2: Raster Pipeline architecture employed for Parallel Tile Rendering. Each Raster Unit has its own private resources.

Since the Tile Fetcher has multiple output queues, a Control Unit is needed to select which Raster Unit to send each primitive to. Note that the scheduling needs to be done at tile-level; otherwise, the primitive order could be broken. In addition, it is required to schedule at tile-level since all the on-chip buffers (Depth Buffer and Color Buffer) are tile-sized, separate for each Raster Unit, and not backed in memory. So, all primitives belonging to the same tile will be scheduled to the same Raster Unit. Otherwise, the Raster Pipeline will render incorrectly. In this work we employ a **static** tile **assignment in an interleaved manner**. Thus, Tile 0 will be assigned to the queue belonging to Raster Unit 0, Tile 1 will be assigned to the queue belonging to Raster Unit 1, Tile 2 will be assigned again to the queue belonging to Raster Unit 0, and so on. In other words, it works in a round-robin fashion. Nevertheless, other approaches have also been explored.

Aside from static scheduling, dynamic approaches have been considered. We have studied a couple of policies, which we believe are reasonable. On the one hand, it has been evaluated a dynamic scheduling based on the fastest Raster Unit. Within this method, a tile is scheduled to the Raster Unit with fewer elements to process in its Tile Queue, which means that the pipeline is more fluid. On the other hand, it has been explored a mixed scheduling policy between the other two: it employs a dynamic scheduling, also favoring the fastest, but in case of a tie, it applies a round-robin policy by selecting the non-last-chosen queue.

Figure 5.3 compares the speedups of PTR over the baseline when applying each of these tile scheduling policies. As it can be seen, all of them are equally effective. Achieving that

both Raster Units are equally workload-balanced is an important goal, and is the main focus of dynamic policies. However, we chose the static scheduling, which is the most straightforward policy and, as shown in Figure 5.4, it already achieves a good workload balancing. In addition, it is the cheapest in hardware even though the dynamic approaches would be simple to implement, but they do not bring any significant benefit over the static.

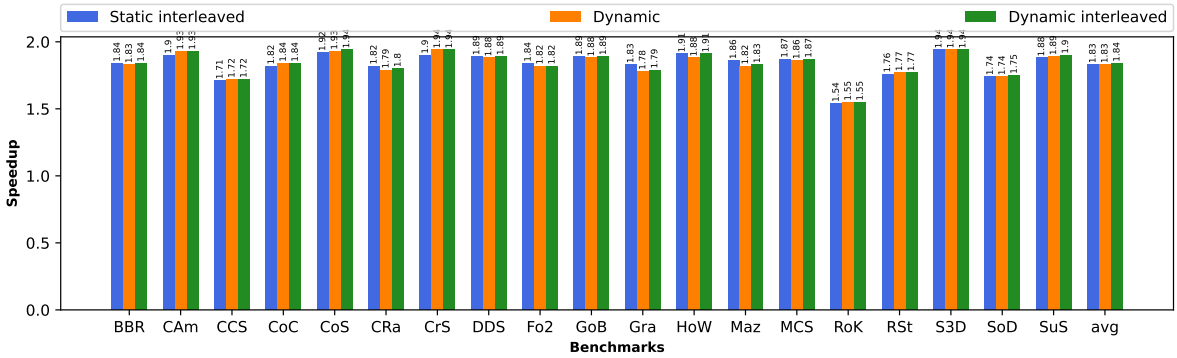


Figure 5.3: Speedup comparison in rasterization when applying different tile scheduling policies.

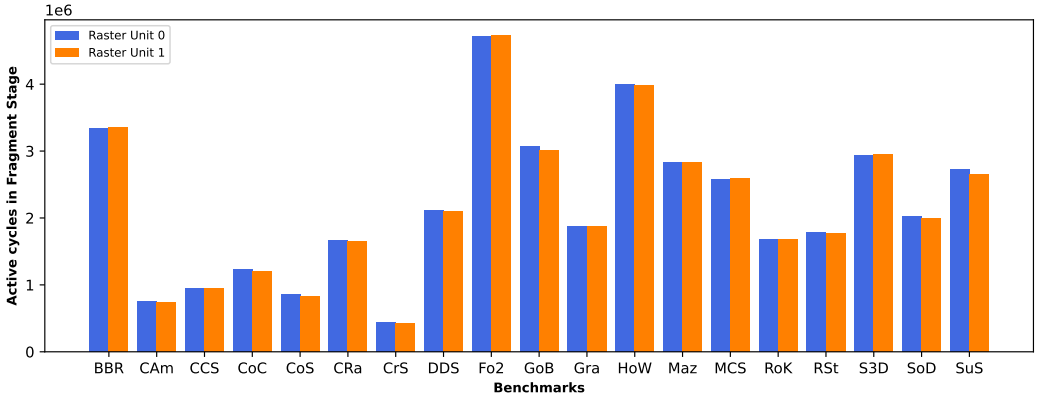


Figure 5.4: Active cycles in Fragment Stage with a static tile scheduling. Both Raster Units are quite well workload balanced.

Last but not least, an arbiter is also required for orchestrating the order between Raster Units when flushing to memory. This arbiter first determines which blending units among the different Raster Units are ready to flush. Then, in case of more than one being ready, it selects which one of them is going to be flushed. An **older-first policy** is applied in case of multiple candidates, giving priority to the slowest Raster Unit.

Improving energy efficiency is an important design aspect when designing a system, but this is even more critical in a mobile one, as we saw in the introduction of this work. Thus, to finish, Figure 5.5 presents the energy consumption for the overall GPU and system memory, for both the baseline and PTR. It can be observed that in most of the cases, important energy savings are achieved when employing PTR, with an average decrease of 9.86%. Note that this is in addition to an 83% performance speedup as shown in Figure 5.3. It must also be observed that some benchmarks show a much higher benefit. For example, *S3D* achieves a 94% increase in performance and a 13.09% decrease in energy. We have also checked the area overhead and we

have seen an increase of 40% in area, according to McPAT, and yet we end up with an almost 10% decrease in energy.

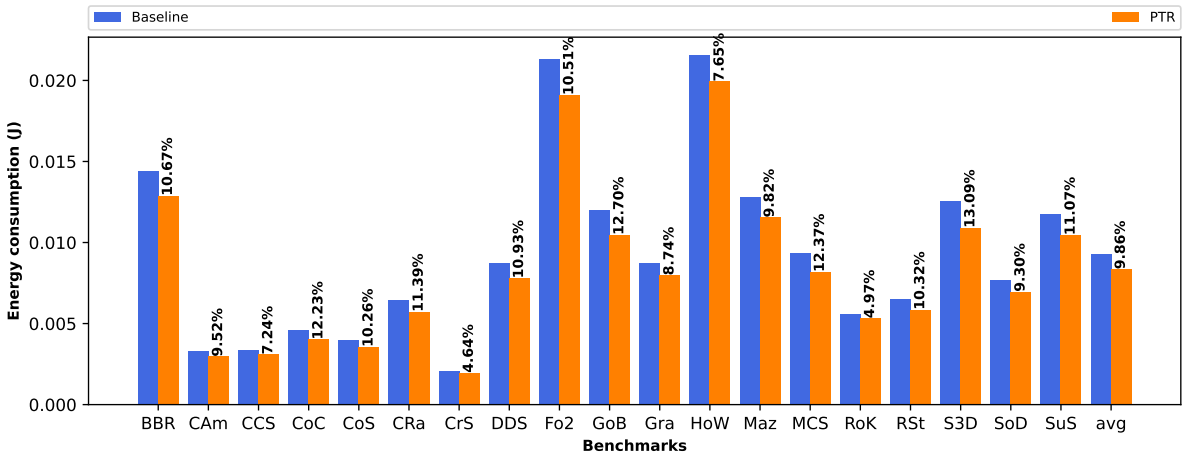


Figure 5.5: Comparison of the overall energy consumption between the baseline and PTR, presenting the energy reduction obtained with PTR.

5.2 Better than just doubling resources

To provide more insights as to Parallel Tile Rendering is a promising approach, it is a must to compare it with the case of a single Raster Unit. To make a fair comparison with PRT, we have doubled the resources in the single Raster Unit setting to have twice as much the resources as the initial baseline. However, along this section we will see that PTR is capable of making a better utilization of the resources and it better exploits parallelism, achieving a higher speedup than simply doubling the computing resources. To this end, the experiments have been carried out changing the following resource capabilities:

- Rasterizer can reach up to a maximum throughput of 8 quads/cycle. Therefore, processing up to 32 attributes/cycle.
- Early Z stage can reach up to a maximum throughput of 8 quads/cycle, with up to 8 concurrent reads and 8 concurrent writes to the Z-buffer. Therefore, having up to 40 in-flight quads.
- Fragment stage is composed of 8 Fragment processors, leading to 8 Instruction caches and 8 Texture caches.
- Blending stage can reach up to a maximum throughput of 8 quads/cycle, with up to 8 concurrent reads and 8 concurrent writes to the color buffer. Therefore, having up to 40 in-flight quads.

Figure 5.6 shows the speedup obtained when rendering with double the resources in a single Raster Unit. For ease of comparison against using PTR, Figure 5.7 compares the speedup of PTR and a single Raster Unit with double the resources, w.r.t. to the baseline single Raster Unit.

We can see from the results that PTR provides better performance than just doubling resources for the case of a single Raster Unit. The reason for this is that PTR is more likely to take advantage of the maximum throughput that the stages can provide while also offering more flexibility in barriers.

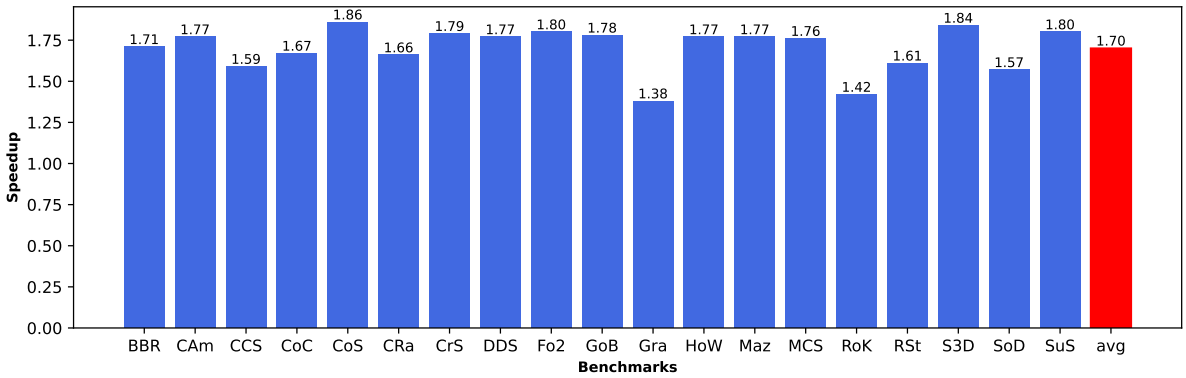


Figure 5.6: Speedup when rasterizing with the double of resources in a single Raster Unit.

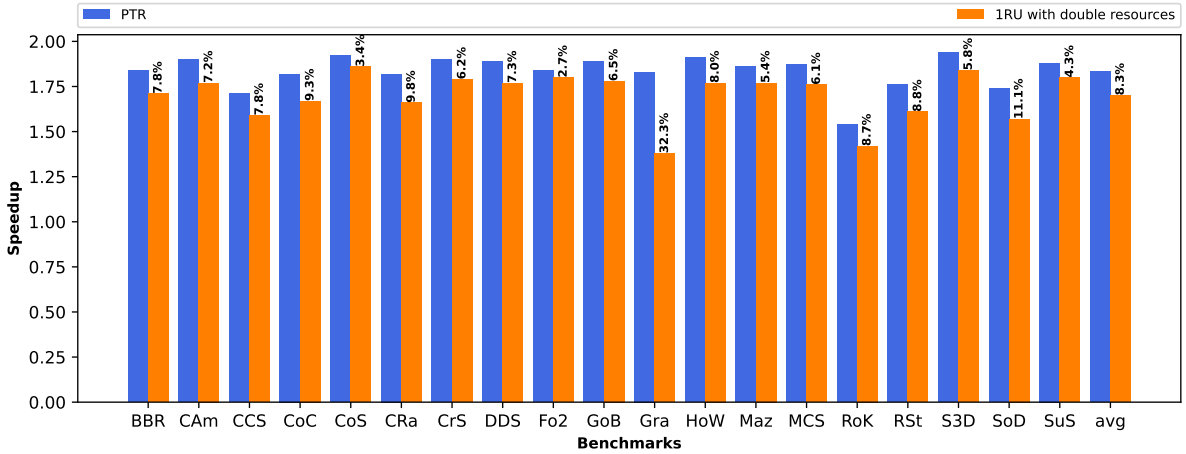


Figure 5.7: Benefits of PTR over just doubling resources.

Nonetheless, it can be observed that when employing PTR we do not reach a 2X performance for any of the benchmarks. The reason is because we have not doubled the bandwidth on upper levels in the memory hierarchy (L2 cache and DRAM). Note that it can be observed two types of games: the ones that are close to reach the 2X speedup, and the ones that are further. Figure 5.8 shows the breakdown of the execution time for each of the benchmarks, which gives us a hint on whether a benchmark is compute- or memory-bound. The greater the part required for memory accesses, the more memory-bound the benchmark is. Contrarily, a significant part required for computation means that the benchmark is compute-bound. This is an important aspect to consider given that we are increasing only the computing capabilities. This implies that the benefits of this doubling will affect the compute-bound benchmarks more than the memory-bound benchmarks, as their performance is still dominated by memory accesses. In other words, as also pointed by Amdahl's law, if only one part of the system is improved while there is another part which is not but it is very influencing, the overall achieved benefit will be limited. In this case, if an application is very memory intensive, less overall benefits will be obtained as the memory

subsystem remains the same. In fact, *CCS* and *RoK* are further from achieving the desired 2X performance speedup, showing a speedup of 1.71 and 1.54, respectively. In addition, both are memory-bound, as can be seen from Figure 5.8. Figure 5.9 more clearly shows this correlation where these benchmarks that are further are the ones that have higher L1 and L2 misses per 1000 instructions. In this case, *CCS* has the highest L1 MPKI and *RoK* has the highest L2 MPKI.

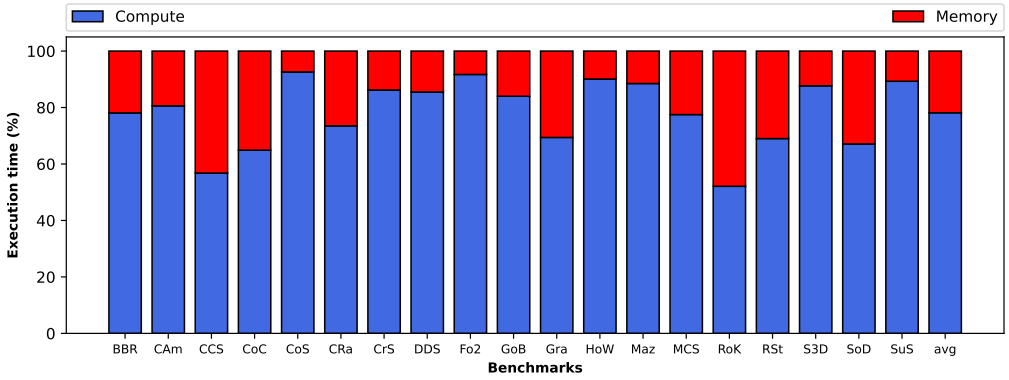


Figure 5.8: Breakdown of the execution time between compute and memory tasks for PTR.

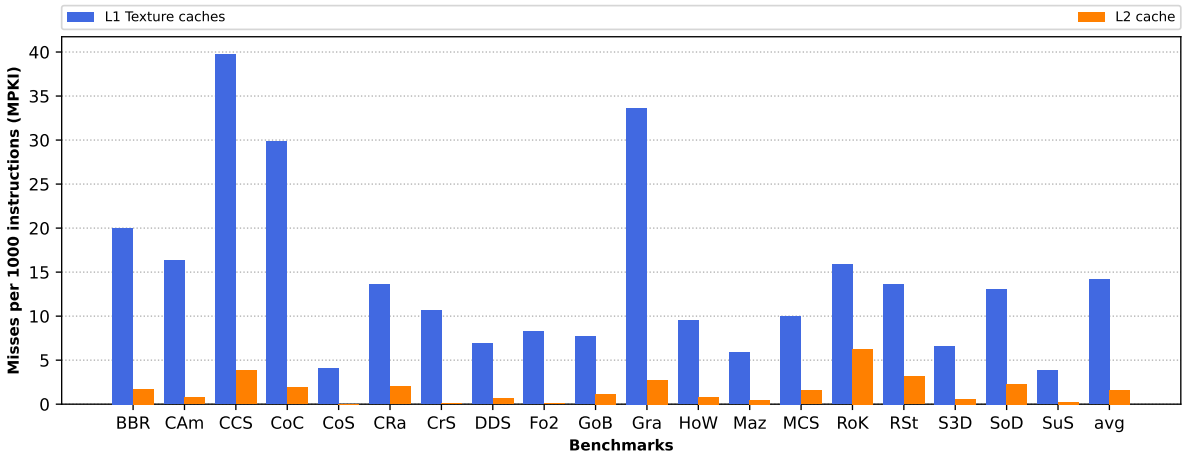


Figure 5.9: Misses per 1000 instructions (MPKI) for the L1 texture caches and L2 cache due to texture accesses for PTR.

Another observation here is that even though it is clear from Figure 5.7 that just doubling the resources does not provide as much speedup as PTR, *GRa* and *RoK* speedups are significantly lower than the other benchmarks. To understand this, Figure 5.10 shows the average number of fragments per tile, which essentially represents the number of threads assigned to the Fragment Processors per tile. By this information, we notice that four benchmarks (*CrS*, *GRa*, *RoK*, and *SoD*) have a significantly low workload per tile. Moreover, the workload is even lesser when it is divided among 8 Fragment Processors, which is the case when having one single Raster Unit with double the resources. Recall that the Fragment Stage cannot start processing a new tile until the previous one has finished. Also note that Fragment Processors, also known as GPU cores, rely on multi-threading to produce a high throughput. When the occupancy of the Fragment Processor is low, the throughput is also low. Long Latency operations like a memory miss renders

a thread "blocked" in the Fragment Processor. As a result, benchmarks with low workload and high MPKI lead to low performance. Out of the four benchmarks mentioned above, all have low workloads but only two of them have high MPKI: *GRa* has a high L1 MPKI, and *RoK* has a high L2 MPKI. Thus, this explains why both these benchmarks show an exceptionally low benefit from just doubling resources within one single Raster Unit.

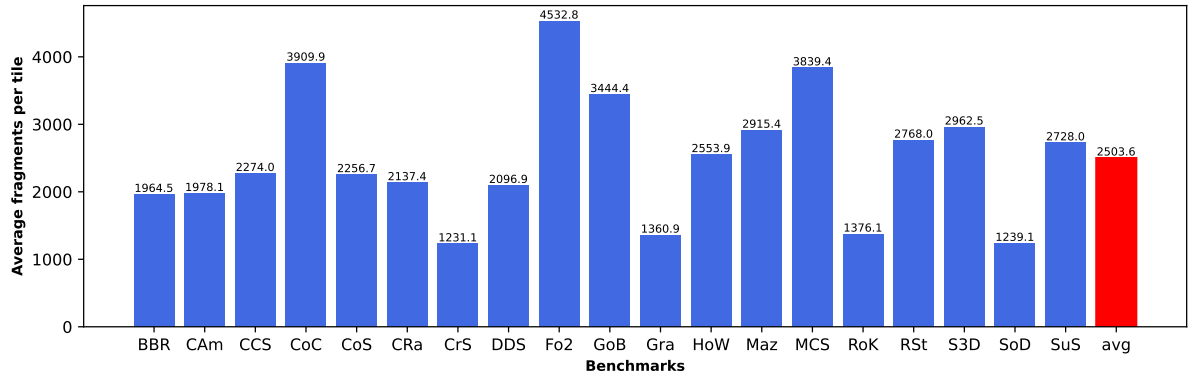


Figure 5.10: Average fragments per tile.

5.3 Opportunities for texture sharing

Throughout this chapter we have verified that employing Parallel Tile Rendering is a good approach for boosting GPU performance. Nevertheless, this does not end here. PTR has opened doors to new explorations such as sharing Texture caches between Raster Units.

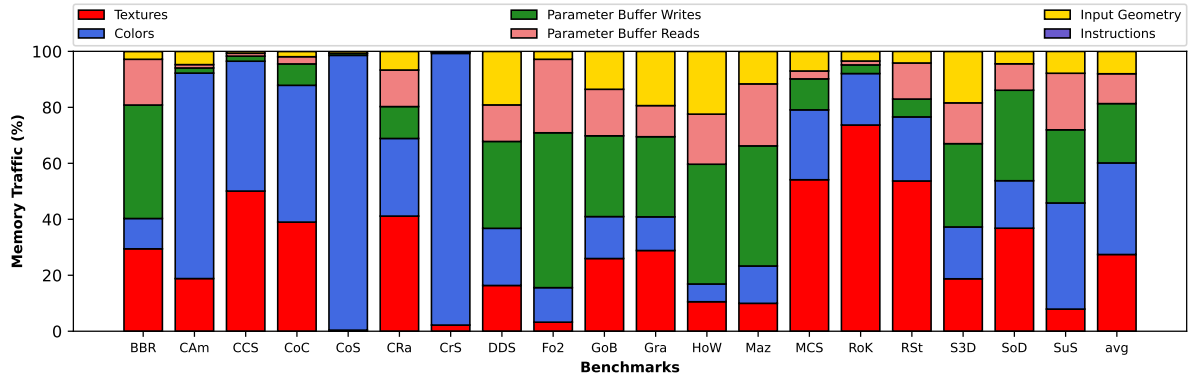


Figure 5.11: Memory bandwidth usage.

Fetching from memory is one of the major sources for energy inefficiency, where texture data is typically one of the main causes of main memory accesses, as shown in Figure 5.11. Therefore, we seek to improve texture locality by sharing texture caches between different Raster Units. Figure 5.12 shows that, on average, almost 50% of the texture addresses are reused inside a frame between different tiles. Moreover, most of these shared texture blocks are accessed in the next tile, which suggests that this reuse likely occurs in the edges of the tiles. Therefore, sharing texture caches could take advantage of the maximum reuse of textures between two Raster Units. In addition, the plot also shows the reuse at a distance of two tiles, which gives us a hint on

the locality in a Raster Unit as it will process tiles at a distance of two. It might seem that this reuse is low, but we have considered only as reuse in distance 1 in the case of being reused in both distance 1 and 2. Hence, the energy efficiency could be improved, but also performance, when enhancing texture locality thanks to inter-tile locality. We have explored this potential, and chapter 6 is the study on the inter-tile texture sharing technique.

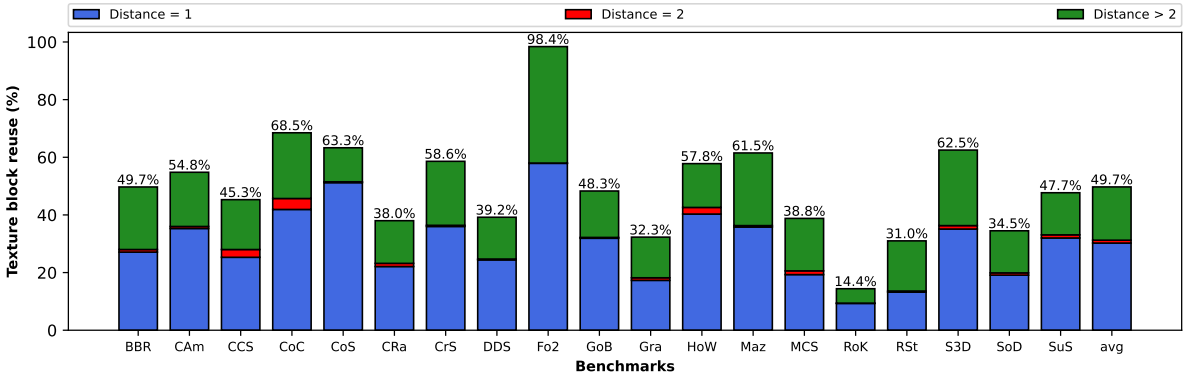


Figure 5.12: Fraction of shared texture blocks among tiles in a frame. On average, 50% of them are reused inside a frame.

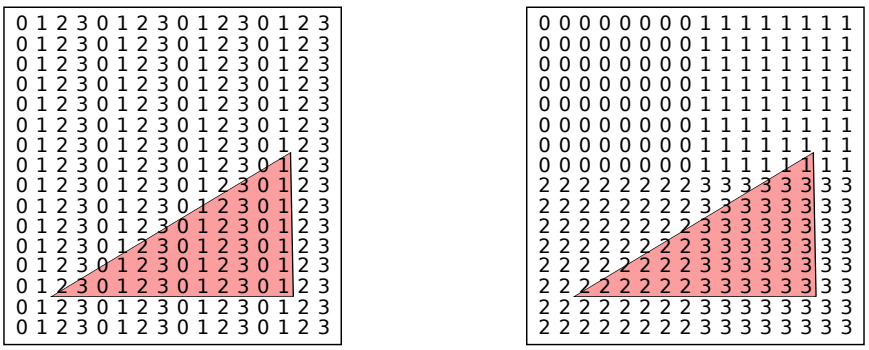
6

Inter-tile texture sharing

This chapter extends the aforementioned Parallel Tile Rendering (PTR) approach by benefiting from texture locality between tiles. First, it is introduced the main motivation behind and why it is challenging. Then, the changes performed in the architecture are presented, including a set of innovations that may enhance the synergy between processors and the texture caches, but also the workload granularity. Finally, the experimental results are reported.

6.1 Motivation

During PTR's evaluation, we realized that texture caches could be a common point between both Raster Units by exploiting texture locality between tiles. Therefore, we seek to reuse texture blocks, as we have seen there are some opportunities on that regard, but we also seek to maintain a balanced workload among the fragment processors, which is crucial. This latter aspect about workload balancing is not trivial as we will see below.



(a) Quad-level (fine-grained)

(b) Quadrant-level (coarse-grained)

Figure 6.1: Different assignments of quads to fragment processors within a tile.

Figure 6.1 illustrates an example where it is shown the **trade-off** between **locality** and **workload balancing**. These diagrams show different alternatives of how quads can be assigned to fragment processors within a tile. Each quad has a number assigned, ranging from 0 to 3, which represents the processor in charge of processing it. Bear in mind that, in the fragment stage, only a single tile is processed at a time. When quads are assigned following a quadrant distribution, as depicted in right-hand scheme, locality is favored. In contrast, workload distribution is harmed, causing performance degradation because at a given moment all the workload concentrates on a few cores, or even on a single one, while the rest remain idle. On the other hand, these workload distribution issues are solved by distributing at a more fine-grained granularity, as represented in the left-hand scheme. However, locality within the texture caches is lost. Hence, workload mapping is challenging, and we have analyzed different quad mapping distributions, which are detailed in section 6.3, to find a balance between workload distribution and locality. In addition, it is not just a problem of workload distribution among processors, but also how to map processors to the shared caches in order to exploit the maximum possible locality.

6.2 Baseline architecture

The idea behind this PTR extension is to benefit from the locality that may arise in the tile frontiers. As before, the baseline GPU implements a Tile-Based Rendering (TBR) architecture, following the modified architecture presented in chapter 5 but with some little changes. In particular, this new design shares one texture cache between two fragment processors, each from a different Raster Unit. As a result, the number of textures caches is halved whereas we keep the same cache size of 8KB for each texture cache. Note that this design allows to reduce the number of power-hungry cache ports to a half. The rest of components remains the same. Figure 6.2 illustrates this new design.

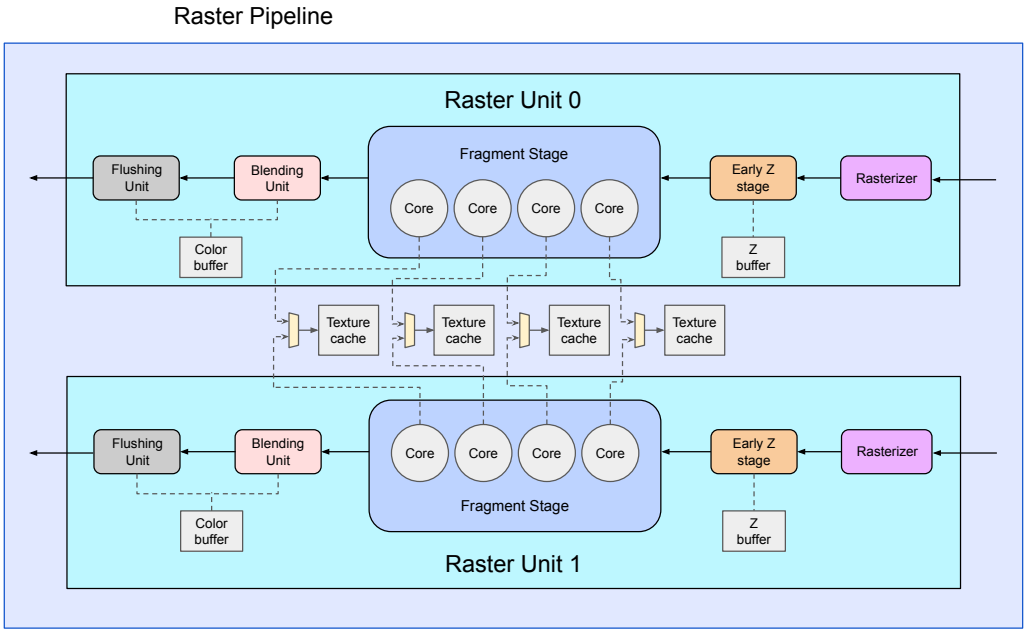
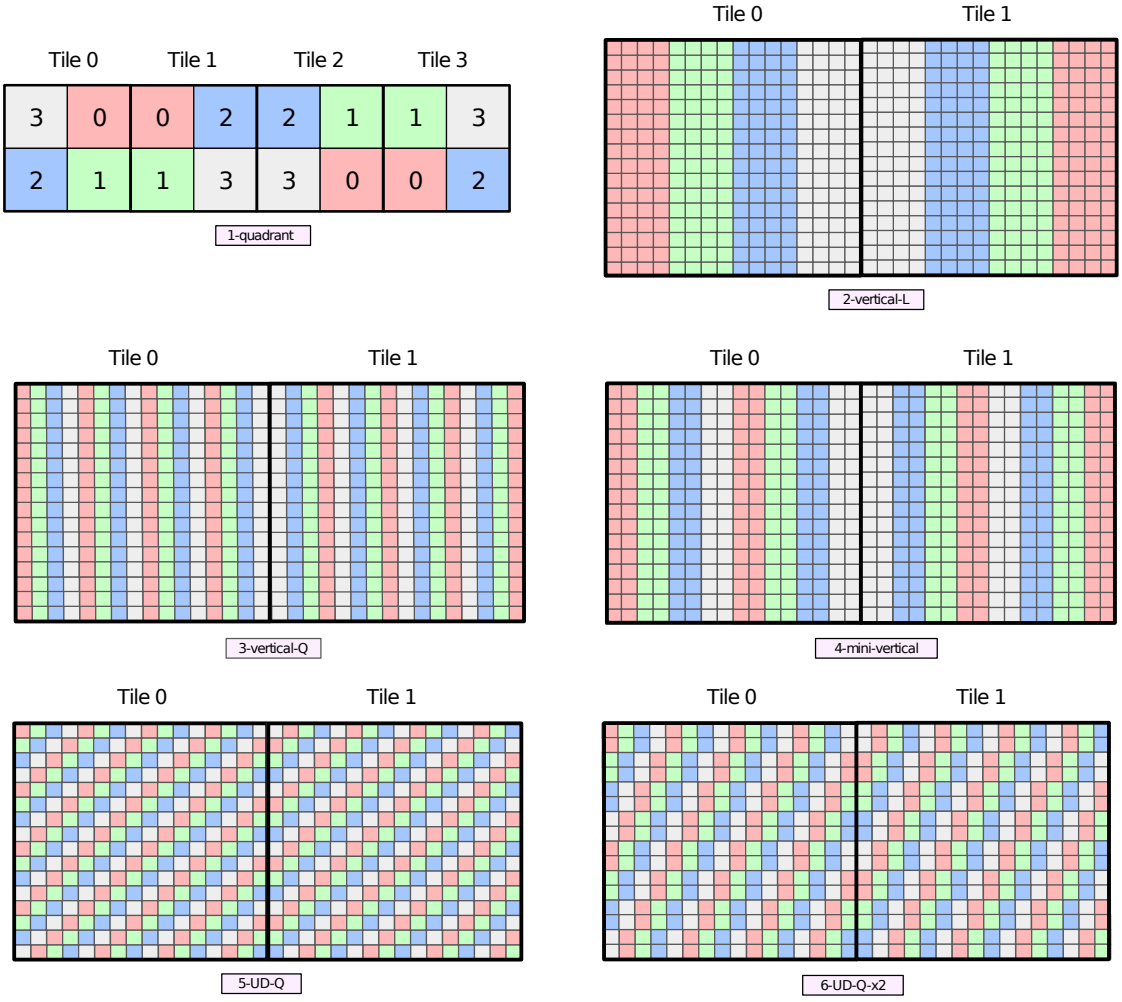


Figure 6.2: Raster Pipeline architecture employed for PTR's extension. Each processor shares the texture cache with a processor from another Raster Unit.

Since there are no longer private texture caches on each processor, but they are rather shared, special care must be taken when assigning quads to processors, but also on how to assign these processors to these shared caches. Therefore, we have studied combinations of workload assignment to processors, but also processors to caches. This analysis is described in the next section.

6.3 Quad mapping distributions

We have seen that enhancing texture locality is not trivial as many factors are involved, such as at which granularity distribute the quads, how quads are assigned to which processor, and which processors are sharing caches. Also, it is important to try that all the caches have the same opportunities for benefiting from the sharing in the borders. For it, in this section we present some of these combinations explored, identified by a reference name to ease the results explanation later. To better clarify the idea behind them, they are depicted in Figure 6.3, where each color represents the texture cache where each quad is assigned to: red, green, blue, and gray represent texture caches 0, 1, 2, and 3, respectively.



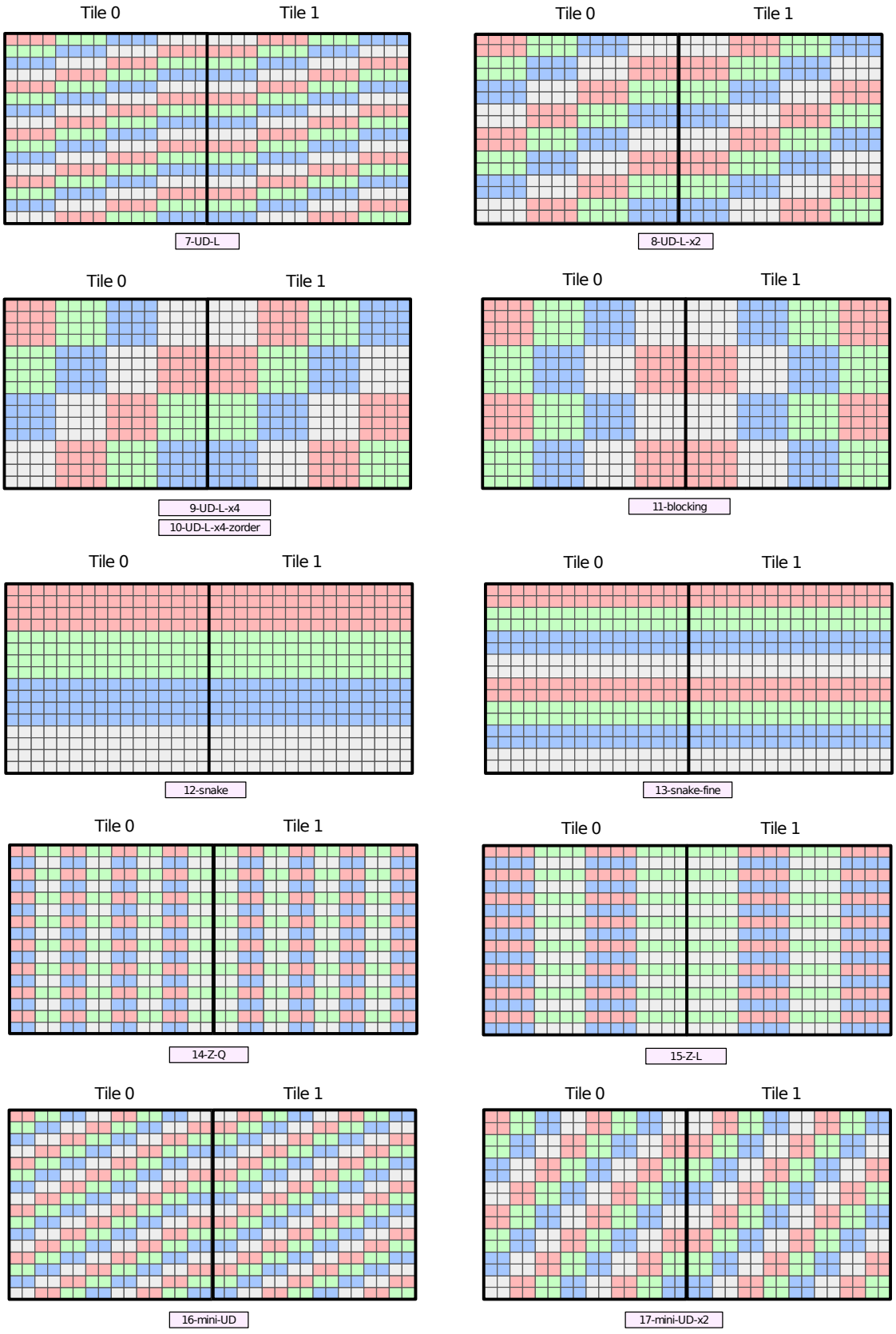


Figure 6.3: Evaluated quad mapping distributions.

We have a studied a plethora of combinations of work distribution but also cache mapping. For each of the combinations is shown the mapping of two consecutive tiles, except for *1-quadrant* that are shown the first four because this mapping rotates. Furthermore, all the combinations are processed in a scanline manner as in the baseline, except *10-UD-L-x4-zorder* which follows a Z-order pattern, and *12-snake* and *13-snake-fine* that follow a snake pattern. Notice that many of the combinations are derivatives of others.

For *1-quadrant* the workload within the tile is distributed, as the name says, in quadrants. To better exploit locality, it performs rotations in a way so all the caches may benefit from reuse. In addition, it is divided in quadrants which also allow to exploit locality within that tile. Then, *2-vertical*, *3-vertical-Q* and *4-mini-vertical* distribute the work vertically and differ in the granularity. The subsequent *5-UD-Q*, *6-UD-Q-x2*, *7-UD-L*, *8-UD-L-x2*, *9-UD-L-x4*, *10-UD-L-x4-zorder* schedule the workload in a round-robin fashion which enhances the inter-tile locality as all the caches share tile edges at each tile. In fact, *10-UD-L-x4-zorder* is the same scheduling as *9-UD-L-x4* but the tile scheduling is processed following a Z-order. *11-blocking* distributes the workload by blocking at a finer grain than by quadrants, maintaining locality within the tile but allowing inter-tile locality alternating caches. *12-snake* and *13-snake-fine* divide the work horizontally, in the sense that each cache has room for inter-tile sharing in each tile. *14-Z-Q* and *15-Z-L* distribute the work in the sense that each cache is assigned forming a Z shape, while at each tile two caches may benefit from tile-edge locality. Finally, *16-mini-UD* and *17-mini-UD-x2* also schedule the workload following a round-robin fashion.

6.4 Experimental results

This section presents the experimental results obtained when employing the different quad mapping distributions. Given that we have evaluated a plethora of benchmarks and various combinations, Figure 6.4 presents a plot that shows the average speedup obtained against the baseline PTR for all the games for each of the different configurations introduced.

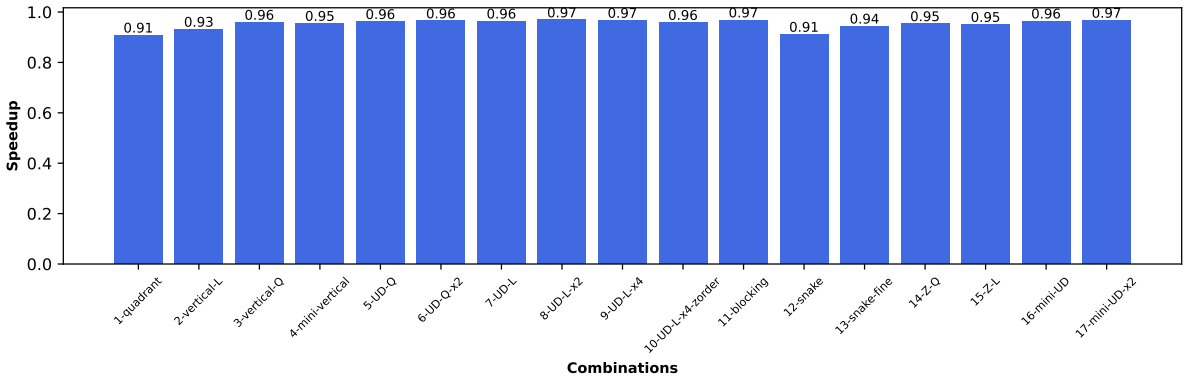
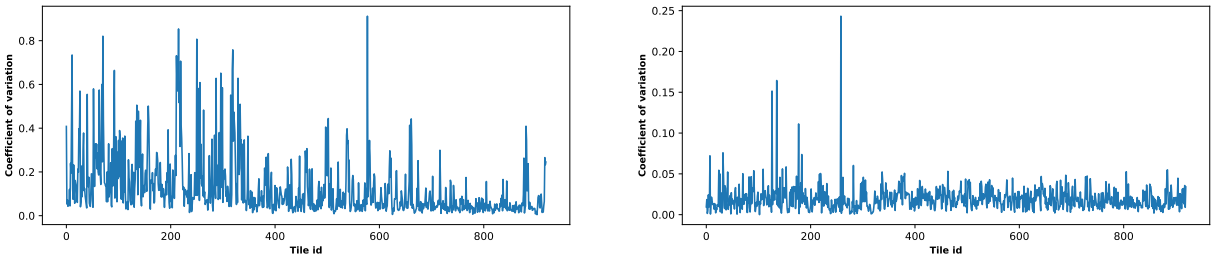


Figure 6.4: Average speedup for all the games for each configuration with respect to the baseline PTR, which owns 8 private texture caches.

As can be seen, for all of the combinations we are losing a bit of performance. Most of them are very close to 1, which would mean that no performance is gained respect having private caches. However, combinations *1-quadrant* and *12-snake* obtain the worst performance results. This is expected because, as we introduced in the beginning of this chapter, workload balancing

gets penalized when the workload distribution is performed at coarse grain. Thus, performance drops, even though they are the best approaches for benefiting from locality.



(a) *BBR* employing *1-quadrant* (b) *BBR* employing *5-UD-Q*

Figure 6.5: Example of processor workload balance using different combinations in a given frame.

For example, Figure 6.5 shows a comparison for the game *BBR* between the processor workload employing *1-quadrant* and *5-UD-Q*, which schedule workload in a coarse- and fine-grained manner respectively. This metric evaluates the standard deviation/mean, which has been measured by computing the average and deviation for each processor's number of cycles active. Thus, we can obtain the correlation between the workload within a tile and the speedup. If the coefficient is close to 0, it means that there is little variability in the data. On the other hand, if they tend to 1, it is a very disperse data. As for the combination with coarser grain we reach near 0.8, whereas with the fine-grained combination it reaches 0.2. Thus, there is imbalance in the coarser grain combination.

In addition, we have evaluated the energy savings compared to the baseline PTR, as shown in Figure 6.6. However, we can see that for most of the cases we obtain a bit less of energy savings. The reason for this is because of the static power consumption, as our execution time lasts a bit longer (as shown in the previous speedup plot). Even though, it also makes sense to have some little savings as we consume less static power because we have reduced the number of caches, while dynamic power remains unchanged as the number of accesses is the same.

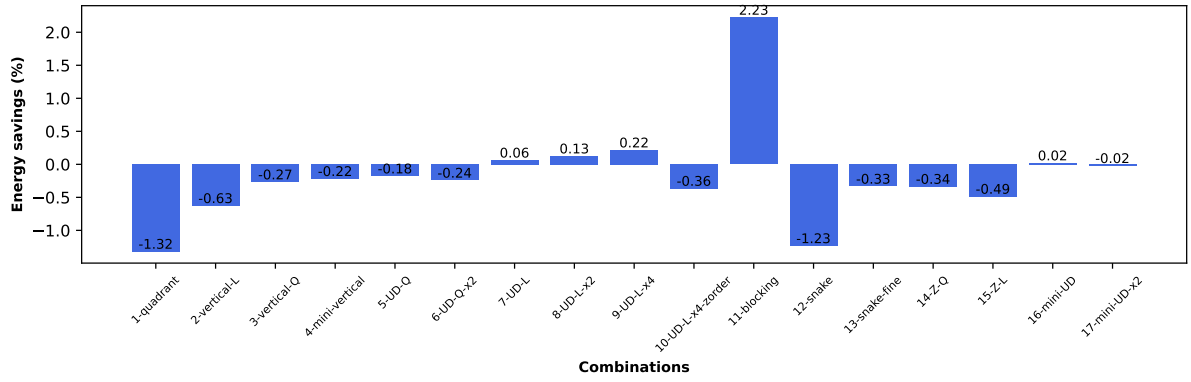


Figure 6.6: Average energy savings for all the games for each configuration respect to the baseline PTR, which owns 8 private texture caches.

11-blocking is one of the combinations that offers better performance, but also is the one with which we achieve better energy savings. Therefore, in Figure 6.7 we compare game by game this best combination with respect to the baseline PTR.

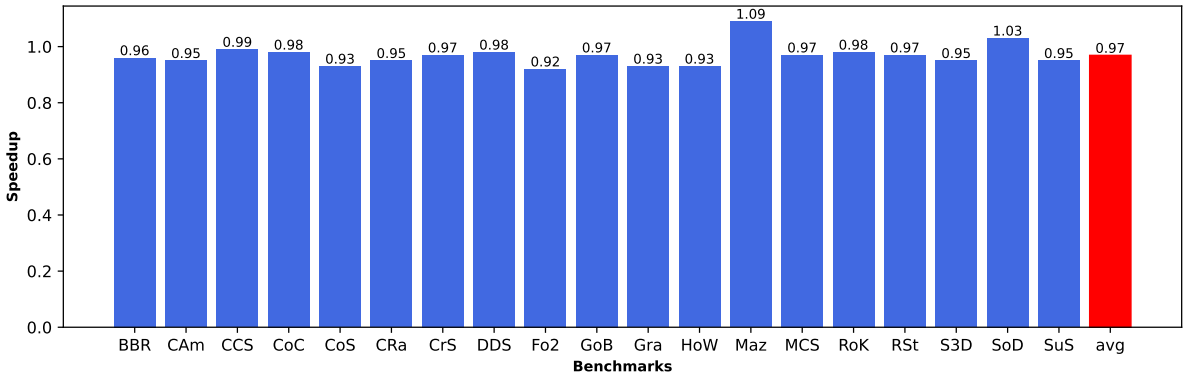


Figure 6.7: Speedup when employing *11-blocking* combination with respect to the baseline PTR, which owns 8 private caches.

All of the games achieve a similar speedup. Even though many of the combinations achieve a similar performance speedup, notice that the ones that provide the best are scheduled in an intermediate granularity, not as coarse as by quadrants neither as fine as by quads. In addition, *11-blocking* is reasonable that it provides one of the best results as it is a combination that seeks to exploit locality within the tile, while also inter-tile with not such a coarse distribution granularity.

We have seen that we have not obtained an improvement in performance as we were expecting, by exploiting inter-tile locality. Even though we did not obtain much benefit, we have reduced the aggregated size of the caches. According to McPAT, we have reduced a 4% of the area without a decrease in performance or an increase in energy. Moreover, Figure 6.8 shows the misses per 1000 instructions (MPKI) obtained when PTR employs private caches compared to our selected better combination, *11-blocking*. As it can be seen, we improve locality in caches, which is another important metric to take into account. However, at this point we cannot see the fully potential of sharing caches in performance terms. Thus, we leave for future research the exploration for this reason, even though we have identified one potential bottleneck: having one port, which we have not doubled for being more energy-efficient. Therefore, we see that we currently have a trade-off between lower cost and performance.

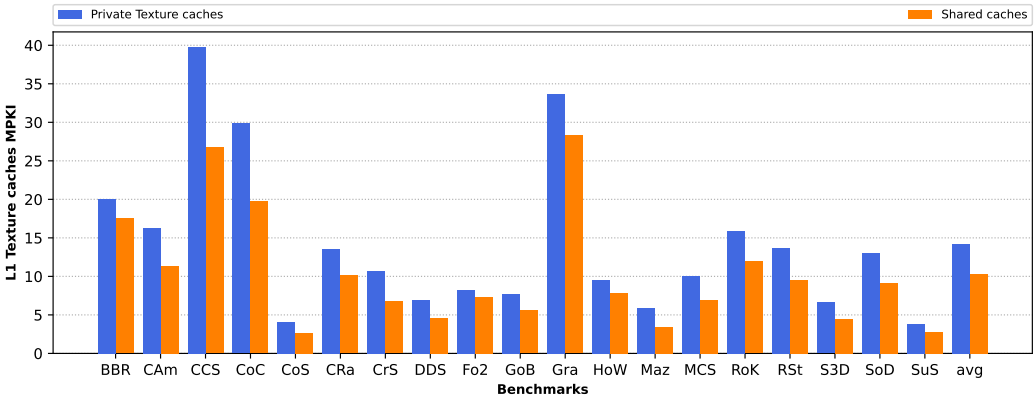


Figure 6.8: Misses per 1000 instructions (MPKI) for the L1 texture caches when employing PTR with private caches compared to shared caches with the *11-blocking* combination.

7

Related work

Although there are plenty of techniques in the literature that seek to boost GPUs performance in several ways, there are few focused on parallel graphics rendering methods. Arnau et al. propose Parallel Frame Rendering (PFR) [9], which splits the GPU into two clusters where two consecutive frames are rendered in parallel. They can do this since they realize that, because of the high degree of similarity between consecutive frames, significant bandwidth can be saved by processing multiple frames at the same time. In this way, they exploit the similarity of the textures that overlap this execution of consecutive frames, which we have seen that has a great impact on both DRAM accesses and the overall energy consumption. This is the most similar proposal to our work as it also outlines the textures as a point of intersection to boost performance. However, our approach pursues to accelerate the rendering process, by processing frame by frame, while exploiting inter-tile locality rather than inter-frame.

GPUpd [22] and CHOPIN [30] are two proposals that aim to scale the graphics rendering performance by distributing the workload among different GPUs. That is, they attempt to speed up the rendering process in the sense that each device has assigned, and processes, a disjoint region of a single frame. Despite speeding the rendering time, they target desktop-like architectures instead of mobile GPUs where energy efficiency is more critical. In addition, it is true that our proposal replicates some GPU components, but it is less aggressive than using multiple GPUs. Apart from that, multi-GPUs add extra costs for inter-GPU communication.

NVIDIA proposes the AFR [26] technique by also distributing the workload among multiple GPUs. Unlike the previous ones, this approach assigns different frames alternatively to multiple independent GPUs in order to increase performance and frame rate. Nevertheless, it also requires an additional cost for coordinating the devices.

On the other hand, Xie et al. [41] explored the use of processing-in-memory architectures to reduce the DRAM traffic motivated by the texture accesses. Corbalan et al. [13] propose a technique to maximize the local accesses within private texture caches by applying a NUCA organization. Hasselgren et al. [19] propose a novel rasterization architecture to improve the texture cache hit ratio targeting stereoscopic displays.

Other recent works have explored memory bandwidth reduction in TBR architectures in various methods. Early Visibility Resolution (EVR) [3] is an HSR technique that speculatively predicts the visibility of objects in a scene before the Raster Pipeline to avoid computation and texture accesses of fragments that will eventually be discarded. Rendering Elimination [4] is a technique that detects tiles that produce the same color across adjacent frames to avoid the computation of the tile in the next frame, thus, saving from redundant computation and texture accesses. Another work, TCOR [20], explores memory bandwidth reduction by targeting another major source of DRAM accesses in TBR architectures, which is the Parameter Buffer.

8

Conclusions and future work

The number of people playing mobile games has been increasing during the last years. Therefore, GPUs have become essential to satisfy the high computing demands that games require. However, mobile GPU design is challenging since this demands for more visually compelling graphics require more energy consumption, which is critical as these devices are battery-operated. Hence, power constraints are tight as it dictates their autonomy.

In this work we have introduced PTR, a novel mobile GPU architecture for rasterizing multiple tiles in parallel in two different Raster Units. It aims to reduce the time required for rasterization, which is the most time-consuming part of the graphics pipeline. In addition, it is also the most energy-consuming part. Experimental results show that PTR is able to achieve an average speedup of 83% for a wide range of different benchmarks, each of them with different characteristics. Moreover, PTR provides significant energy savings with an average decrease of 9.86%. In fact, we have shown that our approach is much more effective than having the same amount of computing resources within a single Raster Unit, leading to an increase in performance of 8.3% on average.

Nevertheless, this proposal has been possible as previously it has been developed a mobile GPU architecture baseline much more realistic and closely resembling modern GPUs, which allowed to explore new techniques. In addition, it is important to have a robust baseline as a starting point, as these new ideas may be conditioned by it. For it, we have improved the main memory model, updated hardware configuration parameters and most of the stages throughput. In addition, the Tile Fetcher has been re-designed together with a new structure that allows the acceleration for tile fetching, which is crucial for PTR, with an insignificant cost.

After employing PTR, we observed that there are still a noteworthy memory accesses caused by textures. As main memory accesses are known to be one of the highest contributors to energy consumption, we seek to reduce them by sharing texture caches among processors from different Raster Units. We found that texture caches can be a common point between the two Raster Units, as we obtained that on average 50% of the texture blocks are reused. Actually, most of this reuse is obtained in the next tile, thus, we proposed an extension to PTR to exploit inter-tile

texture locality. We have managed to reduce the cost of the hardware, the number of MSHRs, the number of ports and the total amount of cache size, with very little loss in performance and energy savings. However, as we still believe that more potential can be achieved, we will do future investigations on this area by more deeply exploring these shared configurations, but also the effects on the number of misses and the reduction of the number of ports.

Bibliography

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[2] Android SDK. https://developer.android.com/studio.

[3] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, and A. González. Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 635–646, 2019.

[4] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González. Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 623–634, 2019.

[5] M. Anglada Sánchez. *Exploiting frame coherence in real-time rendering for energy-efficient GPUs*. PhD thesis, UPC, Departament d'Arquitectura de Computadors, Jun 2020.

[6] Apple App Store. https://www.apple.com/app-store/.

[7] Apple GPU. https://developer.apple.com/documentation/metal/tailor_your_apps_for_apple_gpus_and_tile-based_deferred_rendering. Accessed: June 2022.

[8] ARM Mali. https://developer.arm.com/documentation/dui0555/b/introduction/the-mali-gpu-hardware/tile-based-rendering?lang=en. Accessed: June 2022.

[9] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU. In *Proceedings of the 22nd international Conference on Parallel architectures and compilation techniques*, pages 83–92. IEEE, 2013.

[10] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 37–46. ACM, 2013.

[11] 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*, pages 163–174. IEEE, 2009.

[12] C. 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*, pages 351–360. IEEE, 2010.

[13] D. Corbalan-Navarro, J. L. Aragon, J.-M. Parcerisa, and A. Gonzalez. DTM-NUCA: Dynamic Texture Mapping-NUCA for Energy-Efficient Graphics Rendering. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 144–151. IEEE, 2022.

[14] CUDA. https://developer.nvidia.com/cuda-zone.

[15] Gallium3D. https://www.freedesktop.org/wiki/Software/gallium.

[16] GAPID: Graphics API Debugger. https://gapid.dev/about/.

[17] D. Ginsburg, B. Purnomo, D. Shreiner, and A. Munshi. *OpenGL ES 3.0 Programming Guide*. Addison-Wesley Professional, 2014.

[18] Google Play. https://play.google.com/store.

[19] J. Hasselgren and T. Akenine-Möller. An Efficient Multi-View Rasterization Architecture. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, pages 61–72, 2006.

[20] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González. TCOR: A Tile Cache with Optimal Replacement. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 662–675, 2022.

[21] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho. MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide. *Georgia Institute of Technology*, 2012.

[22] Y. Kim, J.-E. Jo, H. Jang, M. Rhu, H. Kim, and J. Kim. GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 574–586, 2017.

[23] 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*, pages 469–480, 2009.

[24] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[25] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 231–241. IEEE, 2006.

[26] NVIDIA. SLI Best Practices, 2011. https://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf.

[27] OpenCL. https://www.khronos.org/opencl/.

[28] OpenGL ES. https://www.khronos.org/opengles/.

[29] Qualcomm Adreno. https://developer.qualcomm.com/sites/default/files/docs/adreno-gpu/developer-guide/gpu/overview.html. Accessed: June 2022.

[30] X. Ren and M. Lis. CHOPIN: Scalable Graphics Rendering in Multi-GPU Systems via Parallel Image Composition. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 709–722. IEEE, 2021.

[31] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[32] Statista. Most popular Google Play app categories as of 1st quarter 2022, by share of available apps. https://www.statista.com/statistics/279286/google-play-android-app-categories/, 2022. Accessed: May 2022.

[33] Statista. Number of smartphone subscriptions worldwide from 2016 to 2027. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2022. Accessed: May 2022.

[34] Statista. Video game market revenue worldwide in 2021, by segment. https://www.statista.com/statistics/292751/mobile-gaming-revenue-worldwide-device/, 2022. Accessed: May 2022.

[35] Techjury. 23+ Mobile Gaming Statistics for 2022 - Insights Into a $76B Games Market. https://techjury.net/blog/mobile-gaming-statistics/, 2022. Accessed: May 2022.

[36] TGSI. https://docs.mesa3d.org/gallium/tgsi.html.

[37] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.

[38] Z. Wang, L. Lu, and A. C. Bovik. Video Quality Assessment Based on Structural Distortion Measurement. *Signal Processing: Image Communication*, 19(2):121–132, 2004.

[39] M. Wloka. Batch, Batch, Batch: What Does It Really Mean? In *Presentation at Game Developers Conference*, 2003.

[40] Worldometer. https://www.worldometers.info/world-population/. Accessed: May 2022.

[41] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu. Processing-In-Memory Enabled Graphics Processors for 3D Rendering. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE, 2017.

[42] YouGov. The smartphone features that drive Brits' purchase choices. https://yougov.co.uk/topics/technology/articles-reports/2022/02/02/smartphone-features-drive-brits-purchase-choices, 2022. Accessed: May 2022.