Energy-Efficient Mobile GPU Systems

Jose Maria Arnau

Doctor of Philosophy

Department of Computer Architecture

Universitat Politecnica de Catalunya

2014

Abstract

The design of mobile GPUs is all about saving energy. Smartphones and tablets are battery-operated and thus any type of rendering needs to use as little energy as possible. Furthermore, smartphones do not include sophisticated cooling systems due to their small size, making heat dissipation a primary concern. Improving the energy-efficiency of mobile GPUs will be absolutely necessary to achieve the performance required to satisfy consumer expectations, while maintaining operating time per battery charge and keeping the GPU in its thermal limits.

The first step in optimizing energy consumption is to identify the sources of energy drain. Previous studies have demonstrated that the register file is one of the main sources of energy consumption in a GPU. As graphics workloads are highly data- and memory-parallel, GPUs rely on massive multithreading to hide the memory latency and keep the functional units busy. However, aggressive multithreading requires a huge register file to keep the registers of thousands of simultaneous threads. Such a big register file exceeds the power budget typically available for an embedded graphics processors and, hence, more energy-efficient memory latency tolerance techniques are necessary.

On the other hand, prior research showed that the off-chip accesses to system memory are one of the most expensive operations in terms of energy in a mobile GPU. Therefore, optimizing memory bandwidth usage is a primary concern in mobile GPU design. Many bandwidth saving techniques, such as texture compression or ARM's transaction elimination, have been proposed in both industry and academia.

The purpose of this thesis is to study the characteristics of mobile graphics processors and mobile workloads in order to propose different energy saving techniques specifically tailored for the low-power segment. Firstly, we focus on energy-efficient memory latency tolerance. We analyze several techniques such as multithreading and prefetching and conclude that they are effective but not energy-efficient. Next, we propose an architecture for the fragment processors of a mobile GPU that is based on the decoupled access/execute paradigm. The results obtained by using a cycle-accurate mobile GPU simulator and several commercial Android games show that the decoupled architecture combined with a small degree of multithreading provides the most energy efficient solution for hiding memory latency. More specifically, the decoupled access/execute-like design with just 4 SIMD threads/processor is able to achieve 97% of the performance

of a larger GPU with 16 SIMD threads/processor, while providing 20.5% energy savings on average.

Secondly, we focus on optimizing memory bandwidth in a mobile GPU. We analyze the bandwidth usage in a set of commercial Android games and find that most of the bandwidth is employed for fetching textures, and also that consecutive frames share most of the texture dataset as they tend to be very similar. However, the GPU cannot capture inter-frame texture re-use due to the big size of the texture dataset for one frame. Based on this analysis, we propose Parallel Frame Rendering (PFR), a technique that overlaps the processing of multiple frames in order to exploit inter-frame texture re-use and save bandwidth. By processing multiple frames in parallel textures are fetched once every two frames instead of being fetched in a frame basis as in conventional GPUs. PFR provides 23.8% memory bandwidth savings on average in our set of Android games, that result in 12% speedup and 20.1% energy savings.

Finally, we improve PFR by introducing a hardware memoization system on top. As consecutive frames tend to be very similar, not only most of the texture fetches are the same from frame to frame but also a significant percentage of the computations performed on the fragments. We analyze the redundancy in mobile games and find that more than 38% of the Fragment Program executions are redundant on average. We thus propose a task-level hardware-based memoization system that, when architected on top of PFR, provides 15% speedup and 12% energy savings on average.

Acknowledgements

First and foremost, I would like to thank my advisers Joan-Manuel Parcerisa and Polychronis Xekalakis, for teaching me everything I know about Computer Architecture. I feel very lucky to have had the opportunity to work with them over the past years. I am very grateful for their invaluable guidance and support.

I would like to make a special mention to Antonio Gonzalez, who offered me the opportunity to start a research career in the ARCO group. Thanks for your wise advice and support.

I wish to thank all the members of ARCO. I was lucky enough to be part of a large and enriching research group. I was fortunate to share an office with Marc Lupon and Enric Herrero, thanks for teaching me what being a PhD student means. Thanks to Aleksandar Brankovic for showing me what attending to a conference is about. Thanks to Enrique De Lucas for his feedback, I am very glad you decided to work in the exciting field of mobile graphics processors!

I would like to thank Christophe Dubach for having me as an intern in the University of Edinburgh. It was a great experience and a pleasure to work with him. Thanks to all the members of the CArD research group. Thanks to Chris Fensch for his feedback and his valuable help in finding opportunities to develop my career after my PhD.

On a more personal note, I would like to thank my family for their unconditional support, their fondness and their infinite patience. My parents Jose-Medin and Gloria, my brother Ignacio and my sister Maria Lidon have always encouraged me to carry on with my research work. I am grateful for their love, care and affection.

Declaration

I declare that this thesis was composed by myself, that the work contained therein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- "Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2012.

- "TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Conference on Supercomputing, 2013.

- "Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Conference on Parallel Architectures and Compilation Techniques, 2013.

- "Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2014.

(*Jose Maria Arnau*)

Contents

Chapter 1

Introduction

This chapter presents the background and motivation behind this work, a brief description of related work, and an overview of the main proposals and contributions of this thesis.

1.1 Current Trends in Mobile Graphics

Mobile phones have been adopted faster than any technology in history [137]. Smartphones and tablets are becoming primary devices of choice for a variety of activities such as reading email, playing games, taking pictures, interacting with social networks or browsing the web. In recent years, stationary desktop computers have been replaced in many scenarios by mobile devices as they begin to deliver a truly mobile computing experience. Powered by advances in mobile technology and System-on-a-Chip (SoC) design, current smartphones support a plethora of capabilities that cope with consumer expectations.

Undoubtedly, the capability to provide a real computing experience explains to a large extent the success of these mobile devices. Mobile graphics hardware/software improvements play an important role in this respect, as visually compelling graphics and high responsiveness are key to deliver a satisfactory user experience. Current mobile SoCs are able to decode 1080p videos at real time [163], capture and process pictures at resolutions bigger than 12 MegaPixels or render complex 3D graphics at high frame rates, achieving fill rates bigger than 4 GigaPixels/second [150, 161, 162]. On the other hand, smartphones support a plethora of mobile applications, for instance, more than 1.2 million Android applications are available in Google Play by May 2014 [24]. This combination of mobile hardware able to deal with multimedia content at real time frame rates and plentiful mobile software that exploits hardware capabilities makes smartphones/tablets very powerful and versatile devices.

In following sections we will discuss the mobile graphics hardware/software improvements that contributed to the expansion of the smartphones and tablets market. Later in this chapter we will discuss how supporting all these capabilities

affects the energy aspect of an embedded graphics processor, and we will introduce the main problems that have to be dealt with when designing a mobile GPU.

1.1.1 Mobile Graphics Hardware

In this section we will briefly review the improvements in mobile graphics hardware, focusing on the main components of the graphics system: the screen and the Graphics Processing Unit (GPU). In first place, mobile screens have evolved from small text-based displays to high resolution multi-touch displays. High-end smartphones begin to support Full-HD (1080x1920) resolution in 5-inches displays, whereas HD resolution (720x1280) is common in the mid and low-end. Hence, current screens deliver high quality visually compelling graphics, making the smartphone amenable for a broad range of applications. Furthermore, the touch screen is the main input device in a smartphone, being its response time critical for user experience.

On the other hand, mobile GPUs have experienced a significant evolution in recent years, becoming a key component of a SoC. Early mobile phones featured pure software rendering, performing all the graphics on the CPU since no specific graphics hardware was included. Furthermore, fixed-point arithmetic was employed in many cases due to the lack of floating-point units in embedded processors [157]. However, due to the aforementioned evolution in screen resolution the graphics system was required to provide huge fill rates, and the use of hardware acceleration became a hard requirement. The first mobile GPUs were fixed-function pipelines (non-programmable) that implemented the OpenGL ES 1.1 API [53], offering fill rates in the order of 100 MegaPixels/second [161]. Based on these primitive designs, mobile GPUs evolved towards more programmability and parallelism, in a similar way than desktop GPUs did, improving both flexibility and performance. Modern smartphones include programmable multicore GPUs that support the OpenGL ES 2.0/3.0 APIs [54], achieving fill rates in the order of GigaPixels/second and supporting scenes with millions of triangles [150, 161, 162].

It is worth mentioning at this point the main actors in the mobile GPU market. NVIDIA develops the Tegra SoC [26] that includes the Ultra Low-Power GeForce GPU [117], a version of the desktop GeForce optimized for low-power consumption. The ULP GeForce implements a classical Immediate-Mode Rendering architecture [113], and it is included in high-end smartphones and tablets such as the NVIDIA SHIELD [25], the Sony XPeria Tablet S [34] or the HTC One X [17].

Mali [9] is a series of GPUs produced by ARM. Unlike desktop GPUs, Mali implements a Tile-Based Rendering architecture [165], i. e. the screen is divided into small tiles and the scene is rendered tile by tile to maximize locality and minimize bandwidth usage [51]. A more detailed comparison between Immediate-Mode and Tile-Based architectures is provided in Chapter 2. Mali GPUs are employed in

popular SoCs such as the Samsung Exynos [164], included in smartphones like the Samsung Galaxy S3 [31].

Imagination Technologies produces the PowerVR [161] family of mobile GPUs. The PowerVR also implements a Tile-Based Rendering architecture, with special emphasis on addressing the overdraw [56] problem, i. e. unnecessarily computing and writing multiple colors for the same screen pixel due to multiple graphical objects drawn over the top of one another. PowerVR GPUs are included in popular mobile devices such as the Apple iPhone 5S [6], the iPad Air [5] or the Playstation Vita [27].

Adreno [162] is the solution developed by Qualcomm for embedded devices, included in the Snapdragon SoC [163]. Adreno GPUs implement a hybrid architecture able to switch at run-time between Immediate-Mode and Tile-Based by leveraging the FlexRender technology [11]. Google Nexus 5 [22], Samsung Galaxy S5 [32] or Sony XPeria Z2 [35] are examples of popular smartphones powered by Adreno GPUs.

Beyond the products developed by big companies, there exist two other families of mobile GPUs that are not so widespread, but they achieve comparable performance and power consumption. On one hand, Vivante Corporation introduced the Vega 3D architecture [42], implemented in the Vivante GC Series of mobile GPUs. Vivante also proposed the use of a dedicated accelerator for image composition [127], the Composition Processing Cores [41], offloading all the GUI related tasks from the GPU and achieving significant energy savings. Vivante GPUs are included in mobile devices such as the Samsung Galaxy Tab 3 [33] or the Huawei Ascend G615 [18]. On the other hand, Digital Media Professionals created the PICA-200 [160] mobile GPU, the graphics processor of the Nintendo 3DS [156].

1.1.2 Mobile Graphics Software

In this section we will briefly review the graphics software stack of mobile devices including the applications, Operating Systems and graphics APIs. Mobile phones are no longer restricted to making phone calls and sending text messages, but they support a plethora of applications such as web browsing, maps, gaming, social networks or picture and video editing. Mobile SDKs [3, 19] deliver plenty of functionality from networking to graphics including, for instance, libraries for multimedia content processing. Furthermore, they provide access to a great diversity of hardware available in mobile devices like the GPS, the GPU, the camera or the accelerometers. On the graphics side, developers have access to numerous widgets that allow the creation of very rich user interfaces. Furthermore, graphics hardware acceleration is available via OpenGL ES [94], allowing the creation of immersive 3D environments.

OpenGL for Embedded Systems (OpenGL ES) is a subset of the OpenGL API for rendering 2D and 3D computer graphics in low-power systems, developed and

maintained by the Khronos Group [20]. OpenGL ES is the dominant graphics API for accessing the GPU in handheld and embedded devices, as it is widely supported by all the manufacturers mentioned in section 1.1.1. Khronos has released five OpenGL ES specifications so far. The OpenGL ES 1.0 and 1.1 specifications implement a fixed-function pipeline. The OpenGL ES 2.0 specification implements a programmable pipeline, providing full support for vertex/fragment shaders. The OpenGL ES 3.0 specification includes new features to enable acceleration of advanced visual effects, such as occlusion queries [15], transform feedback [36] or multiple render targets [155]. Furthermore, bandwidth saving features like texture compression [140] and framebuffer invalidation [80] are included in the API. Finally, the OpenGL ES 3.1 specification adds the compute shaders [13] to support general purpose computation on the GPU (GPGPU). OpenGL ES specifications are derived from desktop OpenGL revisions, removing complex features and redundant function calls in order to keep the GPU drivers and the graphics hardware as simple as possible.

Regarding the Operating Systems, Android and iOS clearly dominate the smartphone market [92]. On the graphics department both systems provide extensive support for developing graphical user interfaces, in the form of a rich set of widgets and support for 2D/3D rendering via OpenGL ES. Furthermore, the GUI compositor is an essential component in both Operating Systems. Composition [127] is the process of combining multiple surfaces, usually generated by multiple graphical applications running simultaneously, together into a single surface that can be displayed on the screen. In order to achieve high responsiveness, composition is usually hardware accelerated in the GPU [127] or in a dedicated accelerator [41]. Android's GUI compositor, *SurfaceFlinger* [2], employs the OpenGL ES 1.1 API to boost composition. Therefore, the graphics hardware is used in any mobile application that displays content in the screen, since OpenGL ES is either explicitly called by the programmer or implicitly called when using widgets that are later rendered by the OS GUI compositor.

Many popular applications take benefit of hardware accelerated graphics, such as Google Maps, Google Street View or Facebook. The Web Browser also has to deal with tons of multimedia content, and WebGL [99] provides access to the GPU from JavaScript applications. Nevertheless, the applications that especially benefit from hardware acceleration are undoubtedly the numerous games available in Android and iOS. Casual 2D games like *Angry Birds*, with more than 1 billion downloads and 200 million active monthly users [4], or *Candy Crush*, with more than 500 million downloads and 124 million active daily users [16], are among the most popular applications for smartphones. Their game style, based in simple mechanics and short game levels, together with the effective use of the smartphone hardware (touch screen, accelerometers...) are some of the sources of their great acceptance.

On the other hand, smartphones and tables also support multiple desktop-like 3D games that exploit the capabilities of the graphics hardware. Popular franchises such as *Minecraft*, *Call of Duty* or *Need for Speed* are available for mobile devices. This type of games exhibit dynamic shadows [166], reflections [67],

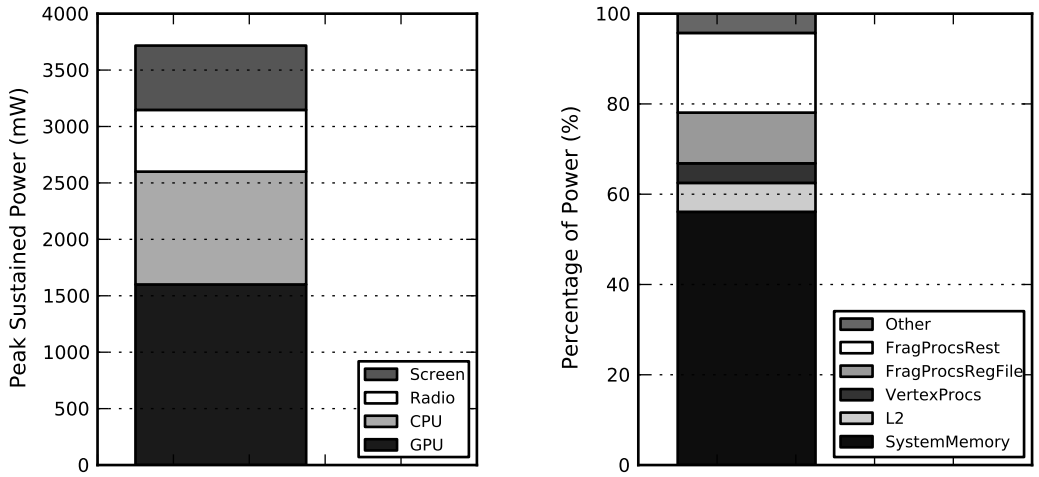
particle systems [130] and many other advanced graphical effects achieved by leveraging programmable shaders.

1.2 Problem Statement

All the mobile hardware/software improvements described in the previous sections are key to satisfy consumer expectations and deliver compelling user experience. However, supporting all these capabilities comes at the cost of a significant increase in energy consumption. Figure 1.1a shows the power breakdown in a Samsung Galaxy S3 when running the Egypt benchmark of the GLBenchmark suite. The GPU is the main battery consumer, requiring more than 1500 mW. Despite technology improvements, achieving the fill rates required to support high image resolutions in mobile devices with a tiny power budget is not an easy task. Mobile GPU design is all about reducing energy consumption because architects are concerned about battery life, but also because they are concerned about heat dissipation.

As smartphones and tablets are powered by batteries, any type of graphics rendering needs to use as little energy as possible. Operating time per battery charge is an important parameter of any mobile device, since users do not want to recharge their devices every few hours. The GPU energy demand increases significantly on each generation of smartphones to achieve the performance requirements, and it increases at a much faster pace than battery improvements [126]. The trend observed in recent years shows an increasing disparity between the energy required and the energy available in a smartphone [111], producing an energy gap that widens on each new generation of mobile devices, resulting in fairly short operating times per battery charge. As an example, Figure 1.1a shows that the Samsung Galaxy S3 requires 3.17 Watts to power the screen, the CPU and the GPU —we assume the radio is disabled while runnig graphical benchmarks. The Galaxy S3 features a 2100 mAh battery [31] at 3.7 V, so the energy available is: $\frac{2100mAh \times 3.7V}{1000} = 7.77Wh$. Converting the Wh to Joules we have: $7.77Wh \times 3600\frac{s}{h} = 27972J$. As the battery is drained at a ratio of 3.17 Joules/sec, the total operating time is 2 hours and 27 minutes. Hence, operating time per battery charge is a big concern since the energy available is consumed in just a few hours when running graphical workloads. Nevertheless, even if some breakthrough in battery technology would provide massive amounts of energy —something that is not likely to happen in the near future— mobile GPU energy consumption would still be a primary concern since architects would have to deal with a huge issue: heat dissipation.

Current smartphones and tablets feature big screens, but they are extremely thin since weight is an important parameter on mobile devices. Hence, smartphones cannot include sophisticated cooling systems such as fans or water cooling radiators. Even if batteries would suddenly become much more powerful, energy consumption could not be increased arbitrarily since the small size of the smartphone means it would need to dissipate large amounts of heat from a small area,



(a) Smartphone power breakdown **(b) GPU power breakdown**

Figure 1.1: The left graph shows the power breakdown measured in a Samsung Galaxy S3. The graph includes the four main battery consumers: screen, radio, CPU and GPU. The peak power of the screen, a 4.8 inches AMOLED display at HD resolution, and the peak power of the radio are provided in [129]. The peak power of the CPU, an ARM-based dual-core Krait at 1.5 GHz, and the GPU, a Qualcomm Adreno 225, are provided in [30]. The peak power was measured while running the Egypt benchmark of the GLBenchmark suite. On the other hand, the right graph shows the power breakdown in a mobile GPU with a Tile-Based Rendering architecture similar to the ARM Mali 400MP4. The graph shows the average power estimated by our power model based on McPAT, described in section 2.1.3, when running the twelve Android games presented in section 2.2.1. The power consumed by the Fragment Processors is split in two categories: the power of the main Register File, "FragProcsRegFile", and the power consumed by the rest of the pipeline, "FragProcsRest".

and thus become too hot to handle [44]. Mobile GPUs are thermally limited [120], since they could easily overheat their packages if the appropriate mechanisms to force them to slow down when they get too hot were not included. Therefore, saving energy is also an effective way of increasing performance on a mobile GPU. As an illustrative example of the importance of energy in mobile GPU design, the ARM Mali GPU requires 35% additional energy efficiency improvements every year to fit new performance requirements within SoC thermal limits [72].

Performance metrics employed for decades such as pixels per second (fill rate [152]) or the controversial triangles per second [121] have been displaced by other metrics that take into account energy. The most important metrics to optimize now are performance per watt [159] and/or nanojoules per pixel [45]. Energy-efficiency is also usually measured as the ratio between the speedup and the normalized energy [70].

The first step in optimizing the energy-efficiency is identifying the main sources of energy consumption in a mobile GPU. Previous studies have found two major sources: the big Register File (RF) required to sustain a huge number of simultaneous threads and the external accesses to off-chip system memory. The numbers

obtained in our mobile GPU simulation infrastructure, described in Chapter 2, support the results of previous studies as illustrated in Figure 1.1b. The RF of the Fragment Processors consumes 25% of GPU power and 11.2% of overall power consumption. On the other hand, the external accesses to off-chip system memory consume 56% of the power of the graphics subsystem.

Regarding the big size of the RF, GPUs are massively parallel programmable processors designed to exploit the concurrency inherent in graphics workloads. Unlike CPUs that generally target single-threaded performance, GPUs aim for high throughput by employing extreme multithreading [73]. Memory latency is hidden by maintaining a huge number of parallel threads, so in case of a cache miss the scheduler switches to another hardware thread to avoid pipeline stalls. Modern desktop GPUs can have thousands of hardware threads. For example, the NVIDIA Fermi architecture supports over 20000 thread contexts [68]. Just holding the registers of these threads requires substantial on-chip storage: a RF of around 2 MBytes that consumes more than 9 Watts [79]. Such a big RF is absolutely prohibitive for a mobile GPU, that usually has at its disposal a small power budget of around 1 Watt [120]. Indeed, mobile GPUs reduce the degree of parallelism to tenths or a few hundreds of threads [23]. The memory latency cannot be completely tolerated as the number of parallel threads is scaled down, introducing pipeline stalls that degrade performance. In short, although multithreading is a very effective technique for hiding the memory latency, we believe that more energy-efficient mechanisms are necessary to meet the energy requirements of mobile GPUs.

Besides the aforementioned issues with the RF, the general agreement in both industry and academia is that the main sources of energy drain in a mobile GPU are the expensive off-chip accesses to system memory [44, 47, 119, 120]. External memory accesses are often the operation that uses the most energy in a computer system [76]. An off-chip memory access consumes more than an order of magnitude more energy than an access to on-chip SRAM in low-power processes. This means that memory bandwidth have to be used with great care in a mobile GPU. Furthermore, this also means that optimizing memory bandwidth usage is an effective way of reducing energy consumption. Multiple bandwidth saving techniques have been proposed in recent years such as texture compression [119, 140] or framebuffer compression [8, 128], Section 1.3.2 provides a review of the state-of-the-art in memory bandwidth usage optimization for low-power GPUs. Most of these schemes are adaptations of techniques previously proposed for desktop GPUs, we believe that bigger bandwidth savings can be achieved by taking into account the specific characteristics of mobile graphics workloads.

In short, the demand for high quality rendering and the requirement of low energy consumption are contradictory. Developing more energy-efficient memory latency tolerance schemes and optimizing memory bandwidth usage are among the most promising ways to ameliorate this conflict. In following sections we will review the solutions proposed in recent years, and later in this chapter we will introduce our proposals to close the energy gap in embedded graphics processors.

1.3 State-of-the-art in GPU Energy Efficiency

Improving the energy-efficiency of mobile GPUs has attracted a lot of attention from the architecture community in recent years. As stated in prior section, reducing the energy consumption of the RF is one of the primary concerns regarding mobile GPU design, previous studies have focused on saving RF energy as described in Section 1.3.1. On the other hand, saving memory bandwidth has been proven to be an effective way of reducing energy consumption and, hence, multiple techniques targeting memory bandwidth usage optimization have been proposed as depicted in Section 1.3.2.

1.3.1 Memory Latency Tolerance Techniques

Graphics workloads have a large amount of inherent parallelism that can be easily exploited by a parallel machine. Texture memory accesses are common operations in graphics workloads and tend to be fine-grained and difficult to prefetch [79]. Graphics workloads have large, long-term (inter-frame) working sets that are not amenable to caching. Therefore, texture cache units focus on conserving bandwidth rather than reducing latency [73]. Because texture accesses are macroscopically unpredictable, and frequent, GPUs rely on massive multithreading to keep arithmetic units utilized. As a result, a huge RF is required to maintain the registers of all the simultaneous threads.

In this section we will first review previous work focused on optimizing the energy consumption of the GPU RF. Second, we will discuss other research work based on the use of prefetching to hide the memory latency, with the objective of reducing the multithreading degree requirements and, hence, the size and the energy consumption of the RF. Finally, we will review related work on Decoupled Access/Execute architectures.

Register File Optimizations

Gebhart et al. [79] propose the use of a Register File Cache to replace accesses to the large main RF with accesses to a smaller structure containing the immediate register working set of active threads. As extreme multithreading requires a complicated thread scheduler, the authors also propose a two-level thread scheduler that maintains a small set of active threads to hide ALU and local memory accesses, and a larger set of pending threads to hide main memory latency. Combined with the RF Cache, this two-level thread scheduler provides a further reduction in energy by limiting the allocation of temporary register cache resources to only the currently active subset of threads. The authors report significant energy savings in both graphics and GPGPU workloads. Note that this approach saves dynamic energy by replacing accesses to a bigger hardware structure by accesses to a smaller structure, but it does not save static energy as the original big RF is kept as in conventional GPUs.

Yu et al. [169] introduce a hybrid memory design that tightly integrates embedded DRAM into SRAM cells to reduce area and energy consumption of a multi-threaded RF. In this memory, each SRAM cell is augmented with multiple DRAM cells so that multiple bits can be stored in each cell. This approach saves both dynamic energy and leakage by keeping data for active threads in SRAM while placing data for pending threads in DRAM. However, such a RF requires explicit data movements between SRAM and DRAM in order to access the registers of pending threads. The thread scheduler has to be modified in order to minimize context switching impact.

Regarding proposals that specifically target mobile GPUs, Sohn et al. [167] propose a mechanism for clock gating an entire RF bank when it is not being accessed. Chu et al. [63] introduce the possibility of further reducing dynamic energy by dividing the register bank into multiple regions and controlling clock gating individually. In a later study, Chu et al [89] extended their work proposing an Adaptive Thread Scheduling mechanism combined with a low-power RF with hybrid power gating [103]. Besides dynamic energy savings, this RF exploits both *gated-*V_{dd} and *drowsy* techniques [75] to achieve long and short term leakage energy savings. Finally, a compiler-assisted demand-driven RF for mobile GPUs is presented in [90]. In this scheme the RF is shared on demand between concurrent threads, and multiple power gating modes are employed to avoid wasting static energy for unused registers.

Prefetching

Besides multithreading, prefetching data into the caches can also help tolerate memory latency. The largest concerns with any prefetching scheme are accuracy and timeliness. Regarding accuracy, if an application's memory stream is irregular or unpredictable cache pollution can occur and increase the number of cache accesses and, hence, energy consumption. Regarding timeliness, prefetch requests have to be issued at the appropriate time in order to be effective. Otherwise data could be prefetched too early and evicted before accessed from the application, or prefetched too late so the memory latency cannot be completely tolerated. Note that prefetching can only provide memory latency tolerance, whereas multithreading can hide both memory and functional units latency.

Several hardware-based data prefetching schemes for CPUs have been proposed in past decades. The next-line [58] and the stride [77] prefetchers are commonly implemented in desktop CPUs as they perform well for typical CPU workloads and they require simple hardware. More sophisticated prefetching schemes try to correlate recent miss addresses, such as the Markov prefetcher [100], whereas other prefetchers try to correlate recent strides, like the distance prefetcher [102]. On the other hand, Nesbit et al. [114] show that the history information employed by the previous prefetchers, that is typically stored in a table, can be kept more efficiently in a new hardware structure called the Global History Buffer.

Classical CPU prefetching results in higher performance, but worse energy in some cases due to unnecessary data being fetched on chip. With the advent of

General Purpose computing on GPUs (GPGPU), several authors have proposed to use prefetching in GPUs with the main objective of saving energy. Sethia et al. [135] proposed APOGEE, a prefetching mechanism able to dynamically detect and adapt to the memory access patterns found in scientific codes that are run on modern GPUs. APOGEE employs multi-thread miss address stream analysis, instead of considering threads in isolation, to improve accuracy and timeliness in highly threaded processors. The net effect is that fewer threads are required to tolerate memory latency and thus sustain performance. Hence, APOGEE saves energy by removing part of the thread contexts in the big and complex RF of a GPU, and including instead the smaller and simpler prefetch tables.

Lee et al. [106] also proposed the use of hardware prefetching for GPGPU applications. They show that conventional CPU prefetchers cannot be straightforwardly applied to GPGPU systems. Instead, they propose a stride based prefetcher tailored to many-core architectures. This prefetcher is based on inter-thread prefetching mechanisms and an adaptive throttling scheme to disable prefetching when it is useless, significantly improving accuracy and reducing energy waste.

The prefetchers previously mentioned target general purpose applications running on CPUs or GPGPU systems. That is why they do not take into account the specific issues with graphics workloads and the ubiquitous and unpredictable texture fetches. Igehy et al. [93] proposed a prefetching architecture for texture caches that is designed to accelerate the process of applying textures on triangles. This texture prefetcher is thoroughly discussed in Chapter 3.

Decoupled Access/Execute

A Decoupled Access/Execute architecture [138] divides the program into two independent instruction streams, one doing memory accesses and the other performing computations. By decoupling memory accesses from computations, access/execute architectures effectively prefetch data from memory much in advance from the time it is required, thus allowing cache miss latency to overlap with useful computations without causing stalls. While this can be viewed as a form of data prefetching, it has a substantial advantage over other prefetching schemes, because it relies on computed rather than predicted addresses, which translates into a higher accuracy and a lower energy waste.

Despite the high potential of access/execute architectures to tolerate a long memory latency at a moderate hardware cost, they have not been widely adopted by current commercial CPUs because their effectiveness is greatly degraded when the computation of an address has a data or control dependence on the execution stream (this occurs, for instance, in pointer chasing codes). In such circumstances, termed loss of decoupling events (LOD), the access stream is forced to stall in order to synchronize with the execution stream. LODs force the access stream to lose its timeliness (i.e. the prefetch distance), so that subsequent cache misses will cause the execution stream to stall as well. Unfortunately, for general purpose

CPUs the frequency of LODs is quite significant in many cases, resulting in fairly restricted performance gains.

Crago et al. [71, 70] propose the use of Decoupled Access/Execute architectures to hide the memory latency in highly threaded workloads. Their scheme is different from classical access/execute architectures as they run the multiple instruction streams in just one core by using multiple thread contexts, instead of having two separate processors for memory accesses and computations. Furthermore, they propose several strategies to mitigate the effect of LODs in multithreaded applications. Overall, they show that decoupled access/execute is an effective energy saving technique in many-core architectures as it requires simple hardware and, in addition, the number of thread contexts required to keep the functional units busy can be significantly reduced. It is worth noting that similar conclusions about the synergy of decoupling and multithreading were already suggested in [122].

Talla et al. [141, 142] describe the benefits of Decoupled Access/Execute for multimedia applications. They introduce the *MediaBreeze* architecture, a system that decouples the useful/true computations from the overhead/supporting instructions that are necessary to feed the SIMD execution units. The *MediaBrezze* architecture includes hardware for efficient address generation, looping and data reorganization. The proposal was evaluated on top of an out-of-order CPU with SIMD extensions and the authors reported significant performance improvements in media applications.

1.3.2 Bandwidth Saving Techniques

The rendering of 3D computer graphics requires the GPU to fetch big datasets from main memory on a frame basis. Retrieving these massive amounts of data is one of the main sources of energy consumption on a mobile GPU and thus optimizing memory bandwidth usage is a primary concern in embedded graphics processors. In this section we will review previous techniques to reduce the number of external memory accesses to the texture datasets and to the framebuffers.

Tile-Based Rendering

Desktop GPUs implement an Immediate-Mode Rendering (IMR) architecture [113]. In IMR, the geometry that describes an object is transformed in the vertex processors and immediately sent down the graphics pipeline for further pixel processing, updating the framebuffer before processing the next object. If a subsequent object is rendered on top of a previous one, the colors of the pixels are computed again and overwritten in the framebuffer. This issue is commonly known as the overdraw problem [56]: the colors of some pixels are computed and written multiple times into main memory due to multiple graphical objects being drawn over the top of one another, resulting in a waste of bandwidth. The average number of times a pixel is written in a single frame is often referred as the depth complexity [69].

Tile-Based Rendering (TBR) architectures [44] are designed to address the overdraw. In TBR the screen is split into rectangular blocks of pixels called tiles, and the frames are generated tile by tile. Tiles are small enough so the portion of the framebuffer corresponding to a tile can be stored in local on-chip memory, avoiding off-chip accesses to a large extent. Unlike IMR architectures, transformed 2D triangles are not immediately rendered in the framebuffer, instead they are stored into main memory in the Scene buffer [51]. Furthermore, 2D triangles are sorted into tiles, so for each tile a triangle overlaps a pointer to that triangle is stored. Once all the geometry for a given frame has been transformed and sorted, the rendering starts tile by tile. Note that pixels are written just once in main memory, as the tiles are generated first in the local on-chip memory and copied to the corresponding region of the framebuffer only when they are ready, i. e. when all the triangles for the given tile have been rendered. TBR provides significant bandwidth savings for generating the framebuffer, however, it requires the 2D triangles to be stored in memory and fetched back for rendering, so it trades geometry bandwidth for pixel bandwidth. Antochi et al [52] report significant bandwidth savings in low-power systems by using TBR. Although IMR dominates the desktop GPU segment, TBR is more popular in the mobile segment [9, 161, 162] as the geometry datasets of mobile workloads are smaller and, hence, the extra cost of storing/fetching 2D geometry is usually less than the bandwidth savings in the framebuffers.

Compression

Compression techniques can reduce both the memory footprint and the bandwidth requirements of graphical workloads. In first place, hardware texture compression was introduced by Knittel et al. [105] and Beers et al. [60]. The core idea is to use lossy compression on the textures, and store the compressed version in system memory. When accessing the texture during rendering, the compressed texture is transferred over the bus, and decompressed on-the-fly as needed, thus saving bandwidth.

Ström et al [140] propose *iPackman*, a texture compression scheme that targets mobile devices. *iPackman* has low-complexity, being amenable for hardware implementation on embedded graphics processors. Furthermore, it achieves high image fidelity and compression ratios of 4 bits per pixel (bpp). The OpenGL ES 3.0 API introduces support for texture compression by using the Ericsson Texture Compression [151] (ETC2/EAC version) algorithm, which is based on *iPackman*.

On the other hand, the major mobile GPU manufacturers provide their vendor-specific texture compression methods. ARM Mali supports the Adaptive Scalable Texture Compression (ASTC) [119], a lossy compression algorithm that provides a high degree of flexibility to trade image quality for bandwidth savings. ASTC achieves bit rates ranging from 8 bpp to less than 1 bpp in very fine steps. Qualcomm Adreno also supports texture compression by using the ATC [1] algorithm, whereas Imagination PowerVR leverages PVRTC [74] and NVIDIA Tegra provides support for DXT compression [109, 123].

Besides texture compression, framebuffer compression can also provide significant bandwidth savings. The framebuffer consists of a set of 2D buffers required for rendering such as the color buffer, that stores a RGBA color for each pixel in the screen, or the depth buffer, that keeps a depth value for each pixel in order to solve visibility. Framebuffer compression works in a similar way than texture compression does, storing the buffers compressed in main memory, transferring the data compressed over the bus, decompressing on-the-fly when reading and compressing on-the-fly when writing to the framebuffer. Note that the color buffer has to be uncompressed in order to be displayed on the screen, in case of using framebuffer compression the display controller has to include the corresponding color buffer decompression hardware.

Regarding IMR GPU architectures, Rasmusson et al. [128] propose both lossless and lossy compression algorithms for the color buffer. Hasselgren et al. [86] introduce a depth buffer compression algorithm to save bandwidth in the depth test. Regarding TBR architectures, ARM Frame Buffer Compression (AFBC) introduces the compression of the tiles before being transferred from local on-chip memory to system memory, achieving compression ratios of 50% with a lossless algorithm. As for the depth buffer, Tile-Based architectures do not usually require to save and restore the depth values of the tiles. In OpenGL ES 2.0 the driver can infer in some cases that the contents of the depth buffer does not need to be preserved from frame to frame, whereas OpenGL ES 3.0 introduces the *glInvalidateFramebuffer* [80] function call to explicitly indicate to the driver that the depth buffer is not required to be saved/restored. Hence, TBR architectures can completely avoid all the transfers to the external depth buffer in main memory.

Transaction Elimination

Transaction Elimination [57, 120] (TE) is a bandwidth saving technique for Tile-Based GPU architectures introduced by ARM and implemented in the Mali GPUs. With TE, the graphics hardware compares the current framebuffer with the previously rendered frame and performs a partial update only to the particular parts of it that have been modified, thus significantly reducing the amount of data transferred per frame to external memory. The comparison is done on a per tile basis, using a Cyclic Redundancy Check signature to determine if the tile has been modified. Tiles with the same signature are identical and eliminating them has no impact in the resulting image quality. TE is highly effective for popular graphical applications such as User Interfaces or casual games, since they usually feature static 2D backgrounds and there are not many changes from frame to frame. However, it is also effective for other graphical applications like 3D games or video.

Multi-View and Multi-Frame Rendering

Stereoscopic and 3D displays require the generation of multiple views or images for every single frame. Hasselgren et al. [87] propose a novel rasterization architecture

that renders each triangle to multiple views simultaneously, with the objective of maximizing the hit rate in the texture caches and thus saving memory bandwidth. When determining which view to rasterize next, the architecture employs an efficiency metric that estimates which view is expected to get the most hits in the texture cache. As the different views of the same frame are very similar, a better utilization of the texture caches is achieved by generating the multiple images in parallel instead of rendering the views sequentially. Instead of processing all the triangles for a given view and then proceed to the next one, the proposed architecture iterates all the views for each triangle to maximize texture locality.

NVIDIA Scalable Link Interface (SLI) [118] is a multi-GPU configuration that offers increased rendering performance by dividing the workload across multiple GPUs. The most common SLI configuration is known as Alternate Frame Rendering (AFR). Under AFR the driver divides the workload by alternating GPUs every frame. For example, on a system with two SLI-enabled GPUs, odd frames would be rendered by GPU 1 whereas even frames would be rendered by GPU 2. The main target of AFR is to increase graphics performance by duplicating hardware resources and increasing memory bandwidth, so power consumption is also significantly increased. However, it is possible to save bandwidth and energy in a multi-GPU system if the multiple graphics processors share a single memory address space as suggested in [112], in that case part of the dataset employed for rendering can be shared by the different GPUs.

1.3.3 Other Related Works

Mochocki et al [111] propose the use of Dynamic Voltage and Frequency Scaling (DVFS) in mobile 3D graphics pipelines. Want et al. [147, 148] describe different power gating strategies for graphics processors. Chu et al [61, 62, 88] propose dynamic precision selection in the GPU shader cores, switching at runtime between fixed-point arithmetic —faster and more energy-efficient— and floating-point arithmetic —better graphics quality.

As described in previous sections, the overdraw [56] is one of the main issues in graphics processors as it results in a waste of bandwidth and energy. Hence, different hardware-based techniques that try to address the overdraw in mobile GPUs have been proposed. The ULP GeForce GPU in the NVIDIA Tegra SoC features Early-Z rejection [115] to preemptly discard fragments that are known to be occluded. When using Early-Z rejection, the depth test is performed before the execution of the fragment shader to avoid computing the colors of non-visible fragments. This scheme achieves maximum efficiency when the triangles are sent to the GPU ordered from front to back. On the contrary, worst case happens when triangles are sent ordered from back to front, as younger fragments always overlap older fragments and, hence, they always pass the depth test and proceed to the fragment shader.

Tile-Based architectures are designed to minimize the impact of the overdraw since pixels are written just once into off-chip system memory due to the use of

local on-chip color buffers. However, the colors of some pixels are still computed and written multiple times in the local on-chip memory. Ideally, the fragment shader should be executed just once for each screen pixel on every frame to completely remove the overdraw [1]. Some mobile GPUs include techniques to remove re-executions of the fragment shader for the same screen pixel, in order to avoid spending time and energy in shading fragments that are not going to contribute to the final image. For example, ARM Mali GPUs implement Forward Pixel Kill (FPK) [21]. In an FPK-enabled GPU, the threads that shade fragments are not irrevocably committed to complete once they are launched. On the contrary, in-flight threads can be terminated at any time if the hardware detects that a later thread will write opaque data to the same pixel location. On the other hand, Imagination Technologies PowerVR GPUs include hardware for Hidden Surface Removal (HSR) [28]. With HSR all the geometry for a tile is processed and the visibility is completely solved before shading any fragment. Once the closest-to-the-observer fragment for each pixel in the tile has been found the fragment shader computations take place and, hence, each pixel is shaded just once.

1.4 Thesis Overview and Contributions

The goal of this thesis is to propose novel and effective techniques that address the issues in mobile GPU design, with the objective of improving the energy-efficiency of embedded graphics processors while keeping complexity low. Our main contributions are a decoupled access/execute-like architecture for the Fragment Processors, a bandwidth saving technique called Parallel Frame Rendering and a hardware-based memoization scheme that avoids redundant computations and memory accesses. All these proposals apply to a conventional mobile GPU architecture, and can be implemented on top of both Immediate-Mode and Tile-Based GPUs. The following sections outline the problems we are trying to solve, describe the approach we take to solve the problem and provide a comparison with related work, highlighting the novel contributions of this thesis.

1.4.1 Mobile GPU Simulation Infrastructure

We have developed our custom mobile GPU simulation infrastructure, that we call TEAPOT. To the best of our knowledge, TEAPOT is the first simulator tailored towards the mobile segment, as none of the previous GPU simulators provide support for running Android graphical applications that employ the OpenGL ES API. TEAPOT consists on a set of tools for evaluating the performance, energy consumption and image quality of mobile graphics processors. Its main target is to drive the evaluation of energy saving techniques for mobile GPUs.

Regarding its features, TEAPOT provides full-system simulation of Android applications. Furthermore, it includes a GPU timing simulator able to model

[1] In case transparent or translucent objects are included in the scene the optimum number of executions of the fragment shader can be greater than one per pixel

both Tile-Based and Immediate-Mode rendering architectures, a power model for mobile GPUs, and automatic image quality assessment by using several metrics. TEAPOT is extensively described in Chapter 2. This infrastructure was presented in a paper published in the proceedings of the 27th International Conference on Supercomputing:

- "TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Conference on Supercomputing, 2013.

The development of tools for evaluating the GPU has attracted the attention of the architectural community the last few years. Recent simulators, such as GPGPUSim [59] or Barra [65], model General Purpose GPU (GPGPU) architectures. These tools support CUDA [116] or OpenCL [172], but they do not support graphics APIs such as OpenGL [97]. GPGPUSim includes a power model, GPUWattch [170], which is also based on McPAT as in TEAPOT. Both power models are similar, but GPUWattch focuses on GPGPU specific features whereas TEAPOT models more specialized graphics hardware. For instance, GPGPUSim models FP units that can be combined to execute 1 double-precision (DP) or 2 single-precision (SP) operations, but TEAPOT relies on SP units since DP is common in scientific workloads but not in games. On the contrary, TEAPOT models specialized Texture Sampling units since texture fetching instructions are frequent in graphical workloads.

ATTILA [113] provides an OpenGL framework for collecting traces of desktop games and a cycle-accurate GPU simulator. A Direct3D [171] driver is also included in the last versions. Although ATTILA provides full support for desktop games, it cannot run applications for smartphones. Furthermore, its GPU simulator models a desktop-like Immediate-Mode Renderer, whereas Tile-Based Rendering is much more popular in smartphones. Finally, ATTILA does not include a power model. Qsilver [136] can also collect traces from desktop OpenGL games and it models a desktop-like NVIDIA GPU, including a power model. GRAAL [101] also provides OpenGL support and a power model for GPUs. Furthermore, it models a low-power Tile-Based Rendering architecture. However, OpenGL ES support is not available in any of these simulators so they cannot run mobile applications for smartphones and tablets. Unlike the aforementioned tools, TEAPOT provides image quality metrics for automatic image quality assessment. In addition, TEAPOT supports full-system GPU simulation, being able to profile multiple applications accessing the GPU concurrently.

1.4.2 The Decoupled Access/Execute Fragment Processor

Extreme multithreading is the solution employed by desktop GPUs to hide the memory latency, as both graphical and GPGPU workloads exhibit a high degree of parallelism. However, aggressive multithreading requires a huge Register

File (RF) to keep the registers of all the parallel threads. Bound by severe energy constraints, mobile GPUs cannot accommodate such a big RF in their tiny power budgets. Hence, memory latency cannot be completely tolerated just by using multithreading due to the smaller number of parallel threads employed in embedded graphics processors.

In first place we evaluate the effects of aggressive multithreading in a mobile GPU, analyzing its impact on performance and energy consumption. As the results show that multithreading is effective but not energy-efficient, due to the big size of the RF, we try to reduce the number of hardware threads by combining multithreading with other memory latency tolerance techniques such as prefetching. We evaluate several state-of-the-art CPU, GPU and GPGPU hardware prefetchers on a mobile GPU running graphics workloads. The results obtained in our cycle-accurate timing simulator indicate that these prefetchers obtain non-negligible performance improvements, but they perform far from ideal as texture accesses are highly unpredictable.

In second place, we propose a decoupled access/execute-like architecture for the fragment processors of a mobile GPU. For GPU fragment programs the memory access patterns are typically free of the dependences that cause Loss of Decoupling (LOD) events. This makes the access/execute paradigm a perfect fit for the requirements of a low-power high-performance GPU: with few extra hardware requirements it can reduce drastically the number of cache miss stalls. In our scheme, all the necessary data for processing the fragments is prefetched into the caches while the fragments are waiting to be dispatched to the GPU cores. By the time a fragment is issued all the data required to process the fragment is hopefully available in the caches, significantly improving the hit rates in the shader cores.

In third place, we improve our base system by introducing remote L1 cache accesses to exploit the high degree of data sharing among fragment processors. When the system detects that a cache line that is going to be prefetched in the L1 cache of a fragment processor has been recently prefetched in another L1 cache, the memory request is redirected to this cache instead of accessing the bigger L2 cache. This optimization saves bandwidth to the L2 cache and saves additional energy by replacing accesses to the bigger L2 cache by accesses to the smaller L1 caches. The end design is able to achieve similar performance to a heavily-threaded GPU by consuming only a fraction of its energy. More specifically, we evaluate our scheme on top of a state-of-the-art mobile GPU by using several commercial Android games, and show that the end design is able to achieve 97% of the performance of a massively multithreaded GPU, while providing 20.5% energy savings. This work has been published in the proceedings of the 39th International Symposium on Computer Architecture (ISCA):

- "Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2012.

Unlike most of the prefetching schemes described in section 1.3.1, our work is focused on graphics workloads instead of general purpose applications. Furthermore, our scheme employs computed addresses rather than predicted addresses to significantly improve accuracy, as in graphics workloads the memory access patterns are typically free of dependences. The work that is closest to ours is the prefetching architecture for texture caches proposed by Igehy et al. [93]. However, our work is different in several ways. First, our system is built on top of a modern mobile GPU pipeline instead of a fixed-function pipeline, so our scheme supports multicore GPUs, programmable shaders and multiple texture fetches per fragment. Second, we take advantage of the Early-Z Test to only prefetch data for visible fragments, increasing the energy efficiency. Third, our proposal allows for remote L1 requests to exploit the high degree of data sharing among fragment processors, providing significant energy benefits.

The efforts to reduce register file power on a GPU include the register file cache and the two-level warp scheduler proposed by Gebhart et al. [79], and the hybrid SRAM-DRAM memory design presented by Yu et al. [169]. We show that there is no real necessity for high degree of multithreading and as such for large register files, as multithreading can be combined with other techniques to hide the memory latency in a more energy-efficient way. On the other hand, the abovementioned research is focused on GPGPU workloads, whereas our study targets graphical applications. General purpose codes employ complex addressing modes that can cause loss of decoupling events, reducing the effectiveness of decoupled access/execute architectures. However we believe that mobile phones are not the ideal platform for scientific applications, so our research is focused on more typical workloads for smartphones, such as games.

Recently, research in the field of mobile GPUs has emerged. Akenine-Moller and Strom [47] propose a rasterization architecture for mobile devices that employs a novel texture compression system to reduce memory bandwidth usage by 53%. Our work is also focused on reducing bandwidth, but we achieve bandwidth savings by exploiting inter-core data sharing.

Tarjan et al. [143] propose the sharing tracker, a simplified directory employed to capture inter-core reusage among the private non-coherent caches of a GPU. Our decoupled system is also able to exploit data sharing, but at a smaller energy cost by using a much smaller hardware structure, the prefetch queue. On the other hand, several tiled-cache approaches have been proposed. Reactive NUCA [85] introduces fixed-center clusters and rotational interleaving on a distributed shared L2 cache, these novel mechanisms provide high aggregate capacity while exploiting fast nearest-neighbour communication. NoC-aware cache design [43] introduces a first-touch data placement policy, a migration policy that moves each block to its most frequent sharer and a replacement policy that is biased towards retaining shared blocks and replacing private ones. DAPSCO [78] consists on a distance-aware cache organization that minimizes the average distance travelled by cache requests. In our system the L2 cache is centralized instead of distributed, since the number of cores in a mobile GPU is much smaller than what is assumed in a many-core system due to power constraints. Furthermore, the tiled-cached

systems use the hardware coherence mechanisms (directory) to detect data sharing among the first level caches, whereas we employ the prefetch queue to detect data reusage among non-coherent L1 caches at a much smaller energy budget (hardware coherent caches are considered too expensive for GPUs [143]).

Crago et al. [71] present OUTRIDER, a decoupled system for throughput-oriented processors. OUTRIDER is similar to our proposal since it also employs a decoupled access/execute architecture to hide the memory latency with fewer threads. However, our system reduces hardware complexity, does not require compiler assistance to generate the instruction streams and it is able to detect inter-core data sharing. On the other hand, OUTRIDER offers better tolerance to LODs by using multiple memory access streams, so it is best suited for scientific applications whereas our system is best suited for graphical workloads.

1.4.3 Parallel Frame Rendering

A large fraction of a mobile GPU energy consumption can be attributed to the external off-chip memory accesses to system RAM. As noted by [87, 119], most of these accesses fetch textures. Our numbers support this claim, as we found that 62% of the memory accesses can be directly attributed to texture data on average in our set of commercial Android workloads. Focusing on the texture dataset of consecutive frames, we realized that there exists a large degree of reuse across frames. Hence, the same textures are fetched frame after frame, but the GPU cannot exploit these frame-to-frame reuses due to the huge size of the texture dataset.

Firstly, we present Parallel Frame Rendering (PFR), a technique to improve texture locality on a mobile GPU. Under PFR, two consecutive frames are processed in parallel by devoting half of the graphics hardware resources to render each frame. By using this organization, each texture is read from memory once and used for rendering two successive frames. Therefore, textures are fetched from memory just once every two frames instead of being fetched on a frame basis as in conventional GPUs. The results show that PFR achieves 23.8% bandwidth savings on average for a set of Android games, providing 14% speedup and 20.5% energy savings.

Secondly, we evaluate the effects of PFR in responsiveness. A valid concern for PFR is that since it devotes half of the resources to render each frame, the time required to render a frame is longer. This is an unfortunate side-effect as it ultimately leads to a less responsive system from the end-user's perspective. Nevertheless, we show that the increase in input lag is acceptable for many mobile applications. Furthermore, we propose a new version of PFR reactive to user inputs that is able to maintain the same levels of responsiveness than conventional mobile GPUs. The reactive versions of PFR monitor user inputs so two frames are processed in parallel just when the user is not interacting with the device, which accounts for most of the time, according to our user interaction analysis. As soon as the user provides inputs the system reverts to conventional rendering,

employing all the GPU resources to render just one frame in order to achieve high responsiveness. In addition, we also explore the possibility of rendering more than 2 frames in parallel. The reactive version of PFR achieves high responsiveness while it provides 23.8% bandwidth savings, achieving 12% speedup and 20.1% energy savings on average. This work has been published in the proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT):

- "Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Conference on Parallel Architectures and Compilation Techniques, 2013.

Although it might appear that PFR is similar in concept to the NVIDIA AFR paradigm [118] mentioned in Section 1.3.2, our approach clearly differs from the AFR, both in its goals and its methods: while AFR aims at increasing performance and frame rate by coordinating multiple independent GPUs with separate address spaces to maximize bandwidth, PFR pursues reducing energy by exploiting inter-frame temporal locality through splitting the computation resources of a single GPU with a shared last level cache and synchronizing memory accesses of consecutive frames. To the best of our knowledge, PFR is the first multi-frame rendering technique that targets energy savings instead of performance improvements.

On the other hand, Hasselgren et al. [87] propose a bandwidth saving technique for systems with stereoscopic displays. Since multiple views of the same frame have to be generated for a 3D display, the authors propose to compute the different views concurrently to improve texture cache hit ratio by using a novel rasterization architecture. Although the objective is the same than in our technique, i. e. saving memory bandwidth, the implementation is significantly different since PFR splits the graphics hardware in multiple clusters, but each cluster is still a conventional mobile GPU that implements a Tile-Based Rendering architecture. Furthermore, PFR is not limited to 3D displays and it achieves bandwidth savings on smartphones with 2D screens. Note that both techniques can be combined in stereoscopic systems: PFR can exploit inter-frame texture similarity by processing consecutive frames in parallel, whereas each GPU cluster can employ the technique described in [87] to maximize intra-frame texture locality.

Our approach is orthogonal to a wide range of memory bandwidth saving techniques for mobile GPUs, such as texture compression [140], texture caching [83], color buffer compression [128], Tile-Based Rendering [52] or depth buffer compression [86].

1.4.4 Eliminating Redundant Fragment Shader Executions

Graphical applications for mobile devices tend to exhibit a large degree of scene replication across frames. Our numbers show that, on average, more than 40% of the fragments computed in a given frame were previously computed in the frame before it. Recent work attempts to exploit this inter-frame locality in order to save memory bandwidth and improve overall energy efficiency, such as the ARM's Transaction Elimination [57]. Removing all these redundant computations and memory accesses would provide significant performance and energy improvements.

In first place, we provide a detailed analysis on the fragment redundancy that exists across frames. We analyze the locality and the complexity of redundant fragments, and conclude that a significant percentage of the redundant fragment shader executions could be avoided by using a simple memoization scheme.

In second place, we propose a task-level hardware-based memoization scheme that, when architected on top of Parallel Frame Rendering, is able to improve energy-efficiency by 12%, while providing 15% speedup on average. Our memoization scheme keeps a hardware structure that computes the signature of all the inputs to a task and caches the value of the corresponding fragments. Subsequent computations form the signature and check against the signatures of the memoized fragments. Hits in the hardware structure result in the removal of all the relevant fragment computations and the corresponding memory accesses.

In third place, we analyze the effects on image quality of using imperfect hash functions to compute the signatures of the fragments, which inevitably leads to collisions in the hardware structure, i. e. fragments that are incorrectly identified as redundant and get a wrong color. We compare the images generated by a conventional GPU with the images generated by a GPU that includes our memoization system, employing image quality metrics that are based on the human visual system and manually confirming the results. The numbers show that even small signatures of just 32 bits are able to achieve high image fidelity. This work has been published in the proceedings of the 41st International Symposium on Computer Architecture (ISCA):

- "Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2014.

Memoization has been subject of research for several decades. A typical problem faced by conventional memoization is to guarantee referential transparency, i. e. that the same set of input values is always going to produce the same output result. The main difficulty here arises from the fact that the memoized instruction blocks must not depend on global memory data that is subject to changes between executions, and they do not produce side-effects. Since it is difficult to track these global changes at runtime, existing hardware memoization approaches apply to

single instructions or blocks of instructions [139, 64, 66, 82, 49, 50, 91, 145], whereas function level memoization has only been exploited in software based solutions [132, 84, 110, 168]. Our memoization scheme is different as it is function level and hardware based, since our work is focused on GPUs and graphical applications where it is easier to track changes to global data and no mutable state or side-effects exist.

Exploiting the high degree of redundancy in graphical applications has attracted the attention of the architectural community in recent years. ARMs Transaction Elimination [57] performs a per-tile comparison of consecutive frames to avoid transferring redundant tiles to main memory. Our work is also focused on exploiting redundancy in GPUs, but besides removing redundant memory accesses our system also avoids redundant computations.

Liktor et al. [108] propose the use of a software managed cache, implemented in GPU local memory, to memoize redundant colors for stochastic rasterization [46]. Our system is different since it is hardware managed and, hence, completely transparent to the programmer. Furthermore, our scheme does not require stochastic rasterization, being able to exploit inter-frame redundancy on top of the conventional rasterization algorithm implemented on current mobile GPUs.

Tseng et al. [144] propose the data-triggered thread programming model to remove redundant computations in general purspose environments. Our work is focused on graphical applications and our hardware memoization system is automatically able to remove a significant percentage of the redundancy without programmer intervention.

Alvarez et al. [173] propose the use of fuzzy memoization at the instruction level for multimedia applications to improve performance and power consumption at the cost of small precision losses in computation. In [50] they further extended tolerant reuse to regions of instructions. Their technique requires compiler support and ISA extensions to identify region boundaries and to statically select regions with redundancy potential. Our approach differs from theirs because we focus on mobile GPUs instead of CPUs, we add PFR to improve reuse distance, and we do not require ISA extensions or compiler support because we consider all fragment shaders without exception, and do not require boundaries since the whole shader is skipped or reexecuted.

1.5 Thesis Structure

The rest of this document is organized as follows:

Chapter 2 describes the evaluation methodology. First, we present our mobile GPU simulation infrastructure and the assumed baseline GPU architecture. Second, we describe the set of Android workloads employed to evaluate our proposals. Finally, we provide the main architectural parameters that were used for the experiments.

Chapter 3 provides an evaluation of several memory latency tolerance techniques, such as multithreading and prefetching, in the context of a mobile graphics processor. Furthermore, it introduces our decoupled access/execute-like architecture for the fragment processors of an embedded GPU.

Chapter 4 describes our bandwidth saving technique called Parallel Frame Rendering (PFR). We first provide an analysis of the memory bandwidth usage on several commercial Android games. Next we introduce PFR, a technique that renders multiple frames in parallel to maximize texture locality and minimize memory bandwidth usage. We also provide a discussion of the impact of PFR in input lag and responsiveness. A study of user interaction in mobile graphics workloads is included in this chapter, the result of this study is the reactive version of PFR that is able to adapt to the amount of input provided by the user.

Chapter 5 proposes the use of hardware memoization to remove redundant computations and memory accesses on a mobile GPU. Firstly, we provide an analysis of the fragment redundancy in commercial Android games. Secondly, we describe our task-level hardware-based memoization system for mobile GPUs. This chapter includes a complete evaluation of our memoization system, including performance and energy estimations, but also a complete analysis of the effects on image quality.

Chapter 6 outlines some of the future steps and open research areas and finally summarizes the main conclusions of this thesis.

Chapter 2

Experimental Environment

This chapter presents the simulation infrastructure developed for estimating performance and energy consumption of an embedded graphics processor, a brief description of the Android applications selected as our set of workloads, and an overview of the main architectural parameters used for the experiments.

2.1 Simulation Infrastructure

GPU simulators typically employed in the architecture community [59, 65, 113, 136, 146] are mainly focused on accurately modeling desktop-like GPUs. Unfortunately, these simulators are not tailored towards the mobile segment. In fact, to the best of our knowledge, none of them support the OpenGL ES API [94], which means that they cannot run smartphone graphical applications. Furthermore, most of them do not provide a power model. This is an important impediment, as all of the mobile segment devices are battery operated and thus energy consumption is a key design aspect as described in section 1.2.

As none of the prior GPU simulators are able to either run mobile software or model low-power graphics hardware, we have developed our own mobile GPU simulation infrastructure that we call TEAPOT. TEAPOT is a toolset for evaluating the performance, energy consumption and image quality of mobile graphics processors. TEAPOT provides full-system GPU simulation of Android applications. It includes a cycle-accurate GPU simulator, a power model for mobile GPUs, based on McPAT [107], and automatic image quality assessment using the models presented in [149].

In terms of the GPU microarchitecture, TEAPOT models both Tile-Based Rendering (TBR) [165] and Immediate-Mode Rendering (IMR) [113]. While IMR is more popular for desktop GPUs, for GPUs targeting energy efficiency TBR seems to be the design of choice [9, 161, 162]. Prior GPU simulators only focused on the IMR approach, which is significantly different from TBR both in terms of performance and power as it will be shown in section 2.1.3.

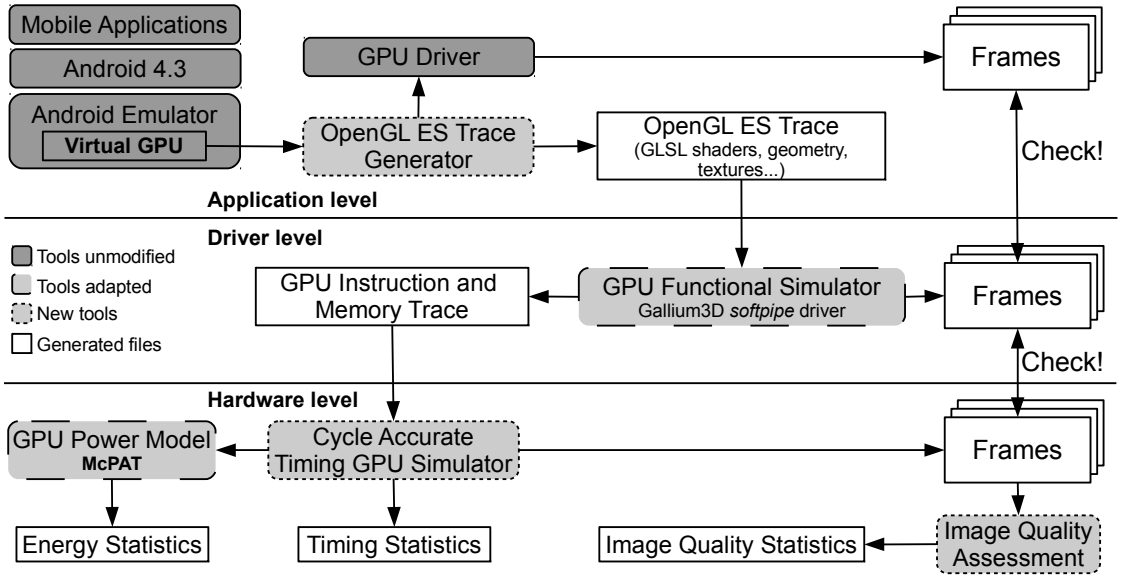


Figure 2.1: Overview of the mobile GPU simulation infrastructure.

Figure 2.1 illustrates the overall infrastructure. TEAPOT leverages existing tools (e.g, McPAT [107] or Gallium3D [12]) that have been coupled with our GPU models and adapted for the low-power segment. The goal of TEAPOT is to drive the evaluation of new energy saving techniques for low-power graphics.

TEAPOT is able to run and profile unmodified commercial Android applications. We have adapted the Gallium3D driver in order to profile OpenGL ES commands and collect a complete GPU instruction and memory trace. This trace is then fed to our cycle-accurate GPU simulator with which we estimate the power and performance for the given application.

Finally, TEAPOT provides several image quality metrics, based on per-pixel errors or based on the human visual perception system. These metrics are useful when evaluating aggressive energy saving techniques that trade image quality for energy such as, for instance, lossy texture compression. The next sections illustrate the workflow of the simulation infrastructure and provide more insights into the components of TEAPOT.

2.1.1 Application Level

TEAPOT uses the Android Emulator available in the Android SDK [3] for running mobile applications on a desktop computer. The Android Emulator is based on QEMU [29] and supports GPU virtualization [40]. Hardware acceleration is thus available for the guest Operating System running inside the emulator, Android in that case. When enabling GPU virtualization, the OpenGL ES commands issued by the applications are redirected to the GPU driver of the desktop machine. Since OpenGL ES is hardware accelerated, state-of-the-art 3D games run at real-time frame rates on the emulator. This also simplifies the GPU profiling since the OpenGL ES commands are not processed inside the emulator but

they are redirected to the desktop GPU driver and, hence, they are completely visible to the host system.

The *OpenGL ES trace generator* component captures the OpenGL ES command stream generated by the Android applications. It saves the GPU commands in a trace file, and redirects them to the appropriate GPU driver. It consists on a library interposed between the Android Emulator and the desktop GPU driver that contains replacements for all the functions in the OpenGL ES 1.1/2.0 APIs, so when a graphical application calls some OpenGL ES function, the one in the trace generator is selected instead of the one in the GPU driver. Hence the *OpenGL ES trace generator* is completely transparent for the emulator and it just causes a small frame rate decrease due to the time necessary for logging and redirecting the GPU commands. Note that the frames generated by the GPU of the desktop system are saved in order to verify that the GPU functional and timing simulators generate the same output than the real hardware.

The OpenGL ES trace file contains the GLSL vertex and fragment shaders, i. e. the high level code executed by the GPU, and all the data employed for rendering including textures, geometry and state information. Therefore, it contains all the necessary data for reproducing the OpenGL ES command stream. Furthermore, the thread identifier is stored in the trace together with each OpenGL ES command, so the cycle-accurate simulator can report per-thread statistics. Note that the *OpenGL ES trace generator* captures commands not from just one application but from all the Android graphical applications concurrently using the GPU, including the *SurfaceFlinger* component of the Android OS that performs image composition (see section 1.1.2), so TEAPOT provides full-system GPU simulation.

2.1.2 Driver Level

The Gallium3D [12] driver provides GPU functional emulation in TEAPOT. Gallium3D is an infrastructure for developing GPU drivers. It includes several front-ends for different graphics APIs, including OpenGL ES, and multiple back-ends for distinct GPU architectures. A modified version of Gallium3D is employed for executing the commands stored in the OpenGL ES trace file. A software-based back-end is selected for rendering since it can be easily instrumented in order to get a complete GPU instruction and memory trace.

Gallium3D translates the high level GLSL shaders into an intermediate assembly code, TGSI [37]. The software back-end of Gallium3D, named *softpipe*, consists of an emulator for this TGSI assembly language. By instrumenting the TGSI emulator all the instructions executed by the GPU and all the memory addresses referenced are collected and stored in a GPU trace file. Note that a software renderer is different from real hardware, so special care is taken in order to trace just the instructions that would be executed in the real GPU, i. e. the instructions in the vertex and fragment programs, and the memory requests that would be issued in a real GPU, i. e. memory accesses to read/write geometry,

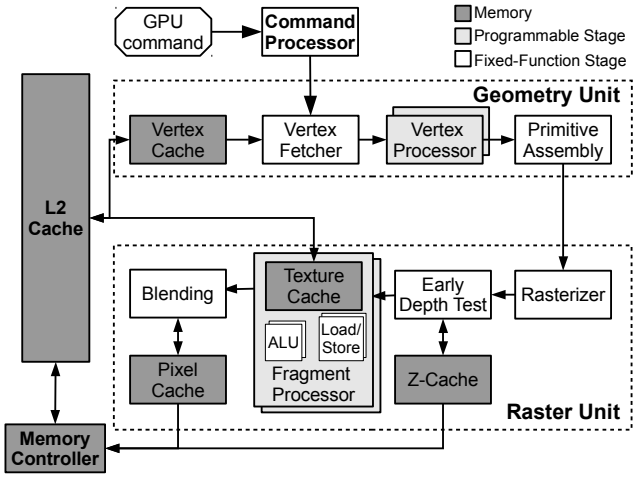


Figure 2.2: Immediate Mode Rendering architecture modeled by the GPU timing simulator.

textures and the framebuffers. On the other hand, off-screen rendering is employed so the frames are written into an image file instead of being displayed on the screen. These frames are compared with the ones generated by the graphics hardware to verify the correctness of the functional emulator.

2.1.3 Hardware Level

TEAPOT includes a cycle-accurate simulator for estimating the GPU execution time taking as input the instruction and memory traces generated by Gallium3D. GPU energy estimations are also provided by using a modified version of McPAT. Finally, the image quality assessment module estimates the visual quality of the output frames.

Baseline Architecture

Our GPU simulator is able to model low-power GPUs based on both IMR and TBR. The IMR architecture implemented in TEAPOT is illustrated in Figure 2.2. This pipeline is based on the architecture of the ULP GeForce GPU included in the NVIDIA Tegra SoC [115] and it works as follows. First, the *Command Processor* receives a command from the CPU and it sets the appropriate control signals so the input vertex stream is processed through the pipeline. Next, the *Geometry Unit* converts the input world-space vertices into a set of transformed and shaded 2D screen-space triangles. Finally, the *Raster Unit* computes the color of the pixels overlapped by the triangles. This architecture is called "Immediate Mode" because once a triangle has been transformed it is immediately sent down the graphics pipeline for further pixel processing. The main problem of this architecture is *overdraw*: the colors of some pixels are written multiple times into memory, wasting bandwidth, because of pixels from younger triangles replacing pixels from previously processed triangles.

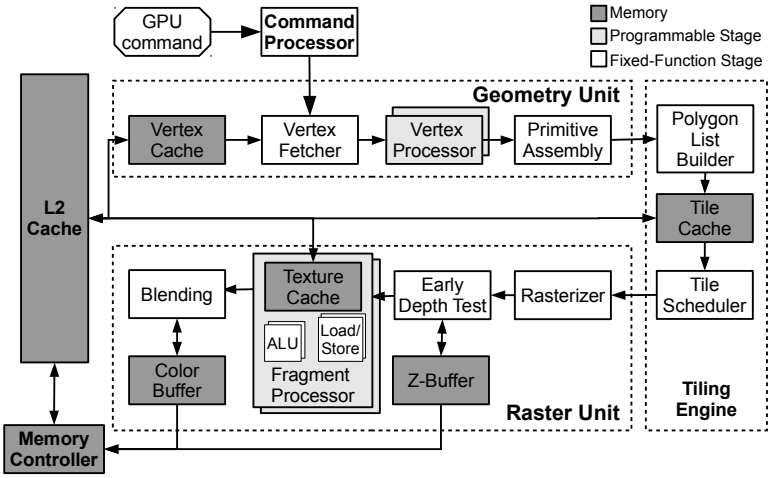


Figure 2.3: Tile-Based Rendering architecture modeled by the GPU timing simulator.

On the other hand, the timing simulator can be configured to model a TBR architecture as the one illustrated in Figure 2.3, that is based on the pipeline of the ARM Mali 400MP [55]. In TBR the screen space is divided into tiles, where a tile is a rectangular block of pixels. Transformed triangles are not immediately sent to the *Raster Unit*. Instead, the *Tiling Engine* stores the triangles in memory and sorts them into tiles, so that for each tile that a triangle overlaps a pointer to that triangle is stored. Once all the geometry for the frame has been fetched, transformed and sorted, the rendering starts. Just one tile is processed at a time in each *Raster Unit*, so all the pixels for the tile can be stored in local on-chip memory and they are transferred just once to the off-chip Color Buffer in system memory when the tile is ready, avoiding the overdraw. However, transformed triangles have to be stored in memory and fetched back for rendering, so there is a trade-off between memory traffic for geometry and memory traffic for pixels. TBR rendering is becoming increasingly popular in the mobile segment. The ARM Mali [9], the Qualcomm Adreno [162] or the Imagination Technologies PowerVR [161] are examples of mobile GPUs that employ some form of TBR.

The IMR pipeline consists of two main components: the *Geometry Unit* and the *Raster Unit*. The TBR pipeline also includes these two units but it adds another component in between: the *Tiling Engine*. These pipelines work as follows. The *Geometry Unit* is the front-end of the GPU in both architectures, IMR and TBR, and it is the first component that is triggered when a new rendering command is received. In the *Geometry Unit* the *Vertex Fetcher* reads first the input vertices from memory. Next, the vertices are transformed and shaded in the *Vertex Processors* by applying the programmed by the user Vertex Program. Finally, the transformed vertices are assembled into triangles in the *Primitive Assembly* stage, where non-visible triangles are culled and partially visible triangles are clipped. In case of an IMR architecture, triangles are immediately sent to the *Raster Unit* for further pixel processing. On the contrary, in case of a TBR architecture triangles are sorted into tiles in the *Tiling Engine*.

The *Tiling Engine* provides support to TBR. First, the *Polygon List Builder* saves the 2D transformed triangles to the Scene Buffer [52] in system memory. Furthermore, triangles are sorted into tiles so for each tile a triangle overlaps, a pointer to that triangle is stored in memory. After all the geometry for the frame has been sorted, each tile contains a list of 2D triangles that overlap that particular tile. Tiles are then processed in sequence by the *Tile Scheduler*: all the triangles overlapping a tile are fetched from memory and dispatched to a *Raster Unit*.

The *Raster Unit* is the back-end of the GPU in both architectures, IMR and TBR, and its main task is to compute the colors of the pixels overlapped by the input 2D triangles. Initially, the *Rasterizer* converts the 2D triangles into fragments, where a fragment is all the state that is necessary to compute the color of a pixel, such as the screen coordinates, texture coordinates or even user defined application specific attributes. The per-fragment attributes are computed by interpolation of the per-vertex attributes of the three vertices of the triangle. Next, fragments are tested for visibility in the *Early Depth Test* stage. Fragments that are occluded by other fragments previously processed are discarded in this stage, since they are not going to contribute to the final image. On the contrary, visible fragments proceed to the *Fragment Processors*, where they are shaded based on the programmed by the user Fragment Program. Finally, the *Blending* unit merges the colors of the fragments with the colors already computed in the Color Buffer by applying the corresponding blending equation [10].

Regarding the memory hierarchy, the IMR architecture employs several first level caches for storing vertices (*Vertex cache*), depth values (*Z-cache*), colors (*Pixel cache*) and textures (*Texture cache*). These caches are connected through a bus to a second level shared cache. On the other hand, the TBR architecture includes caches for storing vertices (*Vertex cache*), triangles (*Tile cache*) and textures (*Texture cache*), whereas it employs local on-chip memories for storing part of the Color Buffer and the Z-Buffer. The Color Buffer is the region of main memory that stores the colors of the screen pixels, whereas the Z-Buffer stores a depth value for each pixel, which is used for resolving visibility. In TBR, local on-chip memories are employed for storing the portion of the Color Buffer and the Z-Buffer corresponding to a tile. The local on-chip buffers are directly transferred to system memory when all the triangles for the tile have been rendered, they are not cached in the L2.

Fragment Processor Microarchitecture

Two of the GPU pipeline stages, the *Vertex Processors* and the *Fragment Processors*, are fully-programmable in-order vector processors whereas the rest are fixed-function stages. Figure 2.4 shows the microarchitecture of the *Fragment Processor*. It consists of a fairly simple 4-stage in-order pipeline. The ISA implemented by the *Fragment Processor* is the TGSI (Tungsten Graphics Shader Infrastructure) [37], the intermediate language employed for describing shaders in the Gallium3D driver. Despite being an intermediate assembly, there is very

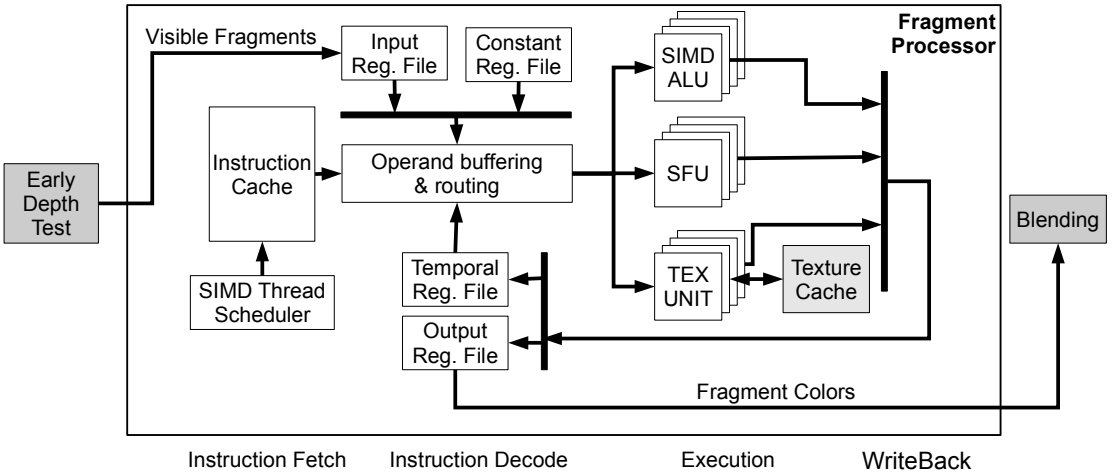


Figure 2.4: Fragment Processor microarchitecture. Vertex processors are similar but they do not include texture units and they take vertices as input instead of fragments.

strong (generally one-to-one) correspondence with the native GPU assembly instructions. Most of the TGSI instructions are vectorial and they operate on vectors of 4 floating point components, as 4-wide vectors are typically employed in graphics. For example, RGBA colors are encoded in 4-wide vectors by using a floating point component to store each channel: reg, green, blue and alpha.

A form of SMT is employed to hide the memory latency by interleaving the execution of several SIMD threads (or warps in NVIDIA terminology [68]). A SIMD thread is a group of threads executed in lockstep mode, that is the same instruction is executed by all the threads but each thread operates on a different fragment. The SIMD width of the pipeline is four, so each SIMD thread consists of four threads that operate on a different fragment, and the input fragments are packed in quad fragments before being dispatched to the processors. Note that individual threads can execute vectorial instructions, for example to multiply two colors that are represented as floating point vectors of four components. Hence, the pipeline requires fetching and decoding just one instruction to issue up to 16 scalar operations in parallel: each SIMD thread consists of four threads and each individual thread executes four scalar operations.

Fragments that pass the Depth Test are packed in groups of 4 fragments or *quads*. A *quad fragment* is processed by a SIMD thread, so each thread in the SIMD thread processes one of the fragments in the *quad*. A *quad fragment* waits at the input of the *Fragment Processor* until a SIMD thread context is available. Once a SIMD thread is free, the per-fragment attributes of the 4 fragments in the *quad* are copied in the Input Register File and the fragment program starts execution: the PC of the SIMD thread context is updated to point to the first instruction of the corresponding fragment program.

In the first pipeline stage, *Instruction Fetch*, the instructions of the fragment program are read from memory by using an Instruction Cache to boost the process. In first place, the *SIMD thread scheduler* determines from which SIMD

thread to fetch an instruction by using a Round Robin policy. Next, the PC of the selected SIMD thread is employed to send a request to the Instruction Cache. Once the instruction is fetched the PC is updated to point to the next instruction. Note that each SIMD thread has its own PC, but the 4 threads in the same SIMD thread share the PC so all of them execute the same instruction on a different fragment.

In the next pipeline stage, *Instruction Decode*, the source operands are fetched from the main register file. Each thread has available a set of 16 input registers for storing per-fragment attributes such as the screen coordinates or the texture coordinates, a set of 12 temporal registers for intermediate computations, and a set of 8 output registers for storing the result of the fragment program. Furthermore, a set of 96 constant registers is shared by all the threads, constant registers contain global state information such as the number of lights enabled. All these registers are vectorial and consist of 16 bytes split in 4 floating point components. One thread has 36 registers (16 input, 12 temporal, 8 output), so it requires 576 bytes of storage and, hence, a SIMD thread context requires 2304 bytes in the main register file. The register file is organized in 4 banks, each one having the registers of one of the threads in a SIMD thread. Furthermore, each bank has 3 read ports and 1 write port. By using this organization, all the source operands for the four threads in a SIMD thread can be fetched in just one cycle, even for the MAD (multiply-add) instruction that requires 3 source operands. The decode logic reads the source operands of the given instruction and dispatches the operands to the functional units.

In the next pipeline stage, *Execution*, the instruction is executed in the corresponding functional unit. Three types of functional units are included in each *Fragment Processor*. The SIMD ALU executes vectorial operations such as additions or multiplications. The SFU (Special Functions Unit) executes more complex operations such as reciprocal or square root. Finally, the Texture Unit computes the color of a texture at a given coordinates. The functional units are pipelined so a new instruction can be issued every cycle. Furthermore, a *Fragment Processor* includes 4 SIMD units, 4 SFU units and 4 Texture Units, so the operations for the 4 threads in one SIMD thread can be issued in parallel in the same cycle. This means that 16 scalar operations can be executed in parallel.

The Texture Unit consists of 3 stages. First, the addresses of the texels (texture elements) are generated by using the texture coordinates of the fragment and the base address of the target texture. In the second stage, the memory requests to fetch the texels are issued to the texture cache. In the final stage, the colors of the texels are combined by applying the corresponding texture filter, this requires linear or trilinear interpolation to obtain the final color. The *Fragment Processor* includes a texture cache to boost the process of fetching the texels from memory. In case of a cache miss in the texture cache, the SIMD thread is marked as blocked and it is not selected for execution by the *SIMD Thread Scheduler*. By doing this, the *Fragment Processor* does not fetch instructions from SIMD threads that are waiting for long latency operations. Once the texture instruction is resolved, the SIMD thread is marked again as ready for execution.

In the last pipeline stage, *Writeback*, the result of the instruction is written in the temporal or in the output register file. Scoreboarding is employed to track dependencies. The destination register is marked as "not available" in the decode stage and it is marked as "available" when its value is written in the writeback stage. On the other hand, no forwarding mechanism is present in the pipeline. However, consecutive instructions usually belong to different SIMD threads since Round Robin is employed and, hence, instructions executed back-to-back are independent.

The end of the fragment program is signaled by a special instruction in the ISA. This "end of program" instruction reads the results of the fragment program, i. e. the colors of the fragments, from the output register file and it dispatches the colors to the next GPU stage, the *Blending*. Furthermore, the SIMD thread context is marked as free so a new *quad fragment* can be assigned to the context.

Vertex Processors are similar to *Fragment Processors*, but the SIMD threads operate on four different vertices instead of fragments. Furthermore, they do not have to handle texture instructions so they do not include Texture units or Texture caches. We assume a non-unified architecture as opposed to a unified architecture where all the processors can handle both vertices and fragments. Usually, unified architectures offer better workload balance, whereas non-unified architectures can exploit the difference between vertex and fragment processing to build more specialized and optimized processors. For instance, the results obtained by using McPAT indicate that a Vertex Processor has just 64% of the area of a Fragment Processor.

TEAPOT reports statistics per-frame and per-thread. Since GPU commands are tagged with the thread identifier, TEAPOT is able to assign fractions of GPU execution time and energy to each application.

System Memory Simulation

The GPU timing simulator of TEAPOT employs DRAMSim2 [133] to accurately model system memory. DRAMSim2 simulates a DDR2/3 memory system, including a detailed cycle accurate model of the memory controller, memory channels, DRAM ranks and banks.

Cycle accurate simulators typically employ fixed memory latencies for the off-chip accesses to main memory. Such a simplistic approach fails to provide a realistic estimation of the cost of accessing main memory, as real memory systems exhibit highly complex behavior that produces significantly different latencies for different memory accesses. For example, the DRAM refresh is a major source of variance in the latency of the memory requests, as read requests that are issued while a refresh is in progress have to wait much longer than other requests. DRAMSim2 is employed in TEAPOT to accurately model the behavior of system memory and, hence, off-chip memory accesses exhibit variable latency.

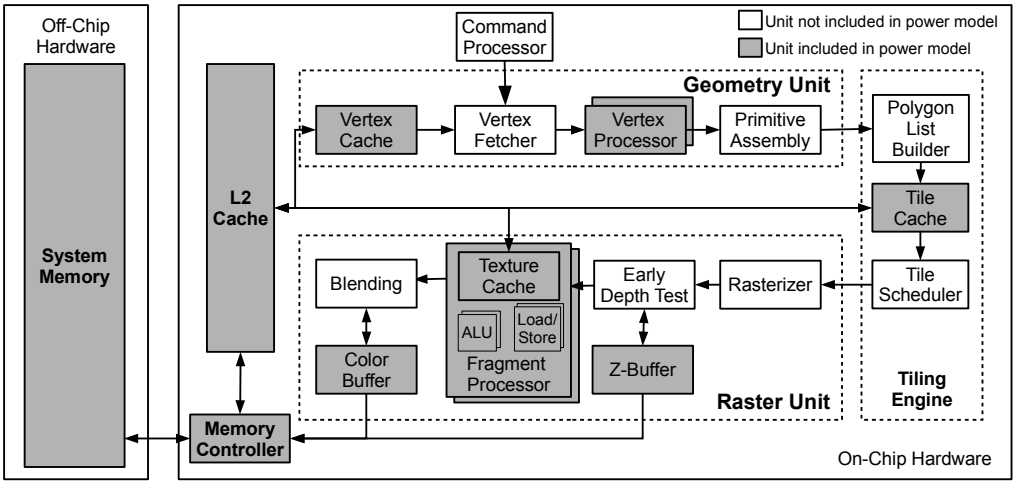


Figure 2.5: The figure shows the GPU units that are included in the power model and the units that are not modeled, for a TBR architecture. Similar models have been included for the IMR architecture.

Power Model

A modified version of McPAT [107] is used for estimating GPU energy consumption. During start-up, the GPU simulator calls to McPAT passing all the microarchitectural parameters, such as the number of processors or the cache sizes, so it can build the internal chip representation. McPAT estimates the dynamic energy required to access each one of the hardware structures and the leakage power. During simulation, the cycle-accurate GPU simulator collects statistics for each unit and, at the end of every frame, it submits all the activity factors to McPAT. The dynamic energy is computed by accounting for events in the GPU simulator and then multiplying these events by a given energy cost estimated by McPAT. The static energy is obtained by multiplying the total GPU leakage by the execution time. McPAT has been slightly modified so it can better model a low-power GPU. For instance, we have included support for read-only L1 data caches to better model the Texture caches of a mobile GPU. Furthermore, we have extended McPAT in order to model Texture sampling units. The texture sampler is implemented by using a combination of Load units, for fetching the texels (texture pixels) from memory, and FP units, for applying the texture filtering.

Figure 2.5 shows the GPU units that are accounted in the power model of TEAPOT. As we can see, the simulator provides energy estimations for the programmable units —Vertex and Fragment processors— and the entire GPU memory hierarchy, including the off-chip system memory. On the contrary, the energy consumed by the fixed-function units is not accounted in our power model. McPAT was designed as a power model for multicore and manycore architectures and, hence, it is easy to model the programmable processors and the caches, register files and the rest of memories included in the GPU. However, McPAT does not provide support for modeling any specialized fixed-function graphics hardware. Nevertheless, the GPU cores and the memory hierarchy are typically the main sources of energy consumption in a mobile graphics processor, representing

more than 90% of the total GPU energy [124, 125].

2.1.4 Automatic Image Quality Assessment

Generating high quality images comes at the cost of significant energy consumption which sometimes is not desirable in the mobile segment, specially for low-battery conditions. Significant energy savings can be achieved by allowing small distortions on image quality, several techniques take this approach such as texture compression or the memoization scheme described in Chapter 5. When trading quality for energy we need some way of evaluating the magnitude of the visual quality decrease. To this extent, TEAPOT provides several metrics for automatic image quality assessment. Image quality is evaluated by comparison with a reference image, usually the result of a high quality rendering. The error is then estimated by comparing the high quality image with the distorted image.

In TEAPOT, the original high-quality frames generated by the real hardware provide the reference images. These images are employed for two purposes. First, they are compared with the frames generated by our GPU architecture models to verify the correctness of the timing simulator. Second, they are employed to assess image quality in case some technique that trades quality for energy is enabled in the simulator, in order to evaluate the magnitude of the distortions.

Two types of metrics are typically employed for image quality assessment, one based on per-pixel errors and the other based on the human visual perception system. Regarding the metrics based on per-pixel errors, TEAPOT implements the MSE (Mean Squared Error) [153] and the PSNR (Peak Signal-to-Noise Ratio) [158]. TEAPOT also includes the MSSIM (Mean Structural SIMilarity Index) presented in [149], a metric based on the human visual perception system. This metric is more desirable since the images generated by the GPU are interpreted by users. Hence, an error in a pixel is a problem just if it can be perceived by the human visual system, i. e. if it causes a degradation of structural information, since the human visual perception system is highly adapted for extracting structural information from a scene. An MSSIM value of 100% means perfect image fidelity, whereas a value of, for example, 90% indicates 10% of perceivable differences between the reference image and the distorted image. The original MSSIM metric only works on gray-scale images, but it can be adapted for RGB format [128].

2.1.5 Assumed Graphics Pipeline

A graphics pipeline consists of three stages: application, geometry and fragment. The application stage performs tasks such as animation of the objects in the scene, physical simulation or collision detection, and it is usually executed on the CPU. Furthermore, the application stage takes care of issuing all the rendering commands to the graphics processor. The geometry and fragment stages are

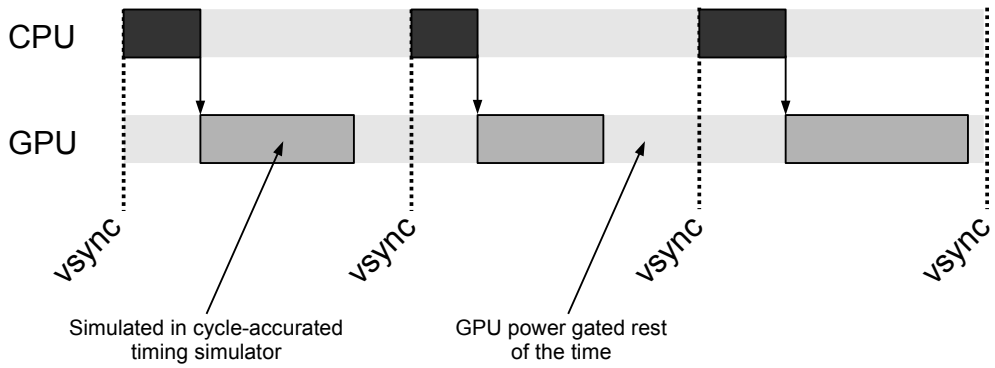


Figure 2.6: The figure illustrates the graphics pipeline assumed in our simulation infrastructure. The timing simulator estimates the cycles where the GPU is busy, the graphics processor is assumed to be power gated the rest of the time.

usually executed on the GPU and perform the rendering of the graphical objects. This pipeline is illustrated in figure 2.6.

Our cycle-accurate GPU simulator estimates the time required to process all the rendering commands issued by the CPU. We assume a constant frame rate that is synchronized with the screen refresh, i. e. we employ vertical synchronization. Hence, the GPU is prevented from generating images at a faster rate than the screen refresh. Note that we do not assign any static energy consumption during long idle periods because we assume that the GPU could drastically reduce it by entering a deep low power state.

As the frame rate is constant and the GPU is synchronized with the screen refresh, the total execution time of the graphics pipeline is the same in all the configurations. Hence, when we report GPU execution times and speedups we focus on active GPU time, that is different depending on the GPU microarchitecture, and we do not account idle periods where the GPU is power gated. The current version of our simulation infrastructure does not implement cycle-accurate simulation of the CPU part of the graphics pipeline, just estimations of the GPU execution time and energy consumption are provided.

In short, a graphical application like a game is composed of a CPU portion and a GPU portion. The timing model in our simulation infrastructure only reports the cycles where the GPU is busy.

2.2 Workloads

In order to evaluate our proposals we have selected twelve commercial Android games that span a wide range of graphics characteristics. We employ real games instead of OpenGL ES benchmark suites such as GFXBench [14] for several reasons. In first place, GFXBench benchmarks are complex 3D kernels that visualize a plethora of advanced 3D effects by exploiting the most recent features of mobile graphics hardware. We believe these kernels are not necessarily representative of

the mobile games that users typically play, as popular games for smartphones tend to be much simpler. Although we include complex 3D games in our set of workloads, we think it is important to also consider 2D games and simpler 3D games as they are very common in the mobile segment. In second place, some energy saving techniques require the evaluation of their impact on user interaction and responsiveness, such as the technique described in Chapter 4. Benchmarks are not well-suited for evaluating user interaction as they are just kernels that run from start to finish without user intervention. Hence, we preferred to rely on real games for this type of user interaction analysis. Next sections introduce our set of games and provide a brief workload characterization.

2.2.1 Workload Selection

We have selected five casual 2D games due to the popularity of this type of applications in smartphones and tablets. Figure 2.7 shows several screenshots of our 2D workloads. A brief description of the 2D games follows:

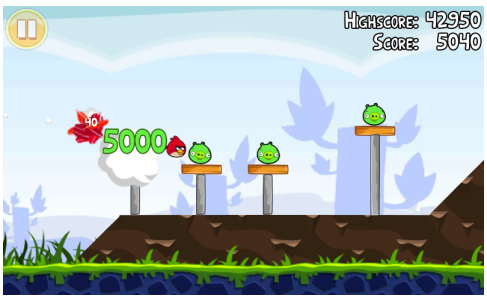
- *Angry Birds*: Undoubtedly one of the most popular games of all time. In this game players use a slingshot to launch birds at pigs stationed on or within various structures, with the intent of destroying all the pigs on the playing field. On the technical side, it features relatively elaborated physical simulation and simple 2D graphics with effects such as alpha blending. The game alternates between scenes with static backgrounds and side-scrolling.

- *Bad Piggies*: Spin-off of Angry Birds, in this puzzle game the objective of the player is to build a contraption that transports the pig from a starting point to the finish line, usually indicated by a map. It features physical simulation of rigid bodies in 2D and it also alternates static scenes with side-scrolling.

- *Cut the Rope*: Physics-based puzzle game. The objective of the game is to feed candy to a little green creature. In each level, the candy hangs by one or several of the titular ropes which the player must cut with a swipe of his finger using the device's touchscreen. The game displays simple 2D sprites on top of static backgrounds.

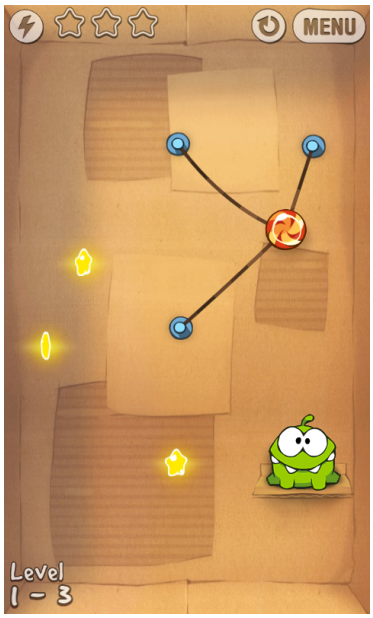
- *Gravity Guy*: Arcade and side-scrolling game in which the player controls a character by tapping the screen to switch gravity. The objective in this game is to run as far as possible while avoiding obstacles that can trap the player, and avoid falling or flying off the screen. This game belongs to the "endless runner" genre, a type of game that is very popular in smartphones and tablets.

- *Jetpack Joyride*: Side-scrolling endless runner game that employs a simple, one-touch system to control a jetpack. When the player presses anywhere on the touchscreen, the jetpack fires and the character rises. When the player lets go, the jetpack turns off and the character falls. The objective of

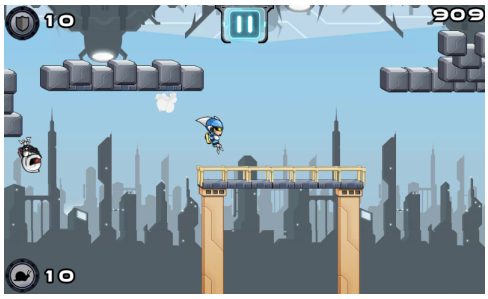


(a) Angry Birds

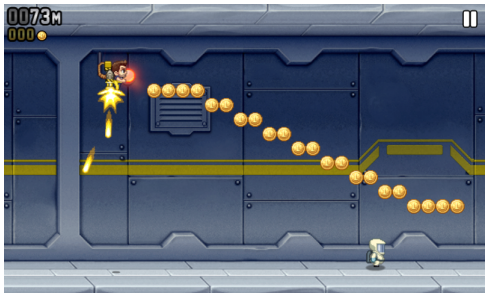
(b) Bad Piggies



(c) Cut the Rope



(d) Gravity Guy



(e) Jetpack Joyride

Figure 2.7: The figure shows a screenshot for each of the 2D Android games selected as benchmarks.

the game is to travel as far as possible, while collecting coins and avoiding obstacles.

On the other hand, we have also selected seven 3D Android games. Despite not being so popular as the aforementioned 2D applications, 3D games still represent a big market in the mobile segment. Furthermore, they exploit the most advanced features of embedded graphics hardware. Figure 2.8 shows several screenshots of our 3D workloads. A brief description of the 3D games follows:

- *Air Attack*: Top-down air combat shooter in which the player controls the plane by touching the screen and dragging. The game displays high definition 3D graphics and advanced effects such as particle systems, powered by the Unity3D engine [38].
- *Captain America*: Side-scrolling action game. The player controls Captain America as he runs, slides, vaults and combats multiple enemies. The

(a) Air Attack (b) Plants War (c) Temple Run

(d) Captain America (e) Crazy Snowboard

(f) Dungeon Defenders (g) Playmobil Pirates

Figure 2.8: Screenshots of the different 3D workloads.

different actions are triggered by touching the screen and dragging. It is powered by the Unity3D engine and it exploits programmable shaders to implement different visual effects.

- *Crazy Snowboard*: Racing game in which the user controls a snowboarder while descending a mountain, performing acrobatics that are triggered via the touch screen. The game features fairly simple 3D graphics and it includes basic lighting and terrain models.

- *Dungeon Defenders*: Mix of Tower Defense and RPG (Role-Playing Game) where between one to four players work together to protect one or more crystals from being destroyed by waves of enemies. Its graphics and gameplay are desktop-like, as it has also been released on PC, Xbox and PS3.

The game features advanced 3D graphics powered by the Unreal Engine [39] and supports a plethora of 3D effects.

- *Plants War*: Real-time strategy game in which the player has to manage resources and build his own army to defeat the enemy troops. Plants War displays simple 3D graphics, as it employs basic lighting models and 3D objects with low polygon counts.

- *Playmobil Pirates*: Combination of strategy and action game. The player has to build and control his own pirate fortress and fleet to protect the treasure of his island. The game displays high definition 3D graphics, including advanced effects such as water rendering, projected shadows or particle systems.

- *Temple Run*: One of the most popular endless runner games. The player takes on the role of an explorer who, having stolen an idol from a temple, is chased by evil monkeys. The player has to swipe to turn, jump and slide to avoid obstacles and collect coins. The game displays relatively complex 3D graphics powered by the Unity3D engine, and it includes some effects such as water rendering.

Our workloads are representative of the mobile graphical applications as we have included some of the most popular games available for smartphones. Furthermore, our set of workloads cover most of the features available in the OpenGL ES 1.1/2.0 APIs [96, 95]. We provide more detailed explanations of why we have selected these concrete games in section 2.2.2. Note that despite Khronos has already released OpenGL ES 3.0/3.1 [97, 98], the new specifications of the API are very recent —December 2013 and March 2014 for 3.0 and 3.1 respectively— and thus all the games available for Android still employ OpenGL ES 1.1/2.0 at the time of writing.

On the other hand, we have focused on the game genres that are more popular in smartphones and tablets. For example, First-Person Shooters (FPS) are very popular in PC and desktop platforms, but they represent a small market in the mobile segment as it is difficult to play these games by using a touch screen. Casual games, physics-based puzzle games or endless runner games are much more common in mobile devices as they better exploit the smartphone's user interfaces.

Note that we have not included any OpenCL/CUDA workload despite the growing interest in General Purpose Computing on the GPU (GPGPU). The support for OpenCL and CUDA in the mobile segment has been non-existent or very poor in the best case during past years, and only the most recent mobile GPUs and Android versions begin to truly support GPGPU. On the other hand, CUDA/OpenCL are typically employed to boost scientific applications. We believe that mobile phones are not the ideal platform for scientific codes, so our research is focused on more typical workloads for smartphones. Other computationally intensive tasks that are common in smartphones and could be

offloaded to the GPU, such as image processing or video decoding, are usually handled by ASICs included in the SoC that are extremely energy-efficient executing these specific tasks. We are not claiming that GPGPU is useless in the mobile segment, since the GPU offers much more flexibility than the ASICs and it is much more energy-efficiency than the CPU for data-parallel workloads. We are only highlighting the fact that it currently has a smaller presence in smartphones. Nevertheless, the current trends in mobile graphics hardware indicate that GPGPU support will be much more widespread and of much higher quality in next years. We believe that, for example, analyzing the use of OpenGL ES 3.1 compute shaders to boost physics simulation or collision detection is an intriguing direction. We leave this type of study for future work, and we propose simple ways of improving our simulation infrastructure to support GPGPU in Chapter 6.

2.2.2 Workload Characterization

In this section we provide a brief workload characterization of our set of games. As these applications are not the conventional benchmarks employed in computer architecture, we think it is worth including a short analysis of their characteristics. We have employed our *OpenGL ES Trace Generator*, described in section 2.1.1, to collect traces of the OpenGL ES calls as seen in our Android games. The results reported in this section have been obtained by analyzing traces of 400 frames for each game, the scenes have been rendered at WVGA (800×480) resolution.

Table 2.1 provides a general description of the workloads. It includes the type of game —2D or 3D—, the date were the last update of the game was released, the approximate number of downloads as reported by Google Play, the game engine and the version of the OpenGL ES API. As we can see, the games have been developed in recent years. Most of them have been updated recently, whereas the oldest game, *captainamerica*, received its last update by 2011. In addition, these games have been downloaded and installed at least more than one million times, and we have included some of the most popular games such as *angrybirds*, *cuttherope* or *templerun* that have been downloaded more than one hundred million times in Google Play.

Regarding the game engine, five of the games employ Unity3D. Unity3D is a state-of-the-art cross-platform game engine developed by Unity Technologies that is commonly employed to create games for mobile devices. Unity3D supports advanced 3D effects such as normal mapping or dynamic shadows. On the other hand three of the games, *cuttherope*, *gravityguy* and *jetpackjoyride*, employ their own custom developed game engine. As mobile games tend to be simpler, some small companies opt by developing their own rendering solutions. This is different from the desktop platform where developing the engines that support modern 3D games would take years and, hence, developers usually rely on mature game development kits. Finally, we have included a game that employs the Unreal Engine 3, *dungeondefenders*. Unreal Engine is one of the best engines available

Table 2.1: General overview of our set of workloads.

Game	Type	Last Update	Downloads	Engine	OpenGL ES
angrybirds	2D	March, 2014	500M	Box2D	1.1
badpiggies	2D	Nov, 2013	50M	Unity3D	2.0
cuttherope	2D	Apr, 2014	100M	Own	1.1
gravityguy	2D	Oct, 2013	10M	Own	1.1
jetpackjoyride	2D	Feb, 2014	50M	Own	1.1
airattack	3D	Jan, 2014	50M	Unity3D	1.1
captainamerica	3D	Jul, 2011	-	Unity3D	2.0
crazysnowboard	3D	March, 2014	10M	Unity3D	1.1
dungeondefenders	3D	Dec, 2012	1M	Unreal Engine	2.0
plantswar	3D	Nov, 2013	5M	-	1.1
playmobilpirates	3D	Aug, 2013	5M	-	2.0
templerun	3D	Jan, 2014	100M	Unity3D	2.0

and it is commonly employed to develop desktop games. We think it is interesting to see how a top-notch engine behaves in this new environment of mobile devices.

Current smartphone's games employ the version 1.1 or the version 2.0 of the OpenGL ES API and, hence, we have covered both versions in our set of workloads as it can be seen in the last column of Table 2.1. OpenGL ES 1.1 implements a fixed-function pipeline, whereas version 2.0 supports programmable vertex and fragment shaders. Note that most of the modern graphics hardware implements a fully-programmable pipeline. Our timing simulator also implements a programmable GPU as described in section 2.1.3. In case an OpenGL ES 1.1 application runs on top of programmable hardware, the GPU driver generates the corresponding shaders that mimic the functionality of the fixed-function pipeline. Despite OpenGL ES 3.0 and 3.1 have already been released, at the time of writing this thesis we could not find any game that employs these recent versions of the API.

Table 2.2 reports information about the complexity of the geometry employed by our set of games. The second column shows the average number of batches sent to the GPU per frame. A batch is a group of vertices that are rendered in a single function call to the API, in case of OpenGL ES by calling *glDrawElements* or *glDrawArrays*. The third column shows the average number of vertices in each of these batches. The fourth column contains the average number of vertices sent to the GPU per frame. The last column shows the average number of static instructions in the Vertex Shader, considering the assembly instructions in the TGSI [37] ISA. TGSI is the intermediate assembly employed by the Gallium3D driver. It is based on one of the first standardized assembly languages for GPUs, the ARB assembly code [7]. Despite being an intermediate assembly, there is very strong (generally one-to-one) correspondence with the native GPU assembly instructions.

In an OpenGL application the objects in the scene are usually described by a set of triangles, and each triangle is defined by three vertices. An instance

Table 2.2: Analysis of the geometry employed by our set of workloads. The last column shows the average number of static assembly instructions in the vertex shader.

Game	Average batches/frame	Average vertices/batch	Average vertices/frame	Average VS insns
angrybirds	11.00	25.07	275.80	6.00
badpiggies	20.00	43.03	860.60	20.70
cuttherope	120.00	7.37	884.00	6.00
gravityguy	12.00	49.33	592.00	7.00
jetpackjoyride	47.40	9.50	450.20	6.46
airattack	26.70	136.16	3635.40	7.57
captainamerica	30.00	156.20	4686.00	13.53
crazysnowboard	19.50	95.10	1854.40	11.15
dungeondefenders	282.50	348.06	98328.20	22.91
plantswar	45.00	191.33	8610.00	9.20
playmobilpirates	28.00	271.86	7612.20	14.25
templerun	15.40	522.21	8042.10	29.91

of the corresponding Vertex Shader has to be executed for each vertex sent to the GPU. The information reported in Table 2.2 provides some hints on the complexity of the scenes and the workload of the vertex processors. As it can be seen, we cover a wide range of geometric complexity, from games with low vertex counts such as *angrybirds* to games that render complex scenes such as *dungeondefenders*. *angrybirds* sends 11 batches to the GPU per frame on average, whereas the scenes in *dungeondefenders* require more than 282 batches, the rest of games lie in between. Furthermore, batches are much bigger in 3D games such as *dungeondefenders* —348 vertices on average— or *templerun* —522 vertices— than in 2D games like *cuttherope* or *jetpackjoyride* —7.37 and 9.5 vertices on average respectively. Regarding the total number of vertices per frame, reported in the fourth column, we find significant differences between the 2D games, shown in the first five rows, and the 3D games. All the 2D games employ less than one thousand vertices per frame, whereas the 3D workloads render scenes with more than one thousand vertices. *dungeondefenders* is the game with the biggest vertex count by far, as it exploits the power of the Unreal Engine to render scenes with close to one hundred thousand vertices per frame in real time. We also observe great diversity in the complexity of the Vertex Shader, from games with just a few instructions per vertex program on average to games with close to 30 static instructions. Hence, our set of games stress the geometry pipeline of the GPU in very different ways.

Once the vertices sent to the GPU have been processed, the next step in the graphics pipeline is to determine the screen pixels covered by the triangles, generate the corresponding fragments by interpolation of the per-vertex attributes and, finally, execute the corresponding fragment shader to compute the color of each pixel. Table 2.3 provides some hints on the workload and the complexity of the fragment stage in the GPU pipeline. The second column of the table shows

Table 2.3: Analysis of the fragment shaders and depth complexity in our set of workloads. The second column shows the average number of static assembly instructions in the fragment shader.

Game	Average FS insns	ALU to TEX ratio	Average texels/frag	Fragment to Vertex ratio	Depth Complexity
angrybirds	3.00	2.67	4.07	3106.33	3.23
badpiggies	4.15	2.46	2.34	1023.22	2.29
cuttherope	1.85	3.35	4.08	734.72	1.69
gravityguy	3.00	2.00	4.14	1304.94	2.01
jetpackjoyride	3.76	2.76	4.14	1215.00	1.42
airattack	4.26	2.59	8.05	234.20	1.64
captainamerica	5.57	2.98	5.11	244.00	2.55
crazysnowboard	5.38	3.88	3.32	383.65	1.41
dungeondefenders	5.35	3.87	7.81	9.88	1.69
plantswar	3.49	2.74	4.05	119.46	2.60
playmobilpirates	12.89	5.81	12.25	196.42	3.89
templerun	8.64	4.32	10.32	92.12	1.54

the average number of static instructions in the fragment shader, considering the TGSI assembly code. The third column contains the ALU to TEX ratio, i. e. the number of ALU instructions divided by the number of texture fetching instructions in the fragment shader. The fourth column shows the average number of texels (texture elements) fetched for each fragment. The number of texels fetched depends on the type of texture filtering selected by using the OpenGL ES API, and it can be different for each batch. Nearest-neighbor filtering requires one texel to be fetched, whereas linear, trilinear and anisotropic filtering require 4, 8 and 16 texels respectively. Furthermore, multitexturing can be employed to access multiple textures from the same fragment shader, so the number of texels per fragment can be bigger than 16. The fifth column shows the Fragment to Vertex ratio, i. e. the total number of fragments processed divided by the total number of vertices. The last column shows the depth complexity, this parameter is computed as the average number of fragments generated for each pixel per frame and it provides a measure of the overdraw.

The numbers reported in Table 2.3 indicate that the fragment programs employed by our set of workloads are significantly different. For instance, *cuttherope* uses fragment shaders with just 1.85 instructions on average, whereas a more complex game like *playmobilpirates* employs shaders with 12.89 assembly instructions on average, the rest of games lie in between. The ALU to TEX ratio is also different for the several games. This ratio provides an idea of how memory-intensive the fragment shaders are, as using texture instructions is the only way of accessing memory in a fragment shader in OpenGL ES 2.0. The bigger the ALU to TEX ratio the better as it will be easier to keep the functional units busy and avoid pipeline stalls due to memory latency. Our games show different values for this ratio ranging from 2 (*gravityguy*) to 5.81 (*playmobilpirates*).

On the other hand, the games employ different texture filters. 2D games

prefer a combination of nearest-neighbor and linear filtering, requiring around 4 texture fetches per fragment. Some 3D games extensively employ trilinear filtering, fetching around 8 texels per fragment such as *airattack* and *dungeondefenders*. Finally, 3D games like *playmobilpirates* and *templerun* make extensive use of multitexturing and trilinear filtering, demanding more than 10 texels per fragment on average.

The fifth column of Table 2.3 contains the Fragment to Vertex ratio. This ratio indicates whether a game is geometry bound or fragment bound, i. e. if the main bottleneck in the pipeline will be the vertex processors or the fragment processors. Games are typically fragment bound since the GPU has to process many more fragments than vertices, this is also the case in our workloads. 2D games are especially fragment bound as the number of fragments to be shaded is three orders of magnitude bigger than the number of vertices. 3D games are also fragment bound, although the number of vertices is much closer to the number of fragments than in 2D games, being just between one and two orders of magnitude smaller. *dungeondefenders* is the most geometry intensive game, as the number of fragments is just 9.88 times bigger than the number of vertices.

The last column of Table 2.3 contains the overdraw. As we mentioned in section 1.3.3, overdraw is an important issue as it can cause multiple writes to each pixel in the Color Buffer and multiple executions of the fragment shader per pixel, wasting bandwidth, time and energy. Our workloads exhibit different depth complexities, ranging from 1.41 fragments per pixel in *crazysnowboard* to 3.89 fragments in *playmobilpirates*.

In short, we have selected a set of workloads that extensively cover the full spectrum of mobile graphical applications. Our workloads include some of the most popular applications for smartphones and, furthermore, they exhibit different degrees of graphics complexity and make completely different use of the geometry and fragment pipelines.

2.3 Summary of Methodology

Regarding our evaluation methodology, we employ TEAPOT to run the Android games described in section 2.2.1. We have collected OpenGL ES traces of 400 frames for each game by using the *OpenGL ES trace generator* available in TEAPOT. Furthermore, we have executed the traces in the GPU functional emulator to generate a complete GPU instruction and memory trace for each game. Finally, the GPU instruction and memory traces are used to drive the cycle-accurate timing simulator. The simulator models both IMR and TBR. The IMR architecture implemented in the simulator is illustrated in Figure 2.2 whereas the TBR architecture is shown in Figure 2.3. Both pipelines are described in section 2.1.3.

Table 2.4 provides the architectural parameters that are common to all the experiments. Parameters such as the technology, the frequency or the characteristics of the system memory are fixed for all the configurations. The values

Table 2.4: Microarchitecture parameters for the experiments.

Global Parameters			
Screen resolution	800×480 (WVGA)	**Technology**	28 nm
Fragment Procs	4	**Frequency**	600 Mhz
Vertex Procs	4	**Tile size**	32×32
Memory Hierarchy			
Main memory	1 GByte, 16 bytes/cycle, 50-100 cycles (variable latency)		
Line size	64 bytes		
L2 Cache	128 KB, 8-way, 1 R and 1 W port, 8 banks, 12 cycles		
Tile Cache	16 KB, 2-way, 1 R and 1 W port, 1 bank, 2 cycles		
Pixel Cache	8 KB, 2-way, 1 R and 1 W port, 1 bank, 1 cycle		
Z-Cache	32 KB, 2-way, 1 R and 1 W port, 1 bank, 2 cycles		
Texture Cache	16 KB, 4-way, 1 R and 1 W port, 1 bank, 2 cycles		
Local Color Buffer	4 KB, 1 R and 1 W port, 1 bank, 1 cycle		
Local Z-Buffer	4 KB, 1 R and 1 W port, 1 bank, 1 cycle		
Vertex Cache	8 KB, 2-way, 1 R and 1 W port, 1 bank, 1 cycle		
Fragment Processor			
Instruction Cache	8 KB, 2-way, 1 R port, 1 bank, 1 cycle		
SIMD Threads	1-16 SIMD thread contexts		
Register File	2.25 KB/SIMD thread, 3 R and 1 W ports, 4 banks		
SIMD ALUs	4 units, ADD/SUB 1 cycle, MUL 6 cycles, DIV 20 cycles		
SFUs	4 units, 6-30 cycles		
Texture Units	4 units, addr gen 1 cycle, texture filter 1 cycle		
Inter-Stage Queues			
Input	**Output**	**Num Entries**	**Size (bytes)**
Vertex Fetcher	Vertex Procs	16	4096
Vertex Procs	Prim Assembly	16	4096
Prim Assembly	Rasterizer	8	6144
Prim Assembly	PolyListBuild	8	6144
Tile Scheduler	Rasterizer	8	6144
Rasterizer	Early Z	8	8192
Early Z	Frag Procs	8	8192
Frag Procs	Blending	16	384

set for these parameters are inspired by the specifications found in current mobile GPUs. More specifically, our IMR model is based on the microarchitecture of the Ultra-Low Power GeForce in the NVIDA Tegra SoC [115], whereas our TBR model is based on the specifications of the ARM Mali 400MP [55]. The experimental results reported in next chapters are always accompanied with the relevant microarchitectural parameters that were employed to run those specific experiments.

Chapter 3

Decoupled Access/Execute Fragment Processor

In this chapter we address the problem of hiding the memory latency in a mobile GPU. Firstly, we argue that conventional techniques such as aggressive multi-threading or prefetching are effective but not energy efficient. Secondly, we propose a decoupled access/execute-like design for the fragment processors. Finally, we apply our proposal on top of a state-of-the-art mobile GPU and show that the end design with just 4 SIMD threads/processor is able to achieve 97% of the performance of a larger GPU with 16 SIMD threads/processor, while providing 20.5% energy savings.

3.1 Memory Latency Tolerance in a Mobile GPU

The fragment processors fetch data from texture memory. Each one of these processors includes a texture cache to speed-up these memory accesses. Texture memory accesses are common operations in graphics workloads and tend to be fine-grained and difficult to prefetch [79]. Graphics workloads have large, long-term (inter-frame) texture datasets that are not amenable to caching. Hence, texture caches focus on conserving bandwidth rather than reducing latency [73].

The fragment processing stage is typically one of the main bottlenecks in the graphics pipeline. As described in the workload characterization provided in section 2.2.2, the GPU has to process millions of fragments per second and shading each fragment usually requires multiple texture fetches. A mobile GPU working at WVGA resolution (800×480) and at a modest 30 frames per second has to process at least more than eleven million fragments per second. Since some games demand more than ten texture fetches per fragment on average (see Table 2.3), this means that the texture sampling subsystem has to sustain hundreds of millions of texture accesses per second. Therefore, the texture sampling is one of the key stages in the graphics pipeline, as the GPU spends significant amounts of time and energy mapping textures on top of the objects.

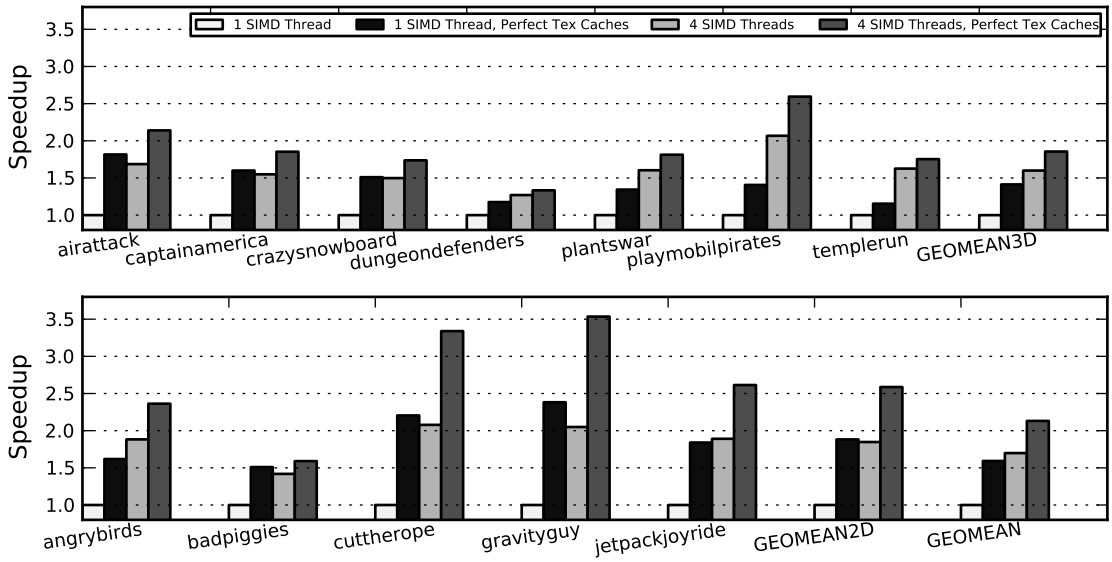


Figure 3.1: Potential performance benefits of removing all the cache misses in the texture caches of the fragment processors. The baseline is an Immediate-Mode Renderer with just one SIMD thread per fragment processor. The top graph shows the speedups for the 3D workloads, whereas the bottom graph provides the results for the 2D games. GEOMEAN3D means average results for the 3D games, GEOMEAN2D is the average for the 2D games and GEOMEAN is the global average considering all the workloads.

Figure 3.1 illustrates the potential benefits of removing all the texture cache misses. The baseline configuration is a mobile GPU that implements an Immediate-Mode Rendering architecture like the one shown in Figure 2.2. The different parameters of the pipeline, such as the number of fragment processors or the size of the texture caches, are provided in Table 3.1. Note that the baseline configuration features single-threaded in-order fragment processors. The baseline does not employ multithreading, prefetching nor out-of-order execution, so no other mechanism than the texture caches is employed to hide the memory latency. Removing all the texture cache misses provides significant performance improvements, as the single-threaded GPU with perfect texture caches achieves 1.59x speedup on average, a maximum speedup of 2.38x —*gravityguy*— and a minimum speedup of 1.15x —*templerun*. On the other hand, Figure 3.1 also includes the speedups achieved by a GPU with a small degree of multithreading, 4 SIMD threads per fragment processor, and the same multithreaded GPU with perfect caches. As we can see, removing all the texture cache misses also provides significant speedups on top of the multithreaded GPU, especially in some 2D games.

Note that Figure 3.1 reports overall GPU speedups considering the entire graphics pipeline. However, including perfect texture caches only improves the fragment stage of the pipeline. Hence, the overall benefit of removing texture cache misses depends on the importance of the fragment stage in the total GPU execution time. As mentioned in section 2.2.2, 2D games are typically more fragment bound than 3D games. As the workload in the geometry pipeline tend to be smaller in 2D, GPU execution time mainly depends on the fragment stage

for 2D games. As a result, including perfect texture caches provides 1.88x speedup for the 2D workloads in the single-threaded GPU. Although the perfect texture caches do not provide such a big speedup for the 3D workloads, they are still able to achieve 1.41x speedup on average and up to 1.81x speedup in *airattack*.

In short, hiding the memory latency of the texture fetches provides significant performance improvements. However, texture accesses are extremely frequent and macroscopically unpredictable, so they are not amenable to caching or prefetching. Texture caches alleviate the problem but their main task is saving bandwidth, as they are able to filter a non-negligible percentage of memory accesses, but they are not as effective as in conventional CPUs. Hardware prefetching is also challenging as we will show in section 3.1.2. Out-of-order execution is considered too complex for embedded graphics processors and not as energy efficient as multithreading for data-parallel applications. As graphics workloads exhibit a high degree of data-level parallelism and memory-level parallelism, desktop GPUs have embraced massively multithreaded architectures.

3.1.1 Aggressive Multithreading

Most desktop GPUs rely on massive multithreading as an effective way to hide memory latency and keep functional units busy during cache misses. Fragment processors interleave the execution of multiple SIMD threads (warps in NVIDIA terminology). In case a SIMD thread is stalled waiting for texture data, the scheduler selects another ready SIMD thread for execution. By leveraging simultaneous multithreading, huge amounts of memory latency can be tolerated just by including the appropriate number of SIMD threads. However, the simultaneous executions of multiple SIMD threads requires an equal number of thread contexts, which greatly increases the number of registers that must be kept in the register file.

The Register File (RF) of a fragment processor stores the registers of all the SIMD threads that are executed in parallel. Each SIMD thread consists of 4 threads that execute the same fragment program on 4 different fragments. According to the minimum resources required for an OpenGL-compliant implementation [7], the RF must provide storage for at least 10 input registers, 10 output registers and 16 temporal registers per thread. Furthermore, each of these registers is a 4-wide vectorial register of 16 bytes (4 floating-point components). Hence, each thread requires 36 4-wide registers, i. e. 576 bytes of storage in the RF. As a SIMD thread contains 4 threads, the RF must provide 2304 bytes of storage for each SIMD thread in the fragment processor. For example, a fragment processor that supports 16 SIMD threads must include a RF of 36 KBytes, much bigger than a typical texture cache size of 16 KBytes (see Table 3.1). On the other hand, GPUs usually include multiple fragment processors. A mobile GPU with 4 fragment processors that supports 16 SIMD threads per processor must provide 144 KBytes of total storage in the RF in order to maintain a total of 256 parallel threads. Therefore, the graphics processor requires a significant amount of storage in the RF in order to sustain massive multithreading.

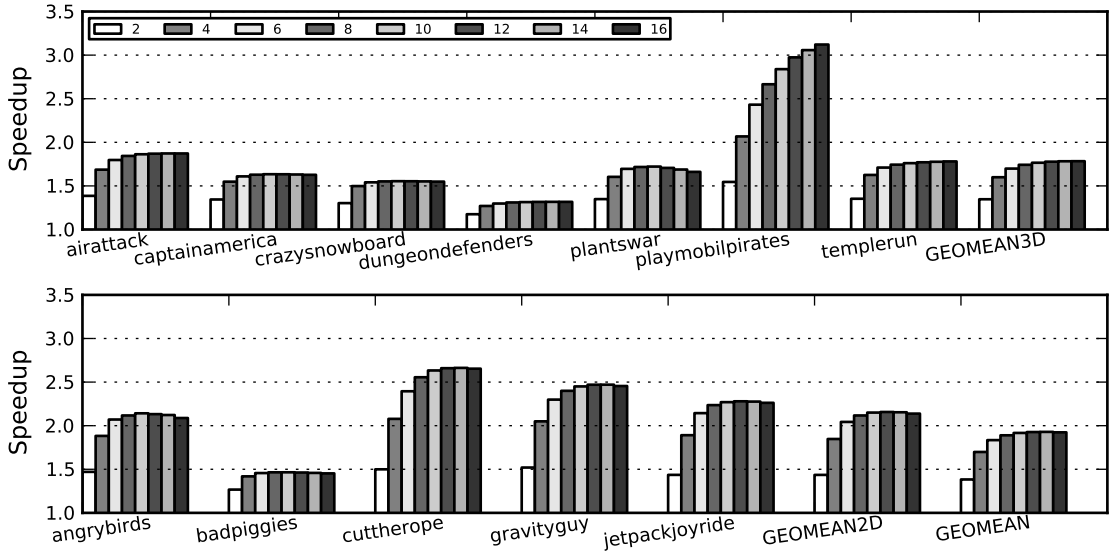


Figure 3.2: Speedups achieved by increasing the number of SIMD threads in the fragment processors of a mobile GPU. The baseline is a mobile GPU with just one SIMD thread per processor.

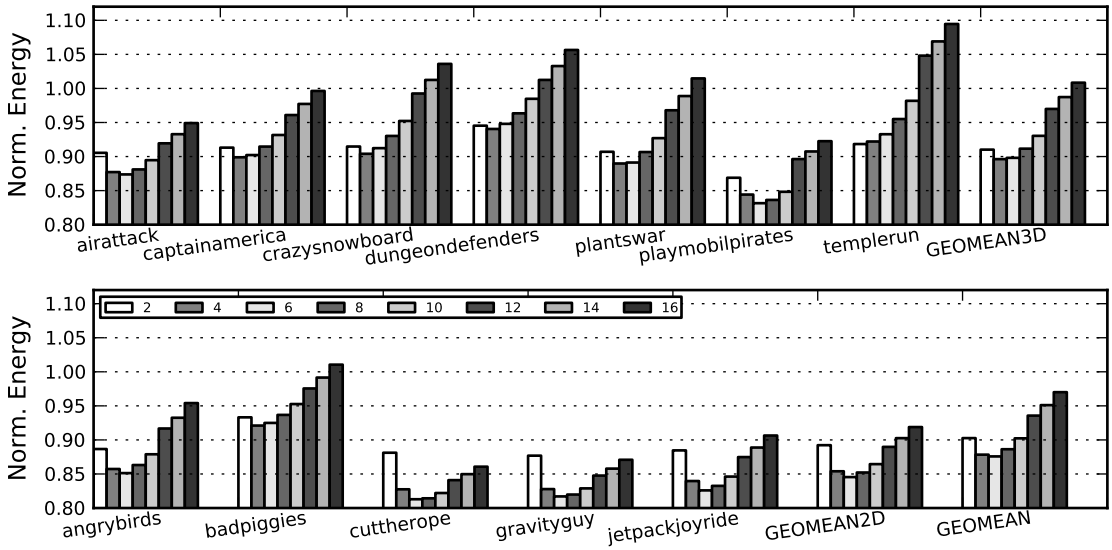


Figure 3.3: Normalized energy obtained when increasing the number of SIMD threads in the fragment processors of a mobile GPU. The baseline is a mobile GPU with just one SIMD thread per processor.

The numbers of a multithreaded mobile GPU may look modest compared to a high-end desktop GPU. As aforementioned, a conventional mobile GPU is able to maintain hundreds of parallel threads with a RF of several KBytes, whereas desktop GPUs support thousands of in-flight threads and feature RFs of several MBytes [68]. Note that the power budget is also completely different in both cases, desktop GPUs typically consume hundreds of Watts [79] while mobile GPUs have at their disposal about 1 Watt [120].

Figure 3.2 shows the speedups achieved when increasing the number of SIMD threads from 1 to 16 in a mobile GPU. The baseline configuration is an Immediate-Mode Renderer pipeline such as the one illustrated in Figure 2.2, with the parameters provided in Table 3.1 and just one SIMD thread per processor. As expected, multithreading provides significant performance improvements, with an average speedup of 1.92x for 16 SIMD threads. However, Figure 3.3 shows that GPU energy consumption also increases as we increase the number of parallel threads. For a small degree of multithreading (2-8 SIMD threads), the savings in static energy due to the speedups achieved are usually greater than the increase in dynamic energy due to the bigger RF. However, more aggressively multithreaded GPUs (10-16 SIMD threads) exhibit bigger energy consumption in most of the applications due to the big RF required to sustain all the parallel threads.

In short, multithreading is a very effective technique to hide the memory latency as it provides significant performance improvements. However, part of the energy savings achieved by the reduction in execution time are compensated by the increase in RF's dynamic energy consumption. We believe that more energy efficient approaches can be employed to tolerate memory latency in a mobile GPU. More specifically, we explore in the next sections the idea of combining multithreading with other techniques such as prefetching or decoupled access/execute, with the aim of achieving similar latency tolerance than massive multithreading but with a smaller number of thread contexts.

3.1.2 Hardware Prefetching

Prefetching data into the caches can help tolerate memory latency. Under prefetching, the hardware prefetcher is triggered in case of a cache miss in order to predict the next addresses that will be requested by the processor, usually by analyzing the miss address stream. The memory latency is then hidden by issuing prefetch requests to those addresses predicted by the prefetcher before they are actually needed. Hardware prefetchers require a relatively low amount of additional hardware.

The two largest concerns with prefetching are **accuracy** and **timeliness**. Inaccurately predicting the memory access stream can produce cache pollution and, furthermore, it increases the number of cache accesses and energy consumption. On the other hand, the prefetcher must be careful to not prefetch the data too early, so it is evicted before requested by the processor, or too late, so memory latency cannot be hidden.

The "aggressiveness" of a prefetcher can be characterized by the **prefetch degree**. The degree of prefetching determines how many requests can be initiated by one prefetch trigger. Increasing the degree can be beneficial, if the prefetched lines are used by the application, or harmful, if the prefetched lines are evicted before being accessed by the application or they are never accessed.

In next sections we review several state-of-the-art CPU prefetchers and we evaluate their performance when they are applied on top of a mobile GPU. Next

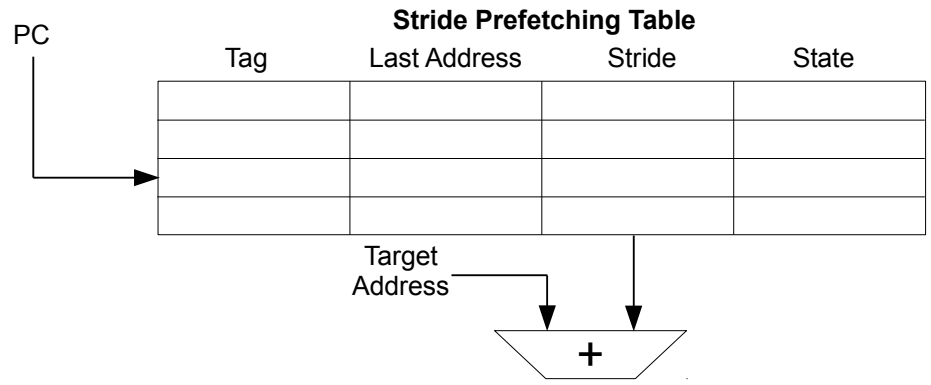


Figure 3.4: Hardware table employed by the Stride prefetcher.

we review a prefetcher specifically tailored to the texture caches of a graphics pipeline.

Conventional Prefetchers

The **Stride Prefetcher** [77] is one of the most commonly employed prefetchers. Conventional Stride Prefetching uses a table to store stride-related local history information, as illustrated in Figure 3.4. The program counter (PC) of a load instruction is employed to index the table. Each table entry stores the load's most recent stride (the difference between the two most recently pending load addresses), last address (to allow computation of the next local stride), and state information describing the stability of the load's recent stride behavior. When a prefetch is triggered, addresses $a + s$, $a + 2s$,..., $a + ds$ are prefetched (a is the load's current target address, s is the detected stride and d is the prefetch degree, an implementation dependent prefetch look-ahead distance).

Although prefetch tables require relatively simple hardware, they store prefetch history inefficiently. In first place, table data can become stale, and consequently reduce prefetch accuracy. In second place, tables suffer from conflicts that occur when multiple access keys map to the same table entry. The more obvious solution for reducing conflicts is to increase the number of table entries. However, this approach increases the table's memory requirements. In third place, tables have a fixed amount of history per entry. Adding more prefetch history per entry creates new opportunities for effective prefetching, but the additional history also increases the table's memory requirements.

A new prefetching structure, the **Global History Buffer**, is proposed in [114]. This prefetching structure decouples table key matching from the storage of prefetch-related history information. The overall prefetching structure has two levels, as shown in Figure 3.5:

- An Index Table that is accessed with a key as in conventional prefetch tables. The key may be a load instruction's PC, a cache miss address, or

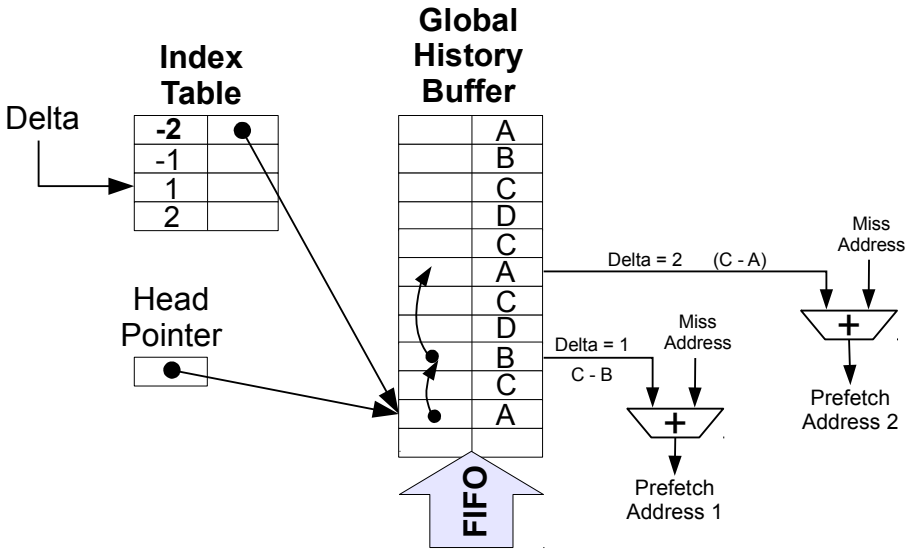


Figure 3.5: Distance prefetcher implemented by using a Global History Buffer. The Head Pointer points to the last inserted address in the GHB. Current delta is -2 (A-C), the last two times that delta -2 was observed by the prefetcher the next deltas were 1 (C - B) and 2 (C-A).

some combination. The entries in the Index Table contain pointers into the Global History Buffer.

- The Global History Buffer (GHB) is an n-entry FIFO table (implemented as a circular buffer) that holds the n most recent miss addresses. Each GHB entry stores a global miss address and a link pointer. Each pointer points to the previous miss address with the same Index Table Key. The link pointers are used to chain the GHB entries into address lists. Hence, each address list is a time-ordered sequence of addresses that have the same Index Table key.

Figure 3.5 illustrates how the GHB can prefetch by using a distance prefetching scheme [102]. This prefetching scheme uses the distance between two consecutive global miss addresses, an address delta, to access the Index Table. The pointer stored in the corresponding entry of the Index Table points to the list of recent miss addresses that have the same delta.

We have implemented in our GPU timing simulator the two previously described prefetchers, the table-based Stride prefetcher illustrated in Figure 3.4 and the GHB-based Distance prefetcher shown in Figure 3.5. We have evaluated the hardware prefetchers on top of the texture caches of an Immediate-Mode Rendering architecture like the one illustrated in Figure 2.2. The parameters employed for the evaluation, including the sizes of the prefetch hardware structures, are provided in Table 3.1.

Figure 3.6 shows the speedups achieved by the hardware prefetchers with different prefetch degrees from 1 to 8. As we can see, the prefetchers provide

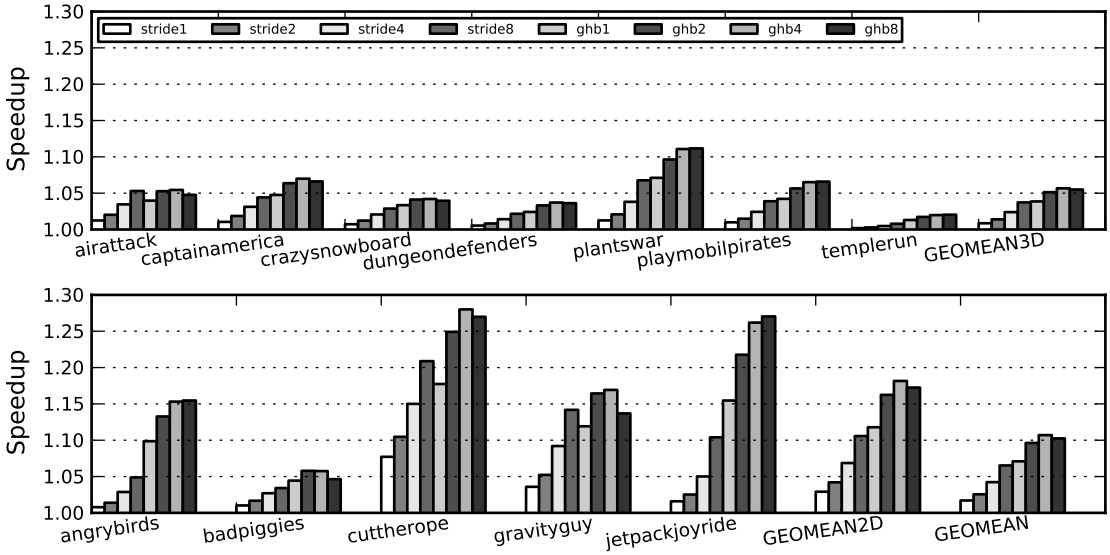


Figure 3.6: Speedups achieved by using Stride and GHB prefetchers with different prefetch degrees from 1 to 8. The baseline is a mobile GPU with just one SIMD thread per processor and no prefetching.

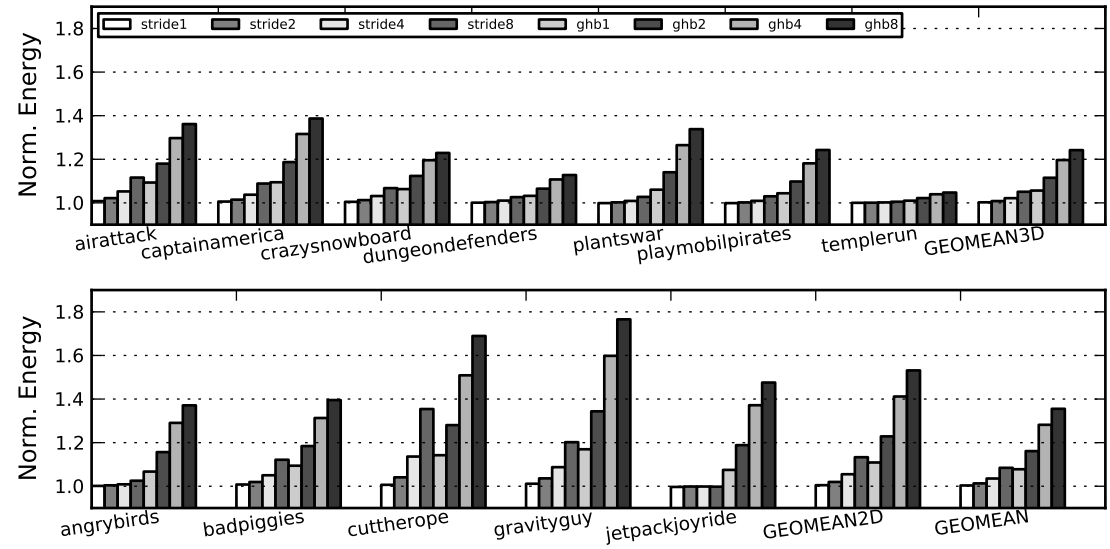


Figure 3.7: Normalized energy obtained by using Stride and GHB prefetchers with different prefetch degrees from 1 to 8. The baseline is a mobile GPU with just one SIMD thread per processor and no prefetching.

performance improvements in all the workloads, no slowdown was observed in any of the games. The GHB prefetcher obtains better results in most of the games and on average. Regarding the effect of the prefetch degree, for the stride prefetcher the bigger the degree the better, whereas the GHB obtains better results with a degree of 4 than with 8 for some workloads such as *airattack* or *gravityguy*. On average, the GHB achieves the best results with a degree of 4, 1.11x speedup, whereas the stride prefetcher achieves best average speedup with

a degree of 8: 1.07x. Despite the performance improvements provided by the hardware prefetchers, the results obtained are far away from the 1.59x speedup achieved by using perfect texture caches (see Figure 3.1), so there is still ample room for improvement.

The most important downside for hardware prefetching in a mobile GPU is its impact on energy consumption, illustrated in Figure 3.7. As we can see, the hardware prefetchers increase energy consumption in all the workloads. The increase in energy consumption is bigger as we increase the prefetch degree, as more memory requests are triggered on every cache miss. The stride prefetcher with a degree of 8 increases energy by 8.4% on average, whereas the GHB with a degree of 4 increases energy by 28.1%. As we have mentioned at the beginning of this chapter, texture accesses tend to be unpredictable and, hence, the accuracy of the hardware prefetchers is not as high as in CPU workloads with more regular memory access patterns. Serving the prefetch requests increases energy consumption, but not all of these requests contribute to hide the memory latency and improve performance due to the reduced accuracy.

In summary, conventional CPU prefetchers can be included in the texture caches of a mobile GPU to obtain moderate performance improvements. However, hardware prefetchers are not energy efficient since their accuracy is fairly constrained due to the unpredictable memory access patterns exhibited by graphics workloads.

Prefetching Architecture for Texture Caches

Igehy et al. [93] propose a custom prefetching architecture that takes advantage of the special access characteristics of texture mapping. This architecture is illustrated in Figure 3.8 and it works as follows. The rasterizer generates first the fragments by interpolation of the per-vertex attributes of the triangles. Next, the texel addresses that are required by each fragment are computed. Note that this is an important difference with respect to the hardware prefetchers previously described, as this architecture is based on computed addresses rather than predicted addresses to improve accuracy. The texel addresses are next looked up in the cache tags. If a tag check reveals a miss, the cache tags are updated with the fragment's texel address and the address is forwarded to the memory request FIFO. The cache addresses associated with the fragment are sent to the fragment FIFO and are stored together with the rest of the data required to process the fragment, such as screen coordinates or filtering information.

As the request FIFO issues memory requests for the missing cache blocks to the memory system, space is reserved in the reorder buffer to store the returning texture memory blocks. This guarantee of space avoids deadlocks in the presence of an out-of-order memory system.

When a fragment reaches the head of the fragment FIFO, it can only proceed if all its texels are available in the cache. Fragments that generated no misses can proceed immediately, whereas fragments that generated one or more misses

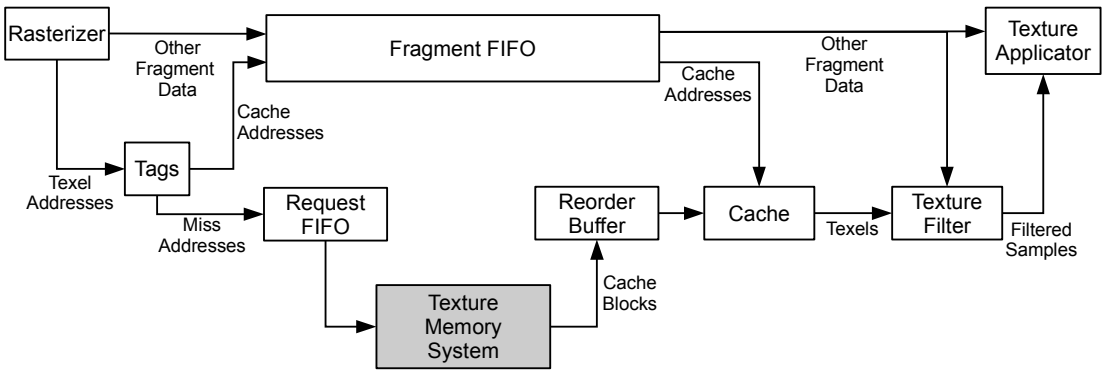


Figure 3.8: Prefetch architecture for texture caches proposed by Igehy et al. [93].

must first wait for their corresponding cache blocks to return from memory into the reorder buffer. In order to avoid new cache blocks overwriting yet-to-be-used older prefetched cache blocks, new cache blocks are committed to the cache only when their corresponding fragment reaches the head of the FIFO. Fragments that are removed from the head of the FIFO have their corresponding texels read from the cache and are further processed through the rest of the texture pipeline.

This prefetching architecture can tolerate increasing amounts of memory latency by increasing the size of the fragment FIFO and the storage capacity of the reorder buffer. Furthermore, it deals with one of the main issues of conventional hardware prefetchers: poor prefetching accuracy due to the unpredictable memory access patterns of graphics workloads. As predicting the texture miss address stream is a complex task, this prefetching architecture is based on computed memory addresses, instead of analyzing recent cache misses to predict the next addresses that will be requested. On the downside, fragments are forced to be processed in order. This means that the pipeline stalls whenever the fragment at the head of the FIFO has not its texture data available in the cache, even if younger fragments in the FIFO have their data ready.

The prefetching architecture for texture caches was proposed in 1998, before programmable GPUs came out, so it is implemented on top of a fixed-function graphics pipeline. Applying this architecture to a modern multicore programmable GPU is not straightforward for several reasons. First, the texture sampling units are usually integrated in the programmable fragment processors, as illustrated in Figure 2.4, and the texture fetches are triggered based on the programmed-by-the-user fragment program. Second, modern GPUs take advantage of the Early Depth Test to try to avoid the overdraw (see Figure 2.2) and, hence, fragments generated by the Rasterizer do not directly proceed to the texture pipeline as it is assumed in Figure 3.8. Third, mobile GPUs include multiple fragment processors with their corresponding texture caches, whereas the original proposal considers just one texture cache as it was the common baseline in 1998. A modern graphics pipeline has to be able to orchestrate texture prefetches to multiple texture caches in parallel, and dispatch the fragments to the appropriate fragment processors where their texels have been prefetched.

3.2 Decoupled Architecture for Fragment Processors

Conventional hardware prefetchers are not energy efficient as we have seen in section 3.1.2, since texture memory accesses tend to be difficult to prefetch [79]. Prefetching schemes designed for graphics pipelines [93] are based on computed addresses rather than predicted addresses to improve accuracy. This approach of computing the memory addresses in advance is similar to Decoupled Access/Execute architectures [138].

Traditionally, an access/execute architecture divides the program into two independent instruction streams, one doing memory accesses and the other performing actual computations. By decoupling memory accesses from computations, access/execute architectures effectively prefetch data from memory much in advance from the time it is required, thus allowing cache miss latency to overlap with useful computations without causing stalls. While this can be viewed as a form of data prefetching, it has a substantial advantage over other prefetching schemes, because it relies on computed rather than predicted addresses, which translates into a higher accuracy and a lower energy waste.

Despite the high potential of access/execute architectures to tolerate a long memory latency at a moderate hardware cost, they have not been widely adopted by current commercial CPUs because their effectiveness is greatly degraded when the computation of an address has a data or control dependence on the execution stream (this occurs, for instance, in pointer chasing codes). In such circumstances, termed *loss of decoupling events* (LOD), the access stream is forced to stall in order to synchronize with the execution stream. LODs force the access stream to lose its timeliness (i. e. the prefetch distance), so that subsequent cache misses will cause the execution stream to stall as well. Unfortunately, for general purpose CPUs the frequency of LODs is quite significant in many cases, resulting in fairly restricted performance gains. However, for GPU fragment programs, the access patterns are typically free of the dependences that cause LODs. This makes the access/execute paradigm a perfect fit for the requirements of a low-power high-performance GPU: with few extra hardware requirements, it can reduce drastically the number of cache miss stalls.

3.2.1 Base Architecture

We propose to employ a decoupled access/execute-like design for the fragment processors of a mobile GPU, as shown in Figure 3.9. After the visibility determination in the Early Depth Test stage (see Figure 2.2), visible fragments are packed into quad fragments, or quads for short. A fragment processor is assigned to each quad by the scheduler, and both the quad and the processor number are inserted into the quad fragment queue to wait its turn until it is dispatched to the processor. Part of the information stored in this quad fragment queue, the

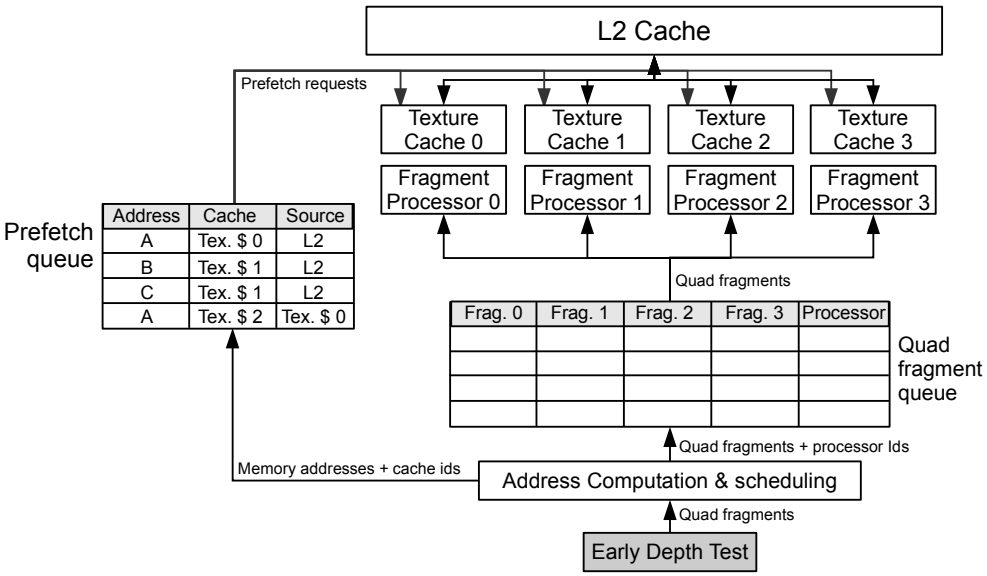


Figure 3.9: Decoupled architecture for the fragment processors.

texture coordinates, will be later used by the fragment processor to compute the addresses that will be issued to the texture cache.

The proposed scheme decouples the memory addresses from the quad fragment queue, so that memory requests for a specific quad can be issued while the quad is still waiting in the queue. This behavior is achieved by inserting all computed addresses of a quad along with their target cache number, into a new queue, the prefetch queue. Notice that the scheduling is performed before queuing the quad, so that the prefetcher knows to which caches it must send the requests. We assume that a new request from the prefetch queue is sent to the corresponding cache every cycle until the queue is drained. For each request, the corresponding cache controller will check the tags (by using a dedicated snoop port), and the request will be either ignored in case of a hit, or trigger a preemptive block fetch to L2, and a subsequent texture cache update. Note also that the proposed scheme performs in-cache prefetching, instead of prefetching into specialized buffers.

By the time a quad is dispatched to the fragment processor, the data required to process the fragments are usually available in the texture cache, so that almost all processor cache accesses hit. Should the prefetch requests not be issued enough in advance, the fragment processor would experience a cache miss stall. This would cause the quad fragment queue to fill up, but it would also allow the prefetcher to increase the prefetch distance again, thus avoiding further stalls. The quad fragment queue must be sized long enough -this mostly depends on miss latency- to allow the prefetcher to gain sufficient prefetch distance to achieve timeliness. But it must also avoid excessive length that could lead to late requests evicting yet-to-be-used prior prefetched data, due to cache conflicts. We have found that lengths between 4 and 32 are appropriate for our workloads. It is also important to design the prefetcher with sufficient throughput to avoid losing the prefetch distance, so we assume it is non-blocking, i. e. multiple pending blocks may be in-flight at any time. The necessary control information for each pending

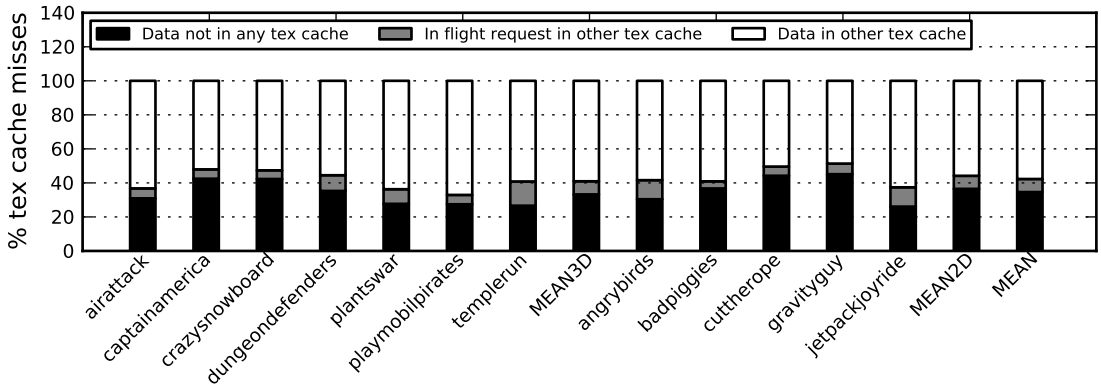


Figure 3.10: The figure illustrates the high degree of data replication among the texture caches of the different fragment processors. The black bars represent texture cache misses to data that is not available in any of the texture caches. The gray bars are misses to data not available in any cache, but with an already triggered in-flight request in some of the texture caches. The white bars are misses to data that is available in another texture cache.

block is held in the prefetch queue.

3.2.2 Remote Texture Cache Accesses

Partitioning the texture cache among the various fragment processors reduces the size of each individual cache and the power required per access, but also produces some degree of replication. The results obtained by using our GPU timing simulator and a commercial set of Android games show a significant degree of texture data sharing among fragment processors, as illustrated in Figure 3.10. On average, 57.7% of the texture cache misses are requests to data that is already available in the texture cache of some other fragment processor. Hence, the decoupled access/execute-like architecture previously described can be further improved because up to 57.7% of the prefetch misses can be satisfied by the texture cache of another fragment processor instead of accessing the bigger L2 cache. This improvement saves bandwidth to the shared L2 cache and reduces energy consumption since accessing to the L2 cache is more expensive than accessing a texture cache.

A naive approach could check the tags of all the caches at the time a prefetch request is dispatched to know which caches could satisfy the request in case of a miss. While this would provide precise information, it would also have a significant energy cost. Instead, we propose to take advantage of the temporal locality that exists in the memory requests in order to achieve similar performance with only a small fraction of the energy. More specifically, we propose to augment each entry in the prefetch queue with a new field, called *Source*, that will hold the predicted alternate location of the block. Each time a new prefetch request is inserted in the queue, the addresses of all the queued requests are associatively compared with the new address. If there is a match with a prefetch request for a different cache, the identifier of this cache is recorded in the *Source* field of the

new entry. If there is no match, the identifier of the L2 cache is recorded instead. When the prefetch request is dispatched to its target cache the *Source* field is included in the request. This information is then used by the cache controller to redirect the request in case of a cache miss. Figure 3.9 shows an example of this behavior, in this case the system has detected a match with an older request for address A and the *Source* field of the last entry in the prefetch queue points to texture cache 0.

Compared with the full tag check approach mentioned earlier, this technique only looks in a small window of recent requests (equal to the number of pending requests held in the queue). Alternatively, we could implement the queue as a circular buffer, where the entries between the head and the tail are considered active and the rest are not. In this case, provided that the inactive entries are not cleared, they still hold the addresses and target cache identifiers. Each new request can then be compared against "all" the queue entries, either active or not, thus widening the window of recent requests to the total length of the queue.

We have implemented the decoupled architecture for the fragment processors in our GPU simulator. We have evaluated our proposal on top of an Immediate-Mode Rendering architecture as the one illustrated in Figure 2.2. The parameters for the experiments, including the size of the prefetch queue and the number of fragment processors, are provided in Table 3.1.

Figure 3.11 shows the speedups achieved in a mobile GPU with single-threaded fragment processors. The *decoupled* configuration shows the results for our base decoupled architecture, whereas the *decoupled_remote* includes the remote texture cache accesses previously described. Multithreading was not employed in any of the experiments, just prefetching or decoupled access/execute. As we can see, the decoupled versions achieve bigger performance improvements than the hardware prefetchers in all the workloads. On average, the GHB prefetcher achieves 1.11x speedup, whereas the decoupled architecture with remote texture cache accesses obtains 1.41x speedup. The performance benefits are due to the improved accuracy, as the decoupled systems are based on computed rather than predicted addresses.

On the other hand, the proposed optimization of allowing for remote texture cache accesses does not provide any significant performance improvement. Nevertheless, its main target is to save energy by replacing accesses to the bigger L2 cache with accesses to the smaller texture caches. Figure 3.12 provides the energy results. The decoupled systems provide consistent energy savings in all the workloads, being much more energy efficient than the hardware prefetchers. In addition, the remote texture cache accesses provide further energy savings in all the games. On average, the decoupled fragment processor achieves 9.6% energy savings, whereas the decoupled system with remote accesses improves energy by 20%.

Our decoupled access/execute-like architecture is based on the ideas of the prefetching architecture for texture caches proposed by Igehy et al. [93] (see section 3.1.2), as our system is also based on computed addresses rather than

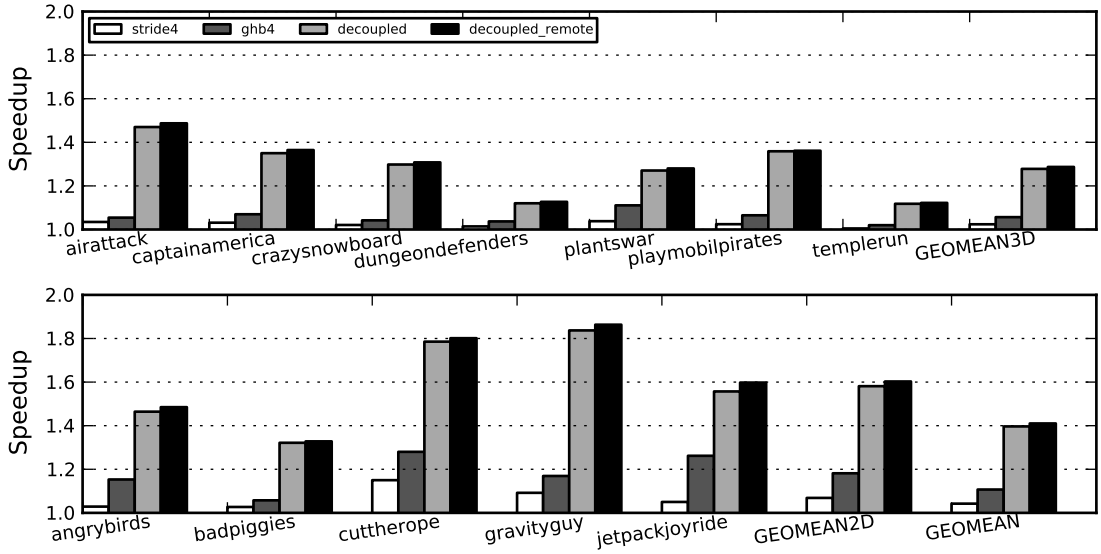


Figure 3.11: Speedups achieved by different hardware prefetchers and our decoupled access/execute-like architecture. The baseline is a GPU with single-threaded fragment processors and no prefetching. The decoupled_remote configuration allows for remote texture cache accesses.

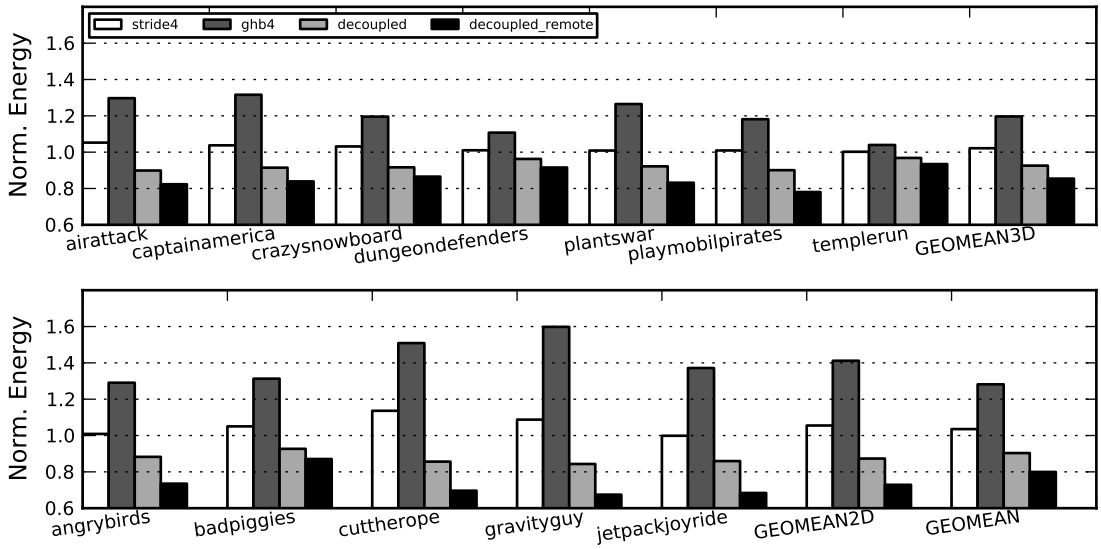


Figure 3.12: Normalized energy for different hardware prefetchers and our decoupled access/execute-like architecture. The baseline is a GPU with single-threaded fragment processors and no prefetching.

predicted. However, our work is different in several ways. First, our proposal is built on top of a modern programmable GPU, and it is able to coordinate texture fetches in an environment with multiple fragment processors. Second, our system prefetches texture data after the visibility test, so we only issue prefetch requests for visible fragments instead of prefetching for all the fragments generated by the Rasterizer, increasing energy efficiency. Third, we do not require the fragments to

Table 3.1: Parameters employed for the experiments.

Main memory	1 GByte, 16 bytes/cycle, 50-100 cycles		
Texture caches	16 KB, 4-way	**L2 cache**	256 KB, 8-way
Fragment Processors	4	**Multithreading**	1-16 SIMD threads
Prefetch degree	4	**SIMD width**	4 threads
Decoupled access/execute		**Register File**	
Prefetch queue size	16 entries	**Register size**	16 bytes (4-FP)
Entry size	40 bits	**Registers/thread**	36
Total size	80 bytes	**Storage/thread**	576 bytes
Stride prefetcher		**GHB prefetcher**	
Stride table size	16 entries	**Index table**	16 40-bit entries
Entry size	98 bits	**GHB**	64 40-bit entries
Total size	196 bytes	**Total size**	400 bytes

be processed in-order, avoiding stalls in case younger fragments have their texels available while older fragments are still waiting for the data. Finally, our proposal allows for remote texture cache requests, which is shown to provide significant energy benefits.

3.3 Multithreading, Prefetching and Decoupled Access/Execute

In prior sections we have shown that massive multithreading is an effective technique to hide the memory latency, but not energy efficient due to the big size of the RF. Furthermore, we have seen that conventional hardware prefetchers provide modest performance improvements due to the poor accuracy. Finally, we have demonstrated that a decoupled access/execute design achieves much bigger energy efficiency than the hardware prefetchers in a single-threaded mobile GPU. In this section we analyze the performance and energy consumption of both hardware prefetching and decoupled access/execute when using different degrees of multithreading, from 1 to 16 SIMD threads per fragment processor.

The baseline configuration is a mobile GPU with an Immediate-Mode Rendering architecture (see Figure 2.2) and the parameters shown in Table 3.1. The baseline GPU supports just one SIMD thread per fragment processor, multithreading or prefetching are not employed to hide the memory latency. We analyze five different schemes implemented on top of this baseline. The first configuration, "Multithreading only", relies on multithreading to hide the memory latency. The second configuration, "Stride", includes a stride prefetcher, illustrated in Figure 3.4, in each one of the texture caches. The third configuration, "GHB", employs a distance prefetcher implemented with a Global History Buffer, shown in Figure 3.5. Both hardware prefetchers employ a prefetch degree of 4. The fourth configuration, "Decoupled", implements our decoupled access/execute design for the fragment processors of a mobile GPU, as described in Section 3.2.1.

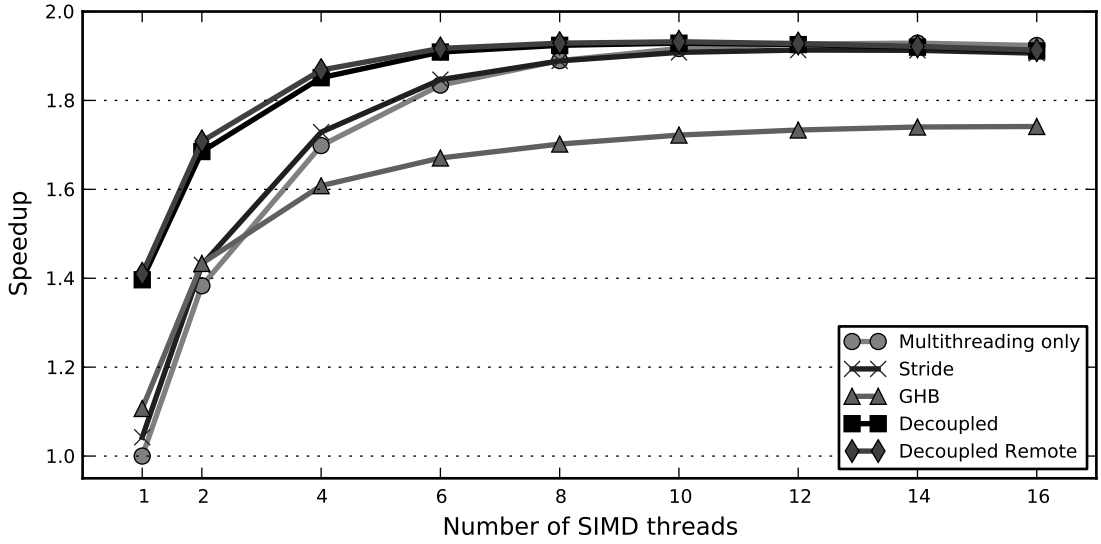


Figure 3.13: Speedups achieved when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching.

Finally, "Decoupled Remote" includes our decoupled architecture and it also implements the optimization that allows for remote texture cache accesses described in Section 3.2.2.

Figure 3.13 shows the speedups achieved by the different configurations, when using multiple degrees of multithreading from 1 to 16 SIMD threads. As we can see, the decoupled schemes achieve bigger performance than the hardware prefetchers when using a small degree of multithreading, from 1 to 8 SIMD threads, whereas the performance converges in most of the configurations when using aggressive multithreading (10-16 SIMD threads). The decoupled design achieves 97.6% of its maximum performance with just 4 SIMD threads, whereas the "Multithreading only" configurations requires 8 SIMD threads to be close to its maximum speedup. The stride prefetcher exhibits similar behavior to the "Multithreading only" configuration. The GHB prefetcher shows worse scalability as it makes more intensive use of memory bandwidth. The graph also shows that the single threaded decoupled architecture can still improve performance by taking advantage of a small degree of multithreading (1-4 SIMD threads). This small amount of threads allows the architecture to hide the latency of the functional units and keep them busy, which is an issue that the access/execute decoupling mechanism does not address. It is worth noting that similar conclusions about the synergy of decoupling and multithreading were already suggested in [122].

Figure 3.14 shows the normalized energy obtained by the different memory latency tolerance schemes. The configurations exhibit significantly different energy consumption, but all the configurations evolve in a similar way when increasing the number of SIMD threads. The first steps (1-6 SIMD threads) provide energy savings, as the reduction in static energy due to the speedups achieved is

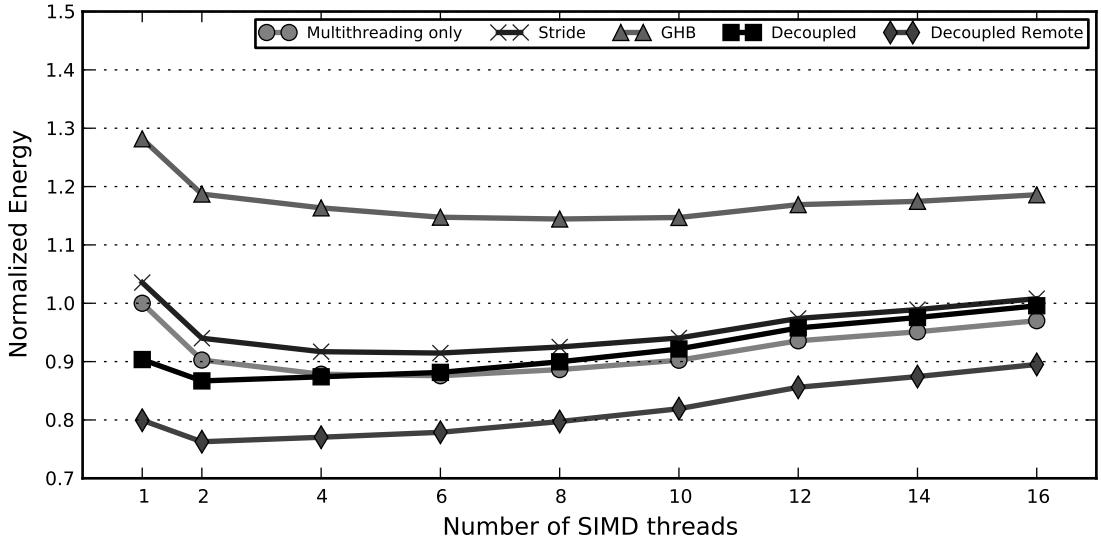


Figure 3.14: Normalized energy obtained when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching.

usually bigger than the increase in RF's dynamic energy. However, the energy consumption increases for bigger degrees of multithreading (8-16 SIMD threads) due to the huge RF required to store the registers of all the parallel threads. The GHB prefetcher exhibits the biggest energy consumption, mainly because the intensive use of the memory bandwidth produced by the huge number of additional memory requests triggered by the prefetcher. The "Multithreading only" and the stride prefetcher exhibit similar behavior, being the energy consumed by the prefetcher bigger due to the additional prefetch requests. The "Decoupled" configuration consumes less energy than the "Multithreading only" GPU for a small degree of multithreading, but the energy consumption converges beyond 2 SIMD threads. Finally, the "Decoupled Remote" configuration exhibits the smallest energy consumption for any number of threads, being able to save more than 20% energy when using between 1 and 8 SIMD threads. The remote texture fetches are an effective way of saving energy, as in case of a texture cache miss the fragment processor can read the texels directly from the 16 KBytes texture cache of another fragment processor, instead of accessing the bigger 256 KBytes L2 cache.

Figure 3.15 shows the GPU energy breakdown for configurations with different numbers of SIMD threads, from 1 to 16 SIMD threads/processor. The energy consumed by the Register File increases significantly as we increase the degree of multithreading. The Register File represents just 3.6% of GPU energy for the "Multithreading only" configuration with 1 SIMD thread/processor, whereas it accounts for 17.4% of GPU energy when using 16 SIMD threads/processor. On the other hand, the "Decoupled Remote" configuration is able to reduce the energy consumed by the L2 cache. On average, L2 cache energy is reduced by 20.6% when using the optimization that allows for remote texture cache accesses.

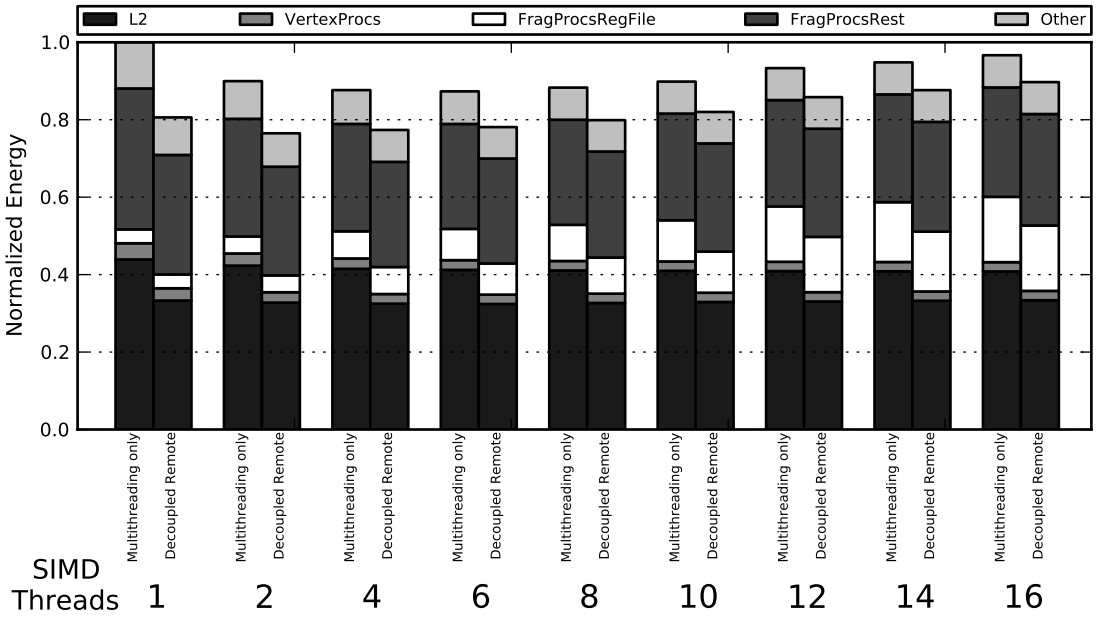


Figure 3.15: Mobile GPU energy breakdown.

Figure 3.16 shows the normalized energy-efficiency. Energy-efficiency of a configuration is defined as the ratio of speedup divided by the normalized energy consumption [70]. This metric reflects both performance improvements and energy savings in a single number, the bigger the speedup and the smaller the normalized energy the better the energy-efficiency. The "Decoupled" configuration obtains better energy-efficiency than the "Multithreading only" and the hardware prefetchers between 1 and 6 SIMD threads. The "Decoupled Remote" configuration achieves bigger energy-efficiency than the rest for any number of threads. This schemes requires a smaller number of thread contexts to achieve high levels of energy-efficiency. For example, the "Decoupled Remote" with just 4 SIMD threads per fragment processor obtains 14% and 22% bigger energy-efficiency than the "Multithreading only" using 8 and 16 SIMD threads respectively.

Previous graphs show average results obtained for our 12 Android commercial games. We believe it is also interesting to analyze the individual results, Figure 3.17 shows the normalized energy-efficiency for each one of the workloads. The "Decoupled Remote" configuration achieves consistent improvements in energy-efficient in all the workloads. It achieves its highest energy-efficiency by using between 4 and 6 SIMD threads in all the games except in *playmobilpirates*, where it achieves optimum energy-efficiency with 10 SIMD threads. The energy-efficiency results are different depending on the workload, as the "Decoupled Remote" with 4 SIMD threads obtains 1.43x and 3.72x improvements in energy-efficiency in *dungeondefenders* and *cuttherope* respectively, the rest of the games lie in between. Note that the results of the "Decoupled Remote" are significantly better than the "Multithreaded only". For example, a larger GPU with 16 SIMD threads is only able to achieve 1.24x and 3.08x improvements in energy-efficiency in *dungeondefenders* and *cuttherope* respectively.

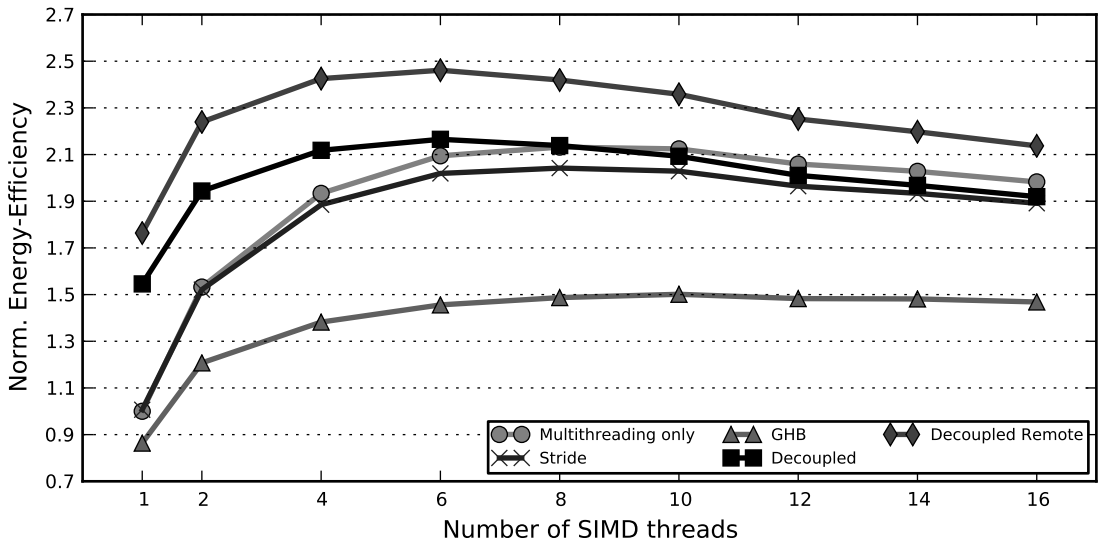


Figure 3.16: Normalized energy efficiency obtained when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching. The energy-efficiency is computed as the speedup divided by the normalized energy.

3.4 Conclusions

The main ambition of this chapter is to demonstrate that high-performing, energy-efficient GPUs can be architected based on the decoupled access-execute paradigm. The proposed scheme does not rely on heavy multithreading so as to hide the memory latency. Although multithreading is still useful, we believe that a significant part of its benefits can be achieved in a more energy efficient fashion. In fact, as it was shown in Section 3.3, a combination of access/execute with multithreading provides the most energy efficient solution. More specifically, we claim that it is better to hide the memory latency using the decoupled access/execute paradigm and hide the functional unit latency using a low degree of threading.

We evaluate the proposed scheme using a set of commercial Android applications and we show that the end decoupled access/execute design with 4 SIMD threads/processor is able to achieve 97% of the performance of a larger GPU with 16 SIMD threads/processor, while providing 20.5% energy savings.

In this chapter we assume an Immediate-Mode Rendering pipeline as our baseline GPU. Nevertheless, our proposal is orthogonal to the type of rendering architecture and it can also be implemented on top of Tile-Based Rendering (TBR). Similar conclusions are obtained when using our decoupled access/execute-like architecture on top of TBR. The experimental results assuming a TBR baseline are provided in Appendix A. The numbers show that the decoupled architecture with just 2 SIMD threads/processor achieves 93% of the performance of a larger GPU with 16 SIMD threads/processor, while providing 28.2% energy savings.

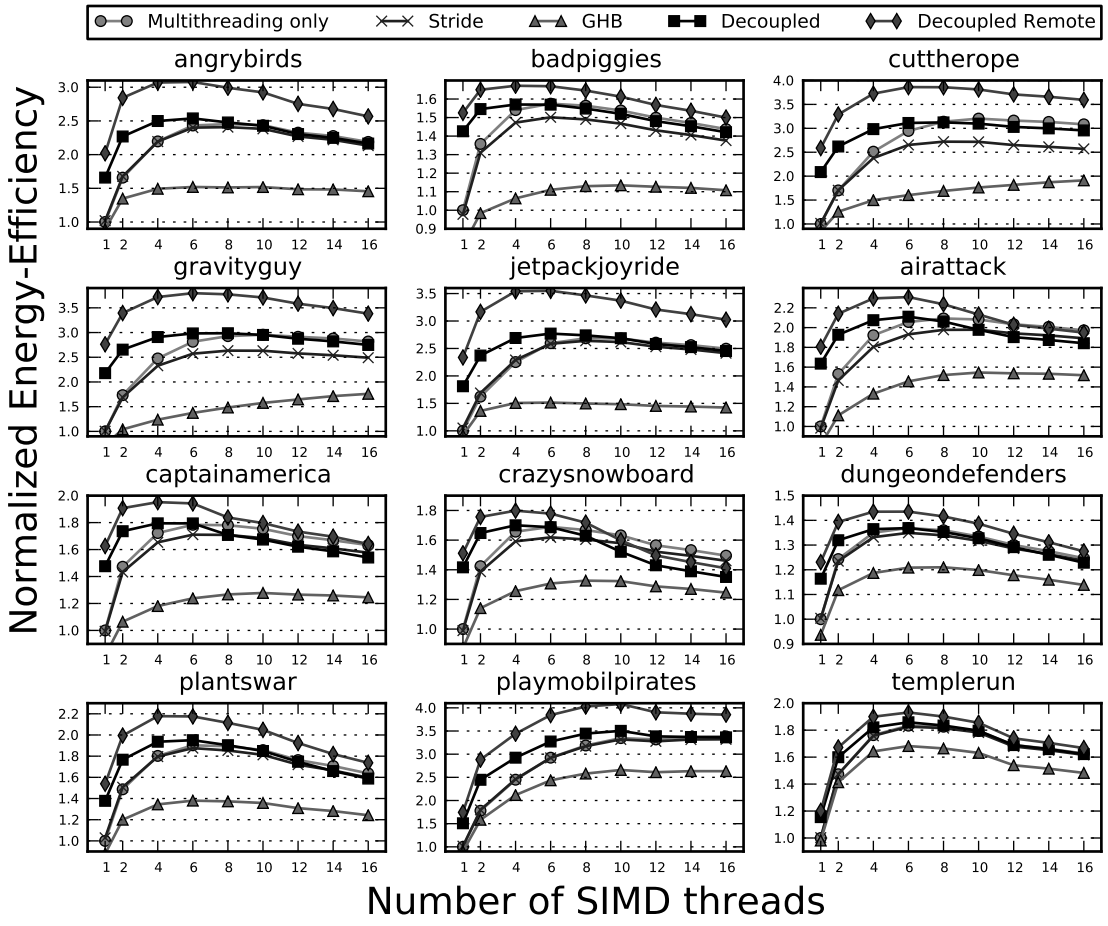


Figure 3.17: Normalized energy efficiency obtained for each one of the workloads. The energy-efficiency is computed as the speedup divided by the normalized energy.

Chapter 4

Parallel Frame Rendering

In this chapter we address the problem of optimizing the memory bandwidth usage in a mobile GPU, as saving bandwidth has been proven to be an effective way of reducing energy consumption. In first place, we analyze the memory bandwidth usage for a set of commercial Android games. In second place, we propose Parallel Frame Rendering, a bandwidth saving technique for mobile graphics processors. Finally, we show that this new technique achieves 23.8% bandwidth savings on average when architected on top of a state-of-the-art mobile GPU, providing 20.1% energy savings.

4.1 Memory Bandwidth Usage on a Mobile GPU

Recent work in [44] revealed that a large fraction of mobile GPU energy consumption can be attributed to external off-chip memory accesses to system RAM. As noted by [87, 119], most of these accesses fetch textures. Figure 4.1 depicts the memory bandwidth usage on a mobile GPU, similar to ARM Mali (see Figure 2.2), for a set of commercial Android games. Our numbers clearly support the prior claims and show that 52.7% of these memory accesses can be directly attributed to texture data. Ideally removing these accesses would result in significant energy savings, while also improving performance substantially.

Focusing on the texture data used by successive frames, we realized that there exists a large degree of reuse across frames. As shown in Figure 4.2 consecutive frames share 94% of the texture addresses requested on average when considering a window of 2 frames. Hence, the same textures are fetched frame after frame, but the GPU cannot exploit these frame-to-frame re-usages due to the huge size of the texture dataset. This is confirmed by inspecting the average re-use distance of the memory accesses (Figure 4.3). Contrary to common belief that GPUs need to deal with the streaming memory behavior of graphical applications, we argue it is perhaps more efficient to change the rendering model leading to this memory behavior.

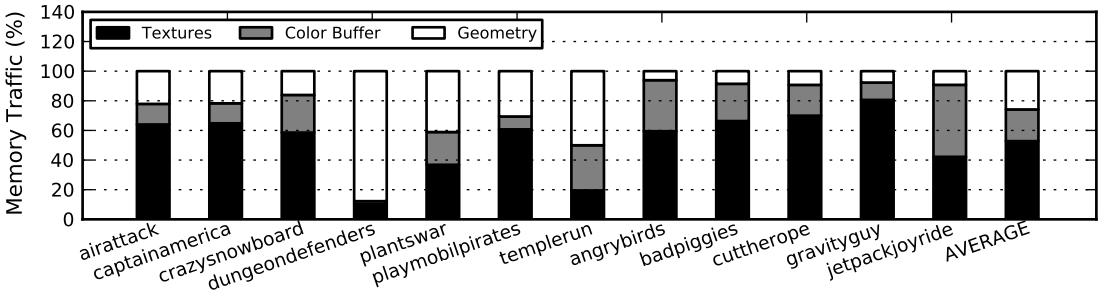


Figure 4.1: Memory bandwidth usage on a mobile GPU for a set of commercial Android games. On average 52.7% of the bandwidth to system memory is employed for fetching textures.

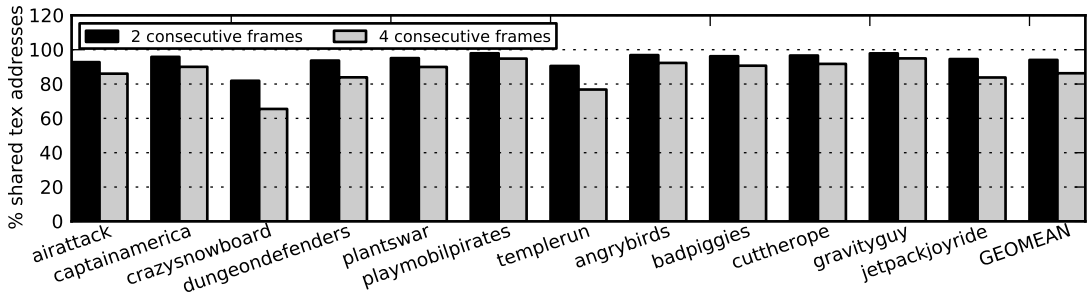


Figure 4.2: Average percentage of shared texture addresses between consecutive frames. Mostly the same texture dataset, 94% on average, is employed from frame to frame since consecutive frames tend to be very similar. Even considering a window of 4 consecutive frames we observe a high degree of texture sharing, 86.2% on average.

In this chapter we propose Parallel Frame Rendering (PFR), a novel technique for improving texture locality on a mobile GPU. In PFR the GPU is split in two clusters where two consecutive frames are rendered in parallel. By using this organization each texture is read from memory once and employed for rendering two successive frames. Therefore, textures are fetched from main memory just once every two frames instead of being fetched on a frame basis as in conventional GPUs. If perfect texture overlapping between both GPU clusters is achieved, then the amount of memory bandwidth required for fetching textures is reduced by 50%. Since textures represent 52.7% of memory bandwidth for our set of commercial Android games as illustrated in Figure 4.1, PFR can achieve up to 26.35% bandwidth savings.

4.2 Trading Responsiveness for Energy

Traditionally GPUs have high memory bandwidth requirements as they need to fetch from memory a very large dataset (textures), which typically thrashes the cache. As shown earlier, texture data locality exists, however most of it can only be exploited across frames. In fact, we have observed that 94% of the texture data is shared between consecutive frames. Most of these accesses miss in cache

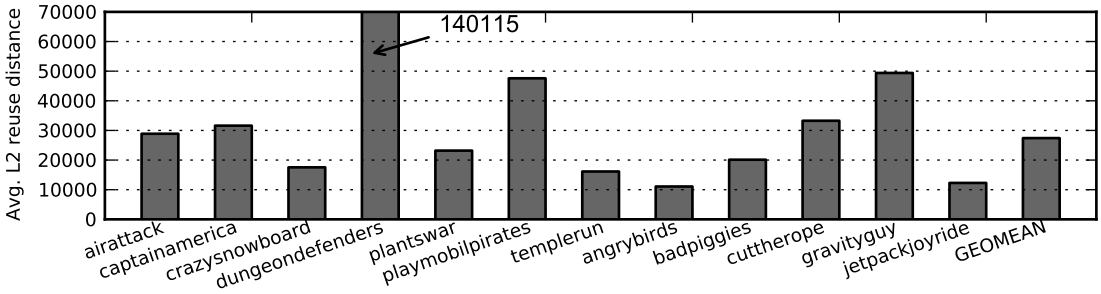


Figure 4.3: Average block reuse distances in the L2 cache for several Android commercial games, computed as number of distinct block addresses referenced between two accesses to the same block.

due to the huge working set involved in rendering every single frame, i. e. they are capacity misses.

Fetching from memory is one of the main sources of energy inefficiency for mobile devices [44, 87, 119], and as such improving the inter-frame locality will likely lead to significant improvements in energy efficiency.

4.2.1 Parallel Frame Rendering

The idea of Parallel Frame Rendering (PFR) consists on rendering two consecutive frames in parallel. If the two rendering tasks are well synchronized then they will hopefully perform similar operations in a similar order, so they will access the same textures within a small reuse distance. If that distance is small enough, the texture access of the first frame will perhaps miss in cache but will result in a cache hit when accessed by the second frame. Ideally, if both frames perform all the same accesses, and data is effectively kept in cache until reused by the second frame, we may expect a texture miss ratio reduction by one half that translates to a large reduction in memory traffic and energy waste.

The assumed baseline GPU implements a Tile-Based Rendering (TBR) architecture, as illustrated in Figure 4.4. To render two frames in parallel, we split the baseline GPU into two separate clusters, cluster 0 and cluster 1, each including one half of the resources, as illustrated in Figure 4.5. Even frames are then rendered by cluster 0, odd frames by cluster 1. Fragment Processors have private first level caches, but the second level cache is shared among all processors of both clusters, so we expect that one cluster prefetches data for the other cluster as discussed above. To further reduce the re-use distance between texture accesses of parallel rendered frames, the two clusters are synchronized by processing tiles in lockstep, since the same screen tile usually employs a very similar texture dataset in consecutive frames.

Figure 4.6 illustrates the difference between conventional rendering and PFR. To simplify the discussion, let us assume that conventional rendering fits in one screen refresh interval, then PFR spans two refresh intervals since each frame is rendered with half the number of processors. Notice that two frames are rendered

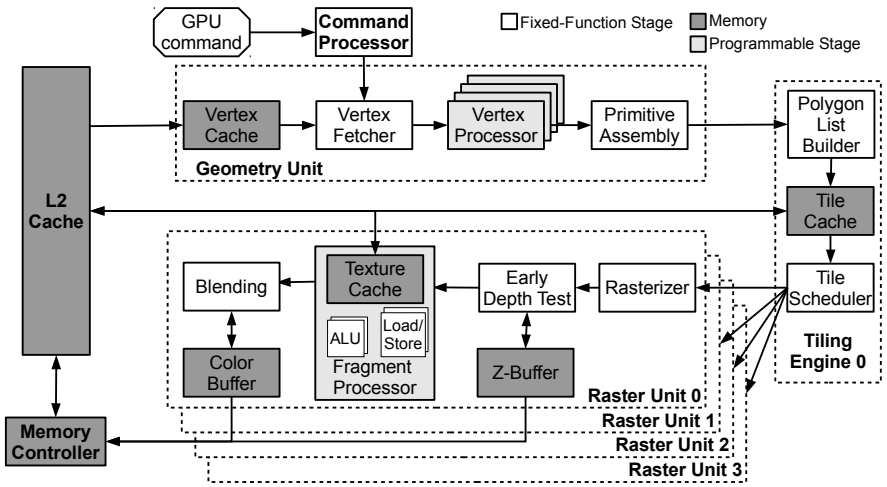


Figure 4.4: Assumed baseline GPU architecture. The baseline GPU employs Tile-Based Rendering and it features four Raster Units to process four screen tiles in parallel.

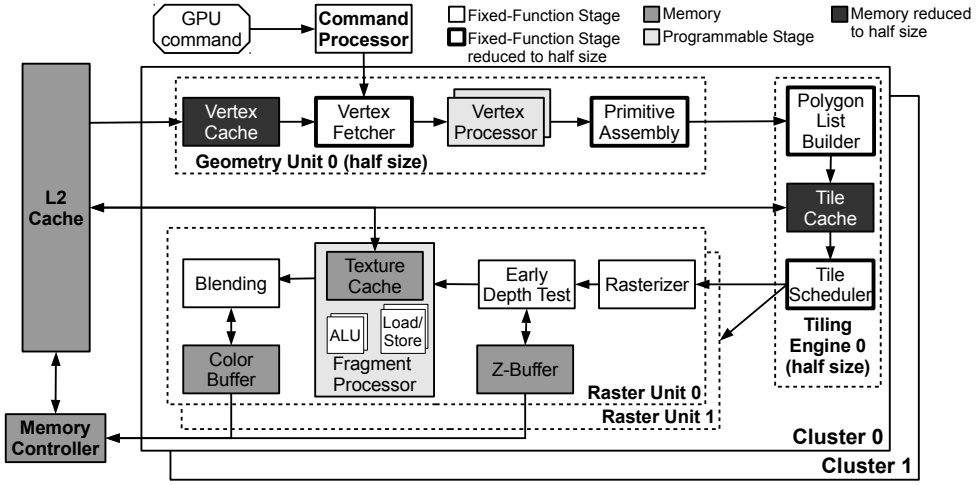


Figure 4.5: GPU architecture employed for implementing Parallel Frame Rendering. The Geometry Unit and the Tiling Engine are reduced to half of their original size.

in parallel, so the frame rate is still the same. With PFR, two consecutive frames are processed simultaneously, so two framebuffers are required instead of one. To avoid flicker or tearing, double buffering is usually employed in conventional rendering, which requires a separate frontbuffer for displaying the rendered image to the screen while the next frame is being drawn to the backbuffer. With PFR, double buffering requires two frontbuffers and two backbuffers. Hence, PFR increases the memory footprint. For a typical smartphone screen resolution of 800×480 (WVGA) and 32 bits per pixel (RGBA), conventional rendering requires 2.92 MBytes of main memory for storing the front and back Color Buffers, then PFR requires 5.84 MBytes. Since current smartphones usually feature 1GB of main memory, the memory footprint increment can be easily assumed. Notice that the bandwidth employed for transferring the Color Buffer to system memory is not increased despite two images are generated in parallel. In PFR

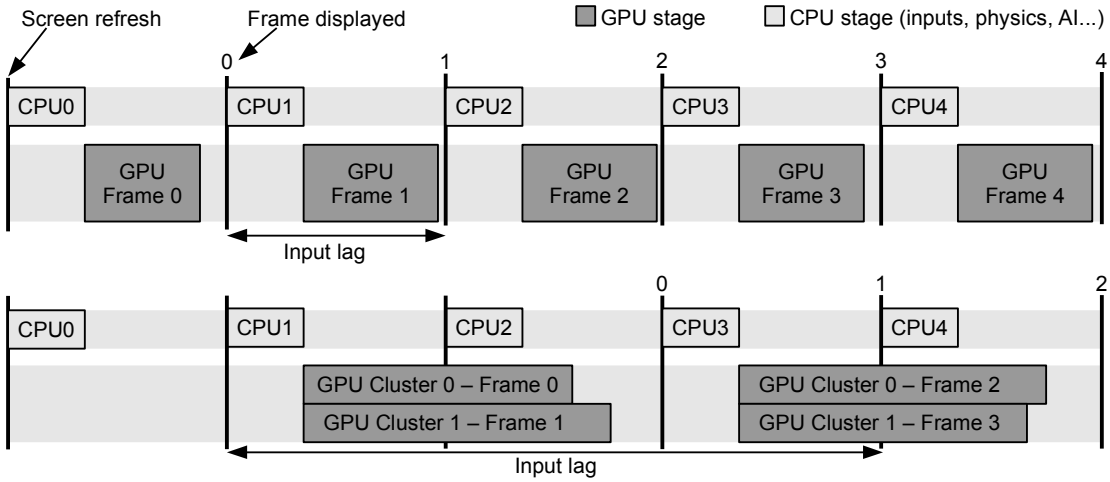


Figure 4.6: Conventional rendering (top) vs Parallel Frame Rendering (bottom). Rendering a frame in a GPU cluster requires twice of the time since each cluster has half of the resources of the conventional GPU.

two different Color Buffers are written into memory during two screen refreshes, whereas in conventional rendering the same Color Buffer is written twice during two refresh intervals. Therefore, the same amount of pixels are transferred in the same amount of time in both cases.

In conventional Tile-Based Rendering (TBR), described in Section 2.1.3, an entire frame has to be captured and the rendering is deferred until all the drawing commands for one frame have been issued by the application. In PFR the GPU driver has to be slightly modified, since two frames have to be buffered instead of one and the rendering is triggered when all the drawing commands for the second frame have been issued by the application and buffered by the GPU driver. Once the rendering of the two frames starts, the graphics hardware reads commands from both frames in parallel. The GPU Command Processor dispatches commands from the first frame to cluster 0 and commands from the second frame to cluster 1. Each GPU cluster behaves as an independent GPU, rendering the frame by using conventional TBR as described in Section 2.1.3.

Moreover, the GPU driver has to allocate two front buffers, two back buffers if double buffering is employed, and two scene buffers for sorting triangles into tiles (see Section 2.1.3). As previously mentioned, the memory footprint is augmented, but the memory bandwidth requirements are not increased, because rendering each frame takes twice of the time.

4.2.2 Reactive Parallel Frame Rendering

A valid concern for the PFR approach is that since it devotes half the amount of resources to render each frame, the time required to render a frame is longer. It is important to point out at this point that the frame rate is not reduced, since only the time between input and display, also known as input lag, increases (see

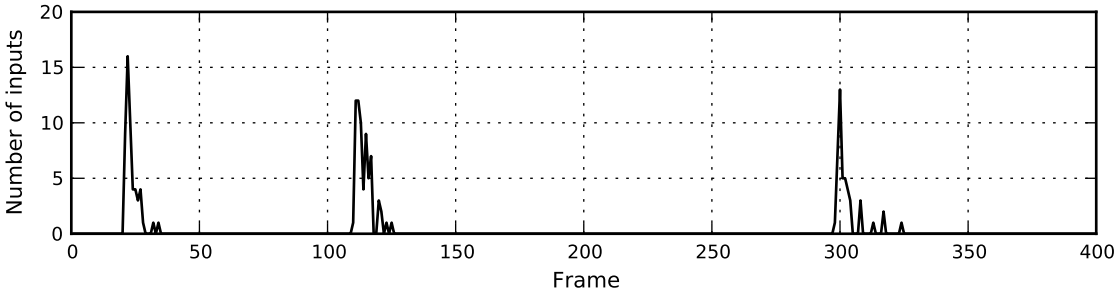


Figure 4.7: Number of inputs provided by the user for 400 frames of Angry Birds. User inputs are heavily clustered, the game exhibits phases that require high responsiveness and phases where no user input is provided.

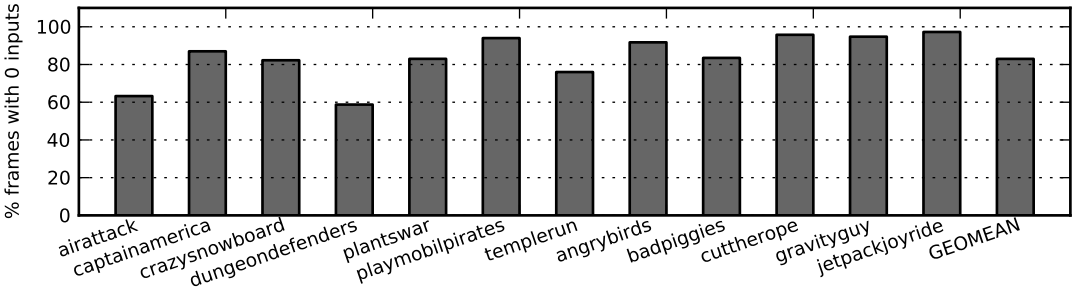
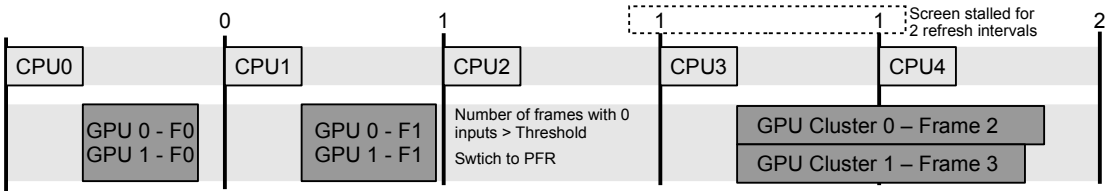


Figure 4.8: Percent of frames where no user inputs are provided for several Android games. The user does not provide any input most of the time.

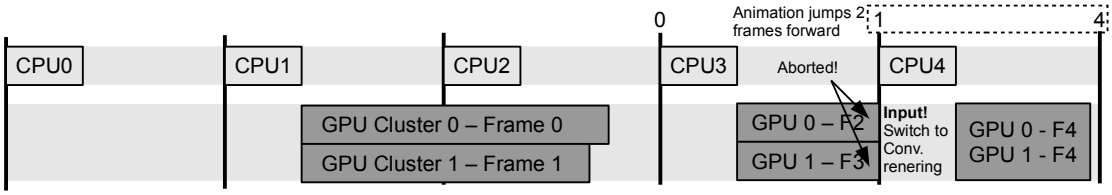
Figure 4.6). However, this is an unfortunate side-effect as it ultimately leads to a less responsive system from the end-users perspective.

Assuming a typical frame rate of 60 FPS, conventional rendering exhibits an input lag of 16 ms (1 screen refresh) whereas in PFR it is increased to 48 ms (3 refresh intervals). Overall, due to the nature of the user interface in mobile devices (touch screens are quite slow), we argue that this will mostly not be noticed by the end-user. In fact most applications tend to require very little input from the user for this exact reason (i.e., Angry Birds is a prime example of this behavior). In fact, this behavior is very common in mobile graphical applications, since they are designed in such a way that the user provides some inputs and then it just observes how the simulation evolves during the following frames in response to the user interaction (Figure 4.7). For these kind of applications PFR can be widely employed without hurting responsiveness during phases of the application where no user input is provided.

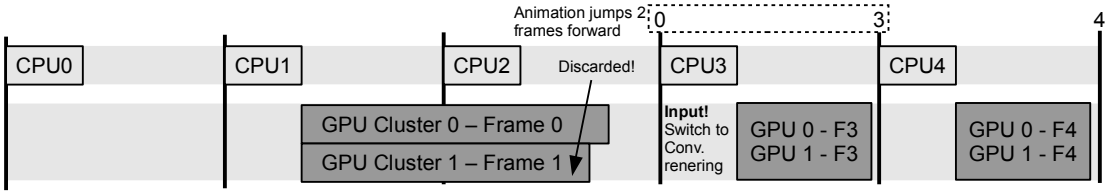
Nevertheless, there exist applications for which high responsiveness is required. As pointed out in [131], lags bigger than 15 ms can be noticeable and produce errors in interaction for applications demanding high responsiveness. In order to maintain the same levels of responsiveness with conventional rendering, we propose to build a system reactive to user inputs. Our basic premise is that for phases of the application in which the user provides inputs, we can disable PFR and the system reverts to conventional rendering, i. e. the two GPU clusters are employed to render just one frame. On the contrary, during phases in



(a) Switch between conventional rendering and PFR.



(b) Switch from PFR to conventional rendering in even frame.



(c) Switch from PFR to conventional rendering in odd frame.

Figure 4.9: The figure illustrates how the system switches between PFR and conventional rendering and the different issues that arise in doing so.

which no user input is provided PFR is enabled and two frames are rendered in parallel to save memory bandwidth. We refer to this system as Reactive-PFR (R-PFR). Figure 4.8 shows that R-PFR can be very effective since no input is provided most of the time for the majority of the mobile graphical applications we examined, so we can employ PFR extensively.

Switching from conventional rendering to PFR produces a very short screen stall. This is illustrated in Figure 4.9a, where two frames have to be buffered and rendered in parallel before the screen is updated again. When switching from PFR to conventional rendering, illustrated in Figures 4.9b and 4.9c, the two frames processed in parallel are discarded and a new frame is generated using both GPU clusters, so the same responsiveness than in conventional rendering is achieved. Hence, two frames are skipped when the system reverts to normal rendering, reducing the smoothness of the animation. Nevertheless, these perturbations happen in a very short time span of two refresh intervals, so an isolated switch is hardly noticeable by the user. Of course, excessive switching between rendering modes can still produce micro stuttering effects [154].

In R-PFR, the number of inputs received from the user are exposed to the GPU driver on a frame basis. The driver decides which rendering mode to employ every frame by monitoring user activity. The GPU maintains a counter with the number of frames without inputs since the last user interaction. Initially, conventional rendering is employed. When the counter of frames without inputs

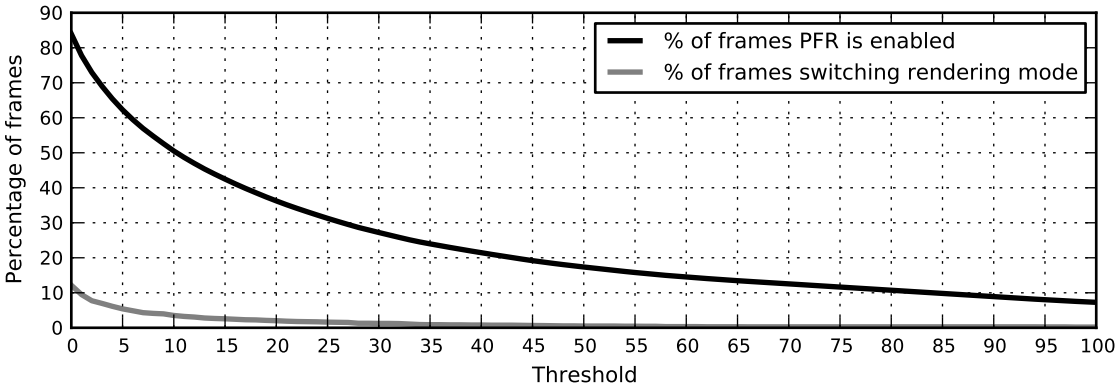


Figure 4.10: Percentage of frames where PFR is applied and percentage of frames where a switch from PFR to normal rendering or vice versa takes place, vs the threshold for R-PFR. Increasing the threshold reduces the frequency of switches but it also reduces the percent of time for PFR.

becomes bigger than a given threshold, the system switches to PFR. Finally, if the user provides some input the system immediately reverts to conventional rendering and resets the counter.

The threshold plays an important role in reducing the excessive number of switches between rendering modes and avoiding micro stuttering, as illustrated in Figure 4.10. If a big value for the threshold is employed, the number of switches is significantly reduced but the opportunities for applying PFR are severely constrained. A big threshold means that an important number of frames without user inputs have to be detected in order to switch to PFR, and all these frames are processed sequentially using conventional rendering. On the contrary, using a small value for the threshold increases the percentage of time PFR can be enabled, providing bigger energy savings, but it also increases the frequency of switches between rendering modes. We have selected 5 as the value for the threshold, since it provides a good trade-off for our set of commercial Android games (see Figure 4.10). By using a value of 5, the switches between rendering modes represent just 5% of the frames, so the small perturbations illustrated in Figure 4.9 are avoided 95% of the time. Even so, PFR can still be applied 62% of the time on average.

4.2.3 N-Frames Reactive Parallel Frame Rendering

R-PFR achieves the same responsiveness as conventional rendering, but at the cost of smaller potential energy savings since PFR is applied just to a fraction of the total execution time. In an effort to recuperate the energy saving opportunities lost during phases of the application with user inputs, the system can apply parallel rendering more aggressively during phases without user inputs. Since input lag is not a problem during these phases the GPU can render 4 frames at a time instead of 2 to achieve bigger memory bandwidth savings. In that case, the GPU is split in 4 clusters that can render 1, 2 or 4 frames in parallel, each

cluster having 25% of the resources of the baseline GPU. Initially, conventional rendering is employed so the 4 GPU clusters are used to render one frame. When the number of frames without inputs is bigger than a given threshold, T_1, the GPU driver switches to PFR so 2 GPU clusters are used to render odd frames and the other 2 clusters process even frames. If the number of frames without inputs becomes bigger than another threshold, T_2, then each GPU cluster renders a different frame so a total of 4 frames are processed in parallel. Finally, as in R-PFR, if the user provides some input the system immediately reverts to conventional rendering, so responsiveness is not reduced. We refer to this technique as N-Frames Reactive PFR (NR-PFR).

The thresholds T_1 and T_2 are set to 5 and 10 respectively, since we found these values provide a good trade-off between the number of switches and the percentage of time PFR is applied for our set of Android applications.

Note that 4 consecutive frames still exhibit a high degree of texture similarity, as shown in Figure 4.2. Ideally, if the 4 frames processed in parallel perform all the same texture accesses within a short time span, up to 75% memory bandwidth savings for texture fetching can be achieved.

4.2.4 Delay Randomly Parallel Frame Rendering

PFR delays user inputs in all the frames to achieve big memory bandwidth savings. R-PFR and NR-PFR do not delay any of the user inputs to achieve high responsiveness, at the cost of smaller bandwidth savings. Between these two extremes, a system can delay just a given percentage of the user inputs in order to trade responsiveness for memory bandwidth savings. We refer to this system as Delay Randomly PFR (DRPFR).

The behavior of DR-PFR is very similar to R-PFR, but when the GPU driver detects user inputs the system does not always revert immediately to conventional rendering. Instead, a random number between 0 and 100 is generated and the system only switches to normal rendering if the random number is bigger than P, where P is the maximum percentage of frames where user inputs are allowed to be delayed. The bigger the value of P, the bigger the memory bandwidth savings but the smaller the responsiveness. The value of P can be set, for instance, depending on user preferences. It can also be set depending on the battery level, so it is dynamically increased as the battery level decreases in order to extend the use-time per battery charge. The frames where the inputs are delayed are randomly distributed to try to reduce the impact on user interaction, since random distortions are usually harder to perceive by the user than distortions that are very localized.

Table 4.1: GPU simulator parameters. All the configurations include the same amount of resources: 1 big GPU cluster, 2 clusters with half the resources for each cluster or 4 clusters with 25% of the resources per-cluster.

L2 Cache	128 KB, 8-way associative, 12 cycles latency		
Texture Caches	16 KB, 4-way associative, 2 cycles latency		
Local Color Buffer	4 KB, 1 cycle latency (32×32 RGBA pixels/tile)		
Main Memory	1 GB, 16 bytes/cycle (dual-channel)		
	Conventional rendering	**PFR, R-PFR DR-PFR**	**NR-PFR**
Number of clusters	1	2	4
Raster units per cluster	4	2	1
Vertex Processors per cluster	4	2	1
Vertex Cache	8 KB, 2-way	4 KB, 2-way	2 KB, 2-way
Vertex Fetcher	16 in-flight vertices	8 in-flight vertices	4 in-flight vertices
Primitive Assembly	4 triangles/cycle	2 triangles/cycle	1 triangle/cycle
Polygon List Builder	4 in-flight triangles	2 in-flight triangles	1 in-flight triangle
Tile Fetcher	4 in-flight tiles	2 in-flight tiles	1 in-flight tile

4.3 Experimental Results

In this section we evaluate normalized off-chip memory traffic, speedups and normalized energy of the PFR variants described in Section 4.2. The baseline does conventional Tile-Based Rendering (TBR) with a single big cluster. The parameters employed for the experiments are summarized in Table 4.1.

In first place, memory traffic is shown in Figure 4.11. The basic version of PFR consistently provides important memory traffic reductions in most of the applications, achieving 23.8% bandwidth savings on average. The reason is that PFR exploits the high degree of texture overlapping between two consecutive frames by processing them in parallel, so that most texture data fetched by one cluster is shortly reused by the other cluster before it is evicted from the shared cache, which converts almost half of the capacity misses produced in conventional rendering into hits. Note that PFR targets bandwidth savings for texture fetching, so the reduction in memory traffic depends on the importance of the texture traffic in the overall bandwidth usage, illustrated in Figure 4.1. Games that devote most of the bandwidth for reading textures, such as *gravityguy* or *cuttherope*, are the ones that obtain the biggest bandwidth savings, 39.3% and 33.3% respectively. On the contrary games such as *dungeondefenders* or *templerun*, that employ most of the bandwidth for fetching geometry, obtain smaller bandwidth savings, 5.3% and 5.7% respectively.

Regarding R-PFR, we have set the threshold to 5, so the GPU driver switches from conventional rendering to PFR after 5 consecutive frames without user inputs. R-PFR achieves more modest memory bandwidth savings, 17.7% on average, because PFR is only enabled during phases of the application without

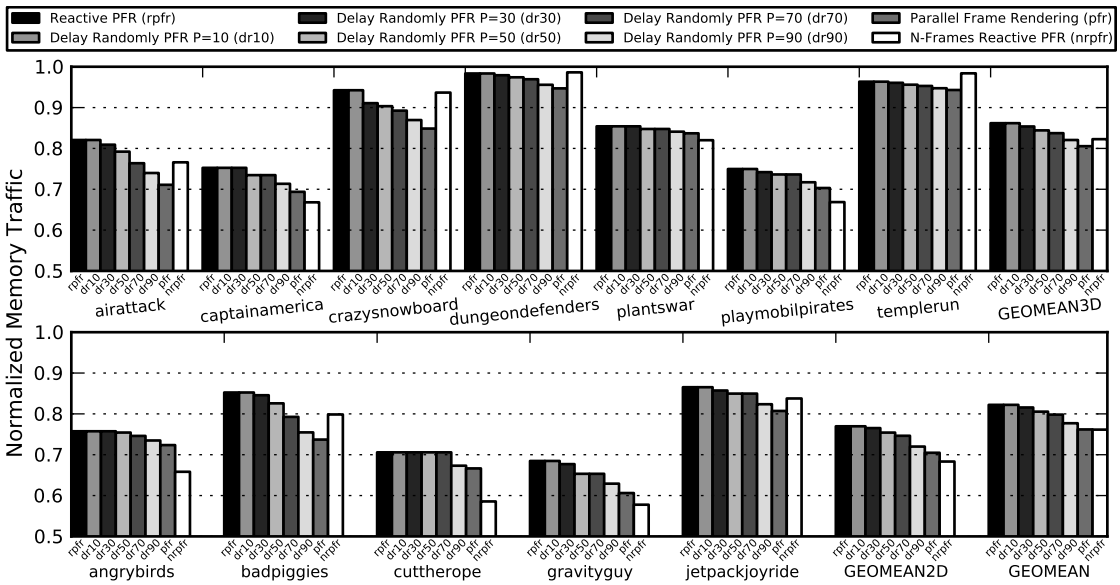


Figure 4.11: Normalized memory traffic. The top graph shows the results for the 3D games, whereas the bottom graph provides the numbers for the 2D workloads. The baseline is a mobile GPU that employs Tile-Based Rendering and processes just 1 frame at a time.

user inputs, which limits the opportunities for processing frames in parallel but completely avoids any loss of responsiveness. R-PFR achieves its best results for games with a small number of user inputs, such as *cuttherope* or *gravityguy* (see Figure 4.8), where parallel processing is enabled most of the time. Conversely, it achieves its worst results for games with more intensive user interaction, such as *dungeondefenders*, that employ sequential frame processing for a bigger percentage of the execution time.

Regarding NR-PFR, we have set thresholds T_1 and T_2 to 5 and 10 respectively, so after 5 consecutive frames without user inputs the GPU driver starts processing 2 frames in parallel and after 10 it switches to 4 frames in parallel. For 6 games (*angrybirds*, *cuttherope*, *gravityguy*, *captainamerica*, *plantswar* and *playmobilpirates*), NR-PFR achieves even bigger memory bandwidth savings than PFR, while maintaining the same responsiveness than conventional rendering. As in R-PFR, the savings are more modest for games with intensive user interaction (*dungeondefenders*). On average, NR-PFR achieves 23.8% off-chip memory traffic savings.

Regarding DR-PFR, we evaluated configurations with the parameter P set at 10, 30, 50, 70 and 90. P specifies the maximum percentage of frames where input lag is tolerated, providing fine-grained control to trade responsiveness for memory bandwidth savings. The results show that the biggest the value of P the biggest the bandwidth savings, but the smallest the responsiveness since more user inputs are delayed.

In second place, speedups are shown in Figure 4.12. It is worth noting that none of the configurations produce slowdowns in any of the games. On the

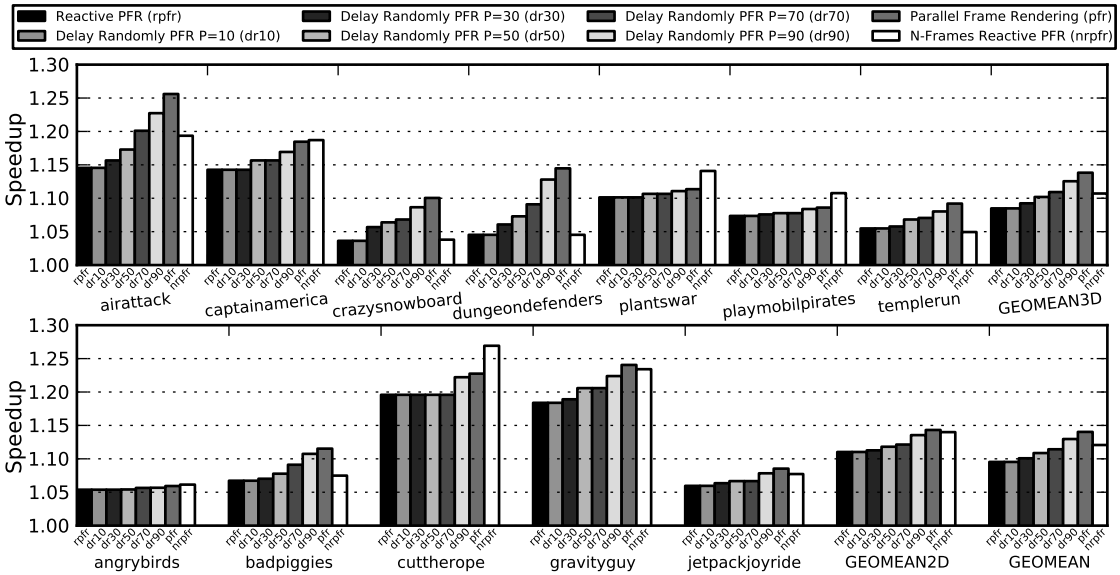


Figure 4.12: Speedups. The baseline is a mobile GPU that employs Tile-Based Rendering and processes just 1 frame at a time.

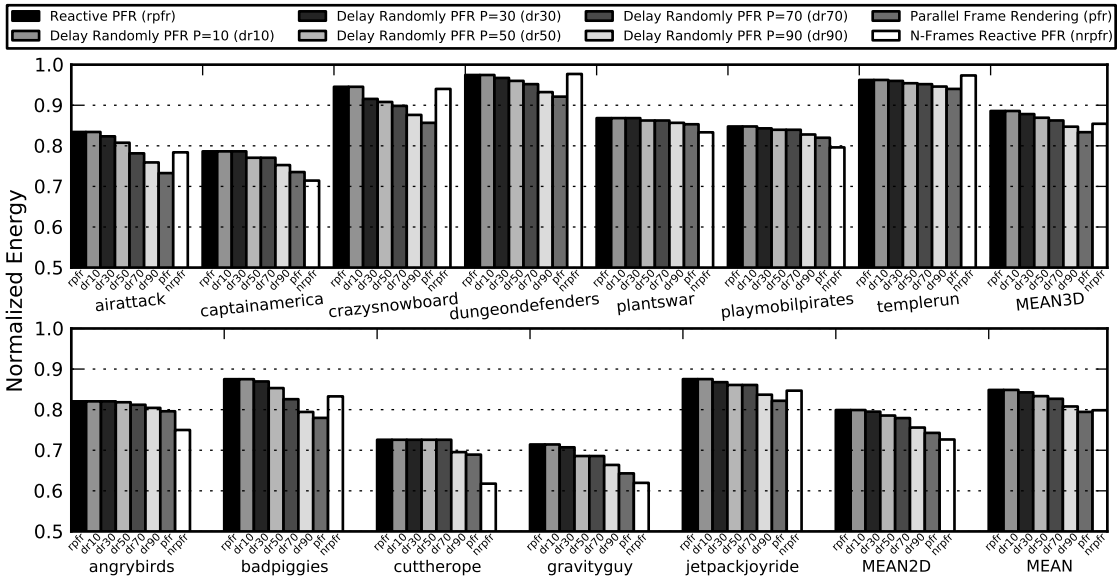


Figure 4.13: Normalized Energy. The baseline is a mobile GPU that employs Tile-Based Rendering and processes just 1 frame at a time.

contrary, PFR provides 14% speedup on average. Although the main objective of PFR is to save energy, performance is also increased as a side effect of reducing the memory traffic in a bandwidth bound system.

In third place, energy consumption is shown in Figure 4.13. Energy results include the static and dynamic energy consumed by both the GPU and system memory. PFR achieves 20.5% energy savings on average that come from two sources. First, the dynamic energy is reduced because 23.8% of the off-chip memory accesses are avoided. Second, the 14% speedup achieved produces a reduction in static energy. Note that we do not assign any static energy consumption during

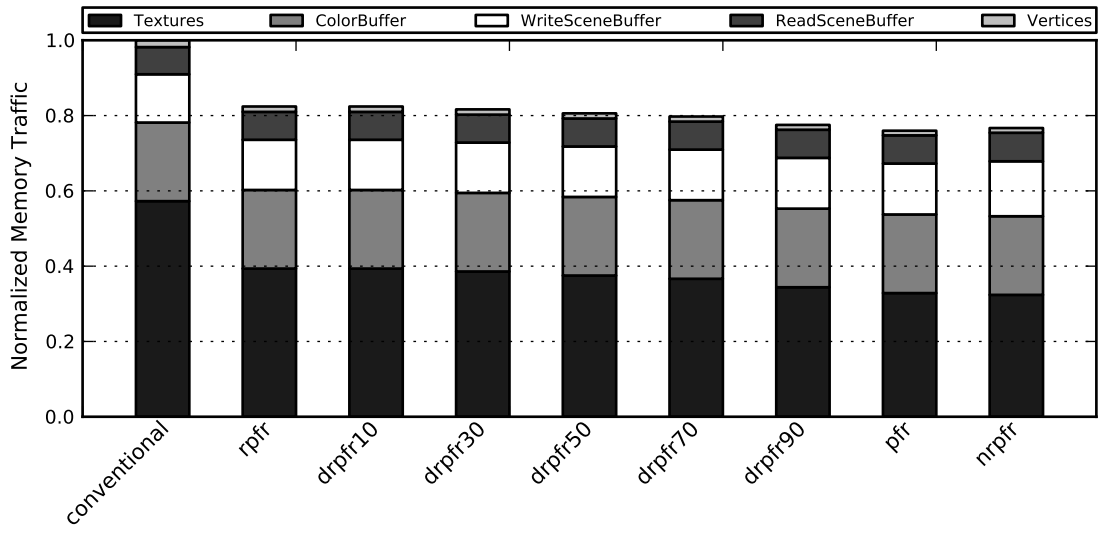


Figure 4.14: Memory bandwidth breakdown for the baseline configuration and the different versions of Parallel Frame Rendering. "WriteSceneBuffer" represents the bandwidth employed for sorting the triangles into tiles in main memory, whereas "ReadSceneBuffer" is the bandwidth required to fetch back the triangles.

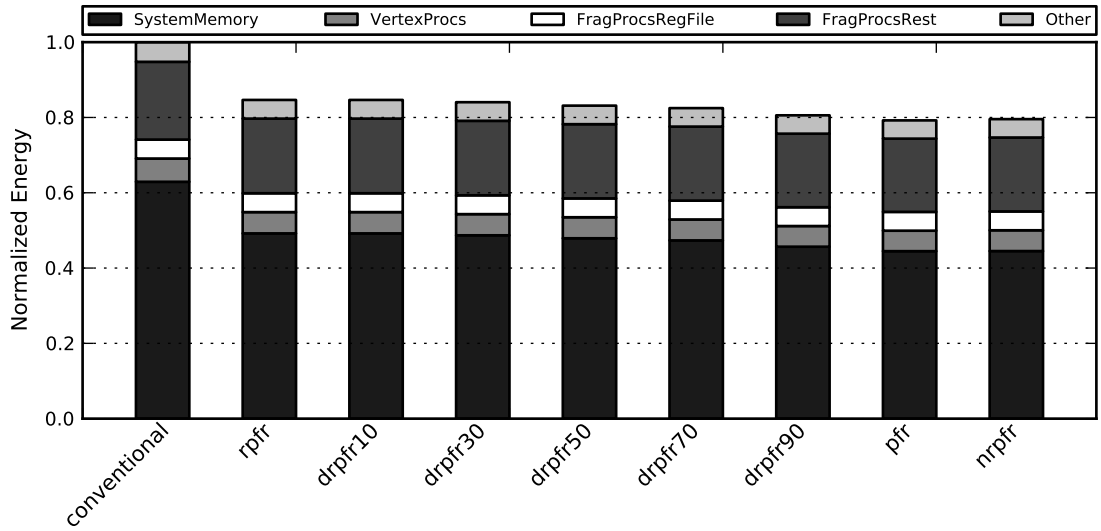


Figure 4.15: Energy breakdown for the baseline configuration and the different versions of Parallel Frame Rendering.

long idle periods because we assume that the GPU could drastically reduce it by entering a deep low power state. Regarding R-PFR and NR-PFR, they achieve more modest energy savings since parallel rendering is enabled just a fraction of the total execution time in order to maintain responsiveness. Finally, DR-PFR provides bigger energy saving as the value of P is increased, at the cost of reducing responsiveness.

Figure 4.14 shows the memory bandwidth breakdown for the different variations of PFR and the baseline GPU. As expected, the bandwidth savings come from the texture fetching, since PFR overlaps the texture accesses of two con-

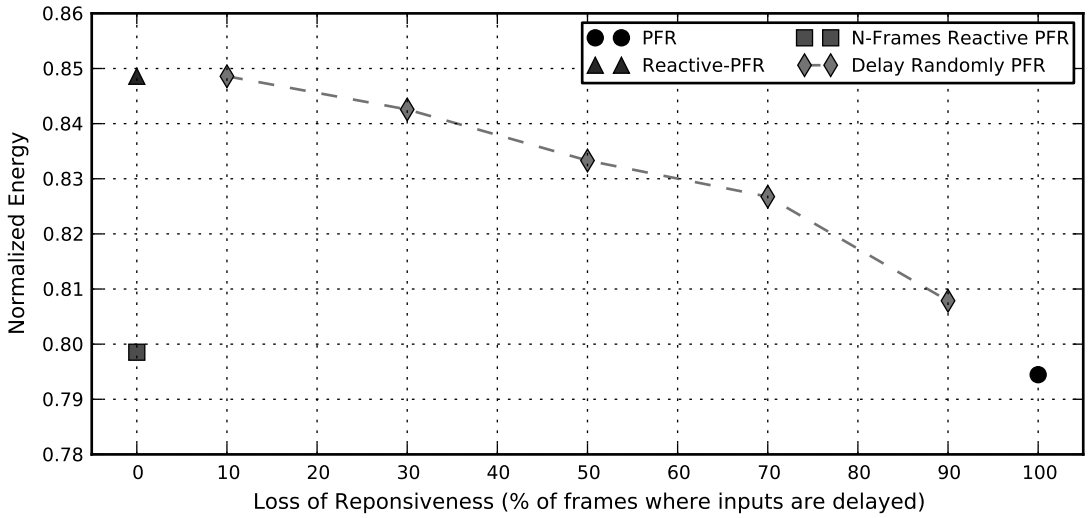


Figure 4.16: Normalized energy vs. loss of responsiveness (percentage of frames where user inputs are delayed).

secutive frames to exploit inter-frame texture similarity. Furthermore, PFR does not increase the bandwidth requirements for the rest of the components despite two frames are generated in parallel. On the other hand, Figure 4.15 shows the energy breakdown for the same configurations. PFR saves system memory energy by reusing textures between consecutive frames to avoid off-chip memory accesses.

Figure 4.16 plots, for all configurations, the average normalized energy consumption versus loss of responsiveness, measured as a percentage of frames where user inputs are delayed. At one end, PFR provides the biggest energy reduction, 20.5% on average, but at the cost of delaying all the user inputs. At the other end, R-PFR does not lose responsiveness at all, since user inputs are never delayed, but achieves smaller energy savings, 15.1% on average. NR-PFR also maintains full responsiveness, but achieves energy savings much closer to PFR, 20.1% on average, as it renders up to four frames in parallel to maximize bandwidth savings. Finally, DR-PFR allows fine-grained control over the energy savings and the responsiveness. The GPU driver can augment the energy savings by increasing the value of P, but then more user inputs are delayed. Conversely, the GPU driver can increase responsiveness by reducing the value of P, but losing part of the energy savings. By varying P, DR-PFR may achieve any intermediate energy versus responsiveness trade-off between PFR and R-PFR.

In this chapter we assume a Tile-Based Rendering architecture as our baseline GPU. However, PFR can also be applied on top of an Immediate-Mode Rendering (IMR) architecture. Appendix B includes the experimental results for PFR assuming an IMR baseline. The numbers show that PFR is able to reduce memory bandwidth usage by 13%, providing 10% energy savings on average when architected on top of IMR.

4.4 Conclusions

In this chapter we argue that the memory bandwidth required for mobile GPUs can be significantly reduced by processing multiple frames in parallel. By exploiting the similarity of the textures across frames, we can overlap the execution of consecutive frames in an effort to reduce the number of times we bring the same texture data from memory. We term this technique Parallel Frame Rendering (PFR).

This, however, comes at a cost in the responsiveness of the system, as the input lag is increased. We argue that in practice this is not really important for most of the applications, as for mobile systems touch screens tend to be slow and thus applications tend to require small interaction with the user. Nevertheless, we show that adaptive forms of PFR can be employed, that trade responsiveness for energy efficiency. We present three variants of these adaptive schemes and show that we can achieve 23.8% reduction in memory bandwidth on average without any noticeable responsiveness hit, achieving 12% speedup and 20.1% energy savings.

Chapter 5

Hardware Memoization in Mobile GPUs

In this chapter we address the problem of removing redundant executions of the Fragment Program. In first place we argue that graphical applications exhibit a high degree of redundancy, since generating an animation typically involves the succession of extremely similar images. In terms of rendering these images, this behavior translates into the creation of many fragment programs with the exact same input data. We measure this fragment redundancy for a set of commercial Android applications and show that, on average, more than 38% of the fragments used in a frame have been already computed in a prior frame.

In second place, we try to exploit this redundancy using fragment memoization. Unfortunately, this is not an easy task as most of the redundancy exists across frames, rendering most hardware based schemes unfeasible. We thus first take a step back and try to analyze the temporal locality of the redundant fragments, their complexity, and the number of inputs typically seen in fragment programs. The result of our analysis is a task level memoization scheme, that outperforms the current state-of-the-art in low power GPUs.

Finally, we evaluate our memoization scheme by using several Android commercial games, and show that our system is able to remove 42.2% of the redundant fragment computations on average. This materializes to a significant speedup of 15% on average, while also improving the overall energy consumption by 12%.

5.1 Redundancy in Mobile GPUs

Graphical applications for mobile devices tend to exhibit a large degree of scene replication across frames. Figure 5.1 shows two consecutive frames of a popular Android game, Bad Piggies. As it can be seen, the input from the user resulted in the main character being shifted, however a significant fraction of the frame remained untouched. Despite being admittedly a well selected example, this

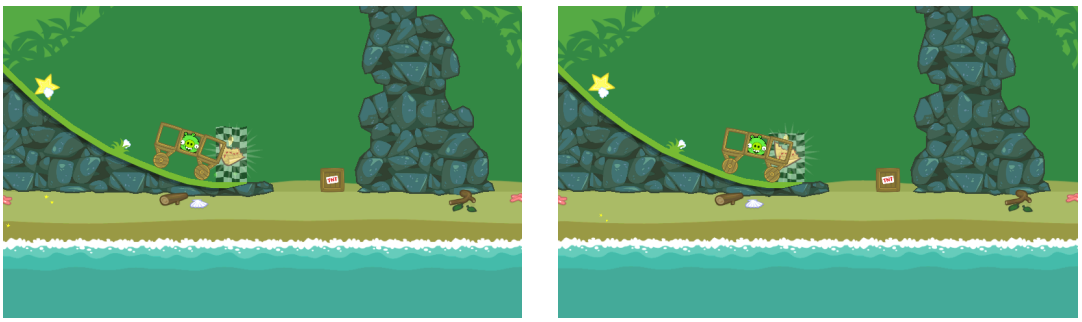


Figure 5.1: Two consecutive frames of the game Bad Piggies. A huge portion of the screen remains unmodified.

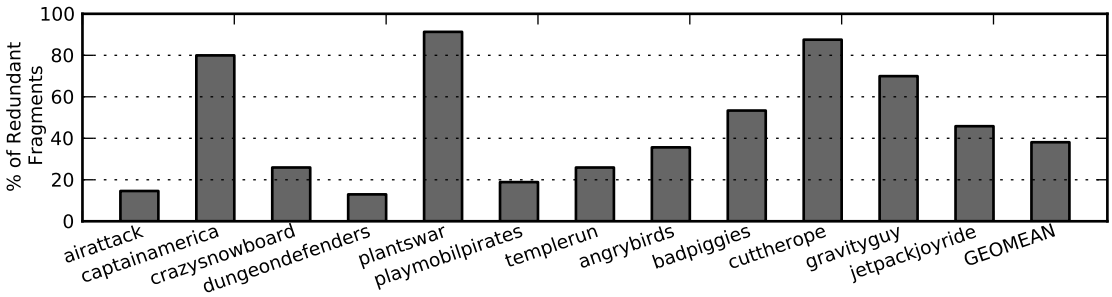


Figure 5.2: Percentage of redundant Fragment Program executions for twelve Android games. On average, 38.1% of the executions are redundant.

behavior is actually quite prevalent for mobile applications. Figure 5.2 depicts the percentage of fragment computation that is common between consecutive frames for twelve popular Android games. Overall, more than 38% of the fragments computed in a given frame were previously computed in the frame before it.

Motivated by this observation, recent work attempts to exploit this inter-frame locality in order to save memory bandwidth and improve the overall energy efficiency. ARMs Transaction Elimination compares consecutive frame buffers and performs partial updates of entire tiles [57]. Parallel Frame Rendering (PFR), presented in Chapter 4, tries to overlap the execution of consecutive frames in an effort to improve the cache locality. Although both schemes are able to significantly improve the overall energy efficiency, they still need to perform all the fragment computations (even if they are not going to store them to the frame buffers) and some of the memory accesses.

In Figure 5.3 we depict the performance benefits that could be attained over PFR if we could avoid all computation of fragments that are exactly the same between two consecutive frames. Removing all redundant computation and memory accesses results in a 28.2% speedup on average over the current state-of-the-art. Moreover, as shown in Figure 5.4, energy is also improved by 32.9%.

In this chapter we remove a significant fraction of this redundant work by adding a task-level memoization scheme on top of PFR. We keep a hardware structure that computes a signature of all the inputs to a task and caches the value of the corresponding fragments. Subsequent computations form the signature and check against the signatures of the memoized fragments. Hits in the

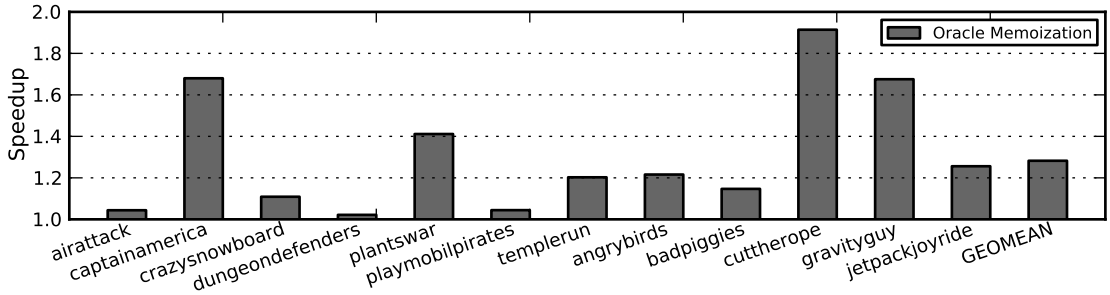


Figure 5.3: Performance increase achieved with an Oracle memoization system. On average, 28.2% speedup can be achieved by removing all the redundant Fragment Program executions. The baseline is a state-of-the-art mobile GPU.

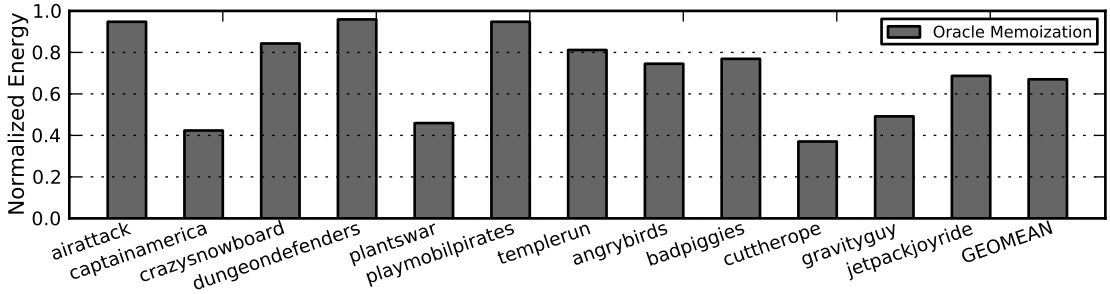


Figure 5.4: Potential energy savings of removing all the redundant Fragment Program executions. On average, 32.9% energy can be saved with respect to a state-of-the-art mobile GPU.

hardware structure result in the removal of all relevant fragment computation. Next sections analyze the fragment redundancy as seen in commercial Android games and describe our memoization scheme.

5.2 Redundancy and Memoization

Memoization is an optimization technique that avoids repeating the execution of redundant computations by reusing the results of previous executions with the same input values, which results in execution speedups and energy savings. The first time a computation is executed, its result is dynamically cached in a Look Up Table (LUT), along with its inputs. Subsequent executions of the same code will probe the inputs in the LUT and in case of hit, the cached result is written to the output rather then recalculating it. Memoization has been employed both at the function level in software [132] and at the instruction (or set of instructions) level in hardware [139]. In both cases, the computed result along with its inputs are cached in a LUT, so that subsequent executions with the same inputs can directly read the memoized result, instead of repeating all the computations. Although the general concept is fairly straightforward, in order for memoization to be efficient a set of requirements have to be met. In the next few sections we try to analyze these restrictions.

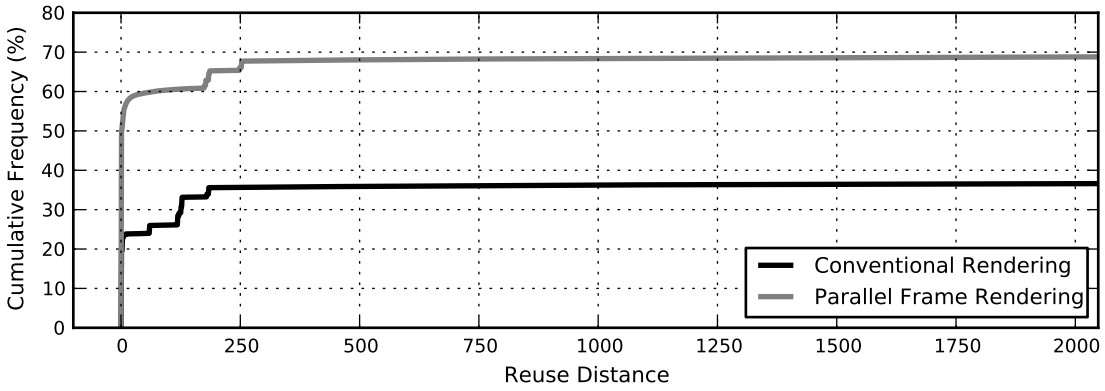


Figure 5.5: Cumulative frequency of distances between redundant fragments for Conventional Rendering and PFR.

5.2.1 Reuse Distance and Parallel Frame Rendering

A prime requirement for any memoization scheme is that the data on which the scheme is applied exhibits a high degree of re-use. Figure 5.2 shows that for the graphical applications that we use as our focus in this paper, 38.1% of the fragments are redundant. However, re-use alone is not enough for hardware based memoization solutions. Bound by power/area limitations, the hardware-based memoization schemes also require that the re-use distance between redundant computations is relatively small.

Throughout the paper we will use the term re-use distance to mean the number of unique fragments processed between two consecutive occurrences of the same fragment, a slightly modified usage of the term from its typical use [104]. We will also say that two fragments are the same if they have identical input attributes and they have to perform the same fragment shader. In case two fragments are the same, we will call the latter redundant.

Figure 5.5 illustrates the distribution of the re-use distances between redundant fragments for the set of Android games we analyzed (see Section 2.2.1) for re-use distances up to 2048 fragments. The "Conventional Rendering" configuration consists on a mobile GPU such as the one illustrated in Figure 4.4, that features four Raster Units to render four screen tiles in parallel, but all these screen tiles belong to the same frame, i. e. it renders just one frame at a time as in conventional mobile GPUs. As we can see, 36.6% of the redundant fragments are re-used at distances that can be captured with some realistic hardware constraints. Although it represents a non-negligible percentage of fragments, this means that 63.4% of the redundancy happens at distances that are not amenable for a hardware-based memoization system. This is somewhat expected, as a significant percentage of the redundancy tend to be inter-frame and frames are relatively big. As such, a fragment memoization scheme can potentially reduce the distances by overlapping the fragment computations from sub-sequent frames.

This observation motivates the adoption of Parallel Frame Rendering (PFR), presented in Chapter 4, as our starting point. PFR tries to render two consecutive

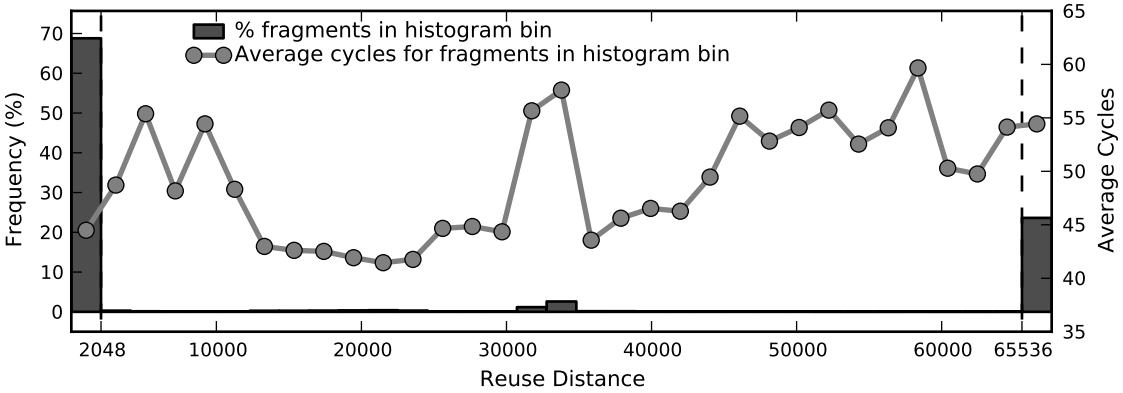


Figure 5.6: Reuse distance histogram with PFR, including percentage of fragments and average complexity for each histogram bin. The last histogram bin includes all the distances that are bigger than 65536. The complexity is measured as the number of GPU cycles required to process the fragment.

frames in parallel. As shown in Figure 4.5, under PFR the baseline GPU needs to be split into two separate clusters, each including half of the resources of the original GPU. The GPU Command Processor dispatches commands from even frames to cluster 0 and commands from odd frames to cluster 1. Each GPU cluster behaves as an independent GPU, rendering the frame by using Tile-Based Rendering as described in Section 2.1.3. To further reduce the distance between redundant computations of parallel rendered frames, the two clusters are synchronized by processing tiles in lockstep. As such they process the same screen tile in two consecutive frames in parallel. Although PFR was originally proposed in order to improve the locality of texture cache accesses, it is a perfect match for our memoization scheme.

Looking again at the re-use distances for a PFR based GPU in Figure 5.5, 59.9% of the redundant fragments have re-use distances smaller than 64 fragments, and 68.8% smaller than 2048. This is a significant improvement over re-use distances of the conventional GPU. Figure 5.6 shows a histogram of fragment re-use distances for PFR (read on the left vertical axis). Distances are discretized in bins of 2048 fragments and, unlike Figure 5.5, all redundant fragment are represented in this graph. As pointed-out before, 68.8% of the re-uses take place at distances smaller than 2048 (first bin), whereas the rest are sparsely distributed across the whole distance spectrum (note that the bipolar appearance of the histogram is just an artifact of grouping all distances greater than 64K into a single "fat" last bin).

5.2.2 Task-level Complexity

In this work, we term complexity the amount of work involved by a fragment, and we measure it as the GPU cycles it takes to compute fragment operations. As pointed out in [134], this concept is important because computation re-use is lucrative only when the cost of accessing the structures used for memoization

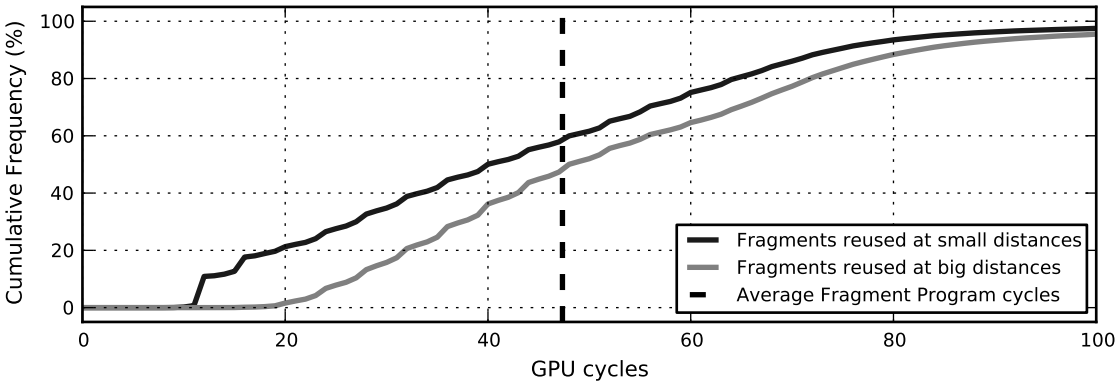


Figure 5.7: Cumulative histogram of fragment complexity for fragments reused at small and at big distances. Complexity is measured as number of GPU cycles required to process the fragment.

is smaller than the benefit of skipping the actual computation. For this reason prior work on memoization either tries to perform memoization for multiple instructions [66, 82, 49, 50, 91, 145] or for long latency operations [64].

Figure 5.6 shows the per-bin average fragment complexity (read on the right vertical axis), so we can see it as a function of the re-use distances. Unfortunately, fragments that are re-used at bigger distances, i. e. the ones that exhibit worse temporal locality, tend to be more complex (54.4 cycles on average). However, fragments at smaller re-use distances, which are the target of our scheme, are also relatively complex (44.4 cycles on average). Figure 5.7 provides a more detailed view of the fragment complexity distribution. 52.6% of the fragments that could be re-used and exist in large re-use distances are more complex than the total average execution time. On the other hand, only 42.1% of the fragments re-used at small distances are more complex than the average. Nevertheless, 100% of the redundant fragments reused at small distances spend more than 6 cycles, which is greater than the amount of time required to perform the lookup to the memoization scheme. As we will show in Section 5.4, this is still enough to provide substantial performance and energy gains.

5.2.3 Referential Transparency

Another problem typically faced by conventional memoization when identifying redundancy between instructions is to guarantee that they are referentially transparent, i.e. that the same set of input values is always going to produce the same output result. The main difficulty here arises from the fact that these instruction blocks must not depend on global memory data that is subject to changes between executions, and they do not produce side-effects. Since it is difficult to track these global changes at runtime, task-level hardware-based memoization usually requires compiler support to carefully select code regions that do not depend on global data or have side-effects, which further reduces the choice of candidate code regions.

Fortunately, our approach does not suffer from this additional complexity for two reasons. The first is that fragment shaders compute a single output color, without side-effects. The second, and perhaps more important, is that changes to global data accessed by the Fragment Program, such as textures or shader instructions, are relatively easy to track by the driver as the programmer does not have direct access to the graphics memory. In fact, API function calls, such as *glTexImage2D* or *glShaderSource*, must be used in order to update textures and shaders.

As such, for a fragment memoization scheme, referential transparency can be guaranteed by simply monitoring the API calls, and discarding the content of the memoization hardware. Although this is a very crude solution, in practice it works quite well as updates to global data are rather infrequent. They also tend to be clustered at the beginning of new levels (for games) when all the textures and shaders required to render the level are loaded, and as such part of the opportunity cost is amortized. We found that the time between updates to global memory in our set of Android games is in the order of hundreds or thousands of frames.

5.3 Task Level Hardware-Based Memoization on a Mobile GPU

5.3.1 Memoization System

Conceptually the proposed memoization scheme is comprised of three principal components: the detection of candidate fragments, the lookup of prior fragment information and the replacement of the fragment computation with the memoized information. Figure 5.8 depicts a block level diagram of the various components and how they operate.

Input fragments are first checked whether they satisfy the input restrictions, in order to be considered as memoization candidates. Fragment Programs with more than 4 input registers or more than 4 texture samplers, are assumed to be bad candidates for memoization. The rationale is that since these inputs will have to be hashed, having more inputs both complicates the hashing logic and degrades the quality of the hash function (dispersion may become worse). Fragments that do not meet the memoization criteria proceed as they would in a normal GPU. The ones that do, pass through a stage where we form a signature out of their inputs. As illustrated in Figure 5.9, 99.9% of the fragments pass the memoization criteria in our set of Android games, so that the overall coverage of our technique is not hindered by the hashing restriction.

Not all input bits are selected for generating the signature, as illustrated in Figure 5.10. Input Registers are vector registers that consist of four 32-bit single precision FP components. Based on the fuzzy memoization paradigm [173], we take off the 8 least significant bits of the mantissa for every component. Hence,

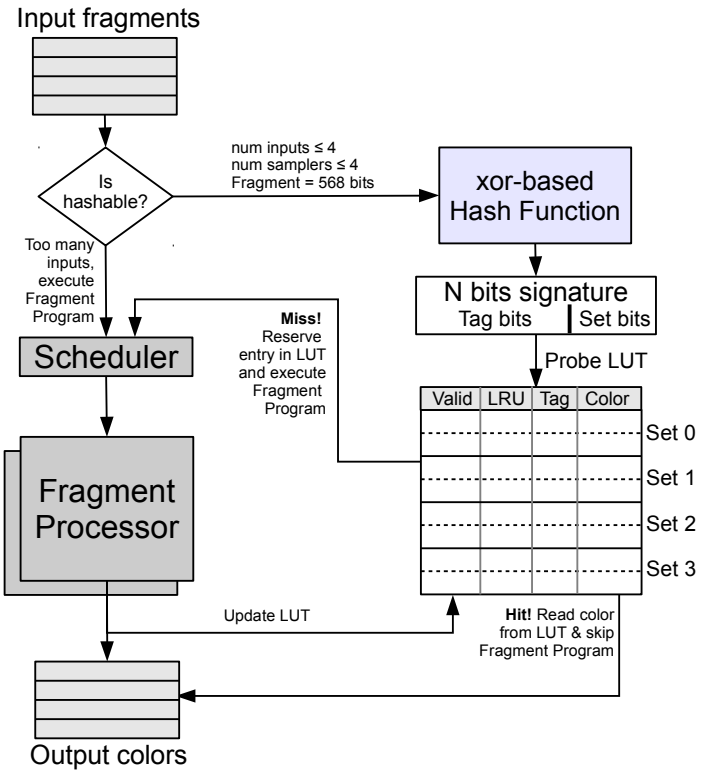


Figure 5.8: Proposed hardware-based task level memoization system.

fragments that are extremely similar are also considered redundant, improving the efficiency of the memoization system. Furthermore, we reduce the input bits that are considered, simplifying the computation of the signature. Note that small precision losses in graphics and multimedia can be tolerated as the end difference is hard to distinguish [81].

Figure 5.10 shows how the hash function generator is implemented. The total number of input bits that we can have based on the input restrictions and the fuzzy memoization feature is 568, including information from the Fragment Program (base PC), the Texture Samplers and the Input Registers. Being F the 568-bit description of a fragment and S the resulting signature of N bits, we have experimentally found that the bitwise XOR operations that follow the next form provide high dispersion even with small signature sizes and require simple hardware:

$$S_i = F_i \oplus F_{i+N} \oplus F_{i+2\times N} \oplus F_{i+3\times N} \oplus ...$$

We hash the input fragment into N bits, being N smaller than 568. For example, for $N = 32$ each of the final bits is a product of an 18-bit XOR. Assuming that we have only 2-input XOR gates, this means that we need a tree of 6 levels of XOR gates. As the number of signature bits grow, the complexity of the hash function lessens, however more storage will be required for signatures (functioning as tags) in the LUT. Moreover, the LUT set index is typically built by bit selection of the signature LSBs, so having a larger signature with a less complex hash

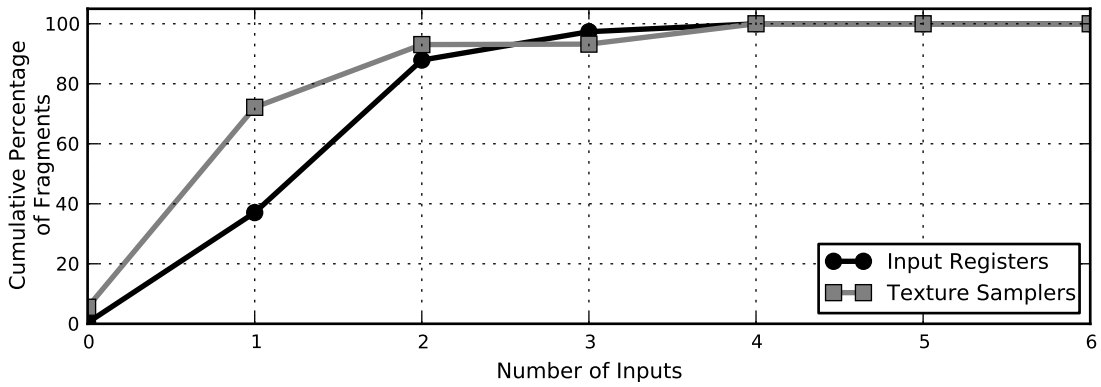


Figure 5.9: Two cumulative histograms for the number of input registers and number of texture samplers. With just 4 of each we cover 99.9% of the fragments, although in OpenGL ES the programmer can use up to 16 input registers and 8 texture samplers.

function makes each index bit to be generated from a smaller number of fragment bits. As we will show in Section 5.4, this puts more pressure to specific sets of the LUT, as the dispersion is much worse. This results in requirements of a bigger LUT, which is obviously not a good trade-off.

Since we perform a hash of many bits into few, we inevitably lose some information. This results in what we call a *false hit*, which ultimately results in a distortion of the frame with respect to the one that would have been computed by a normal GPU. More bits in the hash result in a smaller probability for such collisions, but as we will see in Section 5.4, in practice there is little difference for signatures bigger than 32 bits.

Once the hashing of the inputs is ready, we then perform a lookup to the Look Up Table (LUT). The LUT acts as a cache, in that it uses part of the signature as an index, and the rest as the tag. The LUT is set-associative, and we employ pseudo LRU [48] as its replacement policy. Each entry of the LUT contains control information and data. Regarding the control information, each entry has a Valid bit, an LRU bit for the replacement policy and $N - M$ bits to store the most significant part of the signature that serves as the tag for the entry, being M the number of bits employed to select the set. Regarding the data, each entry stores the 32-bit color, in RGBA format, that corresponds to the given fragment.

A hit in the LUT indicates that we have a memoized value of the prior color of the fragment, and as such we can skip the fragment computation. Fragments that hit in the LUT read out the color and take the bypass to the next GPU stage. On the contrary, a miss in the LUT indicates that the output of the fragment is not available. Missing fragments are redirected to the Fragment Processors where the Fragment Program is applied to compute the color. An entry in the given set is reserved for the fragment and the tags are updated, replacing a previous entry by using pseudo-LRU if there is no free entry available, and the fragment carries the index of the corresponding line.

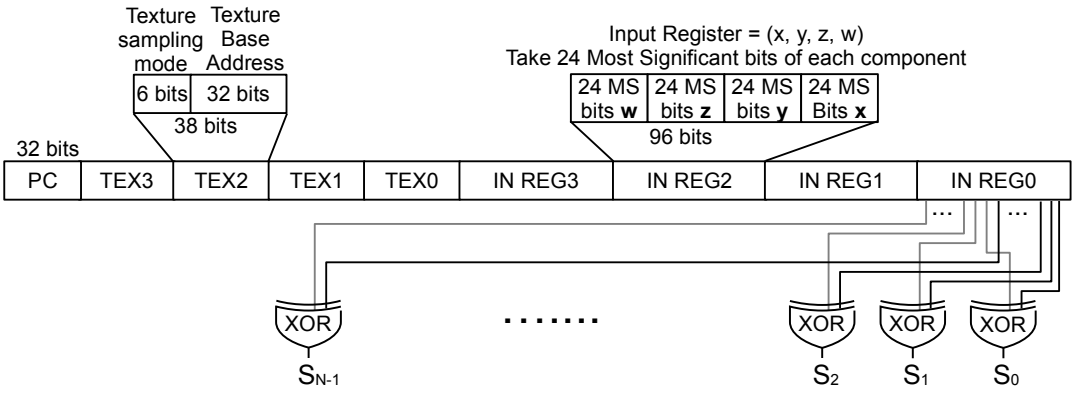


Figure 5.10: Computation of the hash function. The input fragment description contains information from the Fragment Program (base PC), the Texture Samplers and the Input Registers.

Note that in case that two fragments are redundant, the second —younger— fragment can potentially arrive to the fragment stage while the Fragment Program is still in-flight for the first —older— fragment. In that case, the tags of the LUT have already been updated, so the second fragment will hit in the LUT, but the color is not yet available. Our system revolves this situation by blocking the second fragment until the Fragment Program for the first fragment finishes execution. Once the redundant color is ready, we wake-up and bypass the redundant fragment, avoiding the execution of the Fragment Program even for fragments re-used at very small distances.

The Scheduler coordinates the dispatch of fragments to the Fragment Processors. It receives fragments from two sources: fragments that cannot be hashed and fragments that miss in the LUT. It applies a Round Robin policy to dispatch the input fragments to the processors. In the case that all the Fragment Processors are busy, the Scheduler stalls the pipeline. Once the color of a fragment has been computed, it is forwarded to the next GPU stage. Furthermore, the missing fragments update the LUT by using the new color and the index of the line previously reserved.

As the frequency of mobile GPUs is small, it takes only two cycles in order to perform the hash function and access the LUT. As such, there is no significant pressure to this task. Furthermore, the Fragment Processors are usually the main bottleneck in the GPU pipeline [147] due to the complexity of the Fragment Program, that typically includes several complex operations such as texture fetches. Hence, there is significant slack for computing the signature and accessing the LUT while the fragment processors are still computing previous fragments.

The proposed hardware changes are relatively small as a percentage of the total GPU area. For the 32-bit signature, 4-way LUT configuration with 512 sets, we measured it to be 18KBytes in total. Based on estimations using McPAT, the whole memoization scheme including the hash computation logic and the LUT accounts for only 0.47% of the overall GPU area.

The baseline GPU has two clusters to render two frames in parallel in order to

improve the temporal locality of redundant fragments. The hardware LUT (which is the most costly component in terms of area) is centralized and shared by both clusters since most of the redundancy is inter-frame, so the results computed by one cluster must be visible by the other in order to detect and remove redundant computations.

As previously mentioned in Section 5.2.3, we propose the use of task level memoization. The execution of a Fragment Program has no side-effects and no mutable state is allowed. Furthermore, it is easy to track changes to global data. Note that some inputs of the fragments are pointers to global graphics memory, such as the base PC of the Fragment Program or the base address of the textures. Even if two fragments have the same base addresses, the output is not going to be the same if the global data has been updated between two executions of the Fragment Program. Nevertheless, all the updates to graphics memory pass through the GPU driver. We thus extended the driver to detect those updates and send a command to the GPU to clear the LUT, since the memoized values can no longer be trusted. Clearing the LUT consists on setting all the valid bits to 0. We found that this clear operation is very infrequent since textures and Fragment Programs are updated on a game level basis, so they remain unmodified during hundreds or thousands of frames. Moreover, the LUT is warmed-up very fast due to its small size and the small reuse distances exhibited by redundant fragments. As our memoization system is task-level the entire execution of the Fragment Program is avoided for redundant fragments that hit in the LUT. Hence, not only the fragment computations performed in the functional units are avoided, but also the memory accesses required to fetch instructions and textures are removed.

Regarding the granularity of our memoization system, we decided to stay at the fragment level instead of targeting primitive or even object level memoization. We found the fragments to be more amenable to our technique as for primitive memoization more inputs have to be hashed, all the input attributes of three vertices, and more outputs have to be memoized. Triangles can potentially cover big regions of the screen, whereas for fragments only one output color has to be stored in the LUT.

Scalability of our technique to larger GPUs is a valid concern, since a centralized LUT design does not scale to GPUs with tenths or hundreds of cores. Nevertheless, we think there exist viable solutions thanks to the available flexibility in distributing the workload and the highly parallel nature of graphical applications. Scalability can be achieved by distributing among different cores the processing of multiple tiles of the same frame. The LUT could then be distributed into multiple tables, each shared by cores processing the same screen tile in consecutive frames. Other approaches that follow distributed memories ideas are also possible, we leave the implementation of such a distributed memoization system as future work.

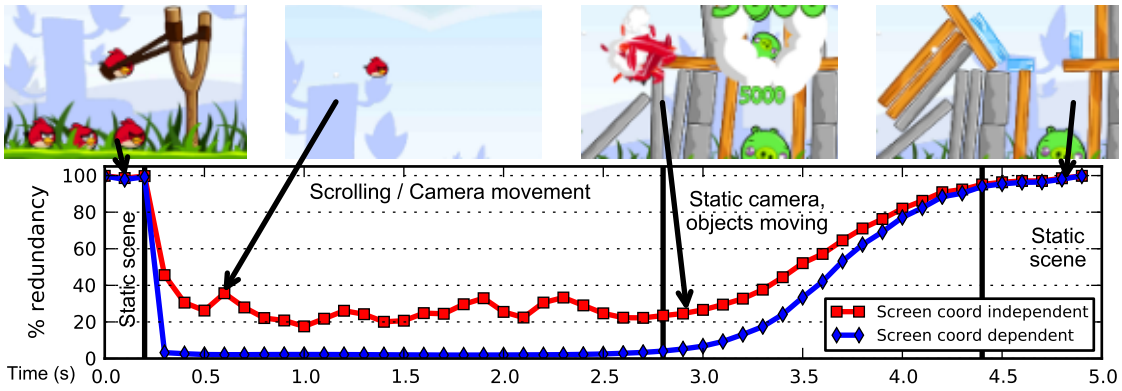


Figure 5.11: The graph shows the percentage of redundant fragments vs time for the game Angry Birds. For the "Screen coord dependent" configuration the screen coordinates are always used to form the signature, whereas for the "Screen coord independent" they are excluded from the signature provided that they are not accessed in the Fragment Program. The graph covers four phases of the application, an image crop of a frame for each phase is included in the figure. For phases with static frames (first and last phases) both configurations achieve levels of redundancy close to 100%. However, the "Screen coord independent" performs better in the presence of camera movements (second phase) or moving objects (third phase), since redundant fragments are not required to be located at the same screen pixel from frame to frame.

5.3.2 Screen Coordinates Independent Memoization

Screen coordinates are 2D coordinates that describe the location of a fragment in the screen, i. e. which pixel overlaps the fragment. The last GPU stage, the Blending stage, employs these coordinates to blend the color of the fragment with the color of the corresponding pixel. However, the screen coordinates are not used in most of the Fragment Programs since the color of the objects does not usually depend on the exact screen pixels where they are located.

Using the screen coordinates to form the signature imposes significant constraints to the memoization system, as only fragments located at the same screen pixels can be identified as redundant. Nevertheless, the screen coordinates can be excluded from the signature to expose more redundancy as long as the outcome of the computation does not depend on the location of the fragment in the screen. In order to implement this optimization our GPU driver generates information about the usage of the input registers. We expose this information to the GPU, so that the screen coordinates are only employed to form the signature in case the compiler indicates they are accessed in the Fragment Program.

The benefits of excluding the screen coordinates from the signature are illustrated in Figure 5.11. By using this optimization the memoization system is able to capture more redundancy if there are camera movements and objects moving on the screen. All the numbers reported in section 5.4 include this optimization, since we found it to be beneficial for all the workloads.

Table 5.1: Parameters employed for the experiments.

GPU clusters	2	**Raster Units/cluster**	2
L2 cache	128 KB, 8-way	**Texture caches**	16 KB, 4-way
Tile Cache	16 KB, 2-way	**Vertex Cache**	4 KB, 2-way
Lookup Table			
Number of sets: 8, 16, 32, 64, 128, 256, 512, 1024, 2048			
Number of ways: 2, 4, 8			
Signature size: 24, 26, 28, 30, 32, 64, 128			

5.4 Experimental Results

We have implemented the hardware memoization system described in Section 5.3 on top our GPU timing simulator (see Section 2.1.3). We have generated GPU instruction and memory traces for the twelve Android games presented in Section 2.2.1. During trace generation we tried to avoid unrealistic situations that favor our technique by artificially increasing the degree of redundancy. For example, in some games if the user does not provide any input the screen does not move and then nothing changes from frame to frame, reaching levels of redundancy close to 100%. On the contrary, we tried to capture normal gameplay, by providing inputs that guarantee forward progress and allow the user to complete the targets of the stage. As we depicted in Figure 5.2, the average redundancy in our traces is 38.1%, and most of the games exhibit redundancy levels that are far from the 100% that would provide artificially biased situations.

2D games with static backgrounds are the perfect fit for our memoization technique, but scrolling 2D games and complex 3D games are also amenable to memoization and we have included both types of games in our set of workloads. More specifically, some of our 2D games like *angrybirds* and *badpiggies* include phases with static background and phases with scrolling as illustrated in Figure 5.11. On the other hand, 3D games also exhibit significant degrees of redundancy that come from static background objects, for example the sky, or from 2D content such as GUI components (scores, life bar, dialogues...) or billboards/impostors. Finally, 3D games also include periods with intensive camera movements and periods where the camera is not moving around. Despite the redundancy levels are higher for the periods with static camera, the optimization described in Section 5.3.2 is still able to expose significant redundancy when the camera is moving.

The baseline for all our experiments is a PFR capable GPU similar to that shown in Figure 4.5. This GPU is able to render two frames in parallel, and as such it is already able to benefit from the improved locality in the memory sub-system. The parameters employed during the simulations are summarized in Table 5.1.

We first evaluate the effect of the signature size on performance, energy and image quality. The Lookup Table employed for this sensitivity analysis is 4-way associative and has 1024 sets.

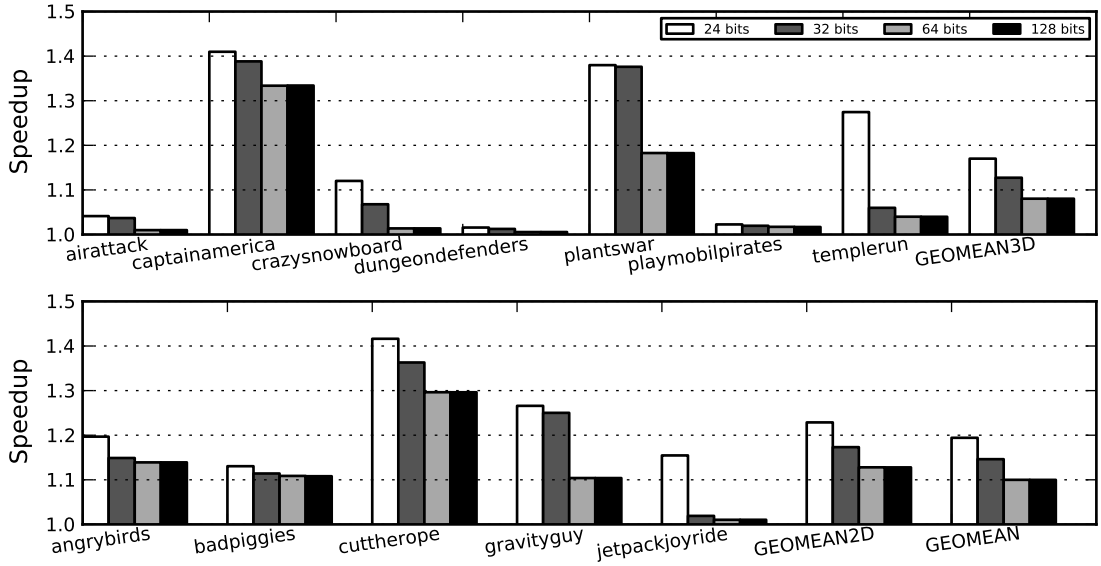


Figure 5.12: Speedups achieved by hardware memoization for different sizes of the signature. The baseline configuration is PFR.

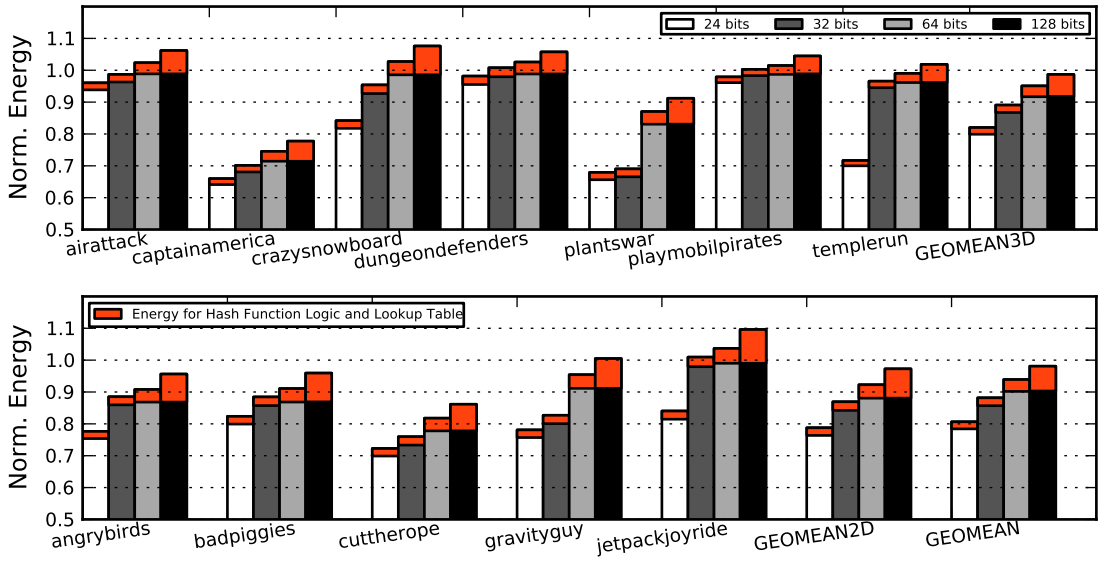


Figure 5.13: Normalized energy for different sizes of the signature. The baseline configuration is PFR.

Figure 5.12 shows the speedups attained for different signature sizes. On average, 9.9% speedup is achieved with a 128-bit signature, whereas 14.6% and 19.5% speedups are obtained by using 32-bit and 24-bit signatures respectively. Reducing the signature increases performance. The smaller the signature the bigger the probability of having a conflict in the LUT (up to some reasonable limit) and, hence, the bigger the hit rate in the LUT. We have a conflict, or false hit, when two different fragments get the same signature and are thus incorrectly identified as redundant, which in turn results in the conflicting fragment getting a wrong color from the LUT. We have measured the number of conflicts and found

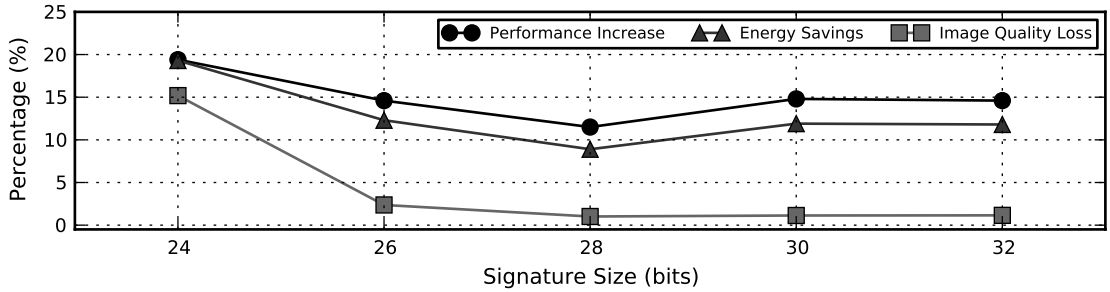
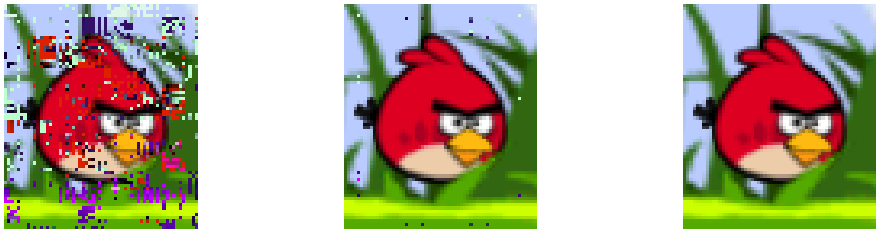


Figure 5.14: Impact of signature size on performance, energy savings and image quality. The baseline configuration is PFR.



(a) 24 bits **(b) 28 bits** **(c) 32 bits**

Figure 5.15: The figure shows a cropped image for a frame of the game Angry Birds for different sizes of the signature: 24, 28 and 32 bits. It illustrates the impact of conflicts in the Lookup Table on image quality.

that for 24-bit signatures we have more than 30k conflicts per frame on average, whereas for 32-bit signatures we only have one conflict every 10 frames. There is no conflict at all for 64-bit and 128-bit signatures. Hence, the 24-bit signature incorrectly removes the execution of more than 30k fragments per frame, achieving significant speedups. Note that the 32-bit signature has a extremely small number of conflicts but it still gets speedups with respect to 128-bit signatures. As we decrease the signature size we also improve the dispersion of the accesses to the LUT, as described in Section 5.3. The bits that are used to select the set in the LUT are computed by using more bits from different sources as the signature size is decreased. This spreads the accesses even when consecutive fragments are just slightly different. Not surprisingly, the 32-bit signature achieves a hit rate of 15.5% on average in the LUT, whereas the 128-bit signature only obtains 9.5% hit rate.

Reducing the signature size improves energy consumption as illustrated in Figure 5.13. The energy savings come from two sources. The obvious source is that the size of the LUT is reduced, since signatures have to be stored in each LUT entry, so both the leakage and dynamic energy required to access the LUT are smaller. Secondly, we avoid more executions of the Fragment Program due to the conflicting fragments that are incorrectly identified as redundant, and also due to the better dispersion of the accesses across the sets of the LUT. On average, switching from 128-bit to 32-bit reduces the energy consumed by the LUT by 68.1% and overall GPU energy by 5.1%.

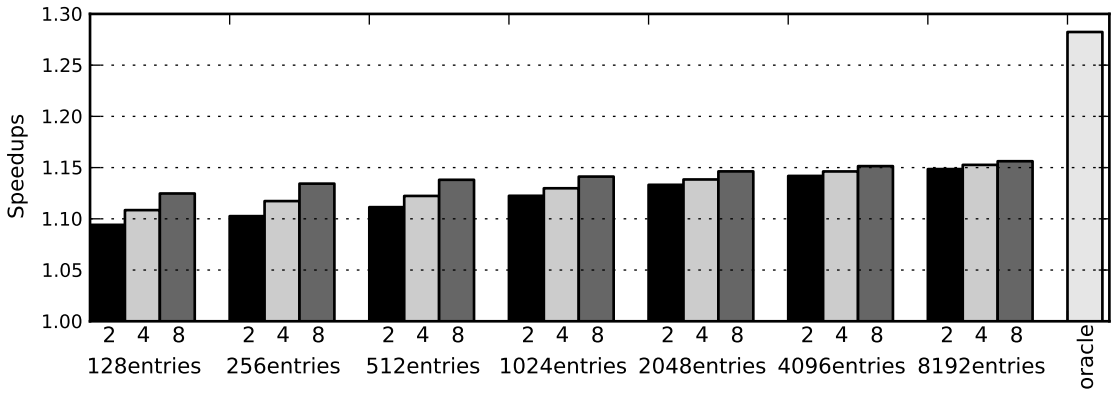


Figure 5.16: Average speedups achieved for LUTs with different sizes and associativity. The baseline configuration is PFR.

Figure 5.14 shows the effect of the signature size on performance, energy and image quality. The 24-bit signature achieves the biggest performance increase and energy savings, but at the cost of a significant percentage of conflicts that introduce noticeable distortions on image quality. More specifically, image quality drops by 15.1%, this is an important percentage that introduces visible artifacts in the images as illustrate in Figure 5.15. For the rest of the experiments we employ 32-bit signatures, since 32-bit still provides significant speedups and energy savings with respect to bigger signatures while achieving high image fidelity.

Figure 5.16 shows the average speedups for 2-way, 4-way and 8-way associative LUTs with a number of entries from 128 to 8192. As expected, increasing the associativity improves performance in all the configurations, especially in the LUTs with a smaller number of entries. For example, for 256 entries the version with 2-ways obtains 10.2% speedup, whereas the version with 8-ways achieves 13.4% performance increase. For a given number of ways, increasing the capacity of the LUT provides noticeable improvements. For instance, for 2-way associativity the version with 128 entries obtains 9.4% speedup, whereas the version with 4096 entries achieves 14.1% speedup. Although the results are far from the Oracle memoization system, all the configurations provide consistent and substantial speedups over the baseline GPU.

Figure 5.17 shows the normalized energy for the same configurations of the LUT. The memoization system provides significant energy savings in all the configurations. The energy savings come from the redundant Fragment Program executions removed, as fragments that hit in the LUT skip all the fragment computations and memory accesses. The savings are in the range from 8.2% —128 entries, 2-way— to 12.08% —4096 entries, 8-way. The figure also shows the percentage of energy consumed by the LUT. LUT energy increases as we increase its capacity and thus part of the savings from removing redundant executions of the fragment shader are lost. Hence, increasing the size of the LUT is only lucrative if the energy savings due to the additional fragments removed are bigger than the extra energy consumed by LUT. As an example, the configuration with 128 entries and 4-way achieves 8.21% energy savings, whereas increasing the number of entries to 4096 provides 11.82% savings. However, switching to 8192 entries

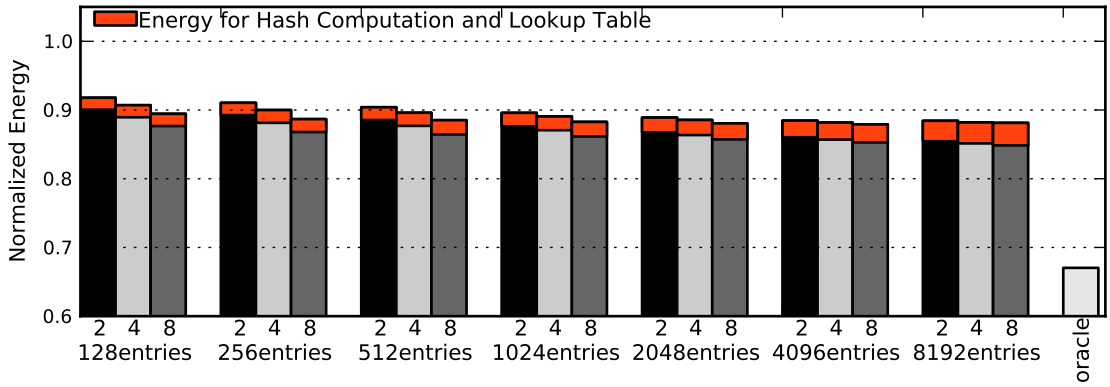


Figure 5.17: Normalized energy for different configurations of the Lookup Table, including both static and dynamic energy. Baseline is PFR.

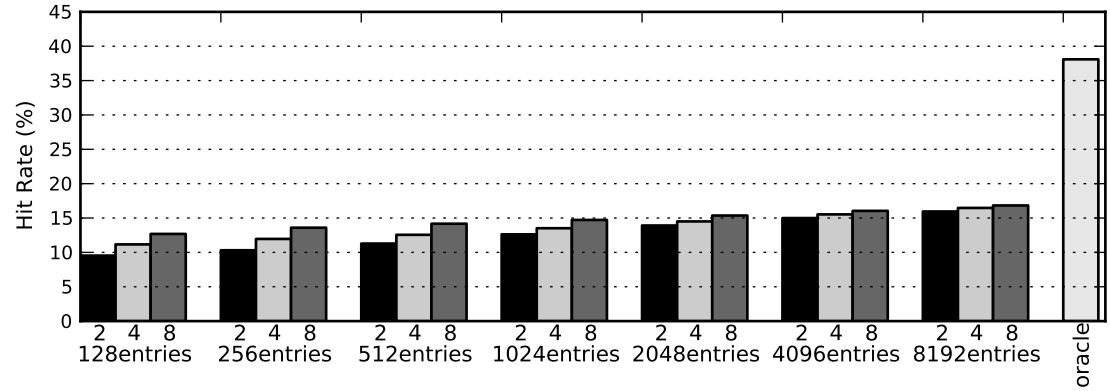


Figure 5.18: Hit Rates for different configurations of the Lookup Table. The baseline configuration is PFR.

results in 11.80% savings. Figure 5.19 shows the normalized energy breakdown for the same configurations of the LUT.

Figure 5.18 depicts the hit rates in the LUT. Again, increasing the associativity produces significant improvements, especially for configurations with a small number of entries. For example, a LUT with 128 entries and 2-way achieves 9.5% hit rate, whereas increasing associativity to 8-way provides 12.7% hit rate. On the other hand, increasing capacity has also a noticeable impact on the hit rate. A 4-way associative LUT achieves 11.1% and 15.5% hit rates with 128 and 4096 respectively. In the best case, we have a 16.8% hit rate, whereas the Oracle achieves 38.1% as it is able to capture redundancy at any re-use distance.

Figure 5.20 depicts the per-game speedups for a hardware memoization system with 32-bit signatures and two configurations for the LUT: 512 sets 8-way and 1024 sets 4-way. As it is shown, the system does not produce slowdowns in any of the games. On the contrary, it is able to achieve 40.1% and 39.1% speedups in *cuttherope* and *captainamerica* respectively. The benefits are small for games such as *dungeondefenders*, 1.3% speedup for an 8-way associative LUT with 512 sets. Hence, the effectiveness of the system is significantly different for different games and it depends on two main factors: the degree of redundancy and the

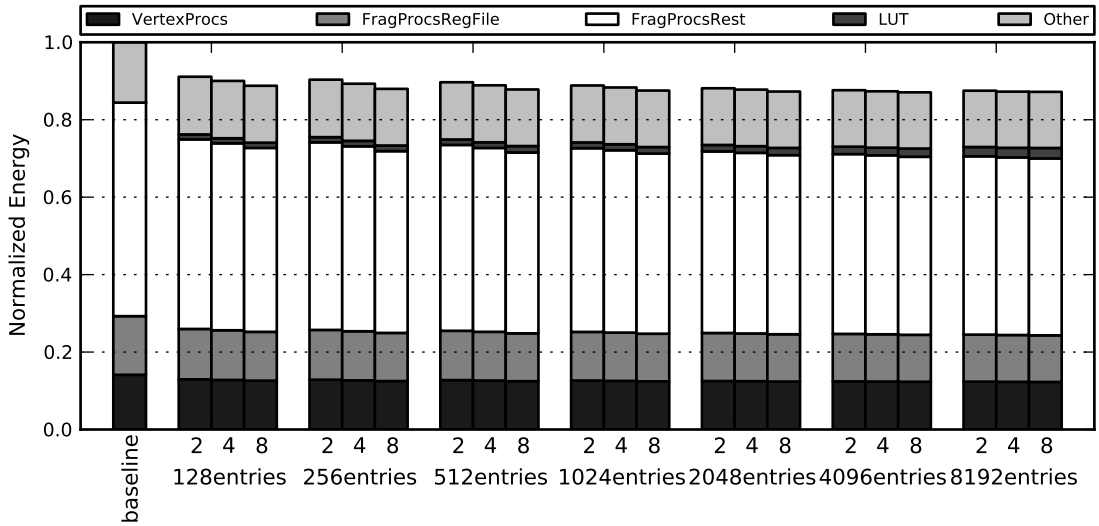


Figure 5.19: Energy breakdown for different configurations of the Lookup Table and the baseline GPU.

importance of the fragment stage in the overall GPU execution time and energy. Figure 5.2 shows that the percentage of redundant Fragment Program executions exhibited by our games is significantly different, as some of the games render scenes with static cameras and backgrounds most of the time, so consecutive frames tend to be extremely similar, whereas other games feature fast scrolling and camera movements that increase frame-to-frame differences. The bigger the degree of redundancy the better for our memoization scheme. Not surprisingly the three games that exhibit bigger degree of redundancy, *cuttherope* (87.5%), *plantswar* (91.2%) and *captainamerica* (79.9%), are the ones that achieve the highest performance improvements. On the contrary, the game with the smallest degree of redundancy, *dungeondefenders* (12.9% redundant fragments), is the one that shows the smallest speedup.

On the other hand, our memoization scheme is an optimization that targets the fragment processing stage of the GPU pipeline and, hence, the overall benefits depend on the importance of the fragment stage in the total GPU execution time and energy. As described in Section 2.2.2, 2D games are fragment bound as the geometry datasets tend to be very simple. 3D games are also fragment bound, but they exhibit much bigger workload in the geometry pipeline and thus the fragment stage represents a smaller percentage. As we can see in Figure 5.20, 2D games achieve better average speedups, 18% and 17.3% for configurations with 512 sets 8-way and 1024 sets 4-way respectively, whereas the 3D games obtain smaller performance improvements of 13.1% and 12.7% for the same configurations.

Finally, Figure 5.21 shows the per-game energy savings. Again, the energy improvements are significantly different for different games, although we do not increase energy consumption in any case. The memoization system provides substantial energy savings for games such as *plantswar* and *cuttherope*, that achieve 31% and 24% savings respectively for a LUT with 1024 sets and 4-way associative. On the contrary, it does not provide any savings for games that exhibit

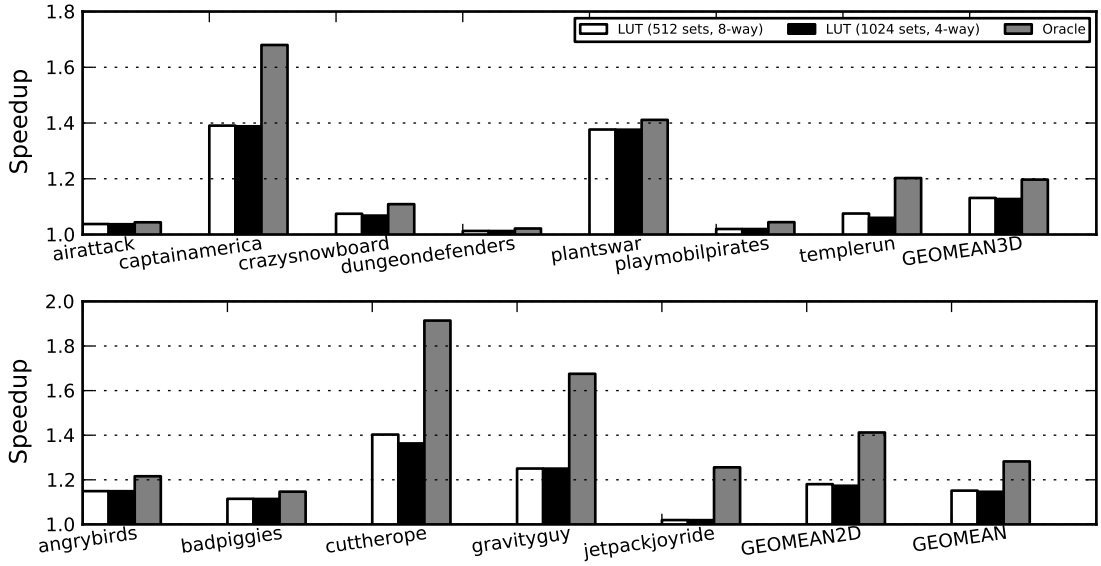


Figure 5.20: Speedups for the twelve Android games and on average. We include two of the best configurations for the LUT and the Oracle memoization. The baseline configuration is PFR.

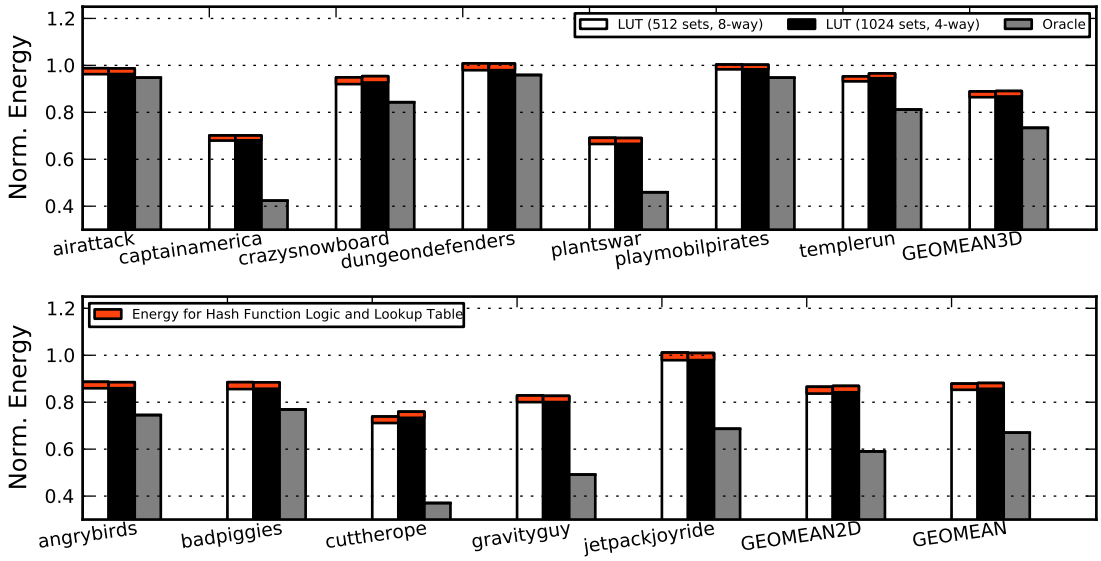


Figure 5.21: Normalized energy for the twelve Android games and on average, including both static and dynamic energy. We include two of the best configurations for the LUT and the Oracle memoization. Baseline is PFR.

small degrees of redundancy, such as *dungeondefenders* or *airattack*.

In this chapter we assume a Tile-Based Rendering architecture as our baseline GPU. Nevertheless, our proposal is orthogonal to the type of rendering architecture and it can also be implemented on top of Immediate-Mode Rendering (IMR). Similar conclusions are obtained when using our memoization system on top of IMR. The experimental results assuming an IMR baseline are provided in Appendix C. The numbers show that the LUT with 512 sets and 8-way associative

achieves 16.4% speedup and 13% energy savings.

5.5 Conclusions

In this Chapter we have shown that more than 38% of the fragment program executions are redundant on average for a set of Android games, suggesting that memoization can be useful to reuse computations in order to save time and energy. However, fragment memoization is no simple task. As we have shown most of the redundancy that could be exploited exists across frames.

We thus proposed to employ fragment memoization on top of techniques that aim to reduce the inter-frame re-use distances of fragments, such as Parallel Frame Rendering (PFR). When we employ PFR, 68.8% of the redundant fragments are brought into re-use distances that make them amenable to hardware memoization. Our memoization scheme is able to achieve significant benefits in both performance and power, with minimal distortion of the frames that are captured. More specifically, when compared with a state-of-art PFR-enabled GPU, our scheme is able to remove enough computation to achieve 15% speedup for a set of commercial Android games. This improves the energy consumption of the system by 12% on average. All this, comes at a negligible cost in terms of image distortion, which as shown in Section 5.4 is not perceivable.

Chapter 6

Conclusions

In this chapter the main conclusions and contributions of this thesis are presented, as well as some open-research areas for future work.

6.1 Conclusions

A complete and comprehensive analysis of mobile graphics processors has been made, together with an evaluation of multiple energy saving techniques for low-power GPUs, yielding the following main conclusions.

In first place, the results obtained by using a cycle-accurate GPU simulator and several commercial Android applications indicate that multithreading and prefetching are effective techniques for tolerating memory latency, but not energy-efficient. Massive multithreading requires huge register files that increase energy consumption, whereas prefetching accuracy is relatively low due to the unpredictable texture fetches. The access-execute paradigm provides a more efficient solution to bridge the memory gap, as it requires simple hardware and it is based on computed addresses rather than predicted addresses to maximize prefetch accuracy. The experimental results presented in this thesis show that a small degree of multithreading combined with a decoupled access/execute-like architecture provides the most energy efficient solution to hide the memory latency. More specifically, the decoupled access/execute-like design with 4 SIMD threads/processor is able to achieve 97% of the performance of a larger GPU with 16 SIMD threads/processor, while providing 20.5% energy savings.

In second place, the energy numbers obtained by using our GPU power model show that the off-chip memory accesses to system memory are one of the main sources of energy consumption in a mobile GPU, thereby supporting the results presented in previous research work. The analysis of bandwidth usage in several Android games reveals that most of the memory bandwidth is typically employed for fetching textures. Furthermore, consecutive frames tend to be extremely similar, sharing most of the texture dataset. However, the GPU cannot exploit

this inter-frame texture re-use due to the big size of the texture dataset for one frame.

Parallel Frame Rendering (PFR) is a bandwidth saving technique designed with the aim of capturing the inter-frame texture re-use by overlapping the execution of consecutive frames. PFR splits the GPU in two clusters that process two consecutive frames in parallel. Both clusters share texture data via the L2 cache, so textures are fetched from system memory just once every two frames instead of being fetched on a frame basis, saving memory bandwidth. This, however, comes at a cost in the responsiveness of the system, since rendering each frame takes twice the time as it is generated with half of the resources, increasing input lag.

Adaptive forms of PFR can be employed to trade responsiveness for energy efficiency. The analysis of user interaction in mobile games shows that the user does not provide any input most of the time. In addition, user inputs tend to be heavily clustered: mobile games exhibit short phases with user inputs and long phases where the user does not provide any input. PFR can be extensively employed during phases without user inputs, where input lag is not a problem. The reactive versions of PFR achieve high responsiveness while providing 23.8% reduction in off-chip memory traffic, that results in 12% speedup and 20.1% energy savings on average.

In third place, an analysis of redundancy in mobile games reveals that more than 38% of the Fragment Program executions are redundant on average, suggesting that memoization can be effective to reuse computations in order to save time and energy. However, capturing this redundancy is not an easy task as most of the redundancy that could be exploited exists across frames at huge re-use distances, rendering most of hardware-based approaches unfeasible.

A fragment memoization system can be architected on top of techniques that aim to reduce the inter-frame re-use distances of fragments, such as the aforementioned PFR. Under PFR, 68.8% of the redundant fragments are brought into re-use distances that make them amenable for hardware memoization. A memoization system based on a small hardware Look Up Table achieves 15% speedup over a state-of-the-art PFR-enabled GPU, reducing energy consumption by 12% on average.

In summary, the experimental results presented in this thesis show that decoupled access/execute architectures, combined with a small degree of multithreading, provide the most energy-efficient solution for hiding the memory latency in a mobile GPU. Furthermore, using PFR to execute multiple frames in parallel provides noticeable energy savings that come from a significant reduction in memory bandwidth usage, as consecutive frames tend to be extremely similar and thus they share most of the texture dataset. GPU energy consumption can be further improved by using a memoization scheme on top of PFR to avoid redundant executions of the Fragment Program.

6.2 Contributions

In this dissertation different energy saving techniques for mobile graphics processors have been proposed, from the development of energy efficient memory latency tolerance techniques to the design of strategies for optimizing memory bandwidth usage. Furthermore, a mobile GPU simulation infrastructure has been developed as part of this thesis. The contributions obtained through this dissertation are summarized as follows.

In first place, we propose the migration of GPU designs towards the access-execute paradigm. This thesis introduces a decoupled access/execute-like architecture for the fragment processors of a mobile GPU. In this scheme, all the necessary data for processing the fragments is prefetched into the texture caches while the fragments are waiting to be dispatched to the fragment processors. By the time a fragment is issued all the data required for its processing is hopefully available in the caches, avoiding cache miss stalls to a large extent. The base decoupled access/execute-like design is further improved by allowing for remote texture cache requests to exploit the high degree of data sharing among fragment processors. The end-design easily outperforms conventional hardware prefetching schemes and achieves similar performance to a heavily multithreaded GPU, but consuming just a fraction of its energy as described in Chapter 3. This work has been published in the proceedings of the 39th International Symposium on Computer Architecture (ISCA):

- "Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2012.

In second place, we propose Parallel Frame Rendering (PFR), a bandwidth saving technique that overlaps the rendering of consecutive frames. Under PFR the GPU is split in two clusters that share texture data via the second level shared cache. The first cluster renders odd frames and the second cluster renders even frames. By processing two frames in parallel, textures are fetched once every two frames instead of being fetched on a frame basis as in conventional GPUs, providing significant bandwidth savings. To minimize the impact on responsiveness, different variations of PFR that are reactive to user interaction are proposed in this thesis. The reactive versions of PFR achieve high responsiveness, while providing significant bandwidth savings that result in improvements in performance and energy consumption as reported in Chapter 4. This work has been published in the proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT):

- "Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.

International Conference on Parallel Architectures and Compilation Techniques, 2013.

In third place, we provide an analysis of the per-fragment redundancy as seen in commercial Android applications, showing that more than 38% of the Fragment Program executions are redundant on average. We propose to exploit this high degree of redundancy by using a task-level, hardware based memoization system able to identify and skip redundant executions of the Fragment Program. The proposed memoization scheme keeps a hardware structure that computes a signature of all the inputs to a task and caches the values of the corresponding fragments. Subsequent computations form the signature and check against the stored signatures of the memoized fragments. In case of a hit in the hardware structure the system skips the execution of the Fragment Program, avoiding all the corresponding computations and memory accesses. When architected on top of PFR, a memoization system based on a small hardware Look Up Table achieves significant performance improvements and energy savings as reported in Chapter 5. This work has been published in the proceedings of the 41st International Symposium on Computer Architecture (ISCA):

- "Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Symposium on Computer Architecture, 2014.

Finally, a mobile GPU simulation infrastructure has been developed as part of this thesis. The simulator is extensively described in Chapter 2. To the best of our knowledge, this is the first GPU simulator able to run and profile Android graphical applications that employ the OpenGL ES API. The simulator consists on a set of tools for evaluating the performance, energy consumption and image quality of mobile graphics processors. This infrastructure was presented in a paper published in the proceedings of the 27th International Conference on Supercomputing:

- "TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems".
  Jose-Maria Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis.
  International Conference on Supercomputing, 2013.

6.3 Open-Research Areas

Mobile graphics processors are a very recent research field that appeared with the huge expansion of the smartphones and tablets market. The bibliography on mobile GPU energy is not extensive, on the contrary, there is just a relatively small number of papers that target smartphones and mobile workloads. Further

research efforts will be necessary to bridge the energy gap in next generations of smartphones and tablets. It seems that most of the energy savings will come from an efficient use of memory bandwidth, as it is stated by previous research work and corroborated in this thesis.

An interesting extension of the work proposed in this thesis would be to try to apply Parallel Frame Rendering (PFR) more aggressively to achieve bigger bandwidth savings. For example, a system could attempt to overlap 8 frames instead of just 2 or 4. The main issue in doing so is responsiveness, as the increase in input lag is proportional to the number of frames rendered in parallel if the hardware resources are not increased. The reactive versions of PFR can be employed to achieve high responsiveness, but they suffer from microstuttering since switching between rendering modes requires buffering or dropping several frames. PFR tries to minimize the number of switches to avoid microstuttering, setting restrictive values for the thresholds that are employed to decide when to switch between rendering modes. However, this severely constrains the percentage of time that frames are overlapped, losing part of the bandwidth savings. Furthermore, the stuttering effects are exacerbated as the number of frames rendered in parallel is increased, since more frames have to be buffered or dropped. More efficient ways of switching between rendering modes are necessary to increase the number of frames rendered in parallel, in order to maximize bandwidth savings while maintaining the same levels of user experience.

On the other hand, an interesting extension of the memoization system presented in Chapter 5 would be to try to bridge the gap with the oracle memoization by capturing redundant fragments at big re-use distances. This would probably require extending the capacity of the Look Up Table (LUT) in some way. For example, the system could employ a virtual LUT stored in main memory and cached on-chip. On the other hand, a software based approach could be used instead. The LUT could be stored in a texture that would be accessed from the fragment shader to identify redundant computations.

The memoization system could be further improved by allowing the software to provide hints to the GPU about the expected degree of redundancy. The memoization system is very effective for phases of the application where the camera is not moving, as consecutive frames tend to be very similar and more redundant computations are captured and avoided. On the contrary, it performs worse in the presence of fast camera movements. As the information about the behavior of the camera is available in the software, the game engine could expose this information to the hardware in order to disable memoization when it is know to be useless, saving energy for accessing the LUT.

In this thesis we focus on optimizing the bandwidth for texture fetching, as textures represent a huge percentage of bandwidth in typical mobile workloads. However, desktop-like games feature more complex scenes with hundreds of thousands or even millions of vertices per frame. In case desktop-like games become popular in smartphones and tables —something that would require significant improvements in mobile user interfaces to provide compelling gaming experience— it could be interesting to consider bandwidth optimizations for geometry fetching.

For example, desktop GPUs employ a technique called geometry instancing to efficiently render objects that are repeated in a crowded scene. By using instancing the geometry is sent to the GPU once and employed to render the multiple copies of the object, saving bandwidth. Geometry instancing could be combined with PFR to perform multi-frame instancing: geometry could be fetched once and employed to render all the instances of the same object in two consecutive frames to further improve bandwidth savings.

GPGPU research is also an intriguing direction, as it seems it will be widely supported in the mobile segment. This thesis is focused in graphics workloads and OpenGL ES since it is the main API for accessing the GPU in smartphones and tablets. Nevertheless, it could also be interesting to consider energy saving techniques for OpenCL/CUDA applications in embedded graphics processors. The mobile GPU simulation infrastructure presented in Chapter 2 can be easily extended to support GPGPU applications. The Gallium3D driver, employed for GPU trace generation in the infrastructure, offers a front-end for OpenCL, GalliumCompute. Although still immature, this front-end could be used to trace OpenCL commands and OpenCL kernels instead of OpenGL ES rendering commands and GLSL shaders. On the other hand, the simulator could be updated to support the version 3.1 of OpenGL ES. This version provides "compute shaders", offering a powerful and flexible form of general purpose computation in a mobile GPU, by using the same graphics API and the same GPU driver that is employed for rendering.

Appendix A

Decoupled Fragment Processor on top of TBR

In chapter 3 we present our decoupled access/execute-like architecture for the fragment processors of a mobile GPU, and we evaluate our proposal on top of an Immediate-Mode Rendering (IMR) architecture. Nevertheless, our scheme is completely orthogonal to the type of rendering architecture and, hence, it can also be implemented on top of a Tile-Based Rendering (TBR) pipeline like the one illustrated in Figure 2.2.

In this chapter we include the evaluation of our proposal using TBR as our baseline GPU instead of IMR. We use the same parameters that we employed for IMR (Table 3.1), but we configure our GPU timing simulator to model a TBR pipeline. Figures A.1, A.2 and A.3 show the average speedup, normalized energy and energy-efficiency respectively. Figure A.4 shows the per-game energy-efficiency. As we can see, the decoupled architectures combined with a small degree of multithreading are the configurations that achieve the highest energy-efficiency. More specifically, the decoupled architecture with just 2 SIMD threads/processor achieves 93% of the performance of a larger GPU with 16 SIMD threads/processor, while providing 28.2% energy savings. This means 29.5% improvement in energy-efficiency (speedup divided by normalized energy).

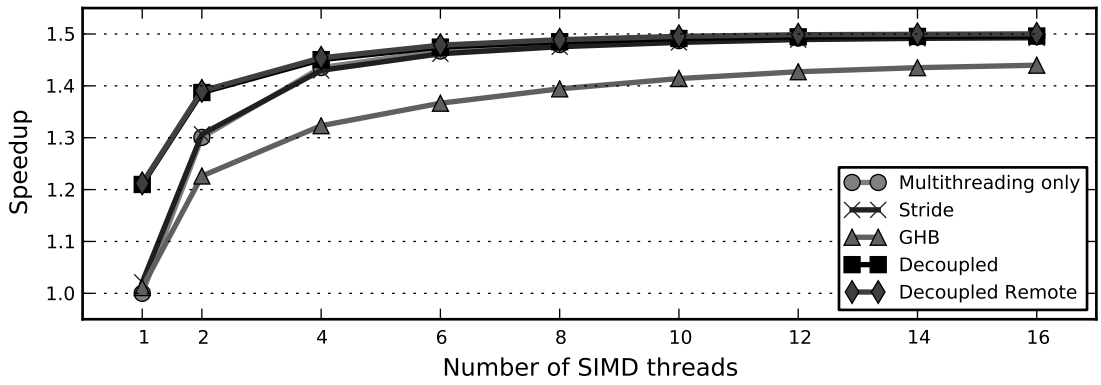


Figure A.1: Speedups achieved when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching.

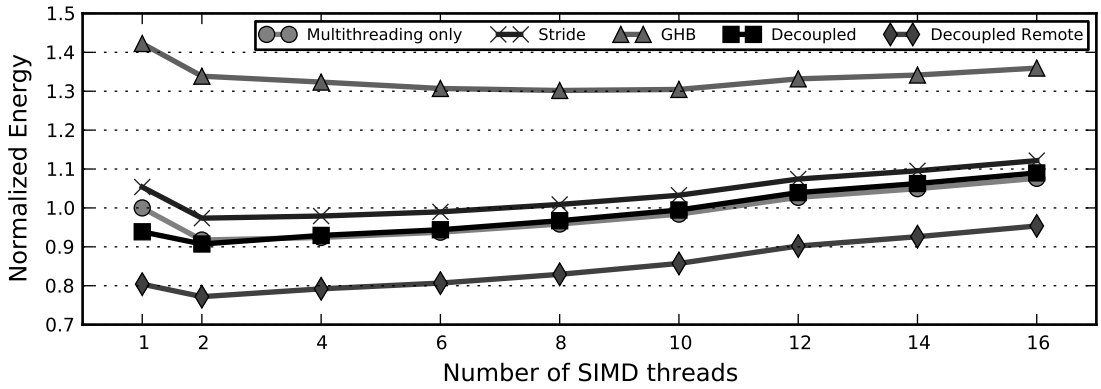


Figure A.2: Normalized energy obtained when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching.

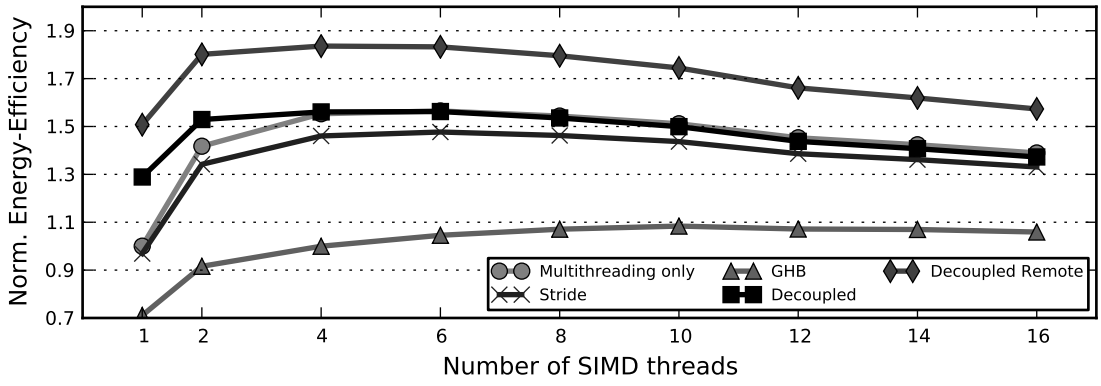


Figure A.3: Normalized energy efficiency obtained when combining multithreading with different prefetching schemes. The baseline is a mobile GPU with just one SIMD thread per fragment processor and no prefetching. The energy-efficiency is computed as the speedup divided by the normalized energy.

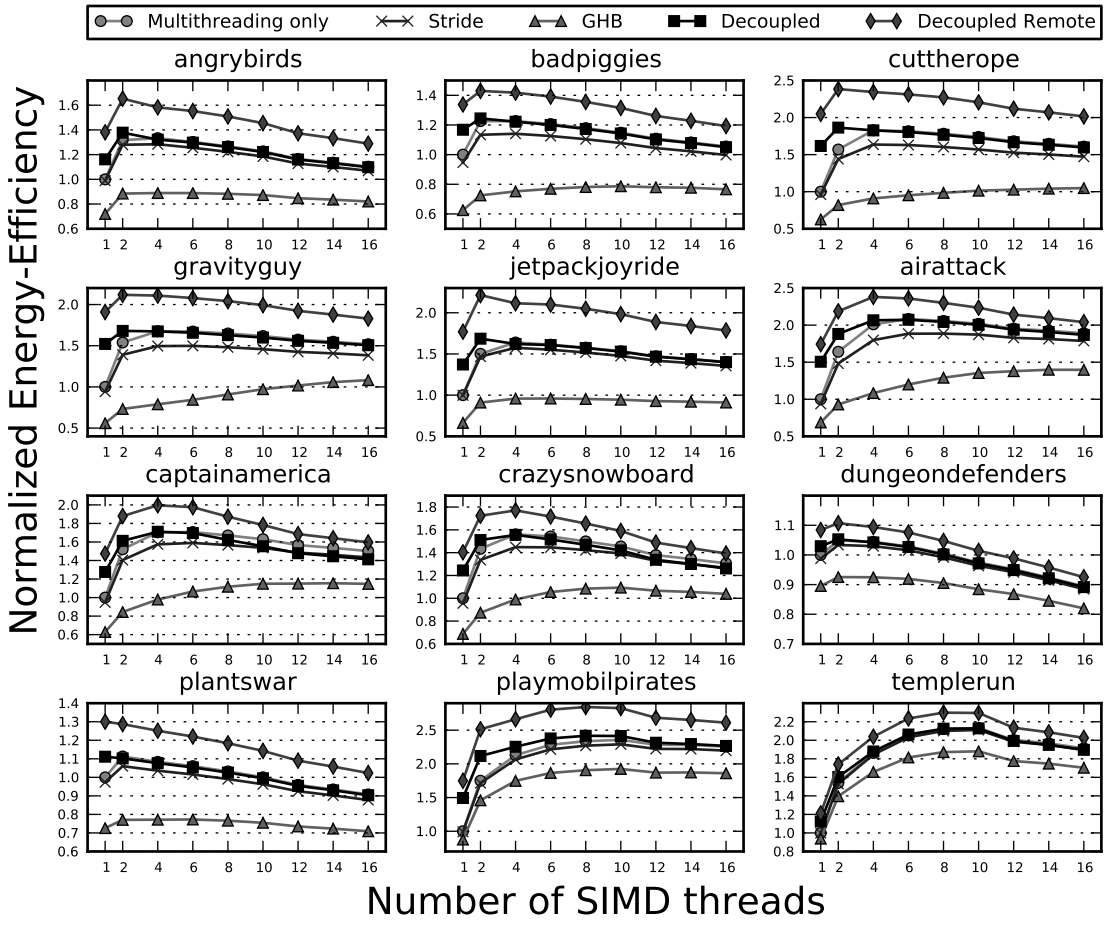


Figure A.4: Normalized energy efficiency obtained for each one of the workloads. The energy-efficiency is computed as the speedup divided by the normalized energy.

Appendix B

Parallel Frame Rendering on top of IMR

In chapter 4 we present Parallel Frame Rendering (PFR), a memory bandwidth saving technique that overlaps the processing of multiple frames. We evaluate our proposal assuming a Tile-Based Rendering (TBR) architecture as the baseline GPU. Nevertheless, PFR is completely orthogonal to the type of rendering architecture and, hence, it can also be implemented on top of an Immediate-Mode Rendering (IMR) pipeline like the one illustrated in Figure 2.2. In this chapter we first describe how PFR can be adapted for IMR architectures. Next, we provide the experimental results when assuming an IMR baseline.

PFR processes multiple frames in parallel to optimize memory bandwidth usage for texture fetching, as the texture datasets for consecutive frames tend to be extremely similar. The GPU is split in two clusters, both clusters sharing the second level cache to exploit texture reuse. Therefore, textures are fetched just once and employed to render the objects in two consecutive frames, instead of being fetched on a frame basis and thus saving bandwidth. The technique is only effective if both clusters access the same textures within a short timespan. In an effort to maximize texture overlapping, we synchronize both GPU clusters at the tile level —we assume a TBR architecture in chapter 4— so both GPU clusters work in the same screen tile in two consecutive frames. As the same screen tile usually contains the same objects from frame to frame, tile-level synchronization provides a good degree of texture overlapping.

IMR architectures do not split the screen in tiles, they consider the entire screen at once. Hence, a different strategy to guarantee texture overlapping is necessary when using an IMR architecture, as we cannot implement tile-level synchronization. Since IMR architectures render the scene object by object instead of tile by tile, we propose to use object-level synchronization and try to render in parallel objects that use the same textures. Hence, our GPU driver for IMR reschedules the drawing commands issued by the application in order to overlap the rendering of objects that access the same textures. For every drawing command in the first frame, the GPU driver tries to find a GPU command in the

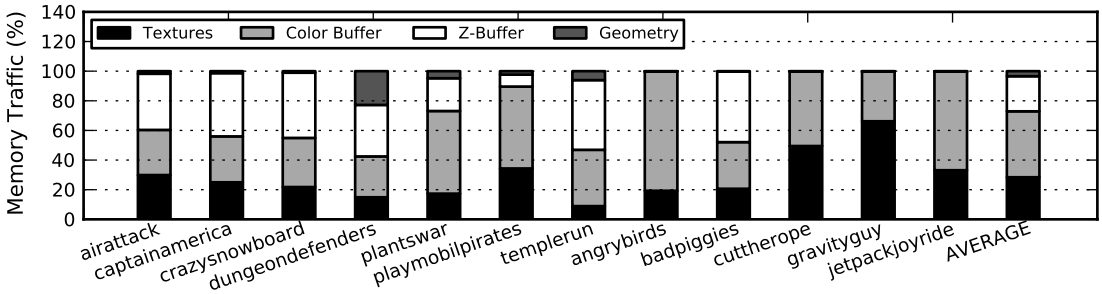


Figure B.1: Memory bandwidth usage on a mobile GPU for a set of commercial Android games. On average 28.4% of the bandwidth to system memory is employed for fetching textures. The mobile GPU implements an IMR architecture.

Table B.1: GPU simulator parameters. All the configurations include the same amount of resources: 1 big GPU cluster or 2 clusters with half the resources for each cluster.

GPU Architecture	Immediate-Mode Rendering	
L2 Cache	128 KB, 8-way associative, 12 cycles latency	
Texture Caches	16 KB, 4-way associative, 2 cycles latency	
Pixel Caches	16 KB, 4-way associative, 2 cycles latency	
Main Memory	1 GB, 16 bytes/cycle (dual-channel)	
	Conventional rendering	**PFR**
Number of clusters	1	2
Raster units per cluster	2	1
Vertex Processors per cluster	2	1
Vertex Cache	8 KB, 2-way	4 KB, 2-way
Vertex Fetcher	16 in-flight vertices	8 in-flight vertices
Primitive Assembly	4 triangles/cycle	2 triangles/cycle

second frame that accesses the same textures. Next, the driver sends the first pair of objects to the GPU, each one to a different cluster. Once both clusters are done with the rendering, the driver dispatches the next pair of objects. By processing objects with the same textures in parallel and synchronizing the GPU clusters at the object level we attempt to maximize the degree of texture overlapping.

On the other hand, the memory bandwidth usage is significantly different in IMR, as illustrated in Figure B.1. Only 28.4% of the memory bandwidth is employed for fetching textures, which is the target of PFR, whereas textures account for 52.7% of the bandwidth when using a TBR architecture (see Figure 4.1). This difference is due to the overdraw. TBR architectures are very efficient in accessing the Color Buffer and the Z-Buffer, as they divide the screen in tiles and use local on-chip memories instead of regular caches. In TBR each pixel in the Color Buffer is written just once per frame in system memory, whereas it can potentially be written multiple times in IMR. Hence, the Color Buffer and Z-Buffer accesses

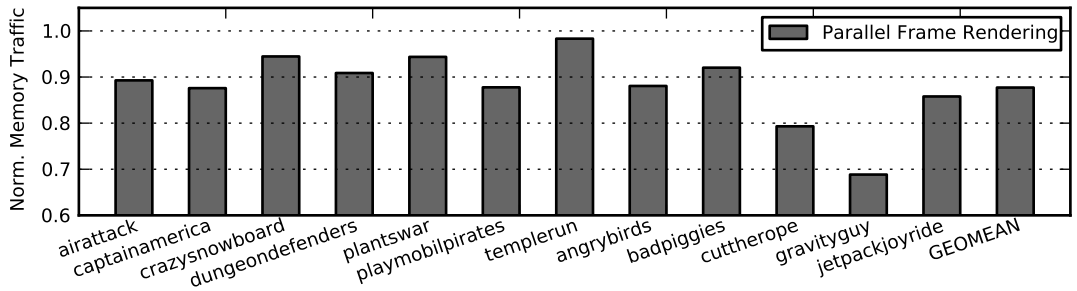


Figure B.2: Normalized memory traffic. The baseline is a mobile GPU that employs Immediate-Mode Rendering and processes just one frame at a time.

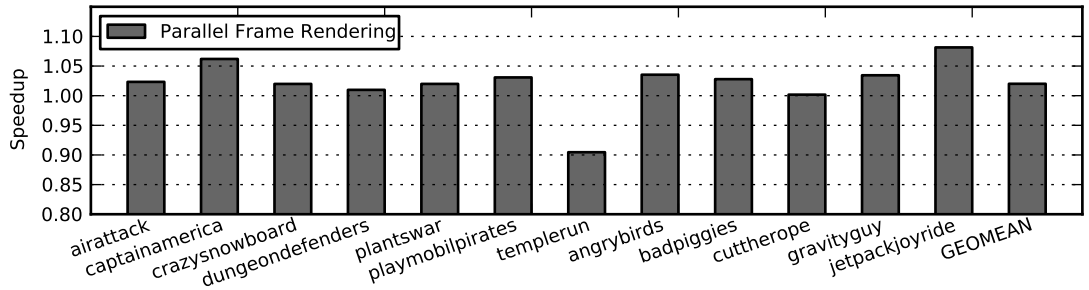


Figure B.3: Speedups. The baseline is a mobile GPU that employs Immediate-Mode Rendering and processes just one frame at a time.

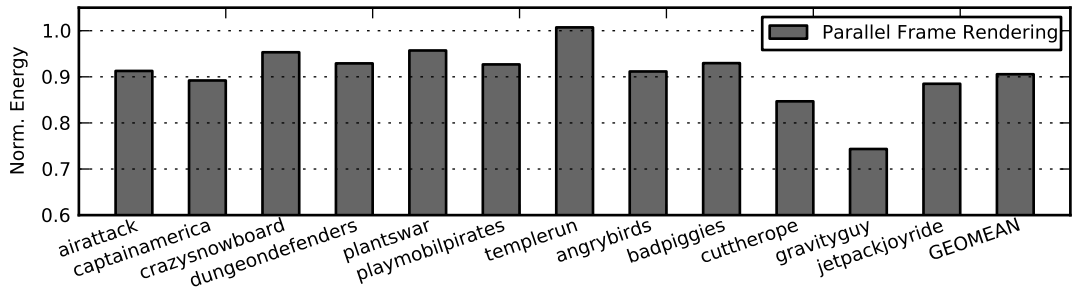


Figure B.4: Normalized energy. The baseline is a mobile GPU that employs Immediate-Mode Rendering and processes just one frame at a time.

represent a bigger percentage of bandwidth in IMR. Nevertheless, textures still represent a significant amount of bandwidth in IMR, 28.4% on average. If perfect texture overlapping can be achieved, then PFR can save up to 14.2% overall GPU bandwidth —half of the texture accesses.

We have evaluated PFR using an IMR architecture as the baseline. We employ the parameters described in Table B.1. The baseline is an IMR architecture with one big GPU cluster that renders just one frame at a time, whereas the PFR configuration includes two half-sized GPU clusters to render two frames in parallel. Figure B.2 shows the normalized memory traffic. PFR saves 13% of memory bandwidth on average, close to the 14.2% theoretical limit. Figure B.3 shows the speedups. The performance is nearly the same on average (2% speedup) and in all the games except *templerun*. The slowdown in *templerun* is due to the object-level synchronizations aforementioned. By synchronizing both clusters at

the object level we maximize texture overlapping. However, if the two objects rendered in parallel exhibit significantly different workload, then the synchronizations introduce workload imbalances that affect performance. This issue can be solved by avoiding the synchronization in case of workload imbalance, or by improving the scheduling of the commands in order to process in parallel objects with similar workload. We leave this improvement as future work. Finally, Figure B.4 shows the normalized energy. PFR saves 10% of overall GPU energy on average.

Appendix C

Hardware Memoization on top of IMR

In chapter 5 we present our hardware memoization system for mobile GPUs, and we evaluate our proposal on top of a Tile-Based Rendering architecture (TBR). Nevertheless, our scheme is completely orthogonal to the type of rendering architecture and, hence, it can also be implemented on top of an Immediate-Mode Rendering (IMR) pipeline like the one illustrated in Figure 2.2.

In this chapter we include the evaluation of our proposal using IMR as our baseline GPU instead of TBR. We use the same parameters that we employed for TBR (Table 5.1), but we configure our GPU timing simulator to model an IMR pipeline. Our baseline is a Parallel Frame Rendering (PFR) capable GPU. PFR is implemented on top of IMR as described in Appendix B.

Figures C.1, C.2 and C.3 show the average hit rates, speedups and normalized energy respectively for different Look Up Table (LUT) sizes. Figures C.4 and C.5 show the per-game speedups and normalized energy respectively. As we can see, the hardware memoization system is able to provide significant speedups and energy savings. More specifically, the LUT with 512 sets and 8-way associative achieves 16.4% speedup and 13% energy savings.

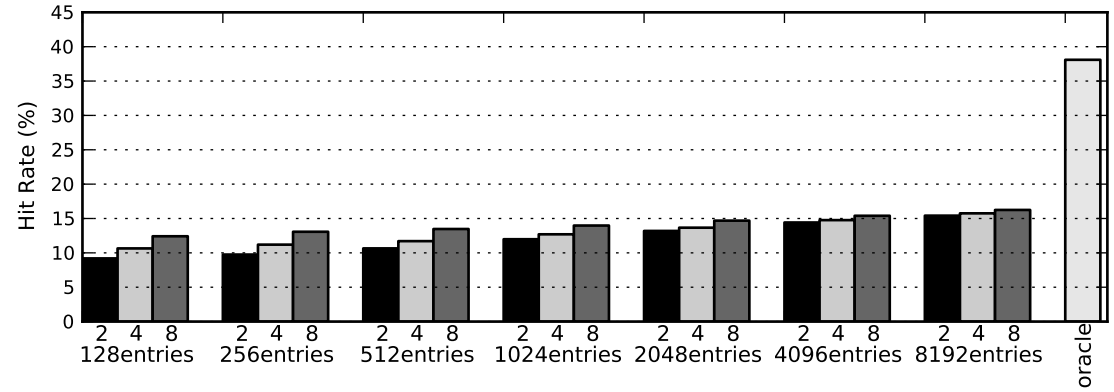


Figure C.1: Hit Rates for different configurations of the Lookup Table. The baseline configuration is PFR.

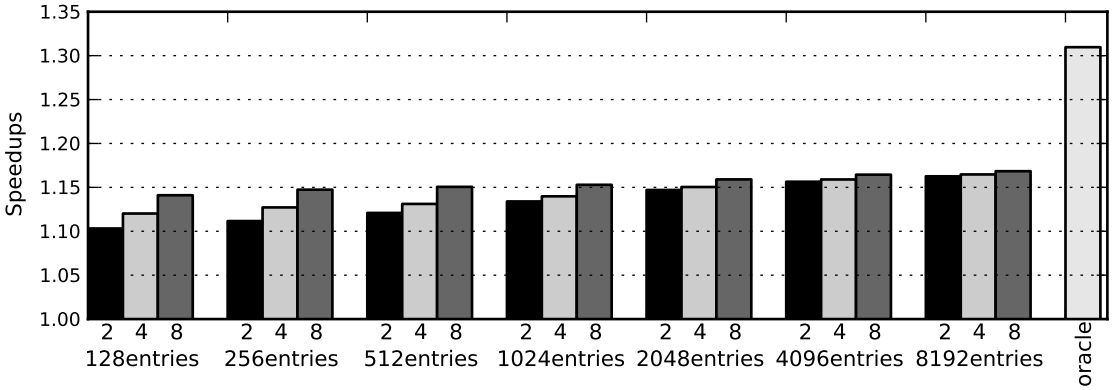


Figure C.2: Average speedups achieved for LUTs with different sizes and associativity. The baseline configuration is PFR.

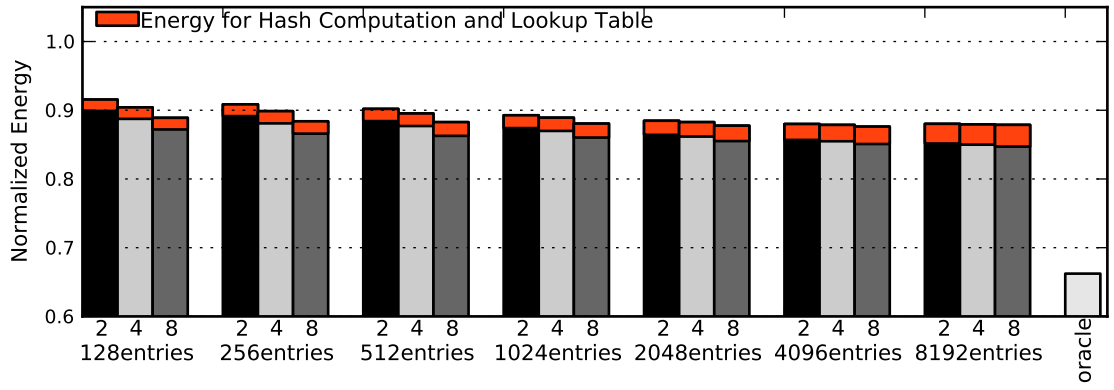


Figure C.3: Normalized energy for different configurations of the Lookup Table, including both static and dynamic energy. Baseline is PFR.

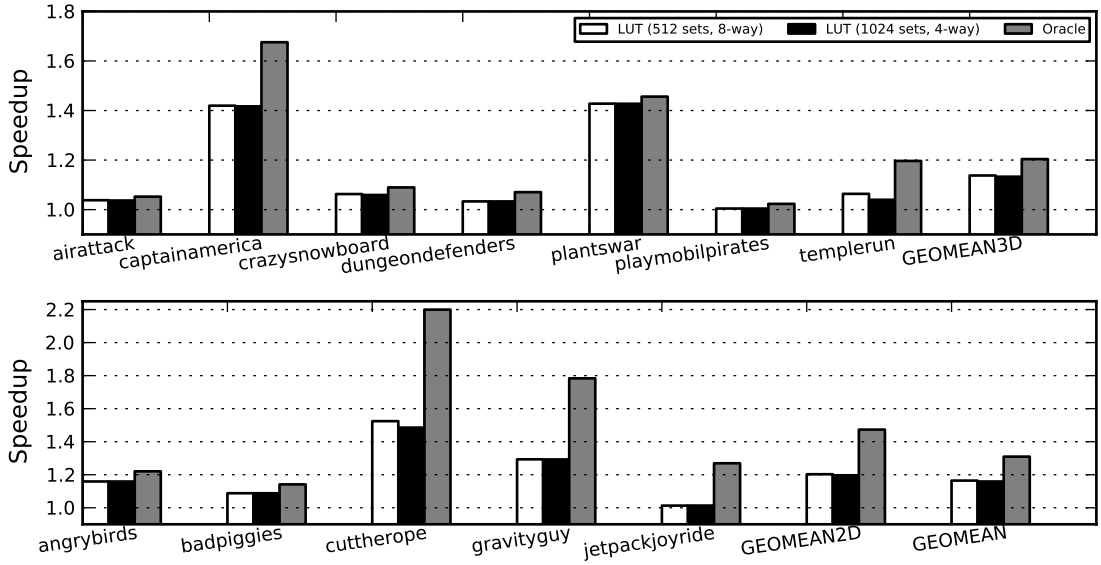


Figure C.4: Speedups for the twelve Android games and on average. We include two of the best configurations for the LUT and the Oracle memoization. The baseline configuration is PFR.

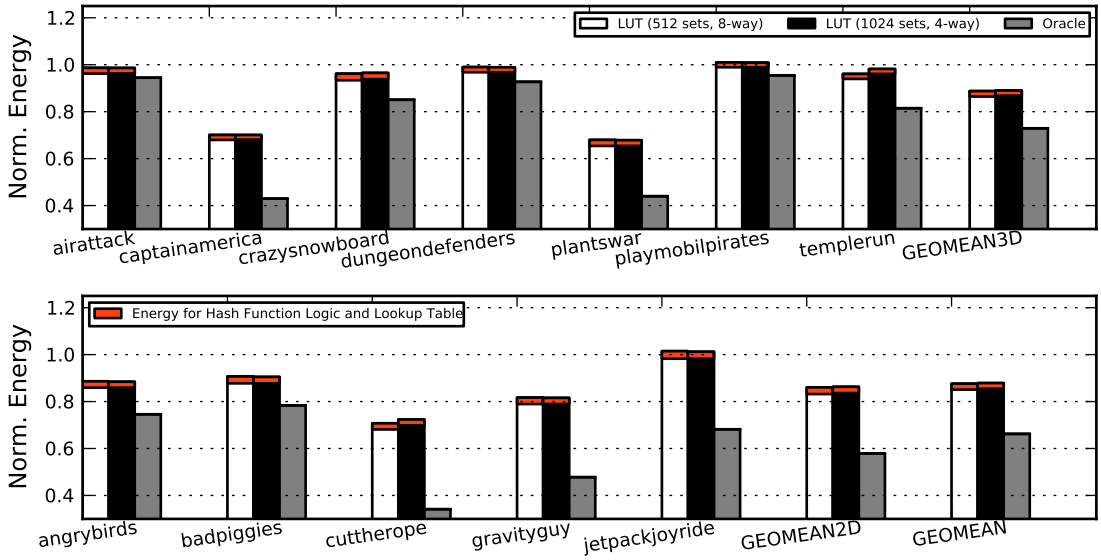


Figure C.5: Normalized energy for the twelve Android games and on average, including both static and dynamic energy. We include two of the best configurations for the LUT and the Oracle memoization. Baseline is PFR.

Bibliography

[1] AMD Compressed ATC Texture. http://www.khronos.org/registry/gles/extensions/AMD/AMD_compressed_ATC_texture.txt.

[2] Android Graphics. https://source.android.com/devices/graphics.html.

[3] Android SDK. http://developer.android.com/sdk/index.html.

[4] Angry Birds has been downloaded more than a billion times. http://bgr.com/2012/05/09/angry-birds-iphone-android-billion-downloads/.

[5] Apple iPad Air. http://www.gsmarena.com/apple_ipad_air-5797.php.

[6] Apple iPhone 5S. http://www.gsmarena.com/apple_iphone_5s-5685.php.

[7] ARB Fragment Program. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

[8] ARM Frame Buffer Compression. http://www.arm.com/products/multimedia/mali-technologies/arm-frame-buffer-compression.php.

[9] ARM Mali. http://www.arm.com/products/multimedia/mali-graphics-hardware/.

[10] Blending in OpenGL. http://www.opengl.org/wiki/Blending.

[11] FlexRender. http://www.qualcomm.com/media/videos/flexrender-rendered-useful.

[12] Gallium3D Technical Overview. http://www.freedesktop.org/wiki/Software/gallium/.

[13] Get started with compute shaders. http://community.arm.com/groups/arm-mali-graphics/blog/2014/04/17/get-started-with-compute-shaders?sf25287757=1.

[14] GFXBench - Unified Graphics Benchmark based on DXBenchmark (DirectX) and GLBenchmark (OpenGL ES). http://gfxbench.com/result.jsp.

[15] Hardware Occlusion Queries Made Useful. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html.

[16] Highlights from the Candy Crush IPO filing: 500 Million Downloads and Counting. http://www.businessweek.com/articles/2014-02-18/king-digitals-ipo-filing-shows-500-million-candy-crush-downloads.

[17] HTC One X. http://www.gsmarena.com/htc_one_x-4320.php.

[18] Huawei Ascend G615. http://www.gsmarena.com/huawei_ascend_g615-5257.php.

[19] iOS Dev Center. https://developer.apple.com/devcenter/ios/index.action.

[20] The Khronos Group Inc. http://www.khronos.org/.

[21] Killing Pixels - A New Optimization for Shading on ARM Mali Gpus. http://community.arm.com/groups/arm-mali-graphics/blog/2013/08/08/killing-pixels--a-new-optimization-for-shading-on-arm-mali-gpus.

[22] LG Nexus 5. http://www.gsmarena.com/lg_nexus_5-5705.php.

[23] The Mali GPU: An Abstract Machine, Part 3 - The Shader Core. http://community.arm.com/groups/arm-mali-graphics/blog/2014/03/12/the-mali-gpu-an-abstract-machine-part-3--the-shader-core.

[24] Number of Android applications. http://www.appbrain.com/stats/number-of-android-apps.

[25] NVIDIA SHIELD. http://shield.nvidia.com/.

[26] NVIDIA Tegra. http://www.nvidia.com/object/tegra.html.

[27] Playstation Vita. http://us.playstation.com/psvita/tech-specs/.

[28] PowerVR Series6 GPU. http://www.imgtec.com/powervr/series6.asp.

[29] QEMU. http://wiki.qemu.org/Main_Page.

[30] Qualcomm Snapdragon S4 (Krait) Performance Preview - 1.5 Ghz MSM8960 MDP and Adreno 225 Benchmarks. http://www.anandtech.com/show/5559/qualcomm-snapdragon-s4-krait-performance-preview-msm8960-adreno-225-bench

[31] Samsung Galaxy S3. http://www.gsmarena.com/samsung_galaxy_s_iii_i747-4803.php.

[32] Samsung Galaxy S5. http://www.gsmarena.com/samsung_galaxy_s5-6033.php.

[33] Samsung Galaxy Tab 3. http://www.gsmarena.com/samsung_galaxy_tab_3_lite_7_0_3g-5975.php.

[34] Sony Xperia Tablet S. http://www.gsmarena.com/sony_xperia_tablet_s-4913.php.

[35] Sony Xperia Z2. http://www.gsmarena.com/sony_xperia_z2-6144.php.

[36] Transform Feedback. https://www.opengl.org/wiki/Transform_Feedback.

[37] Tungsten Graphics Shader Infrastructure. http://people.freedesktop.org/~csimpson/gallium-docs/tgsi.html.

[38] Unity Game Engine. http://unity3d.com/.

[39] Unreal Engine Technology. https://www.unrealengine.com/.

[40] Using Hardware Acceleration in the Android Emulator. http://developer.android.com/tools/devices/emulator.html#acceleration.

[41] Vivante Composition Processing Cores. http://www.vivantecorp.com/index.php/en/technology/composition.html.

[42] Vivante Vega 3D Technology. http://www.vivantecorp.com/index.php/en/technology/3d.html.

[43] Ahmed K. Abousamra, Rami G. Melhem, and Alex K. Jones. Noc-aware cache design for chip multiprocessors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 565–566, New York, NY, USA, 2010. ACM.

[44] T. Akenine-Moller and J. Strom. Graphics Processing Units for Handhelds. *Proc. of the IEEE*, 96(5):779–789, 2008.

[45] Tomas Akenine-Möller and Björn Johnsson. Performance per what? *Journal of Computer Graphics Techniques (JCGT)*, 1(1):37–41, Oct 2012.

[46] Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. Stochastic rasterization using time-continuous triangles. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH'07, pages 7–16, 2007.

[47] Tomas Akenine-Moller and Jacob Strom. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics*, 22:801–808, 2003.

[48] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, 2004.

[49] Carlos Álvarez, Jesús Corbal, Esther Salamí, and Mateo Valero. On the potential of tolerant region reuse for multimedia applications. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 218–228, 2001.

[50] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Dynamic tolerance region computing for multimedia. *IEEE Trans. Comput.*, 61(5):650–665, May 2012.

[51] I. Antochi. *Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators*. PhD thesis, 2007.

[52] Iosif Antochi, Ben H. H. Juurlink, Stamatis Vassiliadis, and Petri Liuha. Memory Bandwidth Requirements of Tile-Based Rendering. In *SAMOS*, pages 323–332, 2004.

[53] ARM. ARM Mali-55. `http://mobile.arm.com/products/multimedia/mali-graphics-hardware/mali-55.php`.

[54] ARM. ARM Mali-T678. `http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute/mali-t678.php`.

[55] ARM. Mali-400 MP: A Scalable GPU for Mobile Devices. `http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf`.

[56] ARM. Mali GPU Application Optimization Guide. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/CHDIAHCC.html`.

[57] ARM. Transaction elimination. `http://www.arm.com/products/multimedia/mali-technologies/transaction-elimination.php`.

[58] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, May 1995.

[59] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. of ISPASS*, pages 163–174, 2009.

[60] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 373–378, New York, NY, USA, 1996. ACM.

[61] Slo-Li Chu, Chih-Chieh Hsiao, and Chen-Yu Chen. A Dual-Mode Unified Shader with Frame-Based Dynamic Precision Adjustment for Mobile GPUs. In *IEEE/IFIP 9th International Conference on Embedded and Ubiquitous Computing, EUC 2011, Melbourne, Australia, October 24-26, 2011*, EUC. IEEE, 2011.

[62] Slo-Li Chu, Chih-Chieh Hsiao, and Chen-Yu Chen. Program-based Dynamic Precision Selection Framework with a Dual-mode Unified Shader for Mobile GPUs. *Comput. Electr. Eng.*, 39(7):2183–2196, October 2013.

[63] Slo-Li Chu, Chih-Chieh Hsiao, and Chiu-Cheng Hsieh. An Energy-Efficient Unified Register File for Mobile GPUs. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pages 166–173, 2011.

[64] Daniel Citron, Dror G. Feitelson, and Larry Rudolph. Accelerating multimedia processing by implementing memoing in multiplication and division units. In *ASPLOS*, pages 252–261, 1998.

[65] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Proc. of MASCOTS*, pages 351–360, 2010.

[66] Daniel A. Connors, Hillery C. Hunter, Ben-Chung Cheng, and Wen-mei W. Hwu. Hardware support for dynamic activation of compiler-directed computation reuse. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 222–233, 2000.

[67] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[68] NVIDIA Corporation. NVIDIAs Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[69] Michael Cox and Pat Hanrahan. Pixel merging for object-parallel rendering: A distributed snooping algorithm. In *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 49–56, New York, NY, USA, 1993. ACM.

[70] Neal C. Crago, Omid Azizi, Steven S. Lumetta, and Sanjay J. Patel. Hybrid latency tolerance for robust energy-efficiency on 1000-core data parallel processors. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 0:294–305, 2013.

[71] Neal Clayton Crago and Sanjay Jeram Patel. Outrider: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[72] Jem Davies. Keynote I: Energy-efficient GPUs from mobile to supercomputers. In *1st annual Workshop on Parallelism in Mobile Platforms*, PRISM-1, 2013.

[73] Kayvon Fatahalian and Mike Houston. A Closer Look at GPUs. *Commun. ACM*, 51(10):50–57, October 2008.

[74] Simon Fenney. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 84–91, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[75] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[76] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom Anderson, and Katherine Yelick. The Energy Efficiency of IRAM Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 327–337, New York, NY, USA, 1997. ACM.

[77] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsl.*, 23:102–110, December 1992.

[78] Antonio García-Guirado, Ricardo Fernández-Pascual, Alberto Ros, and José M. García. Dapsco: Distance-aware partially shared cache organization. *ACM Trans. Archit. Code Optim.*, 8(4):25:1–25:19, January 2012.

[79] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 235–246, New York, NY, USA, 2011. ACM.

[80] Dan Ginsburg, Budirijanto Purnomo, Dave Shreiner, and Aaftab Munshi. *OpenGL ES 3.0 Programming Guide*. Addison-Wesley Professional, 2014.

[81] E. Bruce Goldstein. *Sensation and Perception*. Univ. of Pittsburgh, 6 edition, 2002.

[82] Antonio Gonzlez, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *In Proceedings of the the International Conference on Parallel Processing*, 1999.

[83] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 108–120, New York, NY, USA, 1997. ACM.

[84] Marty Hall and James Mayfield. Improving the performance of ai software: Payoffs and pitfalls in using automatic memoization. In *In Proceedings of the Sixth International Symposium on Artificial Intelligence*, pages 178–184, 1993.

[85] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, New York, NY, USA, 2009. ACM.

[86] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '06, pages 103–110, New York, NY, USA, 2006. ACM.

[87] Jon Hasselgren and Tomas Akenine-Möller. An Efficient Multi-View Rasterization Architecture. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, EGSR'06, pages 61–72, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[88] Chih-Chieh Hsiao, Slo-Li Chu, and Chen-Yu Chen. Energy-aware Hybrid Precision Selection Framework for Mobile GPUs. *Computers & Graphics*, 37(5):431–444, 2013.

[89] Chih-Chieh Hsiao, Slo-Li Chu, and Chiu-Cheng Hsieh. An Adaptive Thread Scheduling Mechanism With Low-Power Register File for Mobile GPUs. *IEEE Transactions on Multimedia*, 16(1):60–67, 2014.

[90] Chih-Chieh Hsiao, Chiu-Cheng Hsieh, and Slo-Li Chu. An energy-efficient demand-driven register file for mobile gpus. *Journal of Circuits, Systems, and Computers*, 23(2), 2014.

[91] Jian Huang and David J. Lilja. Exploiting basic block value locality with block reuse. In *HPCA*, pages 106–114, 1999.

[92] IDC. Android and iOS Continue to Dominate the Worldwide Smartphone Market. `http://www.idc.com/getdoc.jsp?containerId=prUS24676414`.

[93] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, 1998.

[94] The Khronos Group Inc. OpenGL ES. `http://www.khronos.org/opengles/`.

[95] The Khronos Group Inc. OpenGL ES Common Profile Specification Version 2.0.25. `http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`.

[96] The Khronos Group Inc. OpenGL ES Common/Common-Lite Profile Specification Version 1.1.12. `http://www.khronos.org/registry/gles/specs/1.1/es_full_spec_1.1.12.pdf`.

[97] The Khronos Group Inc. OpenGL ES Version 3.0.3. `http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf`.

[98] The Khronos Group Inc. OpenGL ES Version 3.1. `http://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf`.

[99] The Khronos Group Inc. WebGL Specification 1.0. `https://www.khronos.org/registry/webgl/specs/1.0/`.

[100] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[101] B. Juurlink, I. Antochi, D. Crisu, S. Cotofana, and S. Vassiliadis. GRAAL: A Framework for Low-Power 3D Graphics Accelerators. *IEEE Computer Graphics and Applications*, 28(4):63–73, 2008.

[102] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[103] Stefanos Kaxiras, Polychronis Xekalakis, and Georgios Keramidas. In Kaushik Roy and Vivek Tiwari, editors, *ISLPED*, pages 54–59. ACM.

[104] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Proceedings of the 25th International Conference on Computer Design*, ICCD, pages 245–250, 2007.

[105] G. Knittel, A. Schilling, A. Kugler, and W. Strasser. Hardware for superior texture performance. In *Proceedings of the Tenth Eurographics Conference on Graphics Hardware*, EGGH'95, pages 33–40, Aire-la-Ville, Switzerland, Switzerland, 1995. Eurographics Association.

[106] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. *IEEE/ACM International Symposium on Microarchitecture*, 0:213–224, 2010.

[107] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of MICRO*, pages 469–480, 2009.

[108] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D'12, pages 143–150, 2012.

[109] D Mccabe and J Brothers. DirectX 6 Texture Map Compression. *Game Developer Magazine 5*, pages 42–46, August 1998.

[110] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in c++. *SIGPLAN Not.*, 33(8):17–22, August 1998.

[111] Bren Mochocki, Kanishka Lahiri, and Srihari Cadambi. Power Analysis of Mobile 3D Graphics. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, pages 502–507, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[112] Jordi Roca Monfort and Mark Grossman. Scaling of 3D Game Engine Workloads on Modern multi-GPU Systems. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 37–46, New York, NY, USA, 2009. ACM.

[113] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. ATTILA: A Cycle-level Execution-Driven Simulator for Modern GPU Architectures. In *Proc. of ISPASS*, pages 231–241, 2006.

[114] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 25(1):90–97, 2005.

[115] NVIDIA. Bringing High-End Graphics to Handheld Devices. `http://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf`.

[116] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[117] NVIDIA. NVIDIA Tegra 4 Family GPU Architecture. `http://www.nvidia.com/docs/IO//116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf`.

[118] NVIDIA. SLI Best Practices. `http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf`.

[119] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. Adaptive Scalable Texture Compression. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12, pages 105–114. Eurographics Association, 2012.

[120] Tom Olson. How low can you go? Building low-power, low-bandwidth ARM Mali GPUs. `http://community.arm.com/groups/arm-mali-graphics/blog/2012/08/17/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus`.

[121] Tom Olson. Triangles Per Second: Performance Metric or Chocolate Teapot? `http://community.arm.com/groups/arm-mali-graphics/blog/2011/02/22/triangles-per-second-performance-metric-or-chocolate-teapot`.

[122] Joan-Manuel Parcerisa and Antonio Gonzalez. Improving latency tolerance of multithreading through decoupling. *IEEE Trans. Comput.*, 50(10):1084–1094, October 2001.

[123] Anton Pereberin. Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99*, pages 195–199, 1999.

[124] Jeff Pool, Anselmo Lastra, and Montek Singh. A Per-Unit Breakdown of the Energy Consumption in a Graphics Processing Unit. `http://www.cs.unc.edu/~jpool/research/ICCD2010Extension/index.html`.

[125] Jeff Pool, Anselmo Lastra, and Montek Singh. An Energy Model for Graphics Processing Units. In *ICCD*, pages 409–416. IEEE, 2010.

[126] Kari Pulli, Tomi Aarnio, Kimmo Roimela, and Jani Vaarala. Designing Graphics Programming Interfaces for Mobile Devices. *IEEE Comput. Graph. Appl.*, 25(6):66–75, November 2005.

[127] Qualcomm. Composition with Snapdragon. `https://developer.qualcomm.com/sites/default/files/composition-with-snapdragon.pdf`.

[128] Jim Rasmusson, Jon Hasselgren, and Tomas Akenine-Möller. Exact and Error-Bounded Approximate Color Buffer Compression and Decompression. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 41–48, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[129] Vijay Janapa Reddi. Architecting for the Mobile Web: Where We've Been, Where We're Heading, and What We Need to Address. In *Keynote of the 2nd annual workshop on parallelism in mobile platforms*, PRISM-2, 2014.

[130] W. T. Reeves. Particle Systems: A Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.*, 2(2):91–108, April 1983.

[131] Matthew J. P. Regan, Gavin S. P. Miller, Steven M. Rubin, and Chris Kogelnik. A Real-Time Low-Latency Hardware Light-Field Renderer. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 287–290, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[132] Hugo Rito and João Cachopo. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ'10, pages 89–98, 2010.

[133] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, January 2011.

[134] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. Characterizing coarse-grained reuse of computation. In *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, pages 16–18, 2000.

[135] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 73–82, Piscataway, NJ, USA, 2013. IEEE Press.

[136] J. W. Sheaffer, D. Luebke, and K. Skadron. A Flexible Simulation Framework for Graphics Architectures. In *Proc. of the EUROGRAPHICS Conf. on Graphics Hardware*, pages 85–94, 2004.

[137] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[138] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[139] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 194–205, 1997.

[140] Jacob Ström and Tomas Akenine-Möller. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 63–70, New York, NY, USA, 2005. ACM.

[141] Deependra Talla and Lizy K. John. Mediabreeze: A decoupled architecture for accelerating multimedia applications. *SIGARCH Comput. Archit. News*, 29(5):62–67, December 2001.

[142] Deependra Talla, Lizy Kurian John, and Doug Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Trans. Computers*, 52(8):1015–1031, 2003.

[143] David Tarjan and Kevin Skadron. The sharing tracker: Using ideas from cache coherence hardware to reduce off-chip memory traffic with non-coherent caches. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[144] Hung-Wei Tseng and Dean M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, HPCA, pages 181–192, 2011.

[145] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. Design and evaluation of an auto-memoization processor. In *Parallel and Distributed Computing and Networks*, pages 230–235, 2007.

[146] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro López. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[147] Po-Han Wang, Yen-Ming Chen, Chia-Lin Yang, and Yu-Jung Cheng. A predictive shutdown technique for gpu shader processors. *Computer Architecture Letters*, pages 9–12, 2009.

[148] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on gpus. *ACM Trans. Archit. Code Optim.*, 8(3):13:1–13:25, October 2011.

[149] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image Quality Assessment: from Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[150] Wikipedia. ARM Mali. `http://en.wikipedia.org/wiki/Mali_%28GPU%29`.

[151] Wikipedia. Ericsson Texture Compression. `http://en.wikipedia.org/wiki/Ericsson_Texture_Compression`.

[152] Wikipedia. Fillrate. `http://en.wikipedia.org/wiki/Fillrate`.

[153] Wikipedia. Mean Squared Error. `http://en.wikipedia.org/wiki/Mean_squared_error`.

[154] Wikipedia. Micro Stuttering. `http://en.wikipedia.org/wiki/Micro_stuttering`.

[155] Wikipedia. Multiple Render Targets. `http://en.wikipedia.org/wiki/Multiple_Render_Targets`.

[156] Wikipedia. Nintendo 3DS. `http://en.wikipedia.org/wiki/Nintendo_3DS`.

[157] Wikipedia. OpenGL ES. `http://en.wikipedia.org/wiki/OpenGL_ES`.

[158] Wikipedia. Peak Signal-to-Noise Ratio. `http://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio`.

[159] Wikipedia. Performance per watt. `http://en.wikipedia.org/wiki/Performance_per_watt`.

[160] Wikipedia. PICA-200. `http://en.wikipedia.org/wiki/PICA200`.

[161] Wikipedia. PowerVR. http://en.wikipedia.org/wiki/PowerVR.

[162] Wikipedia. Qualcomm Adreno. http://en.wikipedia.org/wiki/Adreno.

[163] Wikipedia. Qualcomm snapdragon. http://en.wikipedia.org/wiki/Snapdragon_%28system_on_chip%29.

[164] Wikipedia. Samsung Exynos. http://en.wikipedia.org/wiki/Exynos.

[165] Wikipedia. Tiled rendering. http://en.wikipedia.org/wiki/Tiled_rendering.

[166] Lance Williams. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[167] Jeong-Ho Woo, Min-Wuk Lee, Hye-Jung Kim, R. Woo, and Hoi-Jun Yoo. A 155-mw 50-m vertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *IEEE Journal of Solid-State Circuits*, 41(5):1081–1091, May 2006.

[168] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 75–82, 2007.

[169] Wing-kei S. Yu, Ruirui Huang, Sarah Q. Xu, Sung-En Wang, Edwin Kan, and G. Edward Suh. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 247–258, New York, NY, USA, 2011. ACM.

[170] GPUWattch. http://www.gpgpu-sim.org/gpuwattch/.

[171] Microsoft Direct3D. http://en.wikipedia.org/wiki/Microsoft_Direct3D.

[172] OpenCL. http://www.khronos.org/opencl/.

[173] Carlos lvarez, Jess Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.