Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor

José-María Arnau[1] jarnau@ac.upc.edu
Joan-Manuel Parcerisa[1] jmanel@ac.upc.edu
Polychronis Xekalakis[2] polychronis.xekalakis@intel.com

[1] *Computer Architecture Department, Universitat Politècnica de Catalunya*

[2] *Intel Barcelona Research Center, Intel Labs Barcelona*

Abstract

Smartphones represent one of the fastest growing markets, providing significant hardware/software improvements every few months. However, supporting these capabilities reduces the operating time per battery charge. The CPU/GPU component is only left with a shrinking fraction of the power budget, since most of the energy is consumed by the screen and the antenna.

In this paper, we focus on improving the energy efficiency of the GPU since graphical applications consist an important part of the existing market. Moreover, the trend towards better screens will inevitably lead to a higher demand for improved graphics rendering. We show that the main bottleneck for these applications is the texture cache and that traditional techniques for hiding memory latency (prefetching, multithreading) do not work well or come at a high energy cost.

We thus propose the migration of GPU designs towards the decoupled access-execute concept. Furthermore, we significantly reduce bandwidth usage in the decoupled architecture by exploiting inter-core data sharing. Using commercial Android applications, we show that the end design can achieve 93% of the performance of a heavily multithreaded GPU while providing energy savings of 34%.

1. INTRODUCTION

The rapid development of smartphones is fueled by the high end-user demand for a continuously improved mobile computing experience. Current smartphones excel in receiving and sending emails, allow for video and picture editing and are able to support a plethora of 3D games. Unfortunately, supporting all these capabilities comes at a high energy cost, which in turn results in fairly short operating times per battery charge. Not surprisingly, a recent study based on real users revealed that most of the energy today is consumed by the communications subsystem and the screen [3]. In Figure 1 we have plotted the energy consumed by each of these two components of a Smartphone along with the capacity of the battery over the past few years. The scaling trend clearly supports the aforementioned observation. The communication subsystem consumes most of the energy today, while with each new generation, the energy requirements grow significantly. The screens exhibit a similar trend. Despite the introduction of the lower-energy AMOLED screens, the increase in their size and resolution, lead to an increase in the energy they consume by 37% each year. The battery capacity on the other hand, increases at a modest pace of 5-10% per year [20].

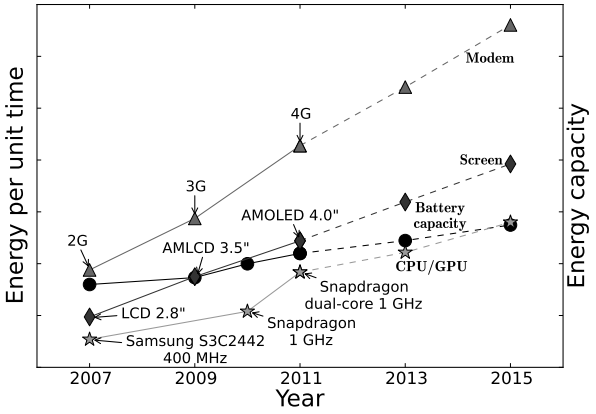


Figure 1. Energy consumption evolution. The gap between the energy provided by the battery and the energy consumed by the main components increases on each generation [3,10,21,25,30,31,32,33,34].

Although the CPU/GPU component is arguably more frequently used than the communication subsystem, we believe that the trend is clear. If the operating time of a Smartphone is going to stay at what it is today, the energy that the CPU/GPU component will have at its disposal will become even more limited with each new generation of Smartphones. Interestingly, better screens will require better graphics rendering, that is, higher performing GPUs. Due to the highly parallel nature of the applications that utilize the GPU, we believe that the main complication that will have to be

dealt with is how to bridge[1] the memory gap in an energy efficient manner.

Believing that today's solutions to the memory problem are mostly adaptations of solutions that exist for desktop systems, we take a fresh approach in the design of the GPU of a mobile system. More specifically, we opt for a solution that does not rely on multithreading to hide the memory latency, as we find it to be effective but not energy-efficient (the register file of an NVIDIA Fermi-like desktop GPU has been reported to be about 9.6 Watts [8]). Moreover, as noted in [12,15], existing prefetching solutions are viable solutions for the pixel cache, but they cannot fully cope with the seemingly random access patterns of the texture cache.

We thus propose to employ a decoupled access/execute design for the fragment processor. We apply our proposal on top of a state-of-the-art GPU, similar to the one that resides in the NVIDIA Tegra 2 chipset [18]. More specifically, the proposed scheme uses specifically designed queues to decouple the memory accesses required for bringing all the pixel and texture data from the computations that have to be performed on them. In an effort to further push for minimizing the accesses to the lower level memory, we also allow for pixel and texture requests to be redirected to remote caches which are known to have the requested data. The combined effect of these two techniques yields significant performance benefits, while keeping the additional energy requirements to a minimal. We evaluate our proposal using commercial Android applications and demonstrate that we can achieve 93% of the performance of a highly threaded GPU, with a much lower energy budget.

This paper focuses on energy efficient, highly performing GPUs. Its main contributions are the following:

- We evaluate several state-of-the-art CPU, GPU and GPGPU hardware prefetchers on a mobile GPU. The results obtained in our cycle-accurate GPU simulator indicate that these prefetchers provide significant benefits on performance although they perform far from ideal, so there is still ample room for improvement.
- We propose a new decoupled access/execute like architecture specifically designed for low-power GPUs and graphics workloads. This decoupled architecture outperforms previous proposals in terms of performance and energy consumption.
- We show that significant amounts of energy can be saved by optimizing the bandwidth usage to the L2 cache. The end design is able to achieve similar performance to a heavily-multithreaded GPU by consuming only a fraction of its energy.

The remainder of this paper is organized as follows: The next section provides background information on the baseline GPU architecture and on the fragment processor. Section 3 presents the proposed decoupled access/execute architecture scheme. Section 4 describes our evaluation methodology and Section 5 presents the performance and power results that were obtained. Section 6 reviews related work on smartphones, GPUs and prefetching and finally, Section 7 provides our conclusions.

2. BACKGROUND

The mobile GPU model that is assumed throughout the paper closely tracks the ultra-low power GeForce GPU in the NVIDIA Tegra 2 chipset [18] (Figure 3). Although the focus of the paper is on the Fragment Processor component, a brief description of the assumed GPU will be presented first so as to ensure completeness.

2.1 Basic GPU Operation

The application OpenGL ES command stream is first converted by the OpenGL ES driver to a sequence of GPU commands. When the *Command processor* receives a command, it installs the corresponding vertex and fragment shaders and sets the appropriate control signals so that the input vertex stream is correctly processed through the GPU pipeline. Having configured the pipeline, the *Vertex Fetcher* is triggered so that the input vertices are fetched from memory with all the per-vertex information and stored in the first *vertex queue*. Next, vertices are transformed and shaded in the *Vertex shader* stage, based on the programmed by the user vertex shader. The transformed and shaded vertices are then passed to the *Primitive Assembly* stage. At this stage the vertices are grouped into the corresponding triangles and clipping and culling are applied. The resulting 2D triangles are inserted into the *triangle queue*. Next, the *Rasterizer* takes these 2D triangles and generates the corresponding fragments that lie inside the triangles. The resulting fragments are inserted into the *fragment queue*, and are processed in the *Fragment shader* stage based on the programmed by the user fragment shader.

There is a variety of caches utilized throughout the pipeline of this GPU. More specifically, the vertex fetcher employs a vertex cache, while the fragment shader includes a Pixel and a Texture cache. These

1. For a Tegra like system, we found that if no threading is utilized to hide the latency, the performance hit for a set of 3D Games for the Android is 140%. A Tegra with perfect caches can provide up to 285% over the non-threaded version.

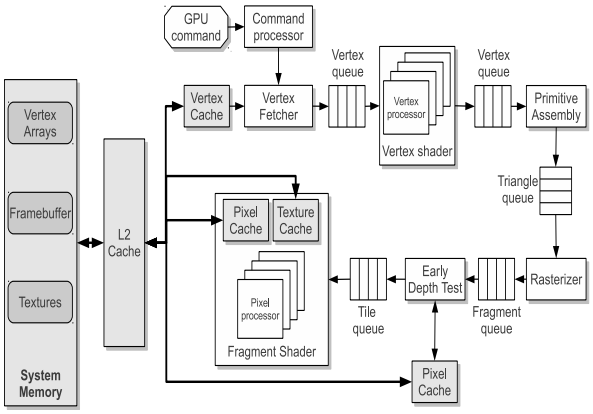


Figure 2. Microarchitecture of Tegra-like GPU.

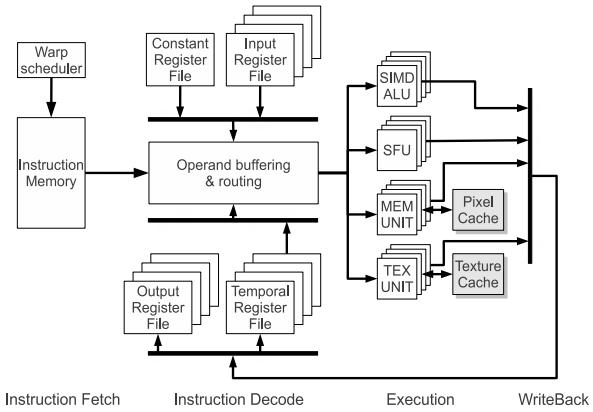


Figure 3. Microarchitecture of a Fragment Processor.

caches are all assumed to be connected via a shared bus to the L2 cache of a CPU/GPU system.

2.2 Fragment Processor

The fragment processor consists of a fairly simple in-order four stage pipeline. Typically, a form of SMT is employed where the threads are organized in groups named warps. All threads in a warp are executed in lockstep mode, that is the same instruction is executed by all the threads but each thread operates on a different input fragment. The warp scheduler determines from which warp to fetch an instruction every cycle by using a Round Robin policy.

After instructions are fetched, they are decoded and their operands are fetched. Depending on the type of the operand, one of the three different register files are probed, depending on the type of value that needs to be read (i.e., constant, input or temporal). In the Instruction Decode stage, the valid bits of the source registers are checked to avoid RAW hazards. Whenever a data dependency is found, the dependent instruction and all the younger instructions of the same warp are squashed and the PC of the warp is set to the PC of the dependent instruction. Moreover, the valid bit of the destination register is also checked to avoid WAW hazards.

Once all the source operands for the threads in a warp are fetched the instruction is dispatched to the corresponding functional unit. Operand buffering is required since the number of functional units can be smaller than the number of threads in a warp, in this case the dispatch to the functional units takes several cycles. Four types of functional units are included in each fragment processor, namely the SIMD ALU (i.e, vector additions), the Special Functions Unit (i.e., reciprocal operations), the Memory Unit (i.e., load/stores to the color buffer), and the Texture Unit (i.e., compute the color of a texture).

In the last pipeline stage the results of the functional units are stored in the temporal or in the output register file. There is no forwarding mechanism, so two dependent instructions cannot be executed back-to-back. However, since the warp scheduler uses a Round Robin policy consecutive instructions usually pertain to different warps.

3. DECOUPLED ACCESS/EXECUTE ON A MOBILE GPU

The fragment processors fetch data from texture memory and access pixels in the framebuffer. Each fragment processor includes two caches to speed-up these memory accesses: a pixel cache and a texture cache. Most desktop GPUs rely on massive multi-threading as an effective way to hide memory latency and keep functional units busy during cache misses. However, the simultaneous execution of multiple warps requires an equal number of thread contexts, which greatly increases the number of registers. Since this approach is power-hungry, it is not deemed appropriate for battery-operated handheld devices such as smartphones with a very limited energy budget. In fact, the number of simultaneous warps of a low-power GPU is usually small due to power constrains, which in turn results in cache misses producing pipeline stalls that cannot be hidden.

3.1 Prefetching

A known solution to tolerate the cache miss latency is prefetching. Since there is no publicly available information about hardware prefetchers in the fragment processors of current GPUs, we assume that they do not include any. Nonetheless, in this paper we review how prefetchers that were previously proposed for CPUs, like the Stride [5] and the GHB [17], work in this new environment. We also study how a more tailored to the GPU needs prefetcher works (Many-Thread-Aware [15]). Our experiments show that these schemes pro-

vide moderate latency tolerance but, compared to an ideal prefetcher, they still leave ample room for improvement. Moreover, their performance improvements come at the cost of a significant increase in energy consumption caused by the useless prefetches they generate.

3.2 Access/Execute Decoupling

We propose to adopt a more energy-efficient prefetching approach to hide memory latency, which is based on the access/execute architectural paradigm [24]. 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.

3.3 The Access/Execute Decoupled Fragment Processor

The decoupled access/execute architecture proposed in this paper is depicted in Figure 4. After the visibility determination in the Early Depth Test stage (shown in Figure 2), visible fragments are packed into tiles, a fragment processor is assigned to each tile by

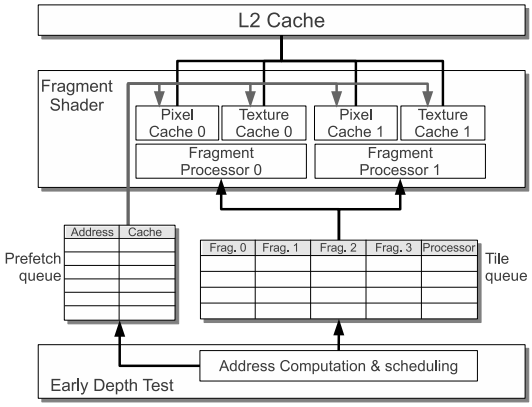
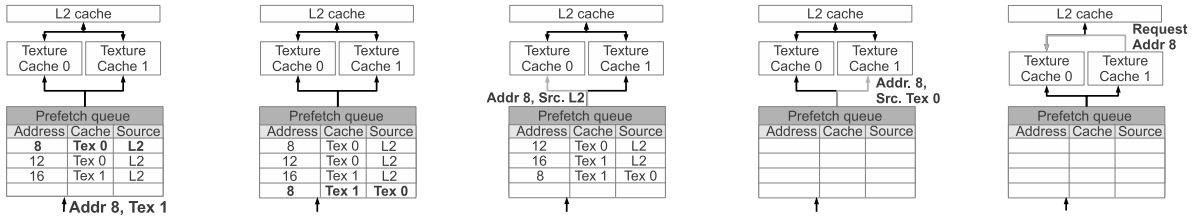


Figure 4. Decoupled access/execute architecture.

the scheduler, and both the tile and the processor number are inserted into the tile queue to wait its turn until it is dispatched to the processor. Part of the information stored in this tile queue (screen coordinates, texture coordinates, etc.) will be later used by the fragment processor to compute the addresses that will be issued to memory.

The proposed scheme decouples the memory addresses from the tile queue, so that memory requests for a specific tile can be issued, while the tile is still waiting in the queue. This behavior is achieved by inserting all computed addresses of a tile along with their target cache number, into a new queue, the prefetch queue. Notice that the scheduling is performed before queuing the tile, 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 L1 cache update. Note also that the proposed scheme performs *in-cache* prefetching, instead of prefetching into specialized buffers.

By the time a tile is dispatched to the fragment processor, the data required to process the fragments are usually available in the pixel and texture caches, 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 tile queue to fill up, but it would also allow the prefetcher to increase the prefetch distance again, thus avoiding further stalls. The tile 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,



(a) A new prefetch request to address 8 in texture cache 1 is sent to the prefetch queue.

(b) The request is inserted in the queue. The Source field points to texture cache 0.

(c) The request at top of the queue is dispatched to the corresponding cache.

(d) After several cycles the request is sent to texture cache 1. The Source field is included.

(e) Cache miss. The prefetch request is redirected to texture cache 0 instead of L2.

Figure 5. Example of how the decoupled access/execute architecture can take benefit of data reusage among L1 caches of different fragment processors to save L2 bandwidth.

due to cache conflicts. We have found that lenghts 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 block is held in the prefetch queue.

3.4 Prefetching from other L1 caches

Partitioning the L1 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 simulator and a commercial set of graphical applications show that, on average, 40.7% of the prefetch requests are cache misses, and that 66.3% of these misses are requests to data that is already available in the pixel or in the texture cache of some other fragment processor. Hence, the decoupled access/execute architecture previously described can be further improved because up to 66.3% of the prefetch misses can be satisfied by the L1 pixel or texture cache of another fragment processor instead of accessing the L2. 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 L1 pixel or 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 5 shows an example of this behavior. 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. In order to increase the energy efficiency, we restrict the associative search to the eight lower bits of the block address. Since the *high bits* of the addresses typically do not change frequently, this approach decreases the cost of the search while keeping the same performance.

4. EVALUATION METHODOLOGY

We have developed a mobile GPU simulation infrastructure which consists of two components: the GPU trace generator and the cycle-accurate GPU simulator. For the trace generation we have instrumented the Android OpenGL ES software renderer in order to capture all the OpenGL ES calls performed by the applications and to gather information about the rendering process. The trace file includes the vertex and fragment instructions and the memory addresses of the texture and vertex data and the framebuffer. Although the instrumented Android OpenGL ES driver is a pure soft-

Table 1: GPU simulator parameters.

Queues	
Vertex Queue (2x)	16 entries, 136 bytes/entry
Triangle Queue	16 entries, 388 bytes/entry
Fragment Queue	64 entries, 57 bytes/entry
Tile Queue	8 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 4-way associative, 8 KB, 4 in-flight requests, 4 banks, 3 cycles
Pixel and Texture Caches	64 bytes/line, 2-way associative, 2 KB, 4 in-flight requests, 4 banks, 2 cycles
L2 Cache	64 bytes/line, 8-way associative, 32 KB, 8 in-flight requests, 8 banks, 12 cycles
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	4 fragments/cycle
Early Z test	8 in-flight fragments, 1 pixel cache
Programmable stages	
Vertex shader	4 vertex processors
Fragment shader	4 fragment processors
Main memory	
Latency	100 cycles
Bandwidth	4 bytes/cycle (dual channel)
Decoupled access/execute	
Prefetch queue size	16 entries
Entry size	40 bits
Total size	640 bits
Global parameters	
Frequency	600 MHz
Voltage	1 V
Screen resolution	800x480 (WVGA)
Technology	45 nm

Vertex processor	
Multithreading	1-16 warps, 4 threads/warp
Register size	16 bytes (4-wide vector)
Constant Reg. File	96 registers
Input Reg. File	64 regs/warp, 4 banks
Output Reg. File	32 regs/warp, 4 banks
Temporal Reg. File	48 regs/warp, 4 banks
Functional Units	4 SIMD ALUs, 4 SFUs
4-stage pipeline	IF, ID, Exec, WB
Fragment processor	
Multithreading	1-16 warps, 4 threads/warp
Register size	16 bytes (4-wide vector)
Constant Reg. File	96 registers
Input Reg. File	64 regs/warp, 4 banks
Output Reg. File	32 regs/warp, 4 banks
Temporal Reg. File	48 regs/warp, 4 banks
Functional Units	4 SIMD ALUs, 4 SFUs, 2 MEM, 2 texture units
Caches	1 texture and 1 pixel cache
4-stage pipeline	IF, ID, Exec, WB
Distance prefetcher	
Index table	16 entries, 40 bits/entry
GHB	100 entries, 40 bits/entry
Prefetch degree	2
Many-thread aware prefetcher with throttling	
Global Stride table	16 entries, 64 bits/entry
Inter-thread Pref. Table	16 entries, 66 bits/entry,
Per-Warp Stride table	16 entries, 164 bits/entry
Prefetch degree	0-5 (dynamically adapted)
Stride prefetcher	
Stride table size	48 entries
Entry size	98 bits
Prefetch degree	2

ware renderer, the trace generator does not save a complete memory trace of all the accesses performed by the driver, but it only saves the addresses of the memory accesses that would be issued in a hardware-based implementation (accesses to fetch vertex and texture data and to read/write the framebuffer). The generated traces are fed to a cycle-accurate GPU simulator which apart from timing, is also able to provide energy estimations. The simulator models the GPU architecture illustrated in Figure 2. For the vertex and fragment processors, the simulated microarchitecture is the one shown in Figure 3. The GPU simulator is also able to model the decoupled access/execute architecture described in Section 3. Moreover, several prefetching schemes extracted from the literature have been implemented in the simulator: the stride prefetcher [5], the Global History Buffer (GHB) [17], and the many-thread aware prefetcher with throttling [15]. The parameters employed during the simulations are summarized in Table 1.

Regarding the workloads, we have selected 8 Android games that are representative of the Android graphical applications since they employ most of the configurations available in the OpenGL ES API (different texture sampling strategies, shading models, antialiasing on/off...). We have included simple 2D games (`angryfrogs`, `icommando` and `pocket-racing`), simple 3D games (`polybreaker` and `shooting`) with simple 3D models and small fragment programs and more complex 3D games (`quake2`, `ibowl` and `tankrecon`) with bigger 3D models and heavy fragment programs.

We have generated traces of 1 billion instructions for each application, these traces consist of 10 checkpoints of 100 million instructions at 10 different locations. We correctly warm up the simulator before collecting statistics for each checkpoint. Regarding the power model, we have employed CACTI [23] to compute the static and dynamic energy consumed by the main hardware structures: caches, queues, register files and prefetching tables.

5. EXPERIMENTAL RESULTS

This section demonstrates the effectiveness of a decoupled access/execute architecture in hiding the memory latency of a mobile GPU. We first evaluate the performance achieved and the energy consumed by different prefetching techniques when executing a commercial set of Android games. We assume a pure decoupled access/execute architecture without multithreading (1 warp per fragment processor) for these experiments. Figure 6 shows the speedups over the *Global History Buffer (GHB)* prefetcher. The simplest prefetching scheme, the *stride* prefetcher, obtains better results than the GPU without prefetching, achieving 65% of the performance of the *GHB* on average. The *Many-Thread Aware* (*MTA*) prefetcher provides similar performance to the *GHB*. Although the experimental conditions (small number of threads, non-regular memory access patterns) are not favorable for the *MTA*

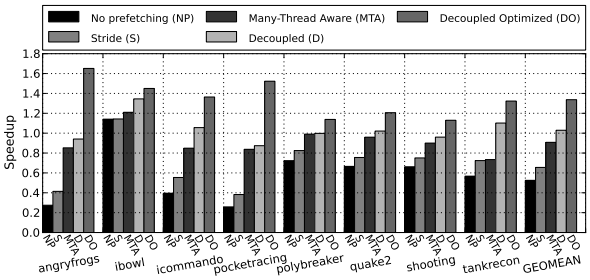


Figure 6. Speedups of different prefetching schemes and the decoupled access/execute architecture. The baseline configuration is the Global History Buffer. Multithreading is not employed.

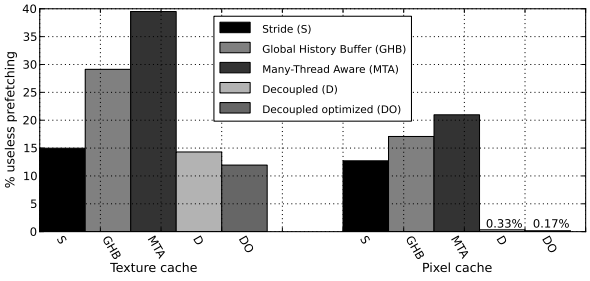


Figure 7. Useless prefetching. The decoupled schemes are more accurate than conventional prefetchers since they are not based on predictions.

prefetcher, it obtains 90% of the performance of the *GHB* prefetcher on average and it achieves better results in one of the applications (`ibowl`). The decoupled access/execute architecture performs better than the *GHB* on average (2.9% speedup). The decoupled access/execute with the bandwidth usage optimizations provides the best performance in all the applications (33% speedup on average). The performance improvements in this graph are due to a large reduction in miss stall cycles achieved by prefetching, specially for the access/execute scheme, since it greatly improves prefetch accuracy, as shown in Figure 7. This graph plots the amount of useless prefetching, i.e. the fraction of prefetched blocks that are never demanded by the processor. The performance advantage of the optimized scheme comes from the reduced latency of prefetch misses that can be satisfied from other local caches, because this improves timeliness of prefetching (more hits) and reduces cache miss stall time (shorter miss penalty).

Limited battery lifetime is a hard requirement for smartphones. Hence, the energy consumption is a primary concern for any architectural innovation. Figure 8 shows the total energy consumed by different prefetching schemes, considering both the static and dynamic consumption. All the prefetchers provide energy savings over a GPU without prefetching. The *GHB* and *MTA* exhibit similar energy consumption, whereas the optimized decoupled access/execute architecture pro-

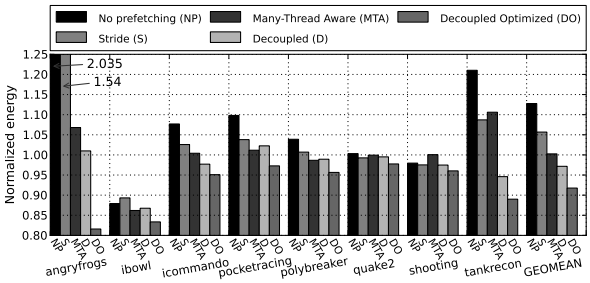


Figure 8. Energy consumed by the different prefetching schemes normalized to the energy consumed by the Global History Buffer. Multithreading is not employed (1 warp per fragment processor).

vides 9% energy savings on average with respect to the *GHB*. Although a bit surprising, the spent energy decreases as performance increases. The access/execute configuration consumes less energy than the baseline because the small increase in dynamic activity (some useless prefetched blocks, double L1 tag checks and associative lookups into the prefetch queue) is more than compensated by a large reduction in execution time (hence a reduction in static energy consumption). Recall that we do not assign any static energy consumption during long idle periods because we assume that the processor could drastically reduce it by entering a deep low power state. To summarize the prefetching analysis, we have seen that the decoupled access/execute architecture provides 33% speedup and 9% energy savings over state-of-the-art prefetchers.

Moreover, we have evaluated the effects of multithreading on a low-power GPU without prefetching. Figure 9 shows the speedups obtained when increasing the number of simultaneous warps from 1 to 16. Multithreading provides significant benefits, with an average speedup of 3.23x for 16 warps. However, aggressive multithreading increases energy consumption as shown in Figure 10. For a small number of warps (2 to 4) we get energy savings in some of the applications because the reduction in static energy, due to the reduction in execution time, is bigger than the increase in dynamic energy due to the larger size of the main register file. However, when using aggressive multithreading (6-16 warps) the energy consumption is increased in most of the applications, with an average increment of 25% for 16 warps.

We have also analyzed the performance and energy consumption of a mobile GPU when combining both prefetching and multithreading. The results are illustrated in Figure 11. This graph shows that the optimized access/execute prefetcher achieves its maximum speedup with just 4 warps, whereas the baseline needs 14-16 warps to achieve the same performance. The other prefetchers lay in between. The graph also shows that the single threaded access/execute architecture can

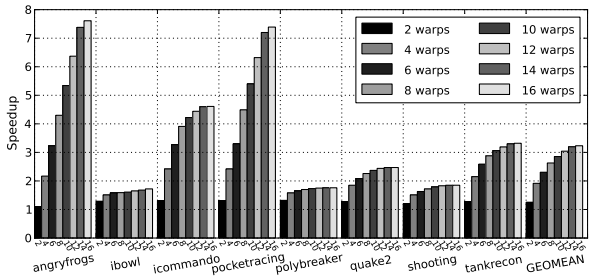


Figure 9. Multithreading speedups. The figure shows the performance benefits obtained when increasing the number of simultaneous warps.

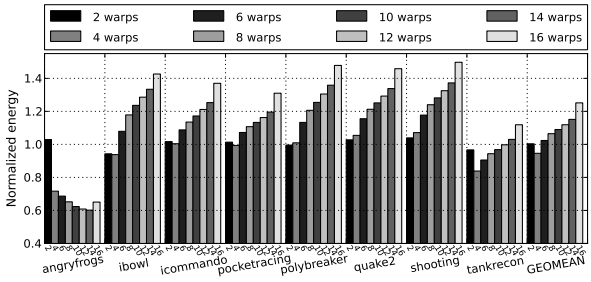


Figure 10. Normalized energy for different number of warps.

still improve performance by taking advantage of a small degree of multithreading (e.g. 4 warps). 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 [19]. Regarding energy consumption, Figure 12 compares the energy and speedup of the different prefetching schemes normalized to the baseline GPU without prefetching. The decoupled architecture with 1 warp achieves 78% of the performance of the baseline GPU with 16 warps, but consuming 35% less energy. Moreover, the decoupled system with 2 warps provides 93% of the performance of a GPU with 16 warps and it consumes 34% less energy. Hence, combining decoupled access/execute -to hide the memory latency- and non-aggressive multithreading -to hide the functional units latency- is an interesting approach to boost performance with a low cost in energy.

Finally, in order to evaluate the effectiveness of the L2 cache bandwidth optimizations described in Section 3 we measured the L2 cache traffic. Figure 13 illustrates the results. The state-of-the-art prefetchers (*stride*, *GHB* and *MTA*) increase L2 cache traffic due to additional memory requests for prefetching. The decoupled access/execute architecture increases L2 cache traffic by 73% on average, whereas the optimized version increases L2 cache traffic just by 24%. Hence, trying to obtain the data from the pixel or texture cache of another processor instead of the L2 cache produces a significant reduction in L2 cache traffic.

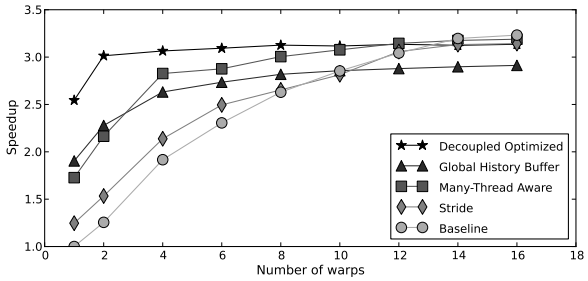


Figure 11. Speedups of different prefetchers when increasing the number of warps from 1 to 16. The baseline is a GPU without prefetching and 1 warp/core.

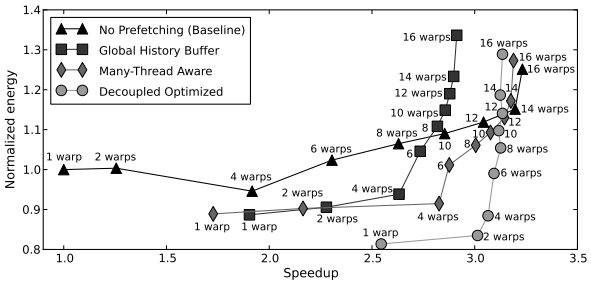


Figure 12. Normalized energy vs speedup. The decoupled architecture achieves nearly the same performance than a heavily multithreaded GPU at a much lower energy budget.

6. RELATED WORK

Increasing the efficiency and performance of GPUs has attracted the attention of the architectural community the last few years. Fung et al. [6] propose the dynamic formation of warps to deal with diverging branch outcomes. Tarjan et al. [26] introduce a hardware technique to allow a subset of the threads in a warp to continue execution while the rest of the threads are waiting on memory. Woo et al. [29] propose the use of the GPU to perform data prefetching for the CPU. Hong et al. [11] present an analytical power and performance model for GPU architectures. 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. [8], and the hybrid SRAM-DRAM memory design presented by Yu et al. [28]. In contrast with our work, all these proposals do not directly focus on the memory aspect of the GPU performance. We show that there is no real necessity for high degree of multithreading and as such for large register files. 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

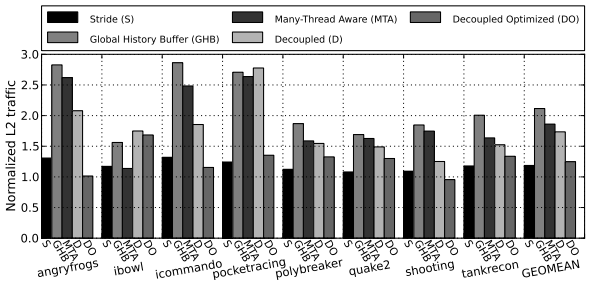


Figure 13. Normalized L2 cache traffic. The optimized decoupled architecture increases the L2 cache traffic by 24% on average, whereas the GHB and the MTA prefetchers cause increments of 111% and 86% on average respectively. The Stride prefetcher increases traffic by just 18% on average with respect to a single threaded GPU without prefetching.

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-Möller and Ström [2] 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. Mochocki et al. [16] explore the use of Dynamic Voltage and Frequency Scaling to reduce the energy consumption of a mobile GPU by as much as 50%.

Lee et al. [15] propose a prefetching scheme specifically designed for many-thread environments. This prefetcher employs several mechanisms such as inter-thread prefetching and stride promotion to boost GPU performance when executing scientific codes. We have shown that similar results can be achieved for mobile applications, however as we have also shown, the proposed scheme is able to be consistently better.

Tarjan et al. [27] 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 the prefetch queue. On the other hand, several tiled-cache approaches have been proposed. *Reactive NUCA* [9] 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 [1] 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* [7] 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 constrains. 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 [27]).

Crago et al. [4] 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.

The work that is closest to ours is that of Igehy et al. [12]. In their paper, they propose a prefetching architecture for texture caches to hide the memory latency. This prefetcher is similar to our decoupled system, however, our work is different in several ways. First, our system is built on top of a modern mobile GPU where the Z-Test stage is performed before the fragment processing. This is an important difference as our system only issues prefetch requests for visible fragments instead of prefetching for all the fragments generated by the Rasterizer, increasing the energy efficiency. Moreover, our scheme is able to coordinate the accesses in an environment with multiple fragment-processors. Finally, our proposal allows for remote requests, which is shown to provide significant energy benefits.

7. CONCLUSIONS

The main ambition of this paper is to demonstrate that high-performing, energy-efficient GPUs can be architected based on the decoupled access-execute design 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 this paper, 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 focus our study in Smartphones, as it is one of the fastest growing

markets, while it is also energy constrained. We evaluate the proposed scheme using a set of commercial Android applications and show that the end decoupled access/execute design with 2 warps/core is able to achieve 93% of the performance of a larger GPU with 16 warps/core, while providing 34% energy savings. Moreover, we show that a significant percentage of L2 cache bandwidth can be saved by redirecting pixel requests to remote texture caches which are known to have the requested data. Compared with prefetchers that were previously proposed for CPUs, the decoupled access/execute architecture clearly outperforms them, as it is 33% faster than the Global History Buffer while it also consumes 9% less energy.

Acknowledgments

This work has been supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Economy and Competitiveness under grant TIN 2010-18368, and Intel Corporation. Jose-Maria Arnau is supported by an FI-Research grant.

REFERENCES

[1] A. K. Abousamra, R. G. Melhem, A. K. Jones. "NoC-aware cache design for chip multiprocessors". In *Proc. of PACT*, pp. 565-566, Sept. 2010.

[2] T. Akenine-Möller and J. Ström. "Graphics for the masses: a hardware rasterization architecture for mobile phones". In *Proc. of SIGGRAPH*, pp. 801-808, July 2003.

[3] A. Carroll and G. Heiser. "An analysis of power consumption in a smartphone". In *Proc. of USENIXATC*, pp. 21-34, June 2010.

[4] N. C. Crago and S. J. Patel. "OUTRIDER: efficient memory latency tolerance with decoupled strands". In *Proc. of ISCA*, pp. 117-128, June 2011.

[5] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. "Stride directed prefetching in scalar processors". *SIGMICRO Newsl.*, pp. 102-110, Dec. 1992

[6] Wilson W. L. Fung, I. Sham, G. Yuan and T. M. Aamodt. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In *Proc. of MICRO*, pp. 407-420, Dec. 2007.

[7] A. García-Guirado, R. Fernández-Pascual, A. Ros and J. M. García. "DAPSCO: Distance-aware partially shared cache organization". In *ACM Trans. on Arch. and Code Optimization*, 8 (4), Jan. 2012.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm and K. Skadron. "Energy-efficient mechanisms for managing thread context in throughput processors". In *Proc. of ISCA*, pp. 235-246, June 2011.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki. "Reactive NUCA: near-optimal block placement and replication in distributed caches". In *Proc. of ISCA*, pp. 184-195, June 2009.

[10] Hewlett-Packard. "HP ProBook 5330m Notebook PC Overview". `http://h18000.www1.hp.com/products/quickspecs/14018_na/14018_na.HTML`.

[11] S.Hong and H.Kim."An integrated GPU power and performance model".In *Proc. of ISCA*,pp.280-289,June 2010.

[12] H. Igehy, M. Eldridge and K. Proudfoot. "Prefetching in a texture cache architecture". In *Proc. of SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pp. 133-142, Aug. 1998.

[13] D. Joseph and D. Grunwald. "Prefetching using markov predictors". In *Proc. of ISCA*, pp. 252-263, June 1997.

[14] G. B. Kandiraju and A. Sivasubramaniam. "Going the distance for tlb prefetching: an application-driven study". In *Proc. of ISCA*, pp.195-206, 2002

[15] J. Lee, N. B. Lakshminarayana, H. Kim and R, Vuduc. "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications". In *Proc. of MICRO*, pp. 213-224, December 2010.

[16] B. Mochocki, K. Lahiri and S. Cadambi. "Power analysis of mobile 3D graphics". In *Proc. of DATE*, pp. 502-507, March 2006.

[17] K. J. Nesbit and J. E. Smith. "Data Cache Prefetching Using a Global History Buffer". In *Proc. of HPCA,* pp. 96-105, February 2004.

[18] NVIDIA. Bringing High-End Graphics to Handheld Devices. 2011. `http://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf`.

[19] J.-M.Parcerisa and A. González. "Improving Latency Tolerance of Multithreading through Decoupling". IEEE Transactions on Computers, vol. 50, no. 10, pp. 1084-1094, October 2001.

[20] K. Pulli, T. Aarnio, K. Roimela and J. Vaarala. "Designing Graphics Programming Interfaces for Mobile Devices". In *Proc. of IEEE Computer Graphics and Applications*, pp. 66-75, Nov. 2005.

[21] Qualcomm. "Two-Headed Snapdragon Takes Flight". `http://www.qualcomm.com/documents/files/linley-report-dual-core-snapdragon.pdf`.

[22] J. W. Sheaffer, D. Luebke and K. Skadron. "A flexible simulation framework for graphics architectures". In *Proc. of SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pp. 85-94, Aug. 2004.

[23] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. CACTI 5.1. Tech. report, HP Laboratories. 2008.

[24] J.E. Smith. "Decoupled access/Execute Computer Architectures". In *ACM Trans. Computer Systems*, vol. 2, no. 4, pp. 289-308, November 1984.

[25] R. M. Soneira. "Smartphone "Super" LCD-OLED Display Technology Shoot-Out". `http://www.displaymate.com/Smartphone_ShootOut_1.htm`.

[26] D. Tarjan, J. Meng and K. Skadron. "Increasing memory miss tolerance for SIMD cores". In *Proc. of SC'09*, pp. 22:1-22:11, Nov. 2009.

[27] D. Tarjan, K. Skadron. "The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches". In *Proc. of SC'10*, pp. 1-10, Nov. 2010.

[28] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan and G. E. Suh. "SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading". In *Proc. of ISCA*, pp. 247-258, June 2011.

[29] D. H. Woo and Hsien-Hsin S. Lee. "COMPASS: a programmable data prefetcher using idle GPU shaders". In *Proc. of ASPLOS*, pp. 297-310, March 2010.

[30] "2G GPRS vs. 3G UMTS connection battery usage on mobile phones". `http://blog.famzah.net/2010/05/24/2g-gprs-vs-3g-umts-connection-battery-usage-on-mobile-phones/`.

[31] `http://en.wikipedia.org/wiki/Neo_FreeRunner`

[32] `http://en.wikipedia.org/wiki/Samsung_eternity`

[33] `http://en.wikipedia.org/wiki/Samsung_Galaxy_S`

[34] `http://en.wikipedia.org/wiki/Samsung_Galaxy_S_II`