Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization

Jose-Maria Arnau* Joan-Manuel Parcerisa* Polychronis Xekalakis†
*Universitat Politecnica de Catalunya
†Intel Corporation
{jarnau,jmanel}@ac.upc.edu, polychronis.xekalakis@intel.com

Abstract

Redundancy is at the heart of graphical applications. In fact, 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 have measured this fragment redundancy for a set of commercial Android applications, and found that more than 40% of the fragments used in a frame have been already computed in a prior frame.

In this paper 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 HW 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 easily outperforms the current state-of-the-art in low power GPUs.

More specifically, our experimental results show that our scheme is able to remove 59.7% of the redundant fragment computations on average. This materializes to a significant speedup of 17.6% on average, while also improving the overall energy efficiency by 8.9% on average.

1. INTRODUCTION

Graphical applications for mobile devices tend to exhibit a large degree of scene replication across frames. Figure 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 behavior is actually quite prevalent for mobile applications. Figure 2 depicts the percentage of fragment computation that is common between consecutive frames for 9 popular Android games. Overall more than 40% 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. ARM's *Transaction Elimination* compares consecutive frame buffers and performs partial updates of entire tiles [7]. *Parallel Frame Rendering* (PFR) [8] 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.

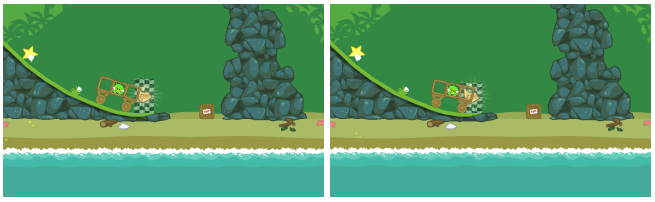

Figure 1: Two consecutive frames of the game Bad Piggies. A huge portion of the screen remains unmodified.

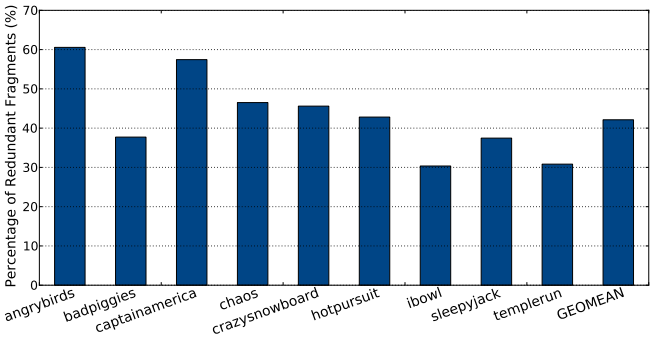

Figure 2: Percentage of redundant Fragment Program executions for 9 Android games. On average, 42.72% of the executions are redundant.

In Figure 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 conscutive frames. Removing redudant computation and memory accesses results in a 39.5% speedup on average over the current state-of-the-art. Moreover, as shown in Figure 4, energy is also improved by 27%.

In this paper, we remove a significant fraction of this redudant work by adding a task-level memoization scheme on top of PFR. We utilize the fact that in graphics programming, it is relatively easy to track changes to global arrays, such as buffers or textures, since the programmer cannot directly access the graphics memory. We thus keep a HW 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

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

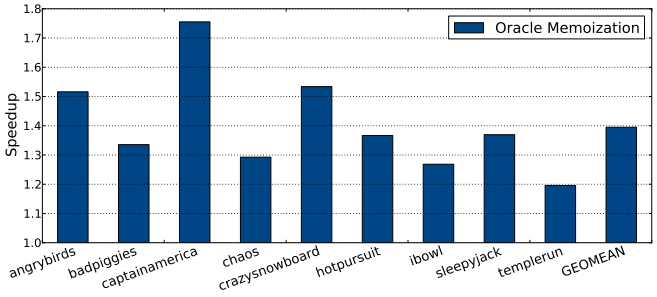

Figure 3: Performance increase achieved with an Oracle memoization system. On average, 39.5% speedup can be achieved by removing all the redundant Fragment Program executions. The baseline is a state-of-the-art mobile GPU.

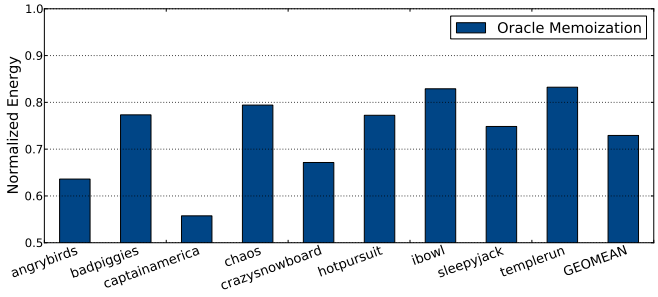

Figure 4: Potential energy savings of removing all the redundant Fragment Program executions. On average, 27% energy can be saved with respect to a state-of-the-art mobile GPU.

the memoized fragments. Hits in the HW structure result in the removal of all relevant fragment computation.

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

1. We provide a detailed analysis on the fragment redundancy that exists across frames, and show that not all fragments are equally predictable due to a non-uniformity in their complexity.
2. We present a task-level memoization scheme that when architected on top of PFR, it manages to improve energy efficiency by 8.9% on average and increases performance by 17.6%.
3. We analyze the effect of using imperfect hash functions on the overall image quality, and show that even small signatures of just 32-bits are able to achieve high image fidelity.

The remainder of this paper is organized as follows: The next section provides background information on the baseline GPU architecture. Section 3 provides an analysis of the fragment redundancy as seen on commercial applications. Section 4 presents the proposed task-level memoization scheme. Section 5 describes our evaluation methodology and Section 6 outlines the performance and power results that were obtained. Section 7 reviews related work and finally, Section 8 sums up with our conclusions.

2. BACKGROUND

The basic architecture assumed along this paper tracks closely the ARM Mali 400-MP [19] mobile GPU. This is a GPU with programmable vertex and fragment shaders, which uses Tile-Based Deferred Rendering (TBDR) [26]. It consists of 3 main components: the Geometry Unit, the Tiling Engine and the Raster Unit, as shown in Figure 5.

In the Geometry Unit, input vertices are read from memory. Next they are transformed and shaded by a user defined Vertex Program running in a Vertex Processor. Finally, they are assembled into the corresponding triangles in the Primitive Assembly stage, where non-visible triangles are culled and partially visible triangles are clipped.

The Tiling Engine gives support to TBDR, a technique that avoids the overdraw problem [21] produced by the conventional Immediate Mode Rendering (IMR). In IMR, each transformed triangle is immediately sent down the graphics pipeline for further pixel processing. Since the colors of pixels where graphical objects overlap are drawn over the top of one another, they are written multiple times into memory causing an increased memory traffic and wasting energy. In contrast, TBDR completely avoids this problem by dividing the screen into small rectangular blocks of pixels or tiles (typically 16 x 16 pixels), and processing them one at a time. Since each tile is small enough that all its pixels may be stored in local on-chip memory, each pixel color is not transferred to main memory until the whole tile is rendered, hence writing it only once. In the Tiling Engine, the Polygon List Builder saves the 2D triangles to the Scene Buffer [6] in memory. For each tile that a triangle overlaps, a pointer to that triangle is stored. After this has been done for all the geometry, each tile contains a list of 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 for rendering. Obviously, TBDR trades pixel for geometry memory traffic, but overall TBDR has proved to be effective in avoiding off-chip memory accesses on a mobile GPU [6]. Since overdraw results in a significant waste of memory bandwidth, hence in energy, it is not surprising that TBDR is becoming increasingly popular in the mobile GPU segment: the ARM Mali [19], the Imagination PowerVR [22] and the Qualcomm Adreno [1] are clear examples.

The Raster Unit computes the colors of the pixels within a tile. Initially, the Rasterizer converts the triangles into fragments, where a fragment is all the state that is necessary to compute the color of a pixel (screen coordinates, texture coordinates or even user defined application specific attributes). Fragments that are occluded by those previously processed in the same tile are discarded in the Early Depth Test stage and the remaining ones are written to the Input Register File of the Fragment Processor.

The Fragment Processor, shown in more detail in Figure 6, runs a user defined Fragment Program that computes the fragment color. It reads the fragment data generated by the Early Depth Test unit from the Input Register File and writes the

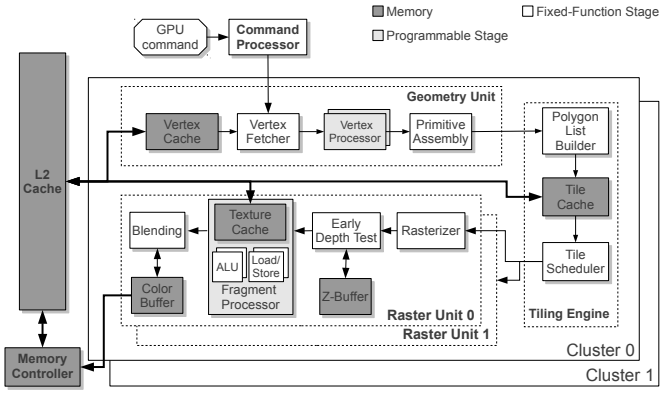

Figure 5: The assumed baseline GPU architecture consists on a state-of-the-art mobile GPU that employs TBDR and Parallel Frame Rendering to process multiple frames at a time.

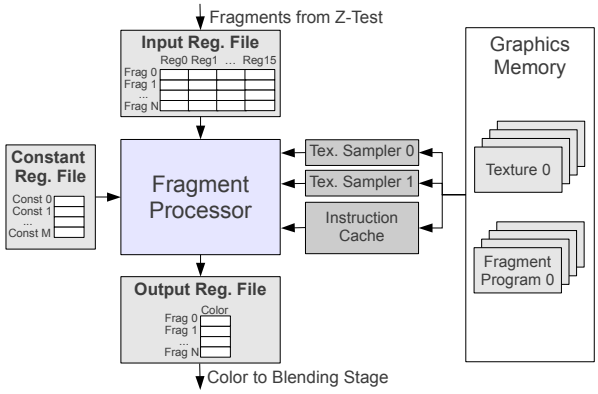

Figure 6: Fragment Stage of the GPU pipeline. The figure shows all the inputs required to process a fragment, including the per-fragment attributes stored in the input registers, the textures samplers, the fragment program and constant memory. The output produced by this stage is the color of the fragment.

resulting output color to the Output Register File, which will later be used in the Blending Stage. The Fragment Processor also includes specialized texture sampling units for processing texture fetching instructions, which access Texture data in memory.

The GPU memory hierarchy includes several first level caches employed for storing geometry (Vertex and Tile Caches) and textures (Texture Cache), and are connected through a shared bus to the L2 cache. 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 TBDR, the GPU employs local on-chip memories for storing the portion of the Color Buffer and the Z-Buffer corresponding to a tile. The local on-chip Color Buffer is directly transferred to system memory when all the triangles for the tile have been rendered. The local on-chip Z-Buffer does not need to be written to main memory.

Two of the GPU components, the Vertex Processors and the Fragment Processors, are fully-programmable simple in-order vector processors whereas the rest are fixed-function stages. A form of SMT is employed, where the processor interleaves the execution of multiple SIMD threads to avoid stalls due to long latency memory operations.

3. 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 [23] and at the instruction (or set of instructions) level in hardware [25]. 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.

3.1. Reuse Distance and PFR

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 2 shows that for the graphical applications that we use as our focus in this paper, 42.72% of the fragments are redudant. However, re-use alone is not enough for HW based memoization solutions. Bound by power/area limitations, the HW-based memoization schemes also require that the re-use distance between redundant computation 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 occurences of the same fragment, a slightly modified usage of the term from its typical use [16]. 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 7 illustrates the distribution of the re-use distances between redundant fragments for the set of Android games we analyzed (see Section 5) for re-use distances up to 2k fragments. Redundant fragments in a conventional GPU, such as the one described earlier, tend to exhibit a very uniform distribution. Unfortunately, this is very bad news for any memoization scheme as only 10% of the redudant fragments can be captured with some realistic HW constraints. This is somewhat expected, as most of the redundancy tends to be inter-frame and frames are relatively big. As such, any fragment memoization scheme needs to somehow overlap the

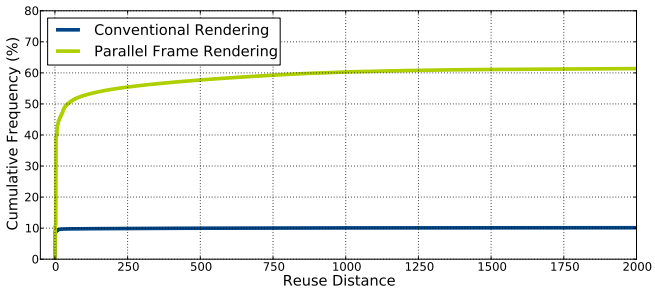

Figure 7: Cumulative frequency of distances between redundant fragments for Conventional Rendering and PFR.

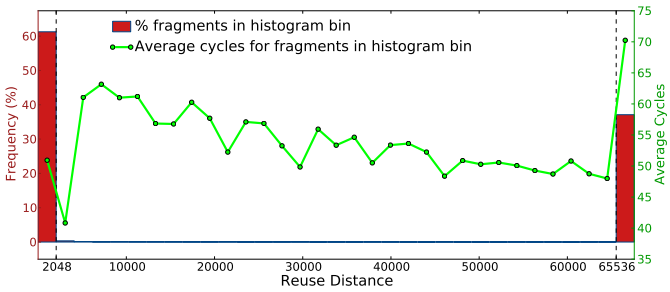

Figure 8: Reuse distance histogram with PFR, including percentage of fragments and average complexity for each histogram bin. The last histogram bin includes all the distances that are bigger than 65536. The complexity is measured as the number of GPU cycles required to process the fragment.

fragment computations from sub-sequent frames.

This observation motivates the adoption of Parallel Frame Rendering (PFR) [8] as our starting point. PFR tries to render two consecutive frames in parallel. As shown in Figure 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 TBDR as described in the previous section. 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 7, 50% of the redundant fragments have re-use distances smaller than 64 fragments, and 61.3% smaller than 2000. This is a significant improvement over re-use distances of the baseline GPU. Figure 8 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 7, all redundant fragment are represented in this graph. As pointed-out before, 61.3% 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).

3.2. 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 [24], this concept is important because computation re-use is lucrative only when the cost of accessing the structures used for memoization 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 [11, 13, 4, 5, 15, 28] or for long latency operations [10].

Figure 8 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 (70.2 cycles on average). However, fragments at smaller re-use distances, which are the target of our scheme, are also relatively complex (50.9 cycles on average). Figure 9 provides a more detailed view of the fragment complexity distribution. 45.6% of the fragments that could be re-used and exist in large re-use distances are more complex than the total average execution time. In contrast, only 29.2% 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 6, this is still enough to provide substantial performance and energy gains.

3.3. 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 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 mem-

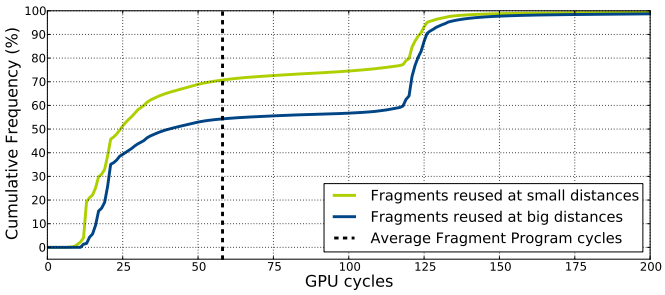

Figure 9: Cumulative histogram of fragment complexity for fragments reused at small and at big distances. Complexity is measured as number of GPU cycles required to process the fragment.

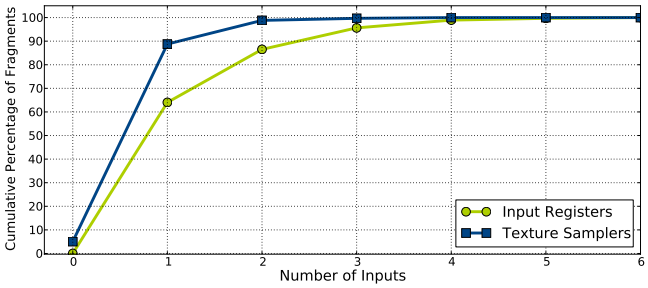

Figure 10: Two cumulative histograms for the number of input registers and number of texture samplers. With just 4 of each we cover 98.9% of the fragments, although in OpenGL ES the programmer can use up to 16 input registers and 8 texture samplers.

ory. 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 average time between updates to global memory in our set of Android games is 93.10 seconds, which amounts to thousands of frames.

4. TASK LEVEL HARDWARE-BASED MEMOIZATION ON A MOBILE GPU

4.1. 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 11 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 10, 98.9% 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 12. Input Registers are vector registers that consist of four 32-bit single precision FP components. Based on the fuzzy memoization paradigm [34], we take off the 8 least significant bits of the mantissa for every component. Hence, 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 [12].

Figure 12 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:

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

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 two input xor gates, this means that we need a tree of 6 levels of xor operations. 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 function makes each index bit to be generated from a smaller number of fragment bits. As we will show in Section 6, 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

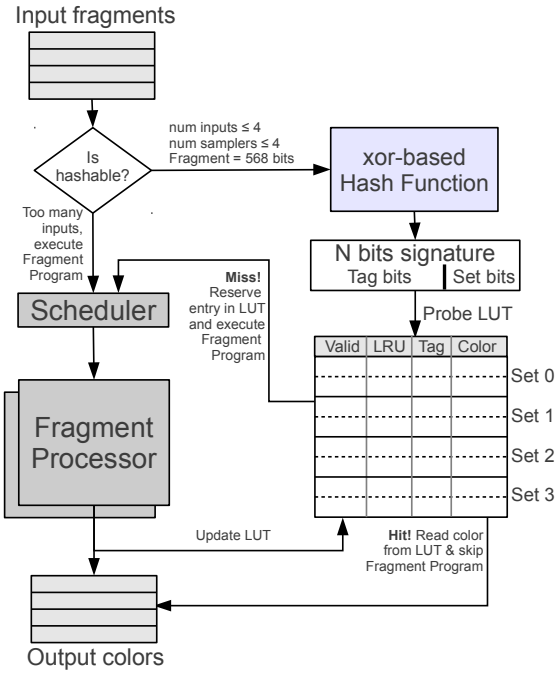

Figure 11: Proposed hardware-based task level memoization system.

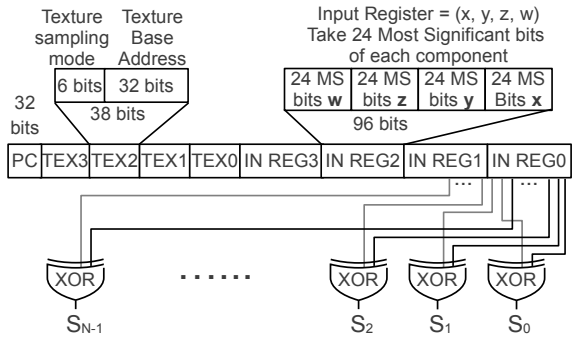

Figure 12: Computation of the hash function. The input fragment description contains information from the Fragment Program (base PC), the Texture Samplers and the Input Registers.

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 6, 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 [3] 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, 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. We block redundant fragments that arrive while the corresponding fragment is still being computed, once the redundant color is ready we wake-up and bypass the fragments.

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 [29] 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 14.25KBytes in total. Based on estimations using McPAT, the whole memoization scheme including the hash computation logic and the LUT accounts for only 0.6% 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 3.3, 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 graph-

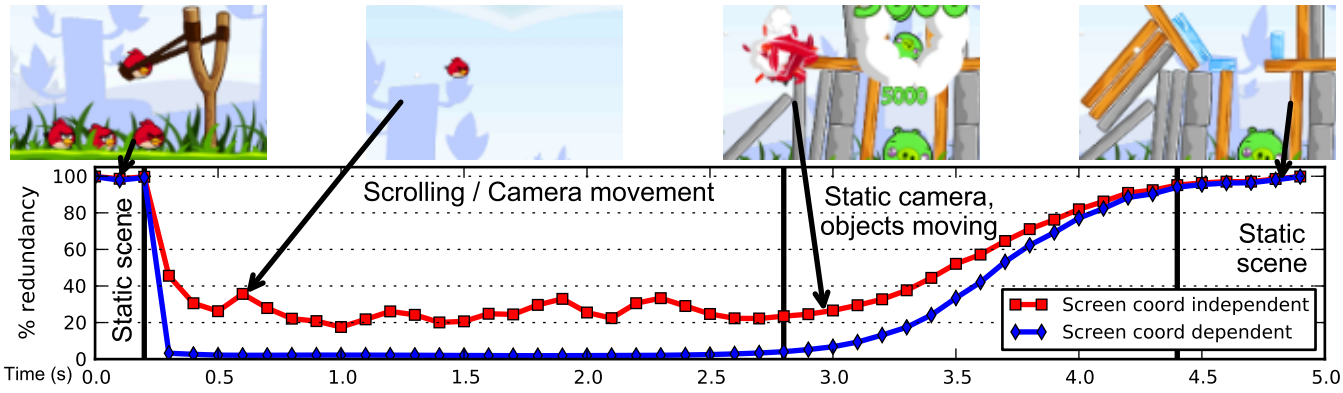

Figure 13: The graph shows the percentage of redundant fragments vs time for the game Angry Birds. For the "Screen coord dependent" configuration the screen coordinates are always used to form the signature, whereas for the "Screen coord independent" they are excluded from the signature provided that 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.

ics 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.

More redundant computations can be removed by processing more frames in parallel. However, this comes at the cost of worse responsiveness [8] if the hardware resources are not increased and diminishing returns as similarity decreases with frame distance.

4.2. 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

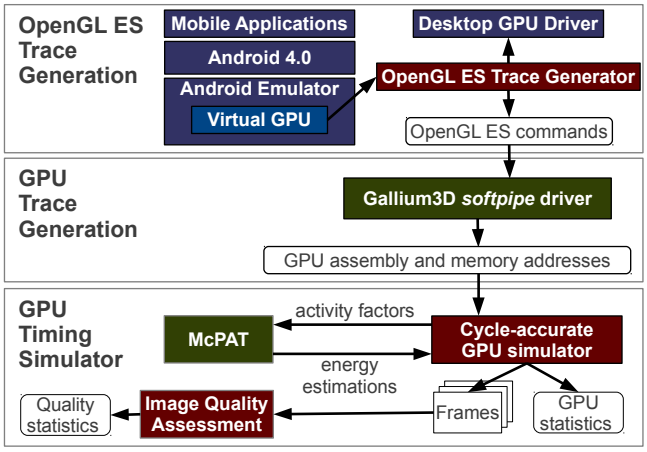

Figure 14: Mobile GPU Simulation Infrastructure.

accessed in the Fragment Program.

The benefits of excluding the screen coordinates from the signature are illustrated in Figure 13. 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 6 include this optimization, since we found it to be beneficial for all the workloads.

5. EVALUATION METHODOLOGY

We employ the TEAPOT toolset [9], a mobile GPU simulation infrastructure that runs unmodified Android applications. The infrastructure is illustrated in Figure 14 and it consists of three main components: the *OpenGL ES trace generator*, the *GPU trace generator* and the *cycle-accurate timing simulator*. Regarding the first component, the Android Emulator available in the Android SDK is used to run the Android OS in a desktop machine. Furthermore the mobile software, i. e. the Android games, are executed on top. The GPU virtualization [33] feature of the Android emulator is employed, so the OpenGL ES commands issued by the mobile applications are redirected to the GPU driver of the desktop machine, providing hardware accelerated graphics for the Android OS running inside the emulator. The *OpenGL ES trace generator* consists of a library interposed between the emulator and the desktop GPU driver. This library captures all the OpenGL ES commands issued by the Android applications, saves the per-command information in a trace file and redirects the commands to the appropriate GPU driver.

The OpenGL ES traces are fed to an instrumented version of the Gallium3D [32]. Gallium3D provides an infrastructure for building GPU drivers and it includes a complete software renderer, *softpipe*. This software renderer runs the OpenGL ES commands on the CPU, providing GPU functional simulation and generating the GPU instruction and memory traces. Note that a software renderer is different from real hardware, so special care is taken to only trace 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 the graphics hardware, i. e. memory accesses to read/write geometry, textures and the framebuffers. All the per-fragment data is also stored in the GPU trace, including the input registers and the state of the texture samplers, so redundant fragments can be identified.

Table 1: Hardware parameters employed for the experiments.

Technology	32 nm
Frequency	300 Mhz
Tile Size	16×16
Screen resolution	800x480 (WVGA)
Number of GPU clusters	2
Raster Units per cluster	2
Vertex Processors per cluster	2
L2 cache	128 KB, 8-way, 12 cycles
Texture caches	8 KB, 2-way, 1 cycle
Tile cache	32 KB, 4-way, 4 cycles
Vertex cache	4 KB, 2-way, 1 cycle
Main memory	1 GB, 16 bytes/cycle
Lookup Table num sets	$8 \rightarrow 2048 : 2*$
Lookup Table num ways	2, 4, 8
Hash size	24, 32, 64, 128, 256

Finally, the GPU instruction and memory trace is used to drive the *cycle-accurate timing simulator*, that models the mobile GPU architecture illustrated in Figure 5. Regarding the power model, the McPAT [17] framework provides energy estimations. The parameters employed during the simulations are summarized in Table 1. We have implemented the hardware memoization system described in Section 4 on top of the timing simulator. The per-fragment data stored in the GPU trace is employed to compute the fragment signatures, access the LUT and avoid re-execution in case of a hit. Note that conflicts in the LUT table can introduce artifacts in the resulting frames. The *Image Quality Assessment* included in TEAPOT provides an estimation of the impact of conflicts on image quality, by comparing the original frames generated by a conventional GPU with the images created by using our memoization system. A well-established metric, the Mean Structural SIMilarity Index (MSSIM) [30] is used to evaluate image quality and we have manually confirmed the results.

The set of benchmarks employed to evaluate our technique include 9 commercial Android games that are representative of the mobile graphical applications since they employ most of the features available in the OpenGL ES 1.1/2.0 API. We have included 2D games (angrybirds and badpiggies), since they are still quite common in the mobile segment. Furthermore, we have included simple 3D games (crazysnowboard, ibowl and templerun), with small Fragment Programs and simple 3D models. Finally, we have selected more complex 3D games (captainamerica, chaos, hotpursuit and sleepyjack) that exhibit a plethora of advanced 3D effects. Regarding the length of the simulations, we have generated traces of 100 frames for each game. During trace generation we tried to

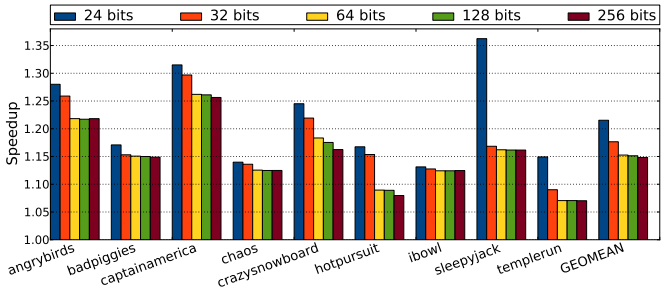

Figure 15: Speedups achieved by hardware memoization for different sizes of the signature. The baseline configuration is PFR.

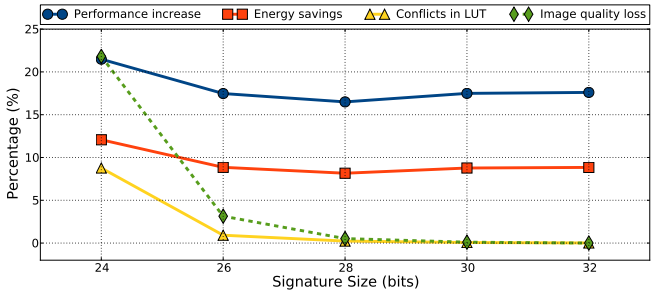

Figure 16: Impact of signature size on performance, energy savings, conflicts in the LUT and image quality. The baseline configuration is PFR.

avoid unrealistic situations that favour 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 2, the average redundancy in our traces is 42.72%, and all the individual 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, our 2D games angrybirds and badpiggies include phases with static background and phases with scrolling as illustrated in Figure 13. 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 4.2 is still able to expose significant redundancy when the camera is moving.

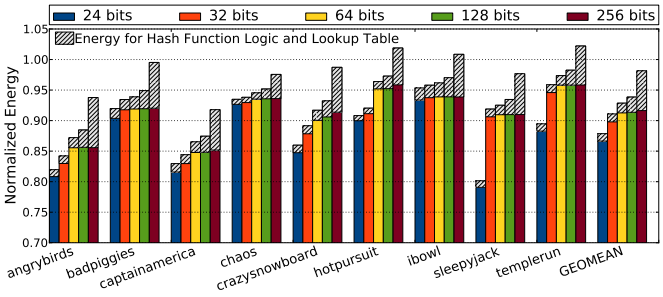

Figure 17: Normalized energy for different sizes of the signature. The baseline configuration is PFR.

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

Figure 18: 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.

6. EXPERIMENTAL RESULTS

In this section we will provide details on the performance and energy efficiency of the proposed scheme. The baseline for all our experiments is a PFR capable GPU similar to that shown in Figure 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.

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 512 sets.

Figure 15 shows the speedups attained for different signature sizes. On average, 14.8% speedup is achieved with a 256-bit signature, whereas 17.6% and 21.5% 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 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. Figure 16 shows the percentage of conflicts for different signature sizes. For 24-bit signatures we have conflicts for 8.77% of the fragments, whereas for 32-bit signatures we only have one conflict every 10 frames on average. There is no conflict at all for 256-bit signatures (not shown in the Figure). Hence, the 24-bit signature incorrectly removes the execution of an additional 8.77% of the fragments with respect to the 256-bit signature, achieving

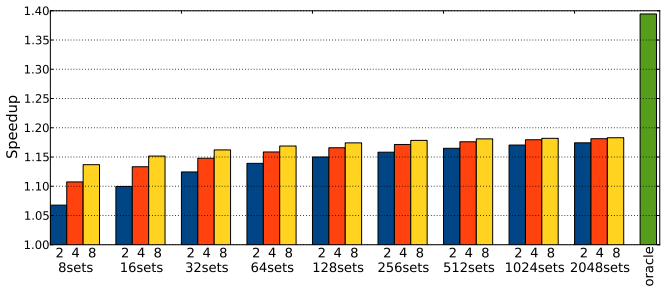

Figure 19: Speedups achieved for LUTs with different number of sets and ways. The baseline configuration is PFR.

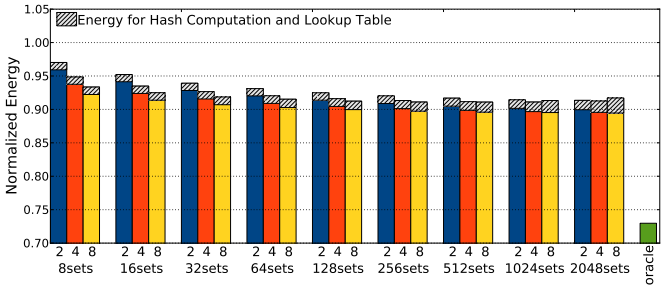

Figure 20: Normalized energy for different configurations of the Lookup Table, including both static and dynamic energy. Baseline is PFR.

significant speedups. Note that the 32-bit signature has a very small number of conflicts but it still gets speedups with respect to 256-bit signatures. As we decrease the signature size we also improve the dispersion of the accesses to the LUT, as described in Section 4. 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 25.2% on average in the LUT, whereas the 256-bit signature only obtains 19.6% hit rate.

Reducing the signature size improves energy consumption as illustrated in Figure 17. The energy savings come from two sources. The obvious source is that the size of the LUT is reduced, since the signature has 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 256-bit to 32-bit reduces the energy consumed by the LUT by 80% and overall GPU energy by 7.2%.

Figure 16 shows the effect of the signature size on performance, energy, number of conflicts 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, 8.77%, that introduce noticeable distortions on image quality. More specifically, image quality drops by

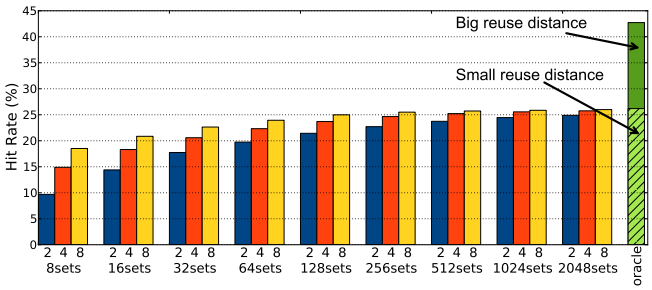

Figure 21: Hit Rates for different configurations of the Lookup Table. The baseline configuration is PFR.

21.8%, this is an important percentage that introduces visible artifacts in the images as illustrate in Figure 18. 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 19 shows the speedup for 2-way, 4-way and 8-way associative LUTs with a number of sets from 8 to 2048. As expected, increasing the associativity improves performance in all the configurations. For example, for 512 sets the version with 2-ways obtains 16.4% speedup, whereas the version with 8-ways achieves 18.1% performance increase. For a given number of ways, increasing the number of sets provides noticeable improvements initially but the speedup saturates at 256-512 sets. Although the results are far from the Oracle memoization system, all the configurations provide consistent and provide substantial speedups over the baseline GPU.

Figure 20 shows the normalized energy for the same configurations of the LUT. Initially, increasing the size of the LUT provides significant energy savings since it improves the hit rate. However, the speedup and the hit rate saturate at 256-512 sets. Beyond this point, increasing the LUT size increases overall energy consumption since it is not going to improve the hit rate but it increases the energy required to access the table. As an example, the configuration with 8 sets and 4-way achieves 5.1% energy savings, whereas increasing the number of sets to 512 provides 8.8% savings. However, switching to 2048 sets results in 8.7% savings, smaller benefits than 512 sets due to the bigger energy requirements of the LUT (73% more energy for the LUT with 2048 sets).

Figure 21 depicts the hit rates in the LUT. Again, increasing the size of the LUT produces significant improvements in the hit rate for the first steps, but the benefits saturate beyond 256-512 sets. In the best case, we have 26% hit rate whereas the Oracle achieves 42.7%. Hence, the hardware LUT is able to remove 60.8% of the redundant fragments detected by the Oracle, but if we look at the speedups the hardware LUT only gets 46.6% of the speedup achieved by Oracle memoization. This is due to the non-uniform complexity of the fragments, as described in Section 3. The LUT can only capture fragments reused at small distances, i.e. fragments with temporal locality, and those fragments exhibit smaller complexity (50.9 GPU cycles on average) than the ones reused

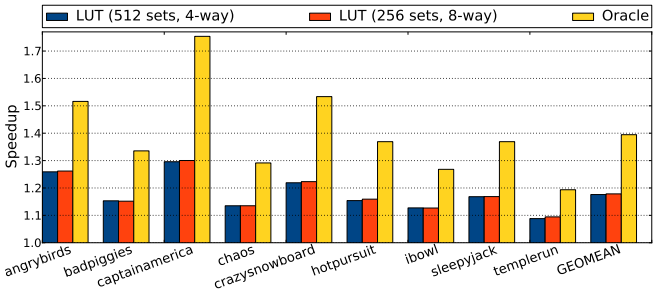

Figure 22: Speedups for the 9 Android games and on average. We include two of the best configurations for the LUT and the Oracle memoization. The baseline configuration is PFR.

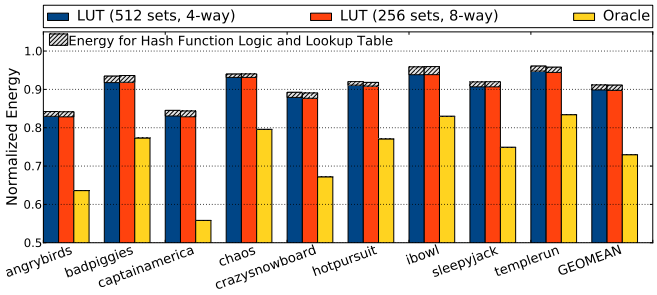

Figure 23: Normalized energy for the 9 Android games and on average, including both static and dynamic energy. We include two of the best configurations for the LUT and the Oracle memoization. Baseline is PFR.

at big distances (70.2 cycles on average). Hence, the Oracle gets significant speedups out of the 39.2% extra fragments that is able to capture.

Figure 21 also splits the Oracle hit rate (total 42.7%) between fragments reused at small and big distances. Since the redundant fragments at small distances amount to 61.3% of all redundant fragments, as mentioned in Section 3, the small distance oracle hit rate is 26.1%. Hence, a small LUT can only aim to capture this 26.1% of fragments with temporal locality. The results shows that a LUT with 256 sets and 8-way avoids 25.5% of the Fragment Program executions, whereas a LUT with 512 sets and 4-way obtains a hit rate of 25.2%. Hence, small LUTs are very effective in capturing redundant fragments with temporal locality.

Finally, Figure 22 and Figure 23 depict the per-game speedups and energy savings respectively, for a hardware memoization system with 32-bit signatures and two configurations for the LUT: 512 sets 4-way and 256 sets 8-way. As it is shown, the system achieves consistent performance improvements and energy savings across all the applications. When focusing on the performance aspect, the biggest speedup is achieved in captainamerica, 30% increase, whereas templerun exhibits the smallest speedup, 8.8%. Regarding the energy, the savings are in the range from 3.93% (templerun) to 15.84% (angrybirds). The energy consumed by the LUT represents 1.42% of the overall GPU energy for the LUT with 512 sets and 4-way and 1.49% for the LUT with 256 sets and 8-way associative, on average. In the worst case the LUT consumes up to 2.19% of total GPU energy, and 0.91% for the best case.

7. RELATED WORK

Exploiting the high degree of redundancy in graphical applications has attracted the attention of the architectural community in recent years. ARM's *Transaction Elimination* [7] performs a per-tile comparison of consecutive frames to avoid transferring redundant tiles to main memory. Parallel Frame Rendering [8] processes multiple frames in parallel to overlap the texture accesses of consecutive frames and improve the locality in the L2 cache. 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. [18] propose the use of a software managed cache, implemented in GPU local memory, to memoize redundant colors for stochastic rasterization [2]. 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. [27] 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.

Memoization has been subject of research for several decades. Hardware memoization applies to single instructions or blocks of instructions [25, 10, 11, 13, 4, 5, 15, 28], whereas software-based solutions aim to memoize entire functions [23, 14, 20, 31]. Our memoization system is function level and hardware-based, since it 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.

Alvarez et al. [34] 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 [5] 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.

8. CONCLUSION

In this paper we have shown that more than 40% 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, 61.3% of the redundant fragments are brought into re-use distances that make them amenable to HW 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 17.6% speedup for a set of commercial Android games. This improves the energy efficiency of the system by 8.9% on average. All this, comes at a neglible cost in terms of image distortion, which as shown in the paper is not perceivable.

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] "Qualcomm Adreno 320," http://www.anandtech.com/show/6112/qualcomms-quadcore-snapdragon-s4-apq8064adreno-320-performance-preview.

[2] T. Akenine-Möller, J. Munkberg, and J. Hasselgren, "Stochastic rasterization using time-continuous triangles," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH'07, 2007, pp. 7–16.

[3] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42Nd Annual Southeast Regional Conference*, ser. ACM-SE 42, 2004, pp. 267–272.

[4] C. Álvarez, J. Corbal, E. Salamí, and M. Valero, "On the potential of tolerant region reuse for multimedia applications," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01, 2001, pp. 218–228.

[5] C. Alvarez, J. Corbal, and M. Valero, "Dynamic tolerance region computing for multimedia," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 650–665, May 2012.

[6] I. Antochi, B. H. H. Juurlink, S. Vassiliadis, and P. Liuha, "Memory Bandwidth Requirements of Tile-Based Rendering," in *SAMOS*, 2004, pp. 323–332.

[7] ARM, "Transaction elimination." Available: http://www.arm.com/products/multimedia/mali-technologies/transaction-elimination.php

[8] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Parallel frame rendering: Trading responsiveness for energy on a mobile gpu," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13, September 2013, pp. 83–92.

[9] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 37–46.

[10] D. Citron, D. G. Feitelson, and L. Rudolph, "Accelerating multi-media processing by implementing memoing in multiplication and division units," in *ASPLOS*, 1998, pp. 252–261.

[11] D. A. Connors, H. C. Hunter, B.-C. Cheng, and W.-m. 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*, ser. ASPLOS IX, 2000, pp. 222–233.

[12] E. B. Goldstein, *Sensation and Perception*, 6th ed. Univ. of Pittsburgh, 2002.

[13] A. González, J. Tubella, and C. Molina, "Trace-level reuse," in *In Proceedings of the the International Conference on Parallel Processing*, 1999.

[14] M. Hall and J. 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*, 1993, pp. 178–184.

[15] J. Huang and D. J. Lilja, "Exploiting basic block value locality with block reuse," in *HPCA*, 1999, pp. 106–114.

[16] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction." in *Proceedings of the 25th International Conference on Computer Design*, ser. ICCD, 2007, pp. 245–250.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[18] G. Liktor and C. Dachsbacher, "Decoupled deferred shading for hardware rasterization," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D'12, 2012, pp. 143–150.

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

[20] P. McNamee and M. Hall, "Developing a tool for memoizing functions in c++," *SIGPLAN Not.*, vol. 33, no. 8, pp. 17–22, Aug. 1998.

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

[22] "PowerVR Technology Overview," http://www.imgtec.com/factsheets/SDK/PowerVR%20Technology%20Overview.1.0.2e.External.pdf.

[23] H. Rito and J. a. 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*, ser. PPPJ'10, 2010, pp. 89–98.

[24] S. S. Sastry, R. Bodik, and J. E. Smith, "Characterizing coarse-grained reuse of computation," in *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, 2000, pp. 16–18.

[25] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97, 1997, pp. 194–205.

[26] "Tiled Rendering," http://en.wikipedia.org/wiki/Tiled_rendering.

[27] H.-W. Tseng and D. M. Tullsen, "Data-triggered threads: Eliminating redundant computation." in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, ser. HPCA, 2011, pp. 181–192.

[28] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima, "Design and evaluation of an auto-memoization processor," in *Parallel and Distributed Computing and Networks*, 2007, pp. 230–235.

[29] P.-H. Wang, Y.-M. Chen, C.-L. Yang, and Y.-J. Cheng, "A predictive shutdown technique for gpu shader processors." *Computer Architecture Letters*, vol. 8, no. 1, pp. 9–12, 2009.

[30] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image Quality Assessment: from Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[31] H. Xu, C. J. F. Pickett, and C. Verbrugge, "Dynamic purity analysis for java programs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07, 2007, pp. 75–82.

[32] "Gallium3D," http://en.wikipedia.org/wiki/Gallium3D/.

[33] "GPU Virtualization in the Android Emulator," http://developer.android.com/tools/devices/emulator.html#acceleration.

[34] C. Álvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.