Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline

Martí Anglada[1] Enrique de Lucas[2] Joan-Manuel Parcerisa[1]
Juan L. Aragón[3] Pedro Marcuello[3] Antonio González[1]
[1]Universitat Politècnica de Catalunya [2]Semidynamics Technology Services, Barcelona [3]Universidad de Murcia
{manglada, jmanel, antonio}@ac.upc.edu
{enrique.delucas, pedro.marcuello}@semidynamics.com jlaragon@um.es

***Abstract*—GPUs are one of the most energy-consuming components for real-time rendering applications, since a large number of fragment shading computations and memory accesses are involved. Main memory bandwidth is especially taxing battery-operated devices such as smartphones. Tile-Based Rendering GPUs divide the screen space into multiple tiles that are independently rendered in on-chip buffers, thus reducing memory bandwidth and energy consumption. We have observed that, in many animated graphics workloads, a large number of screen tiles have the same color across adjacent frames. In this paper, we propose *Rendering Elimination* (RE), a novel micro-architectural technique that accurately determines if a tile will be identical to the same tile in the preceding frame before rasterization by means of comparing signatures. Since RE identifies redundant tiles early in the graphics pipeline, it completely avoids the computation and memory accesses of the most power consuming stages of the pipeline, which substantially reduces the execution time and the energy consumption of the GPU. For widely used Android applications, we show that RE achieves an average speedup of 1.74x and energy reduction of 43% for the GPU/Memory system, surpassing by far the benefits of Transaction Elimination, a state-of-the-art memory bandwidth reduction technique available in some commercial Tile-Based Rendering GPUs.**

***Keywords*-Graphics Pipeline; Energy Efficiency; Tile-Based Rendering**

I. Introduction

Graphics applications for smartphones and tablets have become ubiquitous platforms for entertainment, with more than 2 billion users worldwide and more than a 40% share of the overall games market [1]. The portable nature of such devices drives engagement to games with simple gameplay that can be played in short bursts, such as puzzle, strategy or casual games, genres that represent the greatest number of downloads and played time [2], [3], [4]. While games of those characteristics usually do not involve complex scenes and cutting-edge effects, rendering their scenes still requires a substantial amount of power, a limited resource in battery-operated devices. Consequently, reducing the energy consumption of the GPU is a major concern of hardware and software designers [5], [6], [7], [8], [9].

Figure 1 shows the average power consumption and GPU load for the Android desktop (without animations), for several commercial Android games and the Antutu bench-

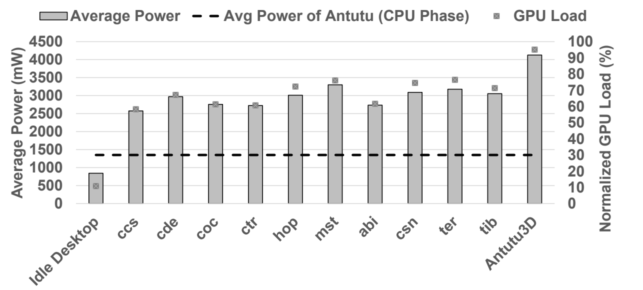


Figure 1. Overall average power consumption. GPU load is normalized by weighting it by the ratio between operating and maximum GPU frequency. Data obtained using Trepn Profiler [10] for a Snapdragon 636 with connections disabled and minimum screen brightness.

mark [11], divided into CPU and GPU phases (*Antutu3D*). As it can be seen, applications with simple scenes such as CandyCrush (*ccs*) require a substantial amount of power and GPU load, comparable to an application designed to stress the GPU. Note that they also drive much more power than the Android desktop, that lets the GPU mostly idle, while consuming twice as much as an application that only stresses the CPU. These experimental results confirm the popular claim that, in graphics applications, the GPU and its communication with main memory (loading textures and storing colors, among other tasks) are the greatest contributors to energy consumption [12], [13], [14].

A state-of-the-art pipeline design employed to reduce bandwidth in mobile GPUs is *Tile-Based Rendering* (TBR). In TBR, a frame space is divided into a grid of tiles that are independently rendered, which allows to do a variety of computations leveraging small, fast, local on-chip memory instead of using main memory. The graphics pipeline in a TBR GPU is divided into two decoupled pipelines: the Geometry Pipeline receives vertices and generates, after a set of transformations, output primitives (triangles) that are sorted into tile bins and stored into the main memory Parameter Buffer; and the Raster Pipeline which traverses the tiles one at a time, fetching each tile's primitives, rasterizing each primitive into fragments, and shading each fragment to obtain a final pixel color.

A main purpose of the GPU is to render sequences of images. In order to produce fluid animations, consecutive frames tend to be similar, i.e., it is usual to find regions in a frame with the same color as in the preceding frame,

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00014

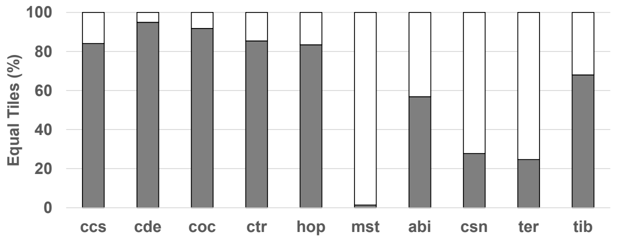


Figure 2. Percentage of tiles producing the same color as the preceding frame across 50 consecutive frames (experimental details in Section IV).

which implies that a significant amount of computations are redundant. Figure 2 illustrates this phenomenon, known as Frame-to-Frame coherence [15], by plotting the average percentage of equal tiles between two consecutive frames for a set of commercial Android games. In games with moderate camera movements (*ccs* to *hop*), over 90% of tiles produce the same color as in the preceding frame. This feature can also be found, albeit less frequently, in games where the scene is in continuous motion (*mst* to *tib*).

Several previous works attempted to exploit frame-to-frame coherence in order to improve energy efficiency. Transaction Elimination [16] (TE) compares a signature of the colors generated after rendering a tile with the signature of the same tile in the preceding frame. If they are equal, the color update to main memory is avoided. Arnau et al. [17] proposed a task-level Fragment Memoization scheme that computes, for each fragment, a signature the shader inputs and caches it, along with the output color, in a LUT. Subsequent fragments form their signatures and check them against the signatures of the memoized fragments. In case of a hit, the shader's computation and associated texture accesses are avoided and the cached color is used instead. Because most redundancy resides between consecutive frames, the huge reuse distance makes impractical to store a frame's worth of signatures and output values. To help reduce the reuse distance, it builds on top of PFR [18], an architecture that renders two consecutive frames in parallel and keeps tiles synchronized. But PFR cuts in half the redundancy detection potential: even frames reuse values cached by the previous (odd) frame, but odd frames cannot because their previous-frame values are already evicted from the LUT by the time they are rendered.

We make the observation that in a TBR GPU, primitives do not need to be discretized into fragments to know that the final result will be the same as in the preceding frame. Instead, by managing redundancy at a tile level, redundant tiles may be discovered much earlier than at a fragment level and bypass the whole Raster Pipeline. Note that the Raster Pipeline computes the pixel colors using as inputs a set of primitives' attributes generated by the Primitive Assembly stage of the Geometry Pipeline plus a set of scene constants, so it knows all the input data required to render a tile when it starts processing it.

Based on the above observation, we propose *Rendering Elimination* (RE), a novel technique that employs the input data of a tile to anticipate if all of its pixels will have the same color as in the preceding frame, and to bypass the complete rendering of the tile. Since an entire frame of these input sets must be stored on-chip, they are compared by means of a signature. In parallel with the sorting of a primitive into tiles, RE computes on-the-fly the signatures of the overlapped tiles and stores them in a local fixed-size on-chip buffer. Then, after the Geometry Pipeline has processed the frame, tiles are dispatched to the Raster Pipeline. For each tile, RE compares its current and preceding frame signatures and, if they match, all the rendering process is bypassed and the colors in the Frame Buffer are reused. Otherwise, the tile is rendered as usual.

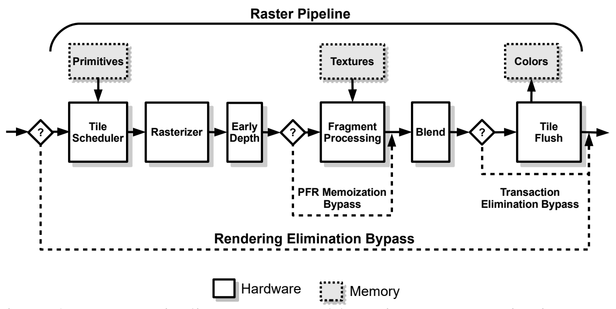


Figure 3. Raster Pipeline stages saved by using TE, Memoization or RE.

By working at a much coarser grain than Fragment Memoization [17], RE can store on-chip all the frame signatures and detect all the available tile redundancy instead of just that of the even frames, which more than compensates for the marginal undetected redundancy at sub-tile level (our results show that RE almost doubles the amount of redundancy discovered). In addition, RE does not need to store output results because tile colors are reused from the Frame Buffer, thus saving storage and bandwidth. Besides this, while TE and Fragment Memoization each skip just a single stage of the Raster Pipeline (as depicted in Figure 3), RE completely skips all the Raster Pipeline stages. Considering that almost 75% of the total GPU memory accesses (textures, colors and primitives) are generated by these stages, our approach is able to greatly reduce memory bandwidth and energy consumption.

The main contributions of this paper are: (1) The observation that frame-to-frame redundancy can be discovered in a TBR GPU at the tile level much earlier in the pipeline than previous techniques do. (2) A detailed proposal of Rendering Elimination, a technique for early discarding of redundant tiles, clearly showing how RE may be seamlessly integrated into the Graphics Pipeline with minimal hardware and performance overheads. (3) An experimental evaluation of RE that shows an average speedup of 1.74x and 43% energy reduction over a conventional mobile GPU, and substantial improvements over previous works.

II. Tile-Based Rendering Baseline

Figure 4 shows the baseline architecture used in this paper, which resembles an ARM Mali-450 GPU [19]. While it is only a single point in the design spectrum of GPUs, it serves perfectly to demonstrate the huge benefits of our technique for a broad class of GPU architectures with the only requirement to follow an OpenGL compliant TBR organization. This architecture is a hardware implementation of the Graphics Pipeline, a conceptual model that describes the stages through which data should be processed in order to render a scene. The application communicates with the GPU using commands, which are used to configure the pipeline state (shader code, constants –"uniforms"–, textures, etc.) and to trigger execution via *drawcalls*, a stream of vertices to be processed with the current state. The pipeline state is held constant during a drawcall invocation but may be altered between invocations.

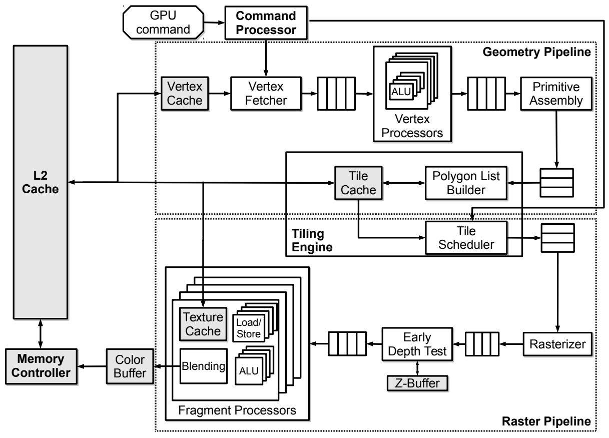


Figure 4. Assumed baseline architecture.

The Command Processor parses drawcalls and determines the format used by the application to submit vertices to the pipeline. Next, the Vertex Fetcher creates an input stream of vertices by reading information with the established format. The per-vertex read information is known as *Vertex Attributes*, and consists of sets of data that specify vertices, such as 3D space coordinates or color. The vertex stream is then *shaded*: the attributes of each vertex are transformed using Vertex Processors that execute programs set by the application. These programs are called shaders, and are shared among all vertices of a drawcall. The shaded vertices are grouped into triangles or other primitives in the final stage of the Geometry Pipeline, known as Primitive Assembly, where some of the non-visible ones are discarded applying clipping and culling techniques.

The primitives resulting from the geometry process are sent to the Tiling Engine, where the Polygon List Builder stores primitives' attributes in a region of memory known as Parameter Buffer. The attributes are stored in a format that exploits locality and enhances performance on the Raster Pipeline. The Polygon List Builder also determines in which tiles each primitive resides. After all the geometry has been sorted into tiles and saved in the Parameter Buffer, the tiles are processed in sequence. The Tile Scheduler is responsible of fetching the primitives' data for a given tile and dispatching it to the Raster Pipeline.

The Raster Pipeline starts by rasterizing primitives. The primitives are discretized into fragments: pixel-sized elements described by interpolated information from vertex attributes. The Early Depth Test is used to discard fragments that would be occluded by previously processed fragments. The fragments that pass the test are sent to the Fragment Processors, which execute application-defined shaders to compute the color for every fragment. The output color computed in the processors for a given pixel is merged with the previously computed colors using the Blending unit, and the resulting color is written into the local on-chip Color Buffer. When the Raster Pipeline has processed all of the primitives of a tile, the contents of the Color Buffer are flushed into the Frame Buffer in system memory and the Raster Pipeline begins processing the next tile.

III. Rendering Elimination

A. Overview

This paper proposes Rendering Elimination, a novel micro-architectural technique that accurately determines if a tile is redundant, i.e., if all of its pixels will have the same color as in the previous frame. Whenever a tile is detected as redundant, its Raster Pipeline execution is completely bypassed and the color from the previous frame is reused.

The Raster Pipeline takes as inputs the scene constants and the attributes of all the primitives that overlap a tile, and produces a color for each pixel belonging to that tile. In order to determine in advance redundancy for a tile, we compare its inputs for the current frame against the inputs for the previous frame: if the two input sets match, the outputs will also be equal. Because of the large volume of these sets, storing them in main memory would be extremely inefficient, even with the support of a cache, because the reuse distance between them is an entire frame. Instead, we use a more efficient approach based on computing a signature for the inputs of the tile and storing it in a local buffer. This buffer, that we call *Signature Buffer*, contains the signatures of all the tiles of the previous and current frames. Figure 5 depicts the Graphics Pipeline flow with the added Signature Buffer.

The Signature Unit computes the signatures employing the primitives that the Polygon List Builder produces and inserts them into the Signature Buffer. At the same time, the Polygon List Builder fills the Parameter Buffer with the data of such primitives, including identifiers of the tiles that contain them. After the geometry of the frame has been processed, the Signature Buffer holds signatures for the inputs of all the tiles. Hereafter, whenever a tile is scheduled in the Raster Pipeline, its Signature Buffer entry

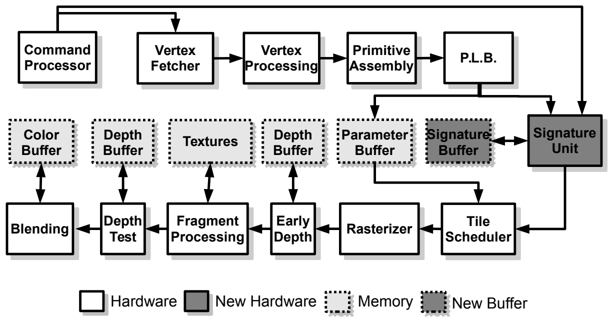


Figure 5. Graphics Pipeline including RE.

is checked: if the current frame signature matches that of the previous frame, the Raster Pipeline execution is skipped and the Frame Buffer locations for that tile are not updated. Otherwise, the Raster Pipeline is executed normally.

B. Implementation Requirements

The signature of a tile is computed by hashing a list of all the inputs of a tile: this includes the vertex attributes and scene constants associated to all the primitives that overlap the tile. Such inputs are produced either by the Command Processor when setting scene constants for a drawcall or by the Polygon List Builder when sorting primitives and storing their vertex attributes into the Parameter Buffer. The stream of primitives produced by the Geometry Pipeline, however, is generated in the order that the GPU received the drawcalls, which is generally not the order in which they appear in the screen. In fact, any primitive from the stream could overlap any number of tiles. This causes that the complete list of inputs for a tile is not known until all the geometry of the scene has been processed. A straightforward implementation that starts computing the signatures when the Geometry Pipeline has processed the whole frame would not be practical. Since vertex attributes are stored in the Parameter Buffer (residing in off-chip memory), retrieving them in order to compute a signature for the tile would require significant time and energy overheads and delaying the execution of the Raster Pipeline.

To be effective, our technique computes the signatures for the current frame in an *incremental* approach. Whenever a primitive is sorted, the temporary signatures for each tile that it overlaps are read. The new signature for each tile is constructed by combining the temporary signature with either the scene constants or the attributes of the vertices of the current primitive and, afterwards, it is rewritten in the appropriate Signature Buffer entry. This on-the-fly signature computation is overlapped with other Geometry Pipeline stages, resulting in minimal overheads in execution time.

The signature function employed by RE is CRC32 [20]. While a plethora of other mechanisms exist, CRC32 outperforms well-known hashing approaches such as XOR-based schemes, as we will show in Section V. We have not observed a single instance of hashing collisions in our benchmarks when using CRC32. Moreover, as a widely-used error detection code, CRC has been extensively researched in the literature and efficient techniques have been developed [21] that allow for an incremental and parallel CRC computation based on Look-up Tables, as outlined below.

C. Incremental CRC32 Computation

As proven in [21], the CRC of a message can be computed even if its length is not known a priori by breaking it down into several submessages and computing the CRC of those submessages independently. Given a message A, composed by concatenating submessages $A_1...An$, of lenghts $b_1...b_n$ bits, the CRC of A can be computed as:

Algorithm 1 Incremental CRC Computation
```
$CRC_A = 0$
**for** submessage $A_i$ in $A$ **do**
    $b = length(A_i)$
    $CRC_{A_i} = ComputeCRC(A_i)$
    $CRC_{Temporary} = ComputeCRC(CRC_A << b)$
    $CRC_A = CRC_{A_i} \oplus CRC_{Temporary}$
**end for**
```

That is, the CRC of the first submessage A_1 is computed. When the length b of the following submessage A_2 is known, we can compute the CRC of the two submessages (a bit string formed by concatenating A_1 and A_2) by computing the CRC of A_2, left-shifting the CRC of A_1 by b bits, computing the CRC of this shifted message, and combining both CRCs via an XOR function. By means of this procedure, CRCs of partial messages of increasing length are computed: first, the CRC of A_1, then the CRC of the concatenation of A_1 and A_2, then the CRC of the concatenation of A_1, A_2 and A_3, and so on, until the last submessage A_n is reached and, therefore, the CRC of the concatenation of the submessages corresponds to the CRC of the original message.

D. Table-based CRC32 Computation

Each iteration in Algorithm 1 would require several cycles if the CRC computation was implemented using the basic Shift Register mechanism [22]. A faster alternative is to use a Look-up Table (LUT) loaded with precomputed CRC values for all possible inputs. However, this approach is unfeasible in terms of storage requirements, since a message of length n requires a LUT of 2^n entries. As shown in [21], a message B of n bits, being n multiple of 8, can be broken into k 1-byte blocks $B_1...B_k$ ($n = 8 \times k$) and use a small LUT to efficiently compute the CRC of each block.

Each LUT takes as input a block B_i and computes the CRC of a message corresponding to left-shifting B_i by $k - i$ bytes. Namely, the first LUT computes the CRC of a message consisting of block B_1 followed by $k - 1$ bytes of zeros, the second LUT comptues the CRC of a message consisting of block B_2 followed by $k - 2$ bytes of zeros and the k^{th} LUT computes the CRC of a message consisting of block B_k. The results of the k LUTs are combined into one CRC via an XOR function.

Since each LUT has 2^8 entries and each entry contains a precomputed CRC value, the size of each LUT is 1 KB and, consequently, computing the CRC of a message of length n bits has a storage cost of k KB.

E. Tile Inputs Bitstream Structure

RE determines if the colors of two tiles are going to be the same by comparing the signature of their inputs. The inputs of a tile are the vertex attributes of the primitives that overlap it and the set of scene constants associated to those primitives. In order to render primitives, the GPU receives a series of commands that define the state of the pipeline (shaders, textures, constants) and drawcalls, which contain a stream of vertices to be processed with the defined state.

Each drawcall can generate any number of primitives and each primitive can overlap any number of tiles. Therefore, the input of a tile consists of a sequence of blocks, one for every drawcall that contains the primitives that overlap this tile. Each block is, in turn, composed of several subblocks: a first subblock corresponding to the constants defined in the drawcall followed by a list of subblocks that correspond to the attributes of the primitives that overlap this tile. Since both the number of primitives overlapping a tile and the number of attributes of those primitives is not fixed, neither are the lengths of the blocks nor is the length of the subblocks.

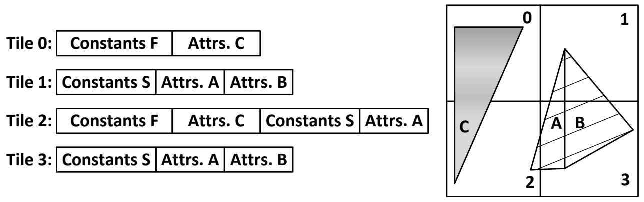

Tile 0: Constants F | Attrs. C
Tile 1: Constants S | Attrs. A | Attrs. B
Tile 2: Constants F | Attrs. C | Constants S | Attrs. A
Tile 3: Constants S | Attrs. A | Attrs. B
0
1
2
3
C
A
B
Figure 6. Example of input message.

Figure 6 provides an example of the described tile inputs for four tiles and the primitives of two drawcalls: Drawcall F (fill) and Drawcall S (stripes). Drawcall F generates Primitive C, which overlaps Tiles 0 and 2. Therefore, the inputs of Tiles 0 and 2 contain the block of Drawcall F, composed of a set of constants and the attributes of Primitive C. Drawcall S generates two primitives, Primitives A and B. These two primitives overlap Tiles 1 and 3, so the inputs of Tiles 1 and 3 contain the block of Drawcall S, composed of a set of constants and the attributes of both primitives. Note that, while two primitives of Drawcall S overlap Tiles 1 and 3, the set of constants of the drawcall is only considered once for those tiles. Primitive A also overlaps Tile 2, so the set of constants of Drawcall S as well as the attributes of Primitive A are added to Tile 2's inputs.

Besides scene constants, primitives have other global associated data that affects the color of a fragment: the shader program and the textures to be used within. RE

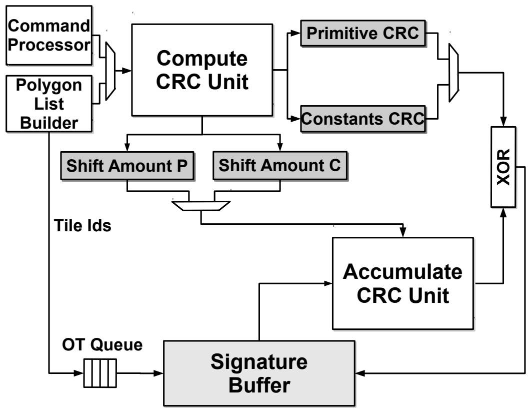


Figure 7. Signature Unit block diagram.

does not include these in the tile signature, since changes to such global data are not common. In our benchmarks, we have observed that shaders and textures remain constant for thousands of frames. Moreover, loading new shaders and textures is done through API calls (such as *glShaderSource* and *glTexImage2D*, for instance) and, therefore, are registered by the driver. Whenever such infrequent API calls occur, Rendering Elimination is disabled for the current frame. Besides this, RE could also be disabled during one frame periodically to guarantee Frame Buffer refreshing. RE should also be temporarily disabled by the driver for scenes that use multiple render targets: RE is specifically targeted to an important segment of less sophisticated applications that cover a large fraction of the mobile market.

F. Signature Unit Architecture

The message that has to be signed for a tile consists of a sequence of blocks, containing either scene constant data or vertex attribute data. The number of blocks of a message is not known until all the geometry of the frame is processed and, therefore, RE uses the incremental signature computation described in Algorithm 1.

The Signature Unit (SU), the piece of logic responsible for the incremental computation of the CRCs of tiles, is shown in Figure 7. Whenever the SU receives a new data block, it computes its CRC and updates the CRC of all the tiles overlapped by the primitive associated to that block.

Let us consider first the case of vertex attributes, which are blocks sent to the SU by the Polygon List Builder. The SU computes the signature of all the vertex attributes of a primitive using the *Compute CRC* unit, and the resulting CRC32 (CRC_{A_i} as described by Algorithm 1) is stored in the *Primitive CRC* register. Since the number of attributes in a primitive is variable, the Compute CRC unit stores the length of the signed block (b in Algorithm 1) in the *Shift Amount P* register. While the SU computes the CRC of a primitive, the Polygon List Builder inserts into the *OT Queue* a list of identifiers of the tiles overlapped by the primitive.

After computing the signature of a primitive, the SU traverses the list of overlapped tiles and updates each tile signature by combining it with the primitive signature. It pops in sequence each entry from the head of the OT Queue and uses this tile *id* to read the corresponding CRC from the Signature Buffer, which is then sent to the *Accumulate CRC* unit. This unit receives as inputs the previous CRC for a tile and the length of the primitive message signed by the Compute Unit. The Accumulate CRC unit computes the CRC of the message that results by left-shifting the previous CRC as many bits as the received length. This CRC corresponds to $CRC_{Temporary}$ in Algorithm 1. Finally, the results of the Compute and Accumulate units are bitwise xored to obtain the new CRC for the tile (CRC_A in Algorithm 1) and the new signature is written back to the Signature Buffer.

The SU can also receive data blocks from the Command Processor, which correspond to scene constants. The signature computation of the constants of a drawcall is done in the same form as the signature computation of the vertex attributes of a primitive: the Compute Unit generates a CRC32 and the length of the signed message and stores them in two registers: *Constants CRC* and *Shift Amount C*, respectively. In order to combine the signature of the constants with the signature of the attributes, several issues need to be addressed. First, every drawcall may define its own set of constants which only affect to that drawcall. Consequently, the Constant CRC register only has to be combined with the CRC of the tiles affected by that drawcall. Besides this, even though multiple primitives of the same drawcall may overlap the same tile, the Constant CRC should be considered only once per tile.

Rendering Elimination uses a bitmap to solve these issues. The bitmap has a length equal to the number of tiles that the Frame Buffer is divided into. If a position of the bitmap is set, it means that the Constant CRC has already been combined into the signature for that tile. Whenever the GPU receives a new set of constants after having processed one or more drawcalls, the bitmap is cleared and the constants are signed and stored in the Constant CRC register. For all the following primitives, for every tile identifier popped from the OT Queue, the bitmap is queried to check whether that tile has already combined the signature of the constants into its signature. If so, the previous CRC of the tile is only updated with the value stored in the Primitive CRC register. Otherwise, the bit in the bitmap position corresponding to that tile is set and the previous CRC of the tile is updated twice: first with the contents of the Constants CRC, and second with the Primitive CRC, by making the Accumulate CRC unit to select the appropriate shift amount in each step.

G. Compute CRC and Accumulate CRC Unit Architectures

The **Compute CRC unit** implements the first two steps in the loop of Algorithm 1, computing the CRC of a block consisting of a primitive or a set of constants and determining the length of the block. Since the length of such blocks is not fixed, the Compute CRC unit is architected to incrementally compute the CRC32 of a block by breaking it into subblocks of fixed length (64 bits) and recursively applying Alg. 1. The resulting procedure is detailed in Alg. 2. Namely, the Compute CRC unit has a similar internal structure as the SU, as shown in Figure 8. It consists of two subunits and the CRC_{Out} register (initialized to zero). The *Sign* subunit computes the CRC32 of a fixed-length subblock and stores it into the CRC_{Out} register after a bitwise XOR with the result of the *Shift* subunit. In parallel, the Shift subunit computes the CRC32 of the message resulting by left-shifting 64 bits the contents of the CRC_{Out} register. This process is repeated for each 64-bit subblock in the input data block received by the Compute CRC unit. The control logic of the Compute CRC unit counts the number of signed subblocks and communicates it to the Accumulate CRC unit using registers Shift Amount P (for Primitives) and Shift Amount C (for Constants), shown in Figure 7.

Algorithm 2 Compute CRC Unit, Incremental Computation

```
$CRC_{Out} = 0$
$ShiftAmount = 0$
for 64-bit subblock $A_i$ in submessage $A$ do
    $CRC_{A_i} = ComputeCRC(A_i)$
    $CRC_{Temporary} = ComputeCRC(CRC_{Out} << 64)$
    $CRC_{Out} = CRC_{A_i} \oplus CRC_{Temporary}$
    $ShiftAmount = ShiftAmount + 1$
end for
```

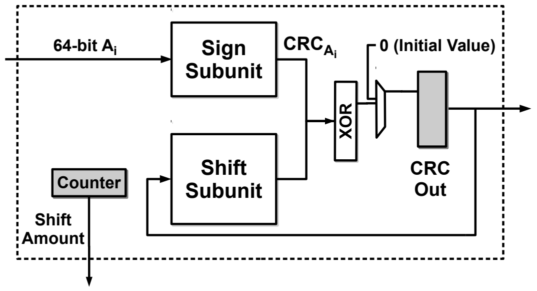


Figure 8. Compute CRC Unit block diagram.

Algorithm 3 Accumulate CRC Unit, Incremental Computation

```
$CRC_{Accum} = SignatureBuffer[tile]$
for $k \leftarrow 1$ to $ShiftAmount$ do
    $CRC_{Accum} = ComputeCRC(CRC_{Accum} << 64)$
end for
```

The **Accumulate CRC unit** implements the third step in the loop of Algorithm 1, that computes the CRC of a message consisting of the partial CRC of a tile (stored in the Signature Buffer) left-shifted by as many zeros as the length of the block to accumulate (the one fed to the Compute CRC unit). Since the length of this block is variable, it is also variable the amount to shift, hence the length of the

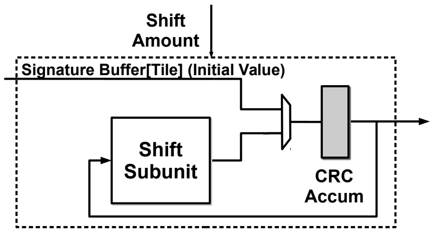


Figure 9. Accumulate CRC Unit block diagram.

resultant message to be signed by the Accumulate CRC unit. Therefore, this unit follows an incremental procedure to compute the CRC, as detailed in Algorithm 3. Note that, while the Accumulate CRC unit follows the same incremental approach as the Compute CRC unit, the accumulated blocks are always zero (they come from a left shift). Therefore, each iteration only requires to shift and re-sign the CRC32 computed on the preceding iteration and, consequently, the Accumulate CRC unit only consists of a Shift subunit, as shown in Figure 9.

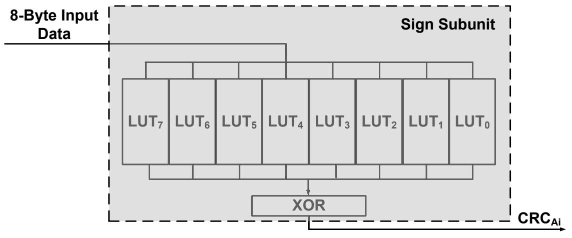


Figure 10. Architecture of the Sign subunit.

Figure 10 shows the **Sign subunit** architecture, which computes the CRC32 of a 64-bit subblock using the table-based approach described in Section III-D. Each byte in the subblock is independently processed by accessing a specific LUT. The output of the Shift subunit is the bitwise XOR of the results of the 8 LUTs.

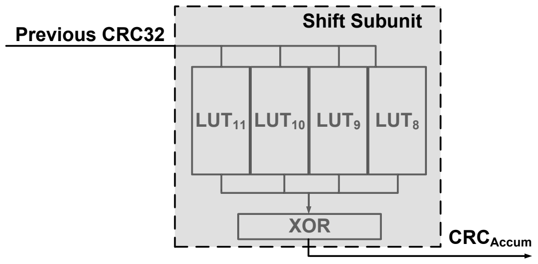


Figure 11. Architecture of the Shift subunit.

Figure 11 shows the **Shift subunit** architecture, which computes the CRC32 of the 64-bit message that results from a 32-bit input block shifted with 32 zeros. The design is analogous to the Sign subunit, and uses the table-based approach described in Section III-D.

The choice of the subblock size for the Compute CRC unit is determined by several tradeoffs: the length of a submessage has to be multiple of the length of the whole message, but very small submessages imply a larger number of cycles to compute the signature. Conversely, long submessages require more LUT storage, which causes energy and area overheads.

Experimentally, we have determined that subblocks of size 8 bytes signed with eight 1-KB LUTs incur in small time and energy overheads, as shown in Section V. The average command that updates constants modifies 16 values. A subblock of length 8 bytes corresponds to 2 of those values and, therefore, computing the signature for the average constant input data requires 8 cycles. Regarding primitives, the size of the data of an attribute is 48 bytes, which correspond to 3 vertices defined by four 4-byte components each. The average number of attributes per primitive is 3 and, thus, computing the signature for the average primitive requires 18 cycles.

IV. Evaluation Methodology

In this section we briefly describe the simulation infrastructure and the set of benchmarks employed in the experiments to evaluate Rendering Elimination and Transaction Elimination (TE) techniques. The implementation of TE is also presented in this section.

A. *GPU Simulation Framework*

In order to evaluate our proposal and TE we employ Teapot [23]. Teapot is a GPU simulation framework that allows to run unmodified Android applications and evaluate performance and energy consumption of the GPU. Table I shows the parameters employed in our simulations in order to model an architecture resembling an ARM Mali-450 GPU [19]. The Mali 400 MP series is the most deployed Mali GPU with around 19% of the mobile GPU market [24].

Teapot is comprised of three main components: an OpenGL trace generator, a GPU functional simulator and a GPU cycle-accurate simulator. The workloads are executed in the Android Emulator deployed in the Android Studio [25]. While the application is running, the OpenGL trace generator intercepts and stores all the OpenGL commands that the Android Emulator sends to the GPU driver. The OpenGL commands trace that is generated is later fed to an instrumented version of *Softpipe*. *Softpipe* is a software renderer included in Gallium3D, a well-known architecture for building 3D graphics drivers. Our instrumented *Softpipe* executes the OpenGL commands and creates a GPU trace including information of the different stages of the graphics pipeline (memory accesses, shader instructions, vertices, fragments, etc). The GPU trace is used by the cycle-accurate simulator, which gathers activity factors of all the components included in the modeled TBR architecture and reports timing as well as power consumption. Regarding the power model, McPAT [26] provides energy estimations for the processors and the caches included in the GPU. We have extended McPAT using its components (SRAM, registers, XORs and MUXes, among others) to describe all the additional structures present in the architecture presented in Section III: the Signature Buffer, the CRC LUTs, the OT Queue and the constant bitmap, as well as all necessary registers and combinational logic. The main memory and the memory controller are simulated with DRAMSim2 [27].

Table I
GPU SIMULATION PARAMETERS.

Baseline GPU Parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1196x768
Tile Size	16x16 pixels
Main Memory	
Latency	50-100 cycles
Bandwidth	4 bytes/cycle (dual channel LPDDR3)
Size	1 GB
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle, Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 2-way, 4 KB, 1 bank, 1 cycle
Texture Caches (4x)	64 bytes/line, 2-way, 8 KB, 1 bank, 1 cycle
Tile Cache	64 bytes/line, 8-way, 128 KB, 8 banks, 1 cycle
L2 Cache	64 bytes/line, 8-way, 256 KB, 8 banks, 2 cycles
Color Buffer	64 bytes/line, 1-way, 1 KB, 1 bank, 1 cycle
Depth Buffer	64 bytes/line, 1-way, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	16 attributes/cycle
Early Z test	32 in-flight quad-fragments, 1 Depth Buffer
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors

Table II
BENCHMARK SUITE.

Benchmark	Alias	Genre	Type
Angry Birds	abi	Arcade	2D
Candy Crush Saga	ccs	Puzzle	2D
Castle Defense	cde	Tower Defense	2D
Clash of Clans	coc	MMO Strategy	3D
Crazy Snowboard	csn	Arcade	3D
Cut the Rope	ctr	Puzzle	2D
Hopeless	hop	Survival Horror	2D
Modern Strike	mst	First Person Shooter	3D
Temple Run	ter	Platform	3D
Tigerball	tib	Physics Puzzle	3D

B. Benchmark Suite

Table II shows the set of benchmarks analyzed to evaluate our technique, which consists of ten commercial Android graphics applications. Our set of benchmarks includes both 2D and 3D games, applications that stress the GPU further than other commonly used applications in battery-operated devices. Among the 3D games we include workloads with simple 3D models such as *tib*, and workloads with more sophisticated 3D models and scenes such as *mst* and *ter*. The workloads included in our set of benchmarks are representative of the current landscape of smartphone games ecosystem as it includes popular Android games. These applications have millions of downloads according to Google Play [28], some of them surpassing 500 million downloads.

C. Transaction Elimination

Transaction Elimination (TE) [16] is a technique that reduces main memory bandwidth by avoiding the flush of the Color Buffer in tiles that have the same color as in the preceding frame. Since the reuse distance of two tiles is an entire frame, tile equality is not performed by comparing the colors of all the pixels of a tile but rather signatures of those colors. Whenever a tile has finished being rendered, its colors (the contents in the Color Buffer) are hashed into a signature and compared to the signature of the same tile for the previous frame. If the two signatures are equal, the newly generated colors are not written into the Frame Buffer. Although the exact details of this technique in commercial systems are not fully disclosed, we have modified our cycle-accurate simulator to model an efficient implementation and compare it with our proposed approach. Figure 12 presents the extra hardware added in the pipeline to perform Transaction Elimination.

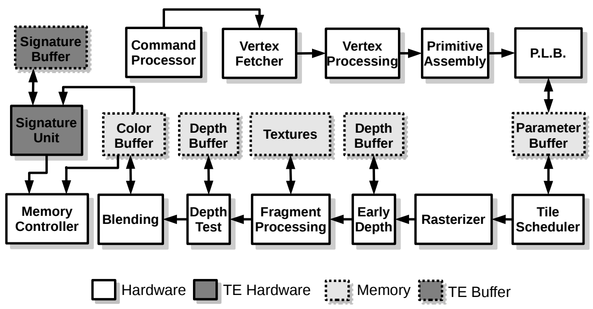
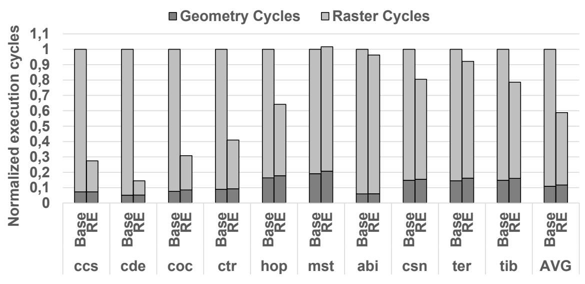


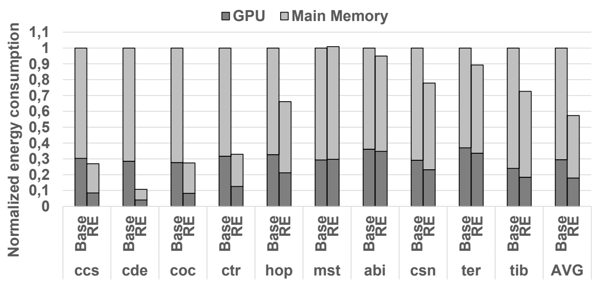
Figure 12. Graphics Pipeline including TE.

In our TE evaluation, we consider the energy overheads caused by the Signature Buffer and the Compute CRC unit, but do not add any execution time overhead: while we count the number of accesses to report energy, we ideally assume that the signature for a Color Buffer does not require any execution cycles.

For both the evaluation of Rendering Elimination and Transaction Elimination, we consider the common case in current GPUs in which the memory system has not only one but two Frame Buffers. This allows the display to read from one (called Front Buffer) while the GPU processes the following frame by writing into a different memory region (Back Buffer) without causing visual artifacts. The Front and Back buffers are periodically swapped so that the display presents new frames at the appropriate frame rate. With this approach, tiles have to be compared not with the frame being displayed but with one prior, since the potential transactions to eliminate occur between the GPU and the Back Buffer. The Signature Buffer, therefore, contains signatures spanning two frames: the set generated when the GPU processes a frame and writes into the Back

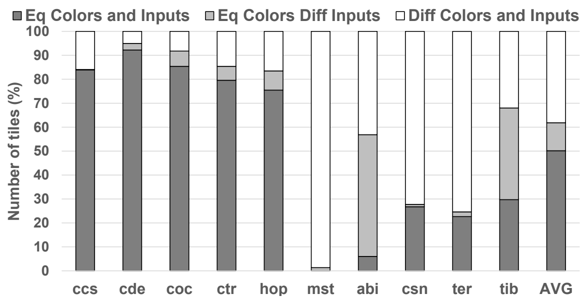


(a)

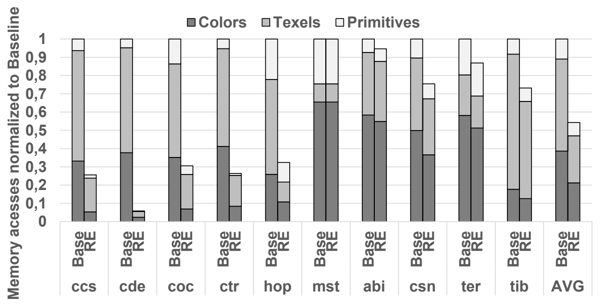


(b)

Figure 13. RE compared against Baseline GPU: (a) Execution cycles. (b) Energy consumption.



(a)



(b)

Figure 14. Sources of execution time and energy reduction (a) Tiles with equal color and inputs, equal color and different inputs, and different color and inputs across neighboring frames. (b) RE memory bandwidth compared to baseline: Parameter Buffer and Texel reads and Color Buffer flushes.

Buffer and the set for the Front Buffer, that will be used to compare tiles when the buffers are swapped.

V. Experimental Results

In this Section we present the main results of RE over the baseline. For comparison purposes, we also evaluate Fragment Memoization [17] and TE [16].

Figure 13a shows execution cycles of RE for our set of benchmarks. The total cycles are normalized to those of the Baseline and divided into cycles corresponding to Geometry and Raster Pipelines. RE achieves an average execution time reduction of 42% (1.74x speedup), yielding reductions of up to 86% (*cde*). The execution of the Raster Pipeline using RE is 2x faster than the Baseline GPU on average, with maximums of more than 10x. On the other hand, the overheads introduced by the technique are almost negligible, since the signature computation is overlapped by previous Geometry Pipeline stages. The pipeline is only stalled when computing signatures for primitives that cover a large amount of tiles, resulting in an overflow of the Overlapped Tiles Queue. These kind of primitives are rare, as can be seen by the fact that, on average, only a 0.64% additional geometry cycles are introduced. The overhead of comparing the signatures is even smaller. Considering that accessing the corresponding Signature Buffer entry and performing a simple comparison takes a few cycles while skipping the entire Raster Pipeline can save thousands, these tiny overheads are more than offset by the large performance gains. Such overheads only result in performance loss in benchmarks that lack redundant tiles and cannot leverage RE at all. Even in those cases, the performance impact is smaller than 1%, as it can be seen in *mst*.

Figure 13b shows the GPU energy consumption (considering both static and dynamic) when using RE for our set of benchmarks, normalized to the baseline. The total energy is split into two parts: energy spent by the GPU in accessing main memory and energy spent in other activities. As shown, RE brings about an average 43% reduction of the energy consumed by the system, with a 38% reduction of the energy consumed by the GPU and 48% reduction of the energy consumed by main memory. Moreover, RE provides enormous energy savings for benchmarks such as *ccs* or *cde*, reducing 90% of the overall energy consumed by the baseline. In *mst*, a benchmark that does not take advantage of RE, the energy overheads are smaller than 1%. Regarding area, McPAT reports that the cost of the hardware added (CRC LUTs, Signature Buffer, Overlapped Tiles Queue and bitmap) incurs in less than 1% area overhead. These reductions in execution time and energy consumption are due to an important number of tiles bypassing the execution of the Raster Pipeline and avoiding their corresponding main memory accesses. Figure 14a shows the average percentage of tiles that, across neighboring frames, produce the same color (the sum of bottom and mid bars) and the average percentage of tiles that change colors (top bar). The bottom bar depicts the percentage of tiles that Rendering Elimination avoids rendering, which is, on average 50% of the tiles of a frame and 81% of the total redundant tiles. The mid bar shows the percentage of tiles that despite having different inputs end up with the same color (12%). The top bar presents the percentage of tiles with different inputs and different colors (38%). Note that there is not a single occurrence of a tile that changes the color while maintaining the same inputs. Furthermore, Figure 14a reveals three different behaviors for the benchmarks analyzed depending on camera movements. The first category, (*ccs* to *hop*) is composed of workloads with mainly static cameras, so their scenes contain lots of redundant tiles. The second category (*mst*) is composed of workloads with highly dynamic camera movements and almost no redundant tiles. The third category (*abi* to *tib*) behaves like the first set in some phases and like the second set in others. It can be seen that there is a strong correlation between the number of detected redundant tiles presented on Figure 14a and the speedup and energy savings reported in Figure 13.

Eliminating redundant tiles not only reduces the activity of the GPU but it also eliminates all the associated memory accesses. Figure 14b plots the amount of main memory traffic generated by the Raster Pipeline, normalized to the baseline. The total traffic is split into three parts: accesses generated by the Tile Cache when reading primitives from the Parameter Buffer, accesses generated by the Texture Cache when fetching textures in the fragment shaders and accesses generated by flushing the on-chip Color Buffer to the Frame Buffer. As it is shown, RE achieves a significant drop in traffic to main memory (48% on average).

A. RE vs Fragment Memoization and TE

Figure 15 compares the number of fragments shaded by RE to those shaded by the technique proposed by Arnau et al. [17], which performs fragment memoization but requires rendering multiple frames in parallel. Note also that our approach is able to skip more pipeline stages and their corresponding main memory accesses (see Figure 3). We run an experiment to compare the amount of reused fragments by each technique. We modelled Fragment Memoization as originally proposed, with 2-frames in parallel and a 32-bit hash that discards the screen coordinates, but we augmented their default 512-entry 4-way LUT to 2048 entries to better compare to the chip area of RE. As shown, RE reuses much more fragments in the majority of benchmarks. One would expect that, by working at a fragment granularity, memoization could discover more redundancy than working at a tile level. However such granularity also requires a bigger storage and, as already pointed out in their paper, a realistic space-limited LUT only captures on average 60% of that potential, whereas RE captures all of the redundant tiles with equal inputs. The only notable exception is *hop*: As a significant portion of the screen is black, the pressure on the LUT storage is heavily reduced by being able to render the scene with a small number of repeated fragments, but this is a rather rare case. Moreover, because of the large reuse distance between redundant fragments, Fragment Memoization requires significant modifications in the pipeline to enable rendering of multiple frames in parallel. While executing two frames in parallel has benefits beyond memoization, it has two major drawbacks that RE does not. First, it implies a significant re-design of the whole GPU. Second, it generates input response lag because of the parallel frame rendering process. To alleviate this side effect it must be disabled during frames where the user introduce inputs.

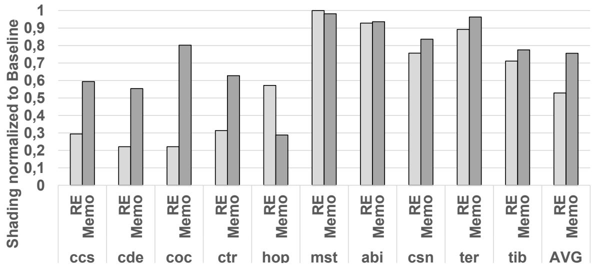
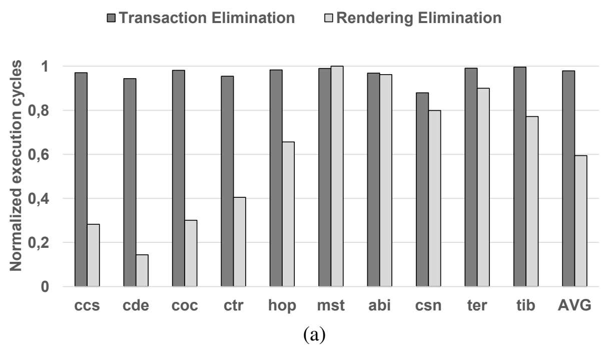


Figure 15. Fragments shaded with RE and PFR-aided Memoization normalized to baseline.

Figure 16 compares the benefits of Rendering Elimination over Transaction Elimination (see implementation details in Section IV-C). Transaction Elimination (TE) avoids only the Color Buffer flushes to main memory, while RE bypasses the whole Raster Pipeline execution for redundant tiles. Therefore, while TE reduces a 9% the energy consumption with respect to the baseline GPU, RE outperforms it and achieves a reduction of 43%. Note that in benchmarks with a large percentage of redundant tiles such as *cde*, RE achieves an additional 65% energy savings compared with TE. Moreover, since the flush of the Color Buffer represents a relatively small portion of the total time of the Raster Pipeline, RE far surpasses the performance benefits of TE.

In some cases, TE may obtain energy savings for benchmarks in which RE cannot. As Figure 14a presents, there is a subset of tiles whose rendering outputs the same color as in the preceding frame but do not have the same inputs as in the preceding frame (depicted in the mid bar). On average, this occurs for 12% of the tiles. This phenomenon may occur, for instance, when the only differences between the two tiles happen on occluded fragments that are eventually culled by the z-test and do not contribute to the final color of the tile, or for scenes with quick camera panning movements where most of the background texture contains a single plain color. Consequently, in benchmarks where RE detects a small percentage of equal tiles, such as *abi*, TE may obtain a slightly better energy savings than RE.

We refer to the above event, where the signature of two tile inputs does not match but the final color of their pixels remains unchanged, as *false negatives*. False negatives do not generate errors, but reveal a broader potential for tile reuse that RE is not capable to detect. On the other hand, since tile inputs are compared using the result of a hash function,



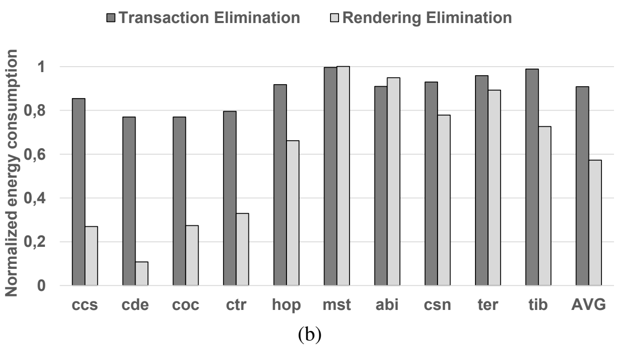


Figure 16. Comparison of RE and TE against Baseline. (a) Cycles. (b) Energy consumption.

there exists the possibility of collisions or *false positives*: pairs of different tile inputs that are mapped to the same signature. A false positive means that the GPU incorrectly reuses a tile that has actually changed in the current frame. However, the probability of such an event with a CRC32 signature is roughly one every 4 billion tiles, i.e., less than one tile per million frames (more than 4 hours playing). Moreover, it would be extremely difficult, or impossible, to spot the incorrect tile by a human, since it would last for only a single frame (less than 20 ms), and it would probably appear very similar to the correct tile due to frame coherency. Actually, we found zero false positives in our experiments with CRC32.

VI. Related Work

Hardware memoization has been widely researched to accelerate general-purpose computing by detecting blocks of instructions that repeatedly produce the same value and caching them in LUTs [29], [30], [31], [32].

Several works exploit frame coherence in order to avoid the processing of redundant fragments. Ragan-Kelley et al. [33] decouple shading from visibility and employ a hardware memoization scheme that caches shading results. Arnau et al. [17] also propose a hardware memoization scheme to reduce redundant fragment shading that is implemented on top of a PFR [18] pipeline to improve reuse distance. A comparison with this technique is provided in Section V-A. RE is similar to memoization in that it remembers the signatures of previous inputs to detect redundancy. However, since RE works at a coarser granularity, instead of caching just a fraction of these signatures, it stores all of them. Furthermore, the outputs do not need to be cached, since they are already present in the Frame Buffer.

Transaction Elimination (TE) [16] is a bandwidth saving feature included in the ARM Mali GPU that detects identical tiles between the current frame being rendered and the previous one. TE computes a CRC signature per tile. If a tile of the current frame has the same CRC as in the preceding frame, its results are not flushed to main memory, which produces significant energy savings. On the other hand, RE not only avoids the flush of redundant tiles to main memory, but also the execution of the entire Raster Pipeline.
Some works aim to reduce fragment shading by means of reducing the number of occluded fragments whose color is computed. Occlussion queries [34], [35] rasterize and test the visibility of Bounding Volumes of the objects to cull the geometry at draw command level granularity. However, the queries need to be sorted in a front-to-back order to perform well, which sets an important limitation. Other works aim to avoid fragment shading for hidden surfaces at fragment level granularity [36], [5]. These methods propose to perform a hidden surface removal phase where geometry is rasterized and depth tested in order to identify the visible geometry that will be later fragment shaded. Unlike these works, RE does not need to perform extra rendering passes to reduce overshading. Furthermore, if a tile is eliminated, both visible and occluded surfaces will not be processed.

VII. Conclusions

In this paper we have presented Rendering Elimination (RE), a novel micro-architectural technique for Tile-Based Rendering GPUs that effectively reduces shading computations and memory accesses by means of culling redundant tiles across consecutive frames. Since RE detects a redundant tile before it is dispatched to the Raster Pipeline, the entire computation (which includes rasterization, depth test, fragment processing, blending, etc.) is avoided, as well as all the associated energy-consuming memory accesses to the Parameter Buffer, Textures and Frame Buffer.

Our results show that RE outperforms state-of-the-art techniques such as Transaction Elimination or Fragment Memoization, which are only able to bypass a single pipeline stage. Compared to the baseline GPU, RE achieves an average speedup of 1.74x and reduces the GPU and main memory energy consumption by 38% and 48%, respectively. The hardware overhead of RE is minimum, requiring less than 1% of the total area of the GPU, while its latency is hidden by other processes of the graphics pipeline. In terms of energy, RE incurs a negligible overhead of less than 0.5% of the total GPU energy. RE is especially efficient in benchmarks with small camera movements, with speedups as high as 6.9x and energy savings up to 90%. Even in benchmarks without any significant amount of redundant tiles, the performance impact is well smaller than 1%.

VIII. Acknowledgements

This work has been partially supportedby the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and by the Generalitat de Catalunya under the AGAUR-FI program. The authors would like to thank Tomas Akenine-Möller, Juan Fernández and Javier Carretero-Casado for their inputs in the genesis of the main idea of this paper.

References

[1] (accessed December 10, 2018) Global games market report. https://newzoo.com/solutions/revenues-projections/global-games-market-report.

[2] (accessed December 10, 2018) Essential facts about the computer and video game industry. http://www.theesa.com/wp-content/uploads/2014/10/ESA_EF_2014.pdf.

[3] (accessed December 10, 2018) Mobile game statistics. https://medium.com/@sm_app_intel/new-mobile-game-statistics-every-game-publisher-should-know-in-2016-f1f8eef64f66.

[4] (accessed December 10, 2018) Who plays mobile games. http://services.google.com/fh/files/blogs/who_plays_mobile_games.pdf.

[5] P. Clarberg, R. Toth, and J. Munkberg, "A sort-based deferred shading architecture for decoupled sampling," *ACM Trans. on Graphics*, vol. 32, no. 4, p. 141, 2013.

[6] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *IEEE/ACM Intl. Conf. on Cyber, Physical and Social Computing*. IEEE, 2010, pp. 344–350.

[7] E. de Lucas, P. Marcuello, J.-M. Parcerisa, and A. González, "Ultra-low power render-based collision detection for cpu/gpu systems," in *Proc. of the 48th Intl. Symp. on Microarchitecture*. ACM, 2015, pp. 445–456.

[8] S.-L. Chu, C.-C. Hsiao, and C.-C. Hsieh, "An energy-efficient unified register file for mobile gpus," in *IFIP 9th Intl. Conf. on Embedded and Ubiquitous Computing*. IEEE, 2011, pp. 166–173.

[9] B.-G. Nam, J. Lee, K. Kim, S. J. Lee, and H.-J. Yoo, "A low-power handheld gpu using logarithmic arithmetic and triple dvfs power domains," in *Proc. of the 22nd Symp. on Graphics Hardware*, vol. 4, no. 05, 2007, pp. 73–80.

[10] (accessed December 10, 2018) Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler.

[11] (accessed December 10, 2018) Antutu benchmark. https://antutu.com.

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

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

[14] J. Nystad *et al.*, "Adaptive scalable texture compression," in *Proc. of the Fourth ACM SIGGRAPH/Eurographics Conf. on High-Performance Graphics*, 2012, pp. 105–114.

[15] H. Hubschman *et al.*, "Frame-to-frame coherence and the hidden surface computation: constraints for a convex world," *ACM Trans. on Graphics*, vol. 1, no. 2, pp. 129–162, 1982.

[16] (accessed December 10, 2018) Transaction elimination. https://developer.arm.com/technologies/graphics-technologies/transaction-elimination.

[17] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ACM/IEEE 41st Intl. Symp. on Computer Architecture*. IEEE, 2014, pp. 529–540.

[18] J.-M. Arnau, J.-M. Parcerisa, and X. P, "Parallel frame rendering: Trading responsiveness for energy on a mobile gpu," in *Proc, of the 22nd Intl. Conf. on Parallel Architectures and Compilation Techniques*. IEEE, 2013, pp. 83–92.

[19] (accessed December 10, 2018) Arm mali-450 gpu. https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu.

[20] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proc. of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[21] Y. Sun *et al.*, "High performance table-based algorithm for pipelined crc calculation," *communications*, vol. 4, p. 5, 2013.

[22] J. Massey, "Shift-register synthesis and bch decoding," *Trans. on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[23] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proc. of the 27th Intl. ACM Conf. on Supercomputing*. ACM, 2013, pp. 37–46.

[24] (accessed December 10, 2018) Hardware gpu market. http://hwstats.unity3d.com/mobile/gpu.html.

[25] (accessed December 10, 2018) Android studio. https://developer.android.com/studio/index.html.

[26] S. Li *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proc. of the 42nd Intl. Symp. on Microarchitecture*, 2009, pp. 469–480.

[27] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[28] (accessed December 10, 2018) Google play. https://play.google.com.

[29] M. H. Lipasti *et al.*, "Value locality and load value prediction," *SIGPLAN Notices*, vol. 31, no. 9, pp. 138–147, 1996.

[30] A. Sodani and G. S. Sohi, *Dynamic instruction reuse*. ACM, 1997, vol. 25, no. 2.

[31] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating multimedia processing by implementing memoing in multiplication and division units," in *ACM SIGPLAN Notices*, vol. 33, no. 11. ACM, 1998, pp. 252–261.

[32] J. Huang and D. J. Lilja, "Exploiting basic block value locality with block reuse," in *Intl. Symp. On High-Performance Computer Architecture*. IEEE, 1999, pp. 106–114.

[33] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, "Decoupled sampling for graphics pipelines," *ACM Trans. on Graphics*, vol. 30, no. 3, p. 17, 2011.

[34] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," *Computer Graphics Forum*, 2004.

[35] D. Sekulic, "Efficient occlusion culling," in *GPU Gems*, Nvidia, Ed., 2004, pp. 487–503.

[36] E. Haines and S. Worley, "Fast, low memory z-buffering when performing medium-quality rendering," *Journal of Graphics Tools*, vol. 1, no. 3, pp. 1–5, 1996.