DTexL: Decoupled Raster Pipeline for Texture Locality

Diya Joseph* , Juan L. Aragón† , Joan-Manuel Parcerisa* and Antonio González*
* *Universitat Politècnica de Catalunya, Barcelona, Spain*
† *Universidad de Murcia, Murcia, Spain*

***Abstract*—Contemporary GPU architectures have multiple shader cores and a scheduler that distributes work (threads) among them, focusing on load balancing. These load balancing techniques favor thread distributions that are detrimental to texture memory locality for graphics applications in the L1 Texture Caches. Texture memory accesses make up the majority of the traffic to the memory hierarchy in typical low power graphics architectures. This paper focuses on improving the L1 Texture cache locality by focusing on a new workload scheduler by exploring various methods to group the threads, assign the groups to shader cores and also to reorder threads without violating the correctness of the pipeline. To overcome the resulting load imbalance, we also propose a minor modification in the GPU architecture that helps translate the improvement in cache locality to an improvement in the GPU's performance. We propose DTexL that envelops these ideas and evaluate it over a benchmark suite of ten commercial games, to obtain a 46.8% decrease in L2 Accesses, a 19.3% increase in performance and a 6.3% decrease in total GPU energy. All this with a negligible overhead.**

***Keywords*-GPU; Caches; Graphics; Scheduling; Texture Locality; Low-power;**

I. Introduction

Graphics applications go through the different stages of the GPU graphics pipeline, being the stage in the shader cores the one that represents the vast majority of execution time and energy consumption. In this stage, the color of each pixel is computed, which requires a significant number of memory accesses, mainly to data structures that define the texture of the different objects, and computations to apply lighting and other visual effects. Graphics workloads are highly parallel since the computations for each pixel are independent, so the application is typically decomposed into a huge number of threads. The shader cores (aka shaders) are highly multithreaded to increase throughput and hide memory latency.

Shader cores (SCs) executing graphics workloads access texture data through L1 Texture Caches to reduce traffic to main memory. Memory latency is less of a problem with graphics workloads because it can be tolerated in current GPUs through multithreading, which is easily exploited by leveraging these workloads' massive inherent parallelism. Performance in that context mainly depends on throughput, i.e. on the ability of multithreading to keep a high occupancy of the shader cores, and is highly correlated to the number of threads simultaneously processing.

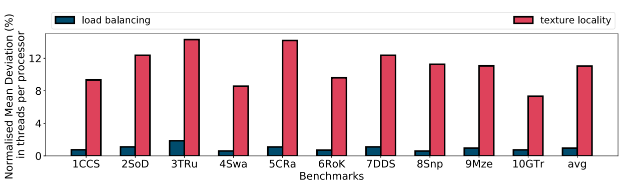


Figure 1: Normalized Mean deviation of number of threads per processor for two different schedulers: one that focuses on Load Balancing versus another that improves Texture Cache locality.

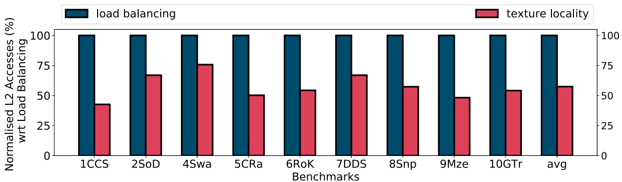


Figure 2: Normalized L2 accesses for two different schedulers: one that focuses on Load Balancing versus another that improves Texture Cache locality.

Tile-Based Rendering (TBR) architectures, the predominant architecture for mobile GPUs, process and render disjoint subsets of the screen called *tiles*. Conventional TBR architectures exclusively process one tile at a time within an SC. This can lead to periods of lower thread occupancy in the SCs if the workload is not evenly distributed among the different SCs, thus reducing the multithreading efficiency and increasing the impact of Texture Cache misses in the core throughput.

Thread schedulers for SCs in contemporary GPUs focus on workload balancing such that SCs are assigned a similar number of threads for each tile. Interestingly, we found that load balancing schedulers lead to memory block replication in the private L1 Texture Caches and thus decrease their aggregated capacity. We explore various thread schedulers that focus on decreasing the memory block replication but we find that they produce a higher workload imbalance in the SCs. Figures 1 and 2 compare two workload schedulers, one focusing on Load Balancing and the other one on Texture Memory Locality. Figure 1 plots the average of the normalized mean deviations in the number of threads assigned to each SC in each tile. On the other hand, Figure 2

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00028

plots the L2 Accesses produced with the scheduler that focuses on texture cache locality, normalized to those of the one focused on load balancing. These results clearly show that where one scheduler wins in load balancing, the other wins in texture cache locality. This provided us with a motivation to explore thread schedulers that focus on both goals, hoping that the combination of both achievements will increase performance and reduce energy.

In TBR GPU architectures, tiles can be processed in any order. For each tile, a number of threads are generated, one for each quad of the scene (a quad is a group of four adjacent fragments, which corresponds to four adjacent pixels). These threads can be assigned to the different SCs following different policies. Overall, by changing the order in which tiles are processed and the assignment of threads to SCs, the memory access patterns can be significantly changed, which can have a great effect on the locality of the memory references.

The Texture Memory Hierarchy of TBR GPUs has a private L1 texture cache associated with each SC. The private L1 Texture Caches are backed by a shared L2 cache which is then backed by main memory. In this paper, we influence the texture memory access pattern by scheduling threads to SCs to favor texture locality over load balancing. We do this in two ways. One by manipulating the distribution of threads among the SCs and hence manipulating the memory accesses among the private L1 texture caches in a way that minimizes cache block replication and increases the aggregated capacity of the Texture Caches. And the second by reordering the memory access pattern (on a tile granularity) in time by controlling the order in which tiles are processed by the graphics pipeline. We also tweak the Raster Pipeline in the TBR architecture to overcome the load imbalance and translate the caching improvements into a cumulative benefit on the GPU's performance and energy.

To summarize, in this paper we propose to improve texture caching in shader cores by means of a novel thread scheduling and a reordering of the memory access pattern to the Texture Memory Hierarchy. In order to translate this improvement into GPU performance, we propose a minor modification in the baseline Graphics Pipeline. Together we call this proposal Decoupled architecture for Texture Locality (DTexL). DTexL achieves an average of 19.3% increase in performance (frames per second) and 6.3% decrease in total GPU energy evaluated over a set of ten commercial graphics applications.

To summarize, this paper makes the following key contributions:

- Proposes and evaluates different workload schedulers for the SCs to improve Texture Caching for graphics workloads in mobile GPUs.
- Proposes and evaluates two tile orders in conjunction with the workload schedulers to improve Texture Caching for graphics workloads in mobile GPUs.
- Provides a decrease of 46.8% in total L2 accesses with the proposed workload scheduler.
- Proposes a minor modification in the graphics pipeline in order to translate the caching improvement into a 19.3% increase in FPS and 6.3% decrease in total GPU energy.

The rest of this paper is organized as follows. Section 2 presents some background on GPUs, with special emphasis on workload scheduling and tile pipelining in TBR GPUs. Section 3 describes DTexL. We describe the tools and workloads used to evaluate our technique in Section 4. In Section 5, we present our experimental results and analysis. In Section 6, we review some related work and Section 7 summarizes the main conclusions of the paper.

II. Background

Mobile GPUs typically implement a Tile-Based Rendering (TBR) architecture. The idea for TBR architectures was initially proposed to facilitate parallel rendering [10], [30]. *Tiles* are disjoint segments of the frame that can be rendered in parallel. TBR is now a common architecture adapted for low-power graphics systems where instead of tiles being rendered in parallel, they are rendered sequentially over small tile-sized on-chip buffers, which allow to exploit locality and significantly reduce power-hungry DRAM accesses and save memory bandwidth. According to a work by Antochi et al. [1], a TBR architecture reduces the total amount of external data traffic by a factor of 1.96 compared to a GPU architecture that is not tile-based (a.k.a. Immediate Mode Rendering).

A. Graphics Pipeline

Figure 3 shows the main stages of the Graphics Pipeline and an overview of the memory hierarchy organization. In Raster Graphics Systems, the Geometry Pipeline transforms the geometric description of a scene and creates all the *primitives* that fall inside the frustum view in accordance with the camera's viewpoint. On the other hand, the Raster Pipeline discretizes each primitive into *fragments* (at pixel granularity) that are then shaded and blended to produce the final screen image.

In a TBR architecture, the Raster Pipeline is designed to render *tiles* rather than the full frame. These tiles are usually square groups of adjacent pixels. This tiling improves locality and allows keeping on chip most bandwidth-intensive memory accesses. In order for this to happen, all the geometry needs to be sorted into subsets that will individually be able to fully render the image for each of these tiles. There are various methods of sorting. In the sorting classification of rendering techniques, as described in [30], TBR can be classified as a Sort-Middle technique. The process of tiling is carried out by a new pipeline stage called *Tiling Engine*.

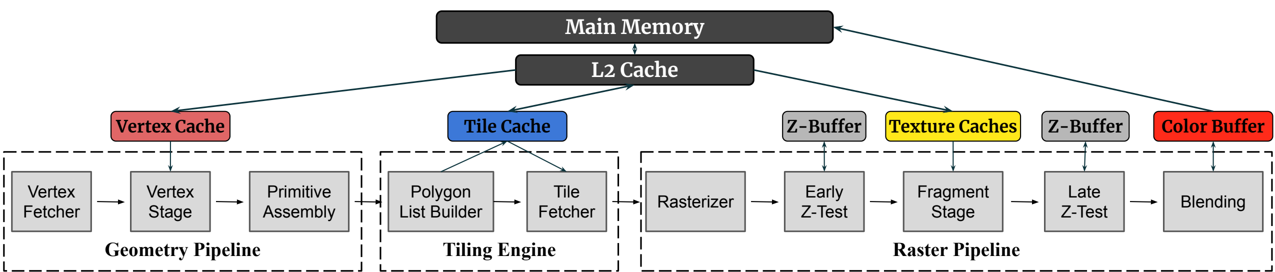


Figure 3: The Graphics Pipeline of a TBR GPU.

Thus, the Graphics Pipeline for TBR architectures consists of three parts, namely the Geometry Pipeline, the Tiling Engine and the Raster Pipeline, as shown in Figure 3.

Input data for the Graphics Pipeline consists of vertices and textures. These vertices join to form different polygons (usually triangles) called *primitives* and the textures are used to enhance details on surfaces while rendering the scene. A *Draw Command* triggers the Geometry Pipeline and the Vertex Stage starts fetching vertices from memory using an L1 Vertex Cache. It then transforms them according to a vertex program provided by the user. The Primitive Assembler takes the vertices in program order and joins them to produce primitives. These primitives are fed as input to the Tiling Engine.

The goal of the Polygon List Builder is to produce a list, for each tile of the screen, containing all the primitives that overlap it. This data is arranged in a structure known as the *Parameter Buffer*. For this purpose, the Polygon List Builder takes each primitive generated by the Primitive Assembly in program order and appends it to each list of every tile it overlaps. For each primitive, the Raster Pipeline needs to know its *attributes*, which describe the characteristics of its vertices (e.g., color, normal vectors, texture coordinates, etc.). Since attributes occupy significant space and primitives may overlap many tiles, the attributes of each primitive are stored only once in the Parameter Buffer, and the per-tile lists contain only the IDs of the primitives.

The Parameter Buffer is built and used up in the same frame. After all the geometry is processed and binned, the Tile Fetcher fetches the primitives corresponding to each tile in the frame, one tile at a time. Tiles are processed in an order specified by the Tiling Engine, and their primitives are put into a FIFO queue for the Raster Pipeline to consume. Since tiles have no data dependencies among themselves, they can be processed in any order.

The Raster Pipeline renders each tile sequentially. For this purpose, the Rasterizer takes each primitive from the FIFO queue and identifies which pixels of the current tile are overlapped by the primitive. It then uses interpolation to calculate attributes for each pixel, a set of data called *fragment*. The fragments of every four adjacent pixels are grouped to form a *quad*, and these quads are sent to the Early Z-Test stage. This stage uses a tile-sized buffer called the *Z-Buffer* to store the minimum depth of previously processed fragments on each tile's pixel coordinate in order to eliminate those that lie behind another previously processed opaque fragment. The non-discarded quads are then sent to a shader core (SC), which computes an initial color for each pixel of a quad, taking into account the lighting and textures provided by the *shader program*. The output colors are then sent to the Blending Unit. This unit computes the final color of pixels depending on the transparency of each quad, and stores them in the Color Buffer. Some rendering techniques require that the SC changes the depth of fragments, in which case the Early Z-Test is disabled and the Late Z-Test is employed. Note that both the Color Buffer and the Z-Buffer have the size of just one tile, and thus can be stored on-chip. Finally, the Color Buffer is flushed to the Frame Buffer in main memory, after a tile has been completely processed. Quads are propagated between stages through FIFO queues.

To increase throughput, the Raster Pipeline, from the Early Z-Test stage onwards, is implemented with several parallel pipelines that operate independently (for simplicity, we will assume four pipelines in the rest of this paper), each having its own SC and a private L1 texture cache. Both the Z-Buffer and the Color Buffer are partitioned into four banks, so each pipeline operates on its disjoint portion of the buffer to avoid access conflicts and improve bandwidth. The partitioning of these buffers implies that the quads generated by the Rasterizer must be scheduled to the four pipelines according to a static mapping that depends on their tile coordinates.

B. *Quad Scheduling*

Contemporary GPUs focus on load balancing for thread scheduling such that SCs are assigned a similar number of threads for each tile [16]. Interestingly, load balancing schedulers lead to memory block replication in the private L1 texture caches and thus decrease their aggregated capacity. This works well for balancing resource utilization. As long as overlapping primitives are not rendered in front-to-back order, the occluded fragments can not be culled by the Early Z-test and they end up rendered one on top of the other, wasting processing resources, a phenomenon known as

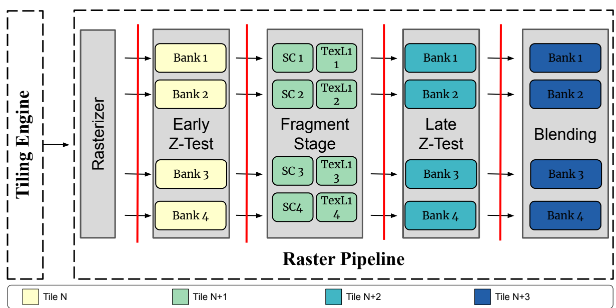


Figure 4: Barriers in the Raster Pipeline to manage the progress of tiles. Each color represents a different tile. Each pipeline stage processes a single tile at a given time.

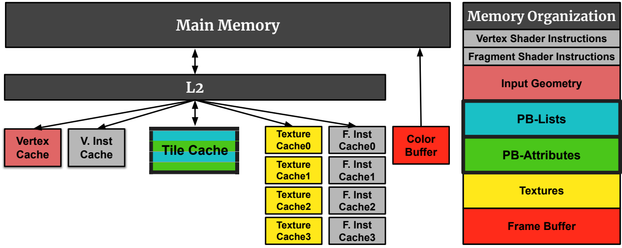


Figure 5: Baseline memory hierarchy and memory organization.

overdraw. Furthermore, some pixels require the computation of multiple fragments if they are transparent. In other words, in most scenes, geometry is not uniformly distributed over the frame[8], but rather some regions are richer than others in depth complexity (number of overlapping surfaces), which makes them prone to suffer more overdraw. Thus, mapping many adjacent quad locations to the same processor could lead to considerable load imbalance because it increases the likelihood that quads of a highly overdrawn region are mapped to the same processor. Therefore, to achieve load balancing w.r.t. the number of quads assigned to each SC, the quads must be scheduled to SCs in a fine-grained fashion such that adjacent screen-neighboring quads do not go to the same SC.

On the other hand, adjacent quads show texture block reuse and thus assigning them to different SCs with private L1 caches leads to block replications which could have been avoided had they gone to the same SC. Each quad accesses various neighboring texels from a large texture map. Adjacent quads frequently access the same texels or texels lying in the same cache line, more so in trilinear and anisotropic filtering than in bilinear filtering [11]. Thus, assigning adjacent quads to different SCs leads to replication of blocks in the private L1 caches, and reduces the effectiveness of the available storage.

This tension between favoring load balancing and locality is the main focus of this work.

Commercial GPUs have not disclosed the details of the quad grouping and subtile mappings they use except the fact that they use fine-grained scheduling to favor workload balance. We thus decided to explore the whole design space by having a reasonable number of representations from fine-grained mappings and choose the one that provides the best load balancing based on empirical data, as the final baseline, as we will see in Section V-A. We also explore a reasonable number of representation from coarse-grained mappings and choose the one that provides the best texture locality for our proposal.

C. Barriers

Since the on-chip buffers (the Z-Buffer and the Color Buffer) are sized to accommodate only one tile at a time, the pipeline stages that use these buffers (Early Z-Test and Blending, respectively) cannot move to the next tile before finishing the processing of the current tile. As for the Fragment Stage, multiple tiles could be processed simultaneously at the cost of providing a reorder mechanism so that the Blending Stage receives all quads belonging to one tile before starting to receive quads from another tile. This is necessary because quads are essentially warps in the GPU core and may not finish in the same order as they started because of various reasons such as Shader Programs, Texture Memory misses and Warp Scheduling. To avoid the high cost of reordering, the Fragment Stage is also limited to the execution of one tile at a time. To enforce these requirements, barriers are placed before the Early and Late Z-Test, the Fragment stage and the Blending stage, such that these stages are only fed quads from a new tile when all the quads from the previous tile have finished processing in that stage. The red lines in Figure 4 illustrate the barriers put in the Raster Pipeline. It must be noted though that these barriers have nothing to do with thread synchronization constructs often used in GPGPUs to synchronize threads within an SC (a.k.a GPU Core) or between pipeline stages to assure Draw Command dependencies set by the programmer. These barriers are internal hardware barriers put in different stages of the Raster Pipeline in order to ensure that each raster stage processes only one tile at a time.

D. Memory Organization

Figure 5 depicts the main memory data structures of a graphics application (on the right) and the memory hierarchy used to store and access them (on the left). As it can be seen, there are multiple L1 caches for instructions and data, backed by a shared L2 cache, which is ultimately backed up by main memory.

III. DTexL

The main contribution of this work is to favor texture locality during quad scheduling and still preserve load

balancing in the SCs. We do this by controlling how the stream of memory accesses is scheduled to the set of shader cores (SCs) so as to improve the caching capabilities within the private L1 Texture caches (favoring locality, reducing replication and maximizing the aggregated capacity) and then tweaking the Raster Pipeline to counter the subsequent load imbalance and achieve a significant improvement in GPU performance and energy.

The main challenges associated with favoring texture locality in quad scheduling while still preserving load balancing are listed below.

- Quads are scheduled to the four SCs according to a mapping that partitions the tile into four disjoint Subtiles. If the *overdrawn* [8] quads are not evenly distributed among the Subtiles then there is a load imbalance w.r.t. the number of quads among the SCs. The easiest way to overcome this imbalance is to use a fine-grained mapping that assigns spatially adjacent quads to different SCs. Six examples of such a mapping are shown in Subfigures 6(a)-6(f).
- Quads that are adjacent in screen coordinates have a high chance for spatial locality in their texture accesses as their texels may be mapped to the same cache blocks. These adjacent quads may also have temporal locality depending on the kind of filtering used in the texturing process (bilinear, trilinear, anisotropic, etc.). I.e., a quad accesses adjacent texels around it which are then reaccessed by a number of neighboring quads depending on the applied texture filtering. So grouping adjacent quads will lead to better caching in the L1 texture caches. Subfigures 6(g)-6(j) show rectangular, triangular and square-shaped groupings (Subtiles), respectively.
- At a higher level, tiles also share edges with their adjacent tiles. These shared edges are a potential area for temporal and spatial locality for textures. Thus, making sure that adjacent Subtiles from consecutive tiles go to the same SC will enhance texture locality.
- Quad schedulers focused on texture locality tend to map coarse-grained regions to the same SC, which causes a significant load imbalance that leads to equal or worse total GPU performance as compared to fine-grained schedulers aimed at load balancing.

A. Our Approach

The mapping of memory accesses among the different private L1 texture caches is synonymous to the mapping of quads among the SCs. To facilitate the understanding of our texture locality-aware mapping, we first explain the grouping of quads into Subtiles and then how these Subtiles are assigned to SCs. For illustrative purposes, we are considering the case of 4 SCs and so quads are assigned to 4 Subtiles.

On the other hand, it is important to recall that the order of these memory accesses on a tile granularity depends on the order that tiles are processed by the Raster Pipeline. Since the Graphics Pipeline does not impose any restriction on the tile processing order, we propose a new tile order aimed at maximizing the Texture Memory locality.

The resulting load imbalance is then overcome by proposing a decoupled Raster Pipeline explained at the end of this Section.

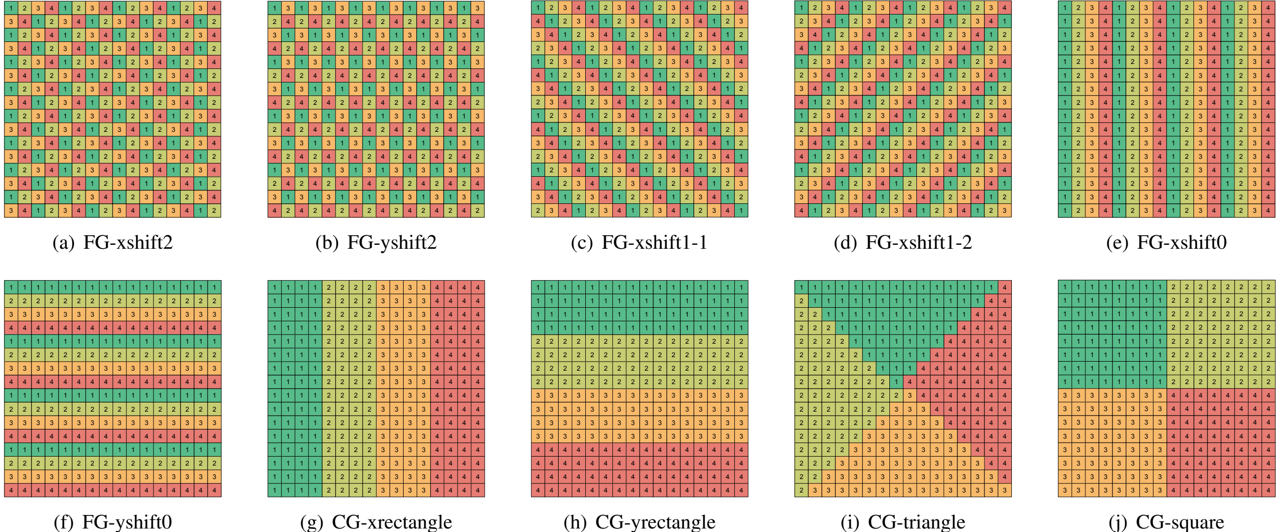
B. Quad Mapping

Each SC has its private L1 Texture Cache. The caching effectiveness of each L1 cache has a correlation with the quads that are assigned to each processor. As explained in Section II, the Tile Fetcher processes tiles in a predefined fixed order after which the Rasterizer produces all the quads belonging to each primitive of a tile in the order that the Tile Fetcher fetched primitives (this order is set by the programmer). These quads are then mapped to one of the four banks of the Z-Buffer. We will henceforth call the group of quads mapped to one bank, a *Subtile*. Each of these Subtiles is then allotted to one of the SCs. Thus, the locality in the L1 of each SC depends on how the Rasterizer maps quads into Subtiles.

1) Quad Grouping: As explained in Subsection II-B, a tile may have more quads concentrated in a particular region because of highly overdrawn primitives concentrated in that region. As the amount of overdraw is unknown beforehand, in order to achieve a balanced number of quads in each Subtile, a simple mapping policy consists of distributing quads in a fine-grained interleaved manner as shown in Subfigures 6(a)-6(f). This balances the number of quads over the SCs at the expense of increasing texture block replication across the different L1 caches, therefore minimizing the potential locality that could be exploited. Note that Subfigures 6(a) and 6(b) make sure that for each quad, no adjacent quad goes to the same SC as itself. On the other hand, for each quad, Subfigures 6(c) and 6(d) allow at most 2 diagonal neighbor quads to go to the same SC and Subfigures 6(e) and 6(f) allow at most 2 vertical or 2 horizontal neighbor quads to go to the same SC. These 6 thus cover a reasonable variation over the infinite possible fine-grained mappings.

In order to reduce texture block replication and to preserve the potential locality in the L1 texture caches, the subtiling must be done so that the maximum number of quads in each Subtile are spatially adjacent. This results in the various possible shapes for the Subtiles illustrated in Subfigures 6(g), 6(h), 6(i) and 6(j). However, these subtilings increase load imbalance due to the overdraw clustering effect of primitives commented above.

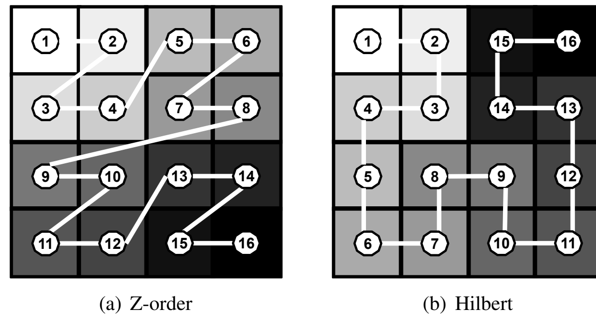
2) Subtile Assignment: Each Subtile could be assigned to any of the SCs. Depending on the tile processing order, many tiles share at least one edge with a previously processed tile. Potential texture locality exists not only among spatially adjacent quads within a tile but also between two



(a) FG-xshift2 (b) FG-yshift2 (c) FG-xshift1-1 (d) FG-xshift1-2 (e) FG-xshift0

(f) FG-yshift0 (g) CG-xrectangle (h) CG-yrectangle (i) CG-triangle (j) CG-square

Figure 6: Various fine-grained (FG) and coarse-grained (CG) quad groupings. Each screen location (a quad) is mapped to one SC. Each color represents a different SC.



(a) Z-order (b) Hilbert

Figure 7: Tile Orders. Each square is a tile. The squares get darker as the sequence progresses. Note that both Z-order and Hilbert process adjacent tiles sooner than tiles far away.

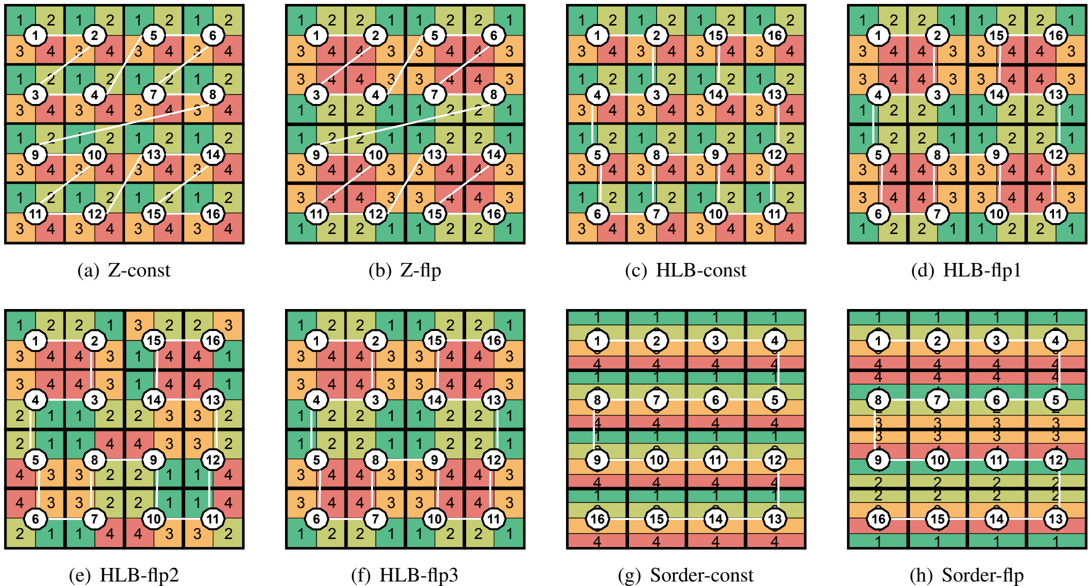
adjacent Subtiles that lie in different tiles but share an edge. Therefore, mapping these Subtiles to the same SC reduces block replication and potentially increases locality in the L1 texture caches. The importance of Subtile assignment is better understood with the example of tile orders as explained next.

C. Tile Orders

The order in which quads are processed by an SC affects the order of the memory accesses to the texture memory hierarchy. On the one hand, quad reordering across primitives violates the correctness of the graphics pipeline. That is to say that quads from another primitive cannot start processing until the ones from the current primitive are finished. However, tiles are totally independent entities and can be processed in any order, therefore, quad reordering on a tile granularity does not affect the correctness of the pipeline. We thus explore different tile orders and their effects on the texture memory hierarchy.

The most well-known orders in graphics are Scan-line and Z-order. Scan-line order processes tiles in a row-by-row fashion. Z-order, shown in Subfigure 7(a), is quite popular in computer graphics and image processing because of certain properties that enhance locality in memory accesses. In mathematical analysis and computer science, space filing curves like Z-order, Hilbert curve or Lebesgue curve map multidimensional data to one dimension while preserving locality of the data points. Figure 7 shows the Z-order and the Hilbert order for a squared frame split into 16 tiles.

Hilbert order has been less explored in computer graphics and it proves to be a strong competitor for preserving locality. Several works like [19], [20] and [13] show that Hilbert performs better than Z-order for particular workloads. But the main trade-off is the calculation of the Hilbert order. While Z-order is easy to calculate with bit swizzling of the scan-line IDs of a data point, Hilbert requires multiple floating point operations to calculate the Hilbert ID of a data point. This is an overhead in many applications that use space-filing curves. However, for the case of Tile orders in the context of a TBR architecture, where the number of tiles is usually in the order of a few thousands (depending on the resolution of the frame to be rendered), this calculation can be precomputed to avoid such an overhead. Another issue is that Hilbert order works best for squared dimensions whereas mobile screens are mostly rectangular. We propose a Hilbert order that has been adapted to a rectangular region.



(a) Z-const (b) Z-flp (c) HLB-const (d) HLB-flp1

(e) HLB-flp2 (f) HLB-flp3 (g) Sorder-const (h) Sorder-flp

Figure 8: Various Subtile mappings with CG-square and CG-yrect. The white circle shows the Tile ID of each tile. Each screen location (subtile) is mapped to one SC. Each color represents a different SC. Subfigures 8(g) and 8(h) use CG-yrect while the rest use CG-square.

We apply the Hilbert order on a square sub-frame with 8X8 tiles and then traverse all the sub-frames in the frame boustrophedonically or, in other words, in the shape of an S.

D. Tile Order and Subtile Assignment

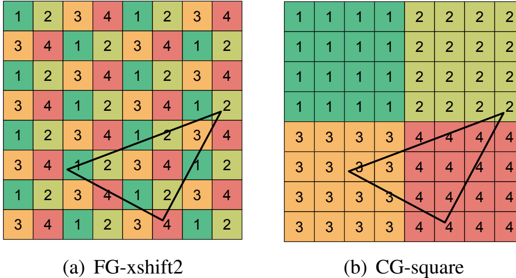
For illustrating purposes, Figure 8 shows a frame containing 16 tiles with eight Subtile assignments, using a combination of one of two quad groupings (CG-square and CG-yrectangle) with one of three tile orders (Z-order, Hilbert order and S-order). In Subfigures 8(a), 8(c) and 8(g), we see that as we progress along the Z-order, the Hilbert order and the S-order respectively, Subtiles that go to the same SC and therefore use the same L1 texture cache, do not share an edge. Whereas, in the five remaining subfigures, we see a rearrangement in the assignment of Subtiles to SCs. The main motivation here is to assign adjacent Subtiles belonging to adjacent tiles to the same SCs without favoring any SC over the course of the frame. In Subfigure 8(d), the Subtile assignment has been simply flipped along the shared edge of consecutive tiles. For example, when going from Tile1 to Tile2 the Subtiles are reassigned so that subtiles in Tile1 that are assigned to SC2 and SC4, share an edge with the subtiles assigned to the same SCs in Tile2. Similarly, going from Tile2 to Tile3, the shared edge is assigned to SC3 and SC4. The same happens in Subfigure 8(b) but with Z-order. In both cases, note that SC4 is favored to always have a shared edge. Whereas, SC1 never gets shared edges. Subfigure 8(e) and 8(h) overcome this imbalance by favoring different SCs over the course of the frame. When going from an even tile to an odd tile, HLB-flp2 flips along the shared edge and then also flips the non-sharing subtiles amongst themselves. For example, in Subfigure 8(e), when going from Tile2 to Tile3 SC3 and SC4 share an edge like before but SC1 and SC2 also interchange their places. The long-term effect of this, considering the whole frame, is that no one SC is favored for edge sharing. Note that in Subfigure 8(f), we flip all four Subtiles every 16 tiles such that the same FP is not at an advantage or a disadvantage every time.

Let us now summarize the benefits and disadvantages of each of analyzed subtile assignments.

- Mapping the same quadrants to the same SCs in all the tiles leads to the L1 texture caches not exploiting all the potential locality (see Subfigures 8(a), 8(c) and 8(g)).
- Flipping tiles around the axis of the shared edge as we traverse the tiles allows adjacent Subtiles to go to the same L1 caches. However, one subtile always gets the advantage of having a shared edge at all times (see Subfigures 8(b) and 8(d)).
- Subfigures 8(e), 8(f) and 8(h) fight this imbalance that gives a fair share of shared edges to all SCs over the course of the frame.

E. Decoupled-Barrier Architecture

In our approach, each SC receives a subset of a given tile. Depending on the number of quads in each SC, their workload complexity, and the state of the private L1 Texture



(a) FG-xshift2 (b) CG-square

Figure 9: Each subfigure shows FG-xshift2 and CG-square quad grouping, respectively, for one tile. The overlapped quads of primitive in 9(a) are evenly distributed among SCs while for 9(b), they are not.

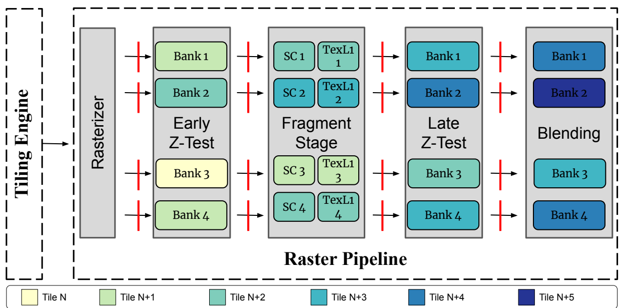


Figure 10: Overview of the proposed Decoupled Barrier Architecture. Each color represents a different tile. Each SC can now process a different subtile, at a given time, depending on the progress of subtiles.

caches at that time, different SCs will finish their set of assigned quads (in a Subtile) at different times. Thus, each SC will have to wait until the last SC finishes its subtile, leading to idle time in the SCs.

Let's look at an example shown in Figure 9 to understand this imbalance. This Figure shows a tile with FG-xshift2 grouping on the left and CG-square on the right. The tile will have multiple primitives overlapping it and they will access various textures. For the sake of simplicity, we only show one primitive in the down-right corner. Let us assume that the primitive has a relatively heavy workload (e.g. long shader programs, multiple long latency texture accesses). The length of the shader program has a proportional impact on the throughput of the SC whereas a main memory access may or may not be hidden by multithreading. In each of the two cases, the primitive overlaps 17 quads. In the FG-xshift2 case (depicted on the left), this heavy workload is more evenly distributed than in the CG-square case. If the rest of the primitives in the tile have a relatively low workload, then in the CG-square case, SC3 got an unfair share of quads with a heavy workload. Thus SC3 has a potential to finish the last whereas SC0 got none of them and has a potential to finish the tile first and will need to sit idle until SC3 finishes. Contrarily, all SCs in the FG-xshift2 case got a near-equal share of the primitive's quads thus reducing the imbalance in execution time.

To avoid this, we propose a new architecture for the pipeline. As explained in Section II, the Raster Pipeline in the baseline places barriers at the input of the final three stages of the Raster Pipeline, namely the Early Z-Test (since we choose Early Z-Test, Late Z-Test is disabled for this example), the Fragment Stage and the Blending, as shown in Figure 10 for the reasons mentioned in Section II. These barriers make sure that the subsequent stage does not start a new tile until the previous one has completed in that stage. Finally, when Blending finishes processing a tile, the whole Color Buffer (all banks) containing the whole tile is flushed into memory, before starting the next tile in Blending. Blending then starts updating the Color Buffer with the new tile's colors. Recall that the Color Buffer and the Z-Buffer are tile-sized but multi-banked. We leverage this banking to decouple the barriers in the final three stages. In our experiments, these three stages have 4 parallel units working independently (4 banks per buffer and 4 SCs), with a maximum throughput of 1 quad per cycle each. The scheduler assigns to each parallel unit a section of the tile that we call subtile. Since these subtiles are disjoint, the baseline requirement of having all the units in the same stage working on a single tile may now be relaxed to just having each parallel unit working on a single subtile. In other words, the barriers among stages are now decoupled so that each stage can start a new subtile when it finishes the previous one, without having to wait until the remainig subtiles are also finalized. Note that the banks are all the same size and thus the subtiles need to be one-fourth the size of a tile.

As an example, let's compare Figure 4 and 10 to better understand the functionality of a decoupled architecture. In Figure 4 we clearly see that each parallel unit of each of the raster stages is processing different subtiles of the same tile. Whereas in Figure 10 going from the back to front order in raster stages, we see that the second parallel unit in the present state is two tiles ahead of the third parallel unit. This just means that the second parallel unit is processing a subtile from Tile N+6 because it finished subtiles from Tiles N+4 and N+5 early and was able to go ahead because of the decoupling. Note that during the execution of the rest of the frame, this parallel unit might slow down and others speed up and finish the frame around the same time.

Decoupling requires only 2 changes as listed below.

- Change the Color Buffer to be able to flush each bank individually. This does not add any additional cost as it just requires the storage of the Tile ID separately for each of the banks so that the bank can be flushed to the correct address in memory. What this achieves is the ability for parallel units in the Blending Stage to start a new subtile from the next tile without waiting

Table I: Evaluated benchmarks from the Google Play Store.

Benchmark	Alias	Installs (Millions)	Genre	Type	Texture Footprint (in MiB)
Candy Crush Saga	CCS	1000	Puzzle	2D	2.4
Sonic Dash	SoD	100	Arcade	3D	1.4
Temple Run	TRu	500	Arcade	3D	0.4
Shoot Strike War Fire	SWa	10	Shooter	3D	0.2
City Racing 3D	CRa	50	Racing	3D	2.8
Rise of Kingdoms: Lost Crusade	RoK	10	Strategy	2D	6.8
Derby Destruction Simulator	DDS	10	Racing	3D	1.4
Sniper 3D	Snp	500	Shooter	3D	1.8
3D Maze 2: Diamonds & Ghosts	Mze	10	Arcade	3D	2.4
Gravitytetris	GTr	5	Puzzle	3D	0.7

Table II: GPU simulation parameters.

Global Parameters	
Tech Specs	600MHz, 1V, 32nm
Screen Resolution	1960x768
Tile Size	32x32
Tile Traversal Order	Z-order
Main Memory	
Latency	50-100 cycles
Size	1GiB
Caches	
Vertex Cache	64-bytes/line, 8KiB, 4-way, 1 cycle
Texture Caches (4x)	64-bytes/line, 16KiB, 4-way, 1 cycle
Tile Cache	64-bytes/line, 64KiB, 4-way, 1 cycle
L2 Cache	64-bytes/line, 1MiB, 8-way, 12 cycles

for all the subtiles in the Color Buffer to be flushed to memory.

- Modify the barriers in the inputs of Early Z-test, Fragment Stage and Blending. The Early Z-Test has 4 parallel pipelines, each used by one subtile. Implementing a decoupled barrier just means ensuring the previous subtitle has finished processing in one parallel pipeline before starting a new subtile in that parallel pipeline. The same applies to the other two stages.

Thus Decoupled architecture has a negligible area overhead. Since we are increasing throughput in the final 3 stages of the pipeline by eliminating this waiting period, this puts pressure on the first two stages and mainly the Tile Fetcher. We find that the Fragment Stage is a major bottleneck of the Raster pipeline. Thus this pressure still does not migrate the bottleneck to the Tile Fetcher, as can be corroborated by the speedup of the whole pipeline in Section V-C2.

Our proposal thus solves the issue of idle time in SCs and thus reduces load imbalance significantly. Thus the idle time between two tiles is reduced to near zero for each SC. Note that in case the Subtile mapping function favors any one SC at all times, this SC will finish the frame before the rest and will sit idle towards the end of the frame. In order to combat this, mapping functions should be impartial to SCs over the period of a frame. This impartiality is depicted by Hilbert-flip2 (Subfigure 8(e)), Hilbert-flip3 (Subfigure 8(f)) and Sorder-flip (Subfigure 8(h)). Whereas Hilbert-flip1 (Subfigure 8(d)) always allots SC3 with a shared edge. While our decoupled architecture will ensure load balancing for the majority of the time, this partiality may lead to an imbalance in the time taken for each SC to finish the frame.

Note here that the Fragment Stage is known to be a bottleneck for the graphics pipeline and SC performance has a major effect on the performance of the GPU as we will see in Section V.

IV. Evaluation Framework

A. GPU Simulation Framework

We use the TEAPOT [3] simulation infrastructure to evaluate our proposal. TEAPOT is a cycle-accurate GPU simulation framework that allows to run unmodified Android applications and evaluate the performance and energy consumption of the modeled GPU. In order to do that, TEAPOT includes timing and power models based on well-known tools: McPAT [28] for power estimation, and DRAMSim2 [32] for modeling DRAM and the memory controllers. Table II shows the parameters employed in our simulations, which resemble those of a contemporary mobile GPU.

B. Benchmarks

We use popular commercial animated applications (games) as benchmarks. We have selected them based on their popularity in the number of downloads in the Google Play Store, and their variety to cover different types of games.

Table I shows the ten Android games used to evaluate our technique. We have 2D games like CCS and 3D games like CRa. Games like RoK has a texture footprint of around 6.8MiB in memory whereas SWa has around 0.2MiB. Upon further characterization of our benchmark suite, we also observed that the reuse of texture memory blocks also varies greatly across different games.

V. Experimental Results

In this Section we first explore the effects of the fine-grained and coarse-grained Quad Groupings on load balancing and on texture access locality. We then choose a representative for load balancing as the baseline and the best

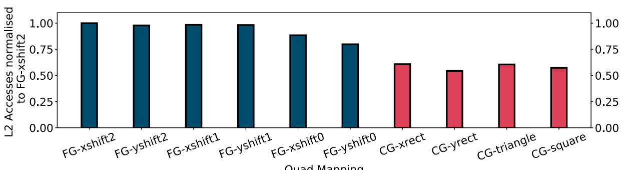


Figure 11: Average L2 Accesses normalized to FG-xshift2 (fine-grained in blue and coarse-grained in pink).

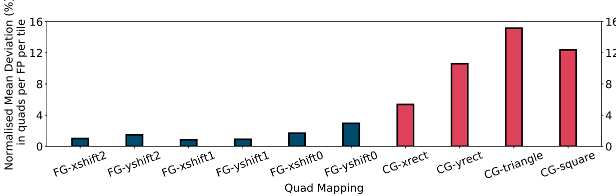


Figure 12: Average normalized Mean Deviation in quad distribution normalized to FG-xshift2 (fine-grained in blue and coarse-grained in pink).

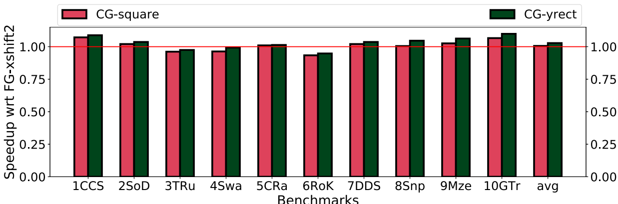


Figure 13: Speedup of CG-square and CG-yrect w.r.t. FG-xshift2.

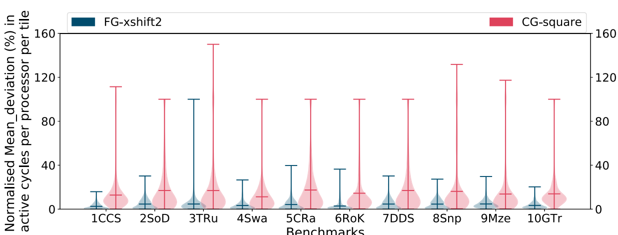


Figure 14: FG-xshift2 vs CG-square execution time imbalance in SCs.

one for texture access locality for our proposed technique and analyze the total performance and workload imbalance of both approaches without a decoupled architecture. Finally, we evaluate DTexL: we explore DTexL with all the Subtile assignments shown in Figure 8 and their effects on caching, performance and energy.

A. *Quad Grouping*

Figure 11 plots the average L2 accesses of the eight quad groupings shown in Figure 6, normalized to those of FG-xshift2. On the other hand, Figure 12 plots the average normalized mean deviation in quad distribution, normalized to that of FG-xshift2. The fine-grained groupings are shown in blue and the coarse-grained ones in pink. Comparing the two figures, the trade-off between texture locality and load balancing is quite stark. As explained in Section III, fine-grained groupings like FG-xshift2 achieve better load balancing because they avoid clustering overdrawn pixels to one SC. Whereas coarse-grained groupings like CG-square help exploit spatial locality of the texels accessed by adjacent quads.

Another observation is that the groupings that show more load balancing vertically than horizontally, and hence more adjacency horizontally than vertically, show better texture locality and worse load balancing. For example, CG-xrect and CG-yrect show a 40% and 45% decrease in L2 accesses and a 6× and 10× increase in load balancing of quads. This implies that averaging over all benchmarks, there is more overdraw clustering horizontally than vertically. One reason could be that in most scenes, gravity forces objects to be more horizontally shaped than vertical or diagonal.

For the rest of the paper, we choose FG-xshift2 as the baseline to represent the fine-grained and CG-square to represent the coarse-grained. We also explore CG-yrect since it shows a decrease in L2 accesses similar to CG-square.

B. *Performance in a Non-Decoupled Architecture*

Figure 13 plots the speedups of CG-square and CG-yrect over the baseline FG-xshift2. It shows that CG-square does not provide any speedup even though we see a 46.8% decrease in L2 Accesses in Figure 11. We could have assumed that this is because multithreading hides memory latencies so well that this improvement in caching does not contribute towards total performance. But Figures 14 and 15 tell a different story: the potential performance improvement enabled by a better cache behavior is offset by a worse load balancing.

Figure 14 plots the mean deviation in SC execution time to complete a tile (normalized to the mean of all SCs, in percent) for both the FG-xshift2 and CG-square mappings, and depicts it in a violin plot including the min, max, average and the distribution. It clearly shows that the normalized mean deviation is higher in the case of CG-square (and it can go up to 150% for TRu) whereas it has an average of around 5% in the case of FG-xshift2.

Figure 15 plots the mean deviation in number of quads processed per SC to complete a tile (normalized to the mean of all SCs, in percent) for both approaches, and here it also shows that the coarse-grained mapping based on texture locality has a higher deviation. This deviation in number of quads per SC contributes towards the deviation in execution time, but it is not the only reason. Each quad comes with a different intensity of workload (depending on things like length of shader programs, number of long

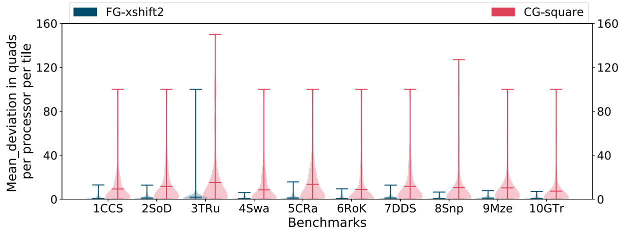


Figure 15: FG-xshift2 vs CG-square Quad distribution imbalance in SCs.

latency texture accesses, etc). Adjacent quads that belong to the same primitive tend to have the same workload intensity. Thus, fine-grained mappings tend to balance the workload intensity among different SCs, too. Both factors together (number of quads and workload intensity) contribute towards the deviation of execution time in different SCs. We counter the impact of this imbalance on aggregated performance by using the Decoupled-Barrier Architecture of the raster pipeline.

C. DTexL

In this subsection we apply the various subtile assignments shown in Figure 8 with DTexL and explore the benefits in texture cache locality, performance and energy w.r.t. the baseline. The baseline here has a fine-grained Quad Grouping (FG-xshift2) and Z-order as its tile order with the non-decoupled barrier architecture.

1) Texture Locality: Figure 16 plots the percent decrease in total L2 accesses w.r.t. the baseline for the eight combinations of Quad Mappings and Tile Orders displayed in Figure 8 as well as an upper bound. It shows that Quad Mapping and Tile Orders can combine to reduce the gap between the L2 Accesses of the baseline and the upper bound by 80%. Note that a higher decrease in L2 accesses indicates lesser replication of texture memory blocks in the private L1 caches.

The upper bound can be computed by running simulations with a single SC and Texture cache instead of 4, but resized to be 4× the size of the original private L1 caches. This arrangement avoids the loss of effective aggregated cache capacity caused by block replication, and the loss of locality caused by mapping quads that access the same texture cache line to different SCs.

It can be observed that without shared-edge aware subtile mapping, the L2 accesses decrease 40.7% on average in both Zorder-const and HLB-const. While HLB-flp1, HLB-flp2 and HLB-flp3 all show around 46.5% decrease in L2 accesses, Sorder-const and Sorder-flp show a slightly better 46.8% decrease. This trend is varying across the benchmarks. And yet the common theme is that HLB-flp1, HLB-flp2, HLB-flp3, Sorder-const and Sorder-flp show the best results. We should also note that these results are very close to the upper bound.

One important point to note here is that most adjacent quads share texture blocks and hence mapping quads into different SCs will always lead to some replication in the L1 texture caches. Thus the upper bound is conservative and not necessarily achievable and yet these mappings are able to reduce the gap between the baseline and the upper bound by 80%.

Having seen such a drastic decrease in L2 accesses, one might expect a decrease in L2 misses too. However, quad mapping variations within a tile, that reduce replication of memory blocks in the private L1 texture caches, are targeted towards improving the short-term reuse of texture memory accesses. So, as expected, we did not see any notable changes in the L2 misses in our results (although not shown here), and hence no changes in main memory accesses.

2) Performance: On analysis, we found that all subtile assignments show a similar effect on performance even with their differences in L2 accesses. This is because multithreading hides memory latency and only drastic changes in L2 accesses can show a significant change in GPU performance. Having said that, HLB-flp2 had the best performance among the proposed subtile assignment in Figure 8 and thus we now study this assignment. Figure 17 plots the speedup of DTexL (HLB-flp2) w.r.t. the baseline (fine-grained FG-xshift2 without decoupled-barrier architecture). We also plot the speedup of a fine-grained (FG-xshift2 with Z-order) with a decoupled-barrier architecture in order to make a wholesome comparison. On average we see a significant 1.2× speedup and in GTr we see close to 1.4× speedup for DTexL. We also see a 1.09× speedup for FG-xshift2. These results corroborate our findings in Subsection V-B that both fine-grained and coarse-grained quad groupings show an imbalance in SC execution time per tile and that the imbalance is more pronounced in the coarse-grained case. What we see here is that the decoupled architecture has been able to overcome this imbalance and provide a speedup for both cases. The higher speedup for HLB-flp2 shows us that benefits in texture caching provided by coarse grained quad groupings (a drastic 46.8% decrease in L2 Accesses) has translated into a speedup despite the fact that GPUs are usually good at hiding memory latencies using multithreading. This can be attributed to two facts. One that improvement in texture cache locality is drastic in coarse grained groupings and two that SC performance in TBR architectures is more susceptible to memory latency due to periods of low occupancy in the SCs.

3) Total GPU Energy: Figure 18 plots the decrease in total GPU energy of FG-shift2 (Z-order) and HLB-flp2, both with a decoupled-barrier architecture, w.r.t. FG-xshift2, without a decoupled-barrier architecture. We see a decrease in total energy of 6.3% on average, and around 8.8% for CCS and 10.6% for GTr. Note that this is a significant

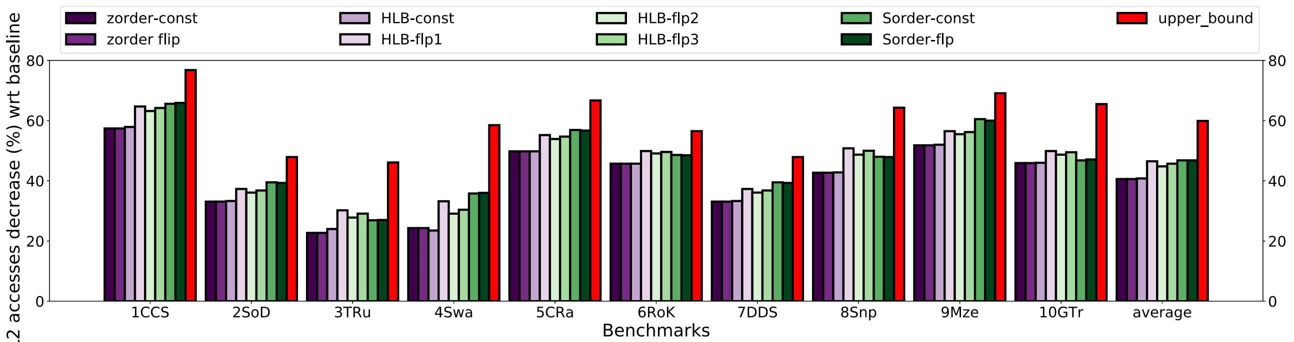


Figure 16: Decrease in L2 accesses for various subtile mappings w.r.t non-decoupled FG-xshift2.

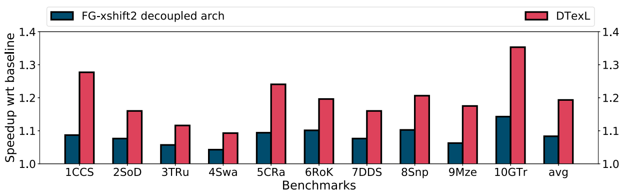


Figure 17: Speedup w.r.t. Non-Decoupled FG-xshift2.

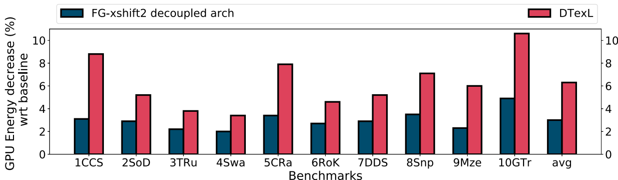


Figure 18: Decrease in total GPU energy w.r.t. Non-Decoupled FG-xshift2.

achievement as energy efficiency is crucial in mobile GPUs and this is achieved with a negligible hardware cost. We also note a 3% decrease for FG-xshift2 with a decoupled architecture. Note that there is a strong correlation between Figure 17 and Figure 18 indicating that reduction in energy comes mainly from a decrease in L2 accesses and execution time.

VI. Related Work

Previous work on workload scheduling in GPUs has focused on load balancing because schedulers are generically designed for graphics as well as general purpose workloads. Our work focuses on proposing a texture locality aware scheduler and a minor modification to the Raster Pipeline to overcome the load imbalance caused by the scheduler.

Kerbl et al.[16] have proposed a static workload scheduler that focused on scheduling tiles among different raster units assuming a parallel tile rendering architecture with multiple Raster Units. This scheduler focused on load balancing to ensure that an equal number of threads are assigned to each Raster Unit. It did this by exploring fine-grained scheduling of tiles that avoided assigning the adjacent tiles to the same Raster Unit. This avoided cluster scheduling of tiles which we know causes load imbalance. Other works like [10] and [29] explore dynamic workload scheduling that also focused on scheduling tiles among different raster units assuming a parallel tile rendering architecture with multiple Raster Units. Yet none of these works focus on workload scheduling based on cache access locality.

Focusing on locality-aware workload scheduling, Ukarande et al [34] report a 4% speedup when exploiting Texture Cache locality on high-end desktop graphics workloads by clustering CTAs that are close in screen coordinates. Other works like [25], [21] and [4] take a similar approach by proposing a CTA scheduling focused on L1 Data Cache locality but for GPGPU workloads, and resulted in performance improvements. It must be noted here that all the works above propose only software modifications and they work on a CTA granularity unlike DTexL that proposes a hardware solution for a quad scheduler for mobile GPUs.

Many works have previously targeted cache locality in GPUs for GPGPU workloads. Some works have targeted cache locality across kernel launches for parent-child kernels [37] or generic dependent kernels [12]. Other works like [18], [5], [25] have also studied the impact of warp throttling on locality for GPGPU workloads. Whereas others like [31], [36], [22], [23], [40], [14], have studied the impact of warp-scheduling within a core on locality, also for GPGPU workloads. Other works like [17], [26], [27], [33], [39], [42], [38], [7], [41], [35], [24] have explored cache bypassing to improve GPU cache locality. Another work [9] proposes a Cooperative Caching Network (CCN), a ring network among L1 caches of a GPU, to reduce L2 bandwidth demand.

As for graphics workloads, there is previous work [2] on prefetching texture memory in the L1 texture caches. Another work [6] proposes a NUCA organization for the L1 texture caches to increase their effective overall capacity. Another work [15] has proposed improving locality of the

L1 Tile Cache in a GPU for graphics workloads.

All the above techniques focus on cache locality in GPUs but most of them are specific for GPGPU workloads and at the same time the techniques are orthogonal to DTexL, which focuses on texure locality aware quad scheduling for graphics workloads in mobile GPUs.

VII. Conclusions

Contemporary GPUs use simple load balancing schedulers in order to assign workloads into the shader cores. These schedulers often prevent the texture caches from exploiting the potential texture locality between adjacent quads by placing them in different cores.

In this work we have proposed a novel workload scheduling for the shader cores in TBR architectures for mobile GPUs focusing on improving Texture Cache locality while still preserving the desired load balancing. We preserve this load balance by also proposing a minor change in the Raster Pipeline of the TBR architecture in order to translate the improvement in caching into an improvement in GPU performance. We call our proposal DTexL.

Experimental results show that the best of DTexL leads to a drastic decrease of 46.8% in L2 accesses, averaged over a benchmark suite that has 10 contemporary gaming applications. Such an L2 access decrease closes the gap between that of the baseline and a conservative upper bound by 80%. We also observed a significant increase in the imbalance in execution time for different cores for each tile and proposed a minor change in the pipeline in order to leverage that imbalance and convert it into a 1.2× speedup in GPU performance on average, and a 1.35× speedup for one case. This also leads to a 6.3% decrease in total GPU energy.

Acknowledgment

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, the ICREA Academia program and the AGAUR grant 2020-FISDU-00287. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Memory bandwidth requirements of tile-based rendering," International Workshop on Embedded Computer Systems, pp. 323–332, 2004.

[2] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," ACM SIGARCH Computer Architecture News, vol. 40, pp. 84–93, 2012.

[3] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems," ACM International Conference on Supercomputing, p. 37–46, 2013.

[4] L.-J. Chen, H.-Y. Cheng, P.-H. Wang, and C.-L. Yang, "Improving gpgpu performance via cache locality aware thread block scheduling," IEEE Computer Architecture Letters, vol. 16, pp. 127–131, 2017.

[5] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2014, pp. 343–355.

[6] D. Corbalán-Navarro, J. L. Aragón, J.-M. Parcerisa, and A. González, "Dtm-nuca: dynamic texture mapping-nuca for energy-efficient graphics rendering," in 2022 30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2022: Valladolid, 9-11 March 2017: proceedings. Institute of Electrical and Electronics Engineers (IEEE), 2022, pp. 144–151.

[7] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor, "A model-driven approach to warp/thread-block level gpu cache bypassing," in 53rd ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, 2016, pp. 1–6.

[8] M. F. Deering, "Data complexity for virtual reality: Where do all the triangles go?" in Proceedings of IEEE Virtual Reality Annual International Symposium. IEEE, 1993, pp. 357–363.

[9] S. Dublish, V. Nagarajan, and N. Topham, "Cooperative caching for gpus," ACM Trans. Archit. Code Optim., vol. 13, 2016.

[10] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories," ACM Siggraph Computer Graphics, vol. 23, no. 3, pp. 79–88, 1989.

[11] P. S. Heckbert, "Survey of texture mapping," IEEE Computer Graphics and Applications, vol. 6, no. 11, pp. 56–67, 1986.

[12] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," ACM Trans. Archit. Code Optim., vol. 17, 2020.

[13] H. V. Jagadish, "Analysis of the hilbert curve for representing two-dimensional space," Information Processing Letters, vol. 62, no. 1, pp. 17–22, 1997.

[14] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," SIGPLAN Not., vol. 48, 2013.

[15] D. Joseph, J. L. Aragón, J. M. Parcerisa, and A. González, "Tcor: a tile cache with optimal replacement," in Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture, 2022, pp. 662–675.

[16] B. Kerbl, M. Kenzel, D. Schmalstieg, and M. Steinberger, "Effective static bin patterns for sort-middle rendering," in Proceedings of High Performance Graphics. Association for Computing Machinery, 2017.

[17] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in gpu," in Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017, pp. 307–319.

[18] H.-K. Kuo, T.-K. Yen, B.-C. C. Lai, and J.-Y. Jou, "Cache capacity aware thread scheduling for irregular memory access on many-core gpgpus," in 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), 2013, pp. 338–343.

[19] J. K. Lawder and P. J. King, "Using space-filling curves for multi-dimensional indexing," in British National Conference on Databases. Springer, 2000, pp. 20–35.

[20] J. K. Lawder and P. J. King, "Using space-filling curves for multi-dimensional indexing," in British National Conference on Databases. Springer, 2000, pp. 20–35.

[21] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2014, pp. 260–271.

[22] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in Proceedings of the 42nd Annual International Symposium on Computer Architecture. Association for Computing Machinery, 2015, p. 515–527.

[23] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. Association for Computing Machinery, 2014, p. 175–186.

[24] S.-Y. Lee and C.-J. Wu, "Ctrl-c: Instruction-aware control loop based adaptive cache bypassing for gpus," in 2016 IEEE 34th International Conference on Computer Design (ICCD), 2016, pp. 133–140.

[25] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2017, p. 297–311.

[26] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and transparent cache bypassing for gpus," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, 2015.

[27] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic gpu cache bypassing," in Proceedings of the 29th ACM on International Conference on Supercomputing. Association for Computing Machinery, 2015, p. 67–77.

[28] 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 many-core architectures," IEEE/ACM International Symposium on Microarchitecture, p. 469–480, 2009.

[29] W.-S. Lin, R. W. H. Lau, K. Hwang, X. Lin, and P. Y. S. Cheung, "Adaptive parallel rendering on multiprocessors and workstation clusters," IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 3, pp. 241–258, 2001.

[30] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," IEEE computer graphics and applications, vol. 14, no. 4, pp. 23–32, 1994.

[31] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. Association for Computing Machinery, 2013, p. 99–110.

[32] 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.

[33] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing," in Proceedings of the 8th Workshop on General Purpose Processing Using GPUs. Association for Computing Machinery, 2015, p. 25–35.

[34] A. Ukarande, S. Patidar, and R. Rangan, "Locality-aware cta scheduling for gaming applications," ACM Trans. Archit. Code Optim., vol. 19, 2021.

[35] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "Dacache: Memory divergence-aware gpu cache management," in Proceedings of the 29th ACM on International Conference on Supercomputing, 2015, pp. 89–98.

[36] B. Wang, Y. Zhu, and W. Yu, "Oaws: Memory occlusion aware warp scheduling," in 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT), 2016, pp. 45–55.

[37] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 583–595.

[38] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on gpus," in 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2013, pp. 516–523.

[39] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 76–88.

[40] Y. Zhang, Z. Xing, C. Liu, C. Tang, and Q. Wang, "Locality based warp scheduling in gpgpus," Future Generation Computer Systems, vol. 82, pp. 520–527, 2018.

[41] C. Zhao, F. Wang, Z. Lin, H. Zhou, and N. Zheng, "Selectively gpu cache bypassing for un-coalesced loads," in 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2016, pp. 908–915.

[42] X. Zhu, R. Wernsman, and J. Zambreno, "Improving first level cache efficiency for gpus using dynamic line protection," in Proceedings of the 47th International Conference on Parallel Processing. Association for Computing Machinery, 2018.