LIBRA: Memory Bandwidth- and Locality-Aware Parallel Tile Rendering

Aurora Tomás*, Juan L. Aragón†, Joan-Manuel Parcerisa*, Antonio González*
* *Universitat Politècnica de Catalunya*
† *Universidad de Murcia*

***Abstract*—The increasing demand for high-quality graphics requires a significant increase in computational power of modern GPUs. The common approach to follow is augmenting the number of compute units (i.e., shader cores). However, this can result in underutilized resources if the workload is not properly balanced. This is particularly challenging in Tile-Based Rendering (TBR) GPUs, the predominant architecture in mobile GPUs, running graphics applications due to limited per-tile workload.**

This work proposes parallel tile rendering to efficiently increase the computational capabilities of TBR GPUs. This solves the problem of not having enough work to utilize the additional compute units but causes memory-intensive applications to underperform due to the increased memory pressure. To this end, we introduce LIBRA, a parallel tile rendering architecture that includes a novel locality-aware approach to schedule tiles to Raster Units to evenly distribute memory requests during the rendering of each frame. This alleviates memory congestion, therefore, reducing memory access time. LIBRA leverages frame-to-frame coherence to predict the memory pressure of each tile of a frame without penalizing the hit ratio of the cache memories. Evaluations over a wide range of commercial gaming applications show that LIBRA reduces the average memory latency by 13.5% and achieves an average speedup of 20.9%. It also provides an 11.4% improvement in throughput (frames per second) and a total GPU energy reduction of 9.2%, while adding negligible overhead.

***Index Terms*—GPU microarchitecture, Graphics, Scheduling, Low-power architectures, Locality.**

I. Introduction

Driven by the increasing demand for high-fidelity animated graphics applications, dedicated or integrated GPUs are now found in a variety of devices, including laptops, smartphones, tablets, smartwatches and AR/VR devices. Users' demand for increased realism is insatiable, and graphics applications use more realistic and sophisticated geometry, lightning and shadowing models in higher-resolution screens along with increased refresh rates year over year. To satisfy all these increasing demands, modern GPUs require more computational power.

In graphics systems, GPUs contain two different parts: a Geometry Pipeline that transforms the scene's geometry, and a Raster Pipeline that computes the final color for each screen pixel from the transformed geometry. To provide an insight of where the cycles go when rendering a 3D scene, Figure 1 shows the breakdown of the execution time in a typical GPU for the set of commercial graphics applications that we have evaluated (see details in Section IV). We can observe that, on average, nearly 88% of the time is spent in the raster process which is the focus of this work.

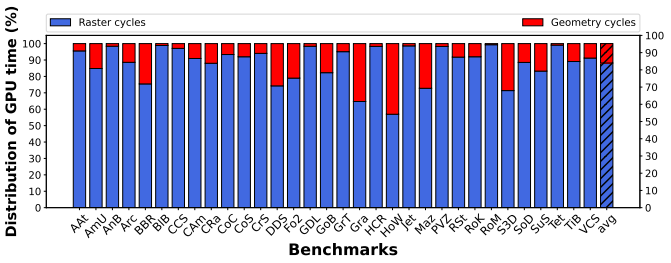


Fig. 1: Distribution of the execution time in the GPU per frame. On average, 88% is spent on the raster process.

Since rasterization dominates the execution time, the most straightforward approach to increase the computing capabilities of modern GPUs and boost their performance is to increase the number of computing units (also known as shader cores). However, the lower levels of the memory hierarchy are shared among these cores and may become a bottleneck, especially in modern gaming applications employing finer geometry (i.e., with more triangles per object) and more detailed and richer textures, leading to a more intensive memory utilization.

To reduce memory pressure, Tile-Based Rendering (TBR) architectures are commonly used for mobile GPUs [63]. They efficiently exploit memory locality by processing small areas of the screen, called *tiles*, one by one to avoid many DRAM accesses by using tile-sized on-chip buffers in charge of storing the intermediate results. Conventional TBR GPUs are limited to processing one tile at a time for simplicity, and all the integrated shader cores are devoted to work on the current tile [4]. Thus, the addition of more shader cores for augmenting the overall GPU computing power can sometimes be ineffective since the limited amount of work in some tiles may result in underutilized resources.

In this paper, we explore a less conventional approach to increase the computing resources in a more effective way. The main idea is to process multiple tiles in parallel. Nevertheless, this is not trivial because this also implies increasing system memory pressure, as more cores are sending requests in parallel, which may lead to a memory bottleneck.

Graphics applications are highly parallel since the computations for each pixel have no dependencies, resulting in the application being composed of a huge number of threads.

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00081

(a) Rendered frame (b) Heatmap of memory accesses

Fig. 2: Example of a rendered frame and the heatmap of the per-tile DRAM accesses for the game Subway Surfers [72].

The shader cores are designed to exploit this feature by being highly multithreaded to increase throughput and hide memory latency. Each shader core has an associated private L1 cache, which is backed by a shared L2 cache and main memory (see Section II-C). Rendering multiple tiles in parallel implies having independent cores that share the access to the memory hierarchy, which may generate congestion in the memory subsystem. This may result in longer latency accesses that cannot be hidden by multithreading which would hurt performance.

In TBR architectures, tiles can be processed in any order. Tiles within a frame are not homogeneous since some regions of a frame have a higher amount of work than others. For instance, it is very common that some areas of the scene contain only background (e.g., the sky) whereas other contain many overlapping objects with a high level of details. Thus, some tiles will perform many more accesses to the LLC and DRAM than others. To better illustrate this effect, Figure 2 shows the heatmap of DRAM accesses generated per each tile for the well-known mobile game Subway Surfers [72]. As it can be observed, we can identify *hot* tiles (performing many memory accesses) that correspond to frame regions where the main character is located, the status bars (aka HUD), and the areas where the fences and coins are located. On the other hand, we can observe *cold* tiles (performing less memory accesses) in other areas of the scene with less details (e.g., the railways, the station roof, and the station stores). This imbalance in the access pattern leads to an opportunity and motivates us to explore novel tile scheduling algorithms based on these observations to ameliorate memory pressure.

This work proposes LIBRA, a *Locality-aware Intelligent Balance Rendering Architecture* that performs parallel tile rendering in such a way that it processes hot and cold tiles concurrently to avoid saturating the memory system. It is well known that the response time of memory increases asymptotically as the utilization factor of the memory bandwidth approaches 100% so, in general, it is better to have a more balanced utilization rather than periods with low utilization followed by others with a very high utilization. On the other hand, processing far away tiles commonly result in a detrimental effect on the locality that can be exploited by the L1 and L2 caches, therefore, LIBRA incorporates a locality-aware mechanism to avoid increasing L1 and L2 cache misses.

To achieve this, we need to have a way to predict the memory pressure that a given tile will generate. Our technique exploits what is known as frame coherence. To create the illusion of movement in animated graphics applications, a high frame rate is required. This results in frames quite similar to their previous one, which is known as *frame-to-frame coherence* [28]. We take advantage of this feature to predict the memory pressure in a given frame based on the collected statistics from the previous one.

To summarize, in this paper we propose to boost GPU performance by first providing an unconventional method for increasing compute resources. To the best of our knowledge, this is the first work exploring parallel tile rendering on GPUs. Our work makes the following key contributions:

- We propose LIBRA, a novel approach to increase the computing capabilities of a GPU by processing multiple tiles in parallel.
- We propose a novel tile scheduler scheme that tries to keep the memory utilization uniformly distributed across the execution time by combining tiles with high and low memory demands, while at the same time not increasing the miss ratio in the private L1 cache and the shared L2 cache.
- We show that LIBRA provides important benefits in terms of performance (20.9% improvement), 11.4% increase in frame rate (frames per second, FPS), and 9.2% decrease in total GPU energy consumption.

The rest of the paper is organized as follows. Section II provides some background on GPUs. Section III presents LIBRA. Section IV describes the evaluation methodology. Section V presents our experimental results and analysis. Section VI reviews some related work. Finally, Section VII summarizes the main conclusions of this work.

II. Background

TBR architectures were originally proposed to smooth parallel rendering since tiles do not overlap in the scene [26], [53]. Nowadays, mobile GPUs normally implement a Tile-Based Rendering (TBR) architecture in order to reduce main memory accesses. This rendering approach is very popular in low-power graphics and memory-bandwidth-limited systems. TBR is characterized by dividing the screen space into a grid of smaller rectangular regions of adjacent pixels, called *tiles*. Tiles are small enough to perform many operations on tile-sized on-chip buffers. Since off-chip main memory is usually

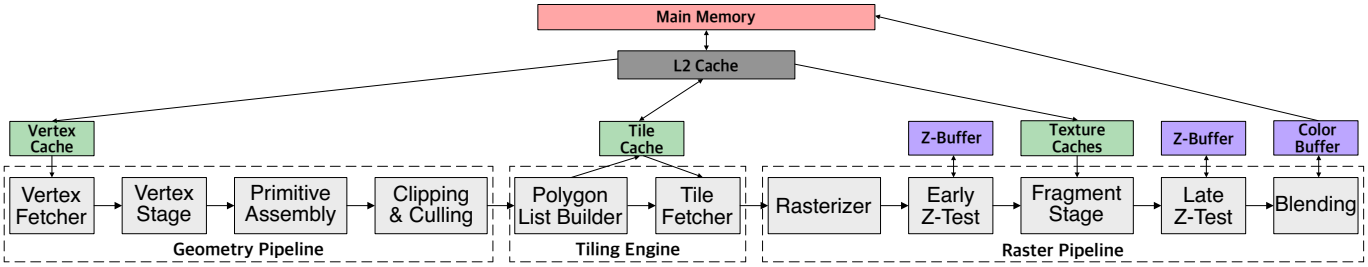


Fig. 3: The Graphics Pipeline of a TBR GPU.

the primary source of power consumption in GPUs [14], [16], [25], these small on-chip buffers significantly reduce these power-hungry accesses to off-chip main memory and reduce memory traffic. For instance, Antochi et al. [5] show that TBR considerably reduces the total amount of external data traffic compared to traditional architectures that are not tile-based, also known as Immediate-Mode Rendering (IMR) GPUs.

A. Graphics Pipeline

Figure 3 depicts the main stages of the Graphics Pipeline and the memory hierarchy organization of a TBR architecture. The rendering process typically has two major pipelines: Geometry and Raster. The Geometry Pipeline performs all the geometry-related operations over the triangles that compose the objects in the scene and generates all the corresponding *primitives* that fall into the visible frustum. Then, the Raster Pipeline discretizes each primitive into pixel-sized *fragments* which are then shaded and blended to produce their final output color for the final screen image.

In TBR architectures, the Raster Pipeline renders tiles rather than the full frame to keep an important number of memory accesses on chip by increasing locality. To make possible this tiling process, all the geometry is sorted into sub-regions to be later processed by the rasterization stages. Different sorting taxonomies exist, but TBR is classified in the literature as sort-middle [2], [53]. These architectures rely on an intermediate phase where the tiling process is carried out, called *Tiling Engine*. Thus, TBR architectures have three main pipelines, as shown in Figure 3.

The Geometry Pipeline is triggered by *draw calls*, which are commands that demand for the rendering of a batch of objects. The Vertex Fetcher fetches the objects' vertices from memory. Then, the Vertex Processors transform these vertices by executing a user-defined *vertex shader* program. Once processed, these vertices are taken in program order and assembled to generate different polygons (usually triangles). Afterward, for each primitive, it is determined whether the primitive lies within the frustum view, according to the camera's point of view. This Culling process discards the triangles that are detected to be entirely outside of this viewing volume. However, in case a triangle is partially visible, a Clipping operation is applied, in which the primitive is split into smaller triangles and only those that entirely fall inside this visible region are kept. The resulting primitives are the input data to the Tiling Engine.

The Polygon List Builder is in charge of binning the primitives into tiles, i.e., to produce a list in program order for each tile with all the primitives that totally (or partially) fall inside it. These per-tile primitive lists are stored in a main memory region called *Parameter Buffer*. Once all the geometry has been processed and binned into tiles, the Tile Fetcher starts working in a tile-by-tile fashion, fetching the primitives that belong to the current working tile which are served as inputs to the Raster Pipeline.

The Raster Pipeline renders tiles sequentially one after another. The Rasterizer determines the pixels that are overlapped by each primitive in the current tile and discretizes each primitive into a set of *fragments*. In addition, the Rasterizer interpolates the values of the primitive's attributes. Fragments are assembled into groups of 2x2 adjacent fragments to form *quads* which are sent to the Early Z-Test stage. This stage aims to eliminate fragments that are known to be occluded by a previously processed one. This is accomplished by employing a tile-sized on-chip buffer called *Z-Buffer* that stores the depth value of the closest fragment processed for each tile's pixel position so far. The *presumably* visible quads proceed to the Fragment Stage, where the shader cores compute the color for each fragment by executing a user-defined *fragment shader* program that provides the corresponding lightning model and textures. Finally, output colors are processed by the Blending Unit to properly combine them with the ones already in the same position in the *Color Buffer*, and allows to achieve transparency effects. In some situations, the shader cores may need to modify the depth values of the fragments, in which case the Early Z-Test is disabled and the visibility test is performed after shading (Late Z-Test stage).

Finally, once all the primitives in the current tile have been completely rendered, the content of the Color Buffer is flushed to the *Frame Buffer*, a region in main memory used to hold the data that will be displayed in the screen. Therefore, the Color Buffer is entirely written into main memory once for each tile. After all the tiles of a frame have been processed, the frame is ready to be displayed. Recall that both the Z-Buffer and Color Buffer are tile-sized, which means that they can be held in on-chip memory, thus, significantly reducing accesses to off-chip DRAM memory.

B. Tile Scheduling

As mentioned previously, the Tile Fetcher is in charge of fetching the primitives corresponding to each tile on a frame – one tile at a time – that are stored in the Parameter Buffer. The tile's primitives are pushed onto a FIFO queue for the Raster Pipeline to consume. The next tile to process is selected in an order specified by the Tiling Engine. Note, however, that tiles can be processed in any order as they are independent. In any case, primitives within each of these tiles need to be processed in program order to guarantee correctness.

The most common tile traversal orders in computer graphics are scanline and Morton order [56]. Scanline follows a row-major order, while Morton order follows a Z-shaped pattern. Although Morton order is more complex, it is considered more cache-friendly as it helps improve spatial locality. For this reason, we assume the Morton order (or Z-order) as the one used in the baseline GPU of this work.

C. Memory Organization

Figure 3 also depicts the memory hierarchy of a TBR GPU. There are multiple L1 caches to store geometry (Vertex cache and Tile cache) and textures (Texture caches), which are connected to a shared on-chip LLC. This L2 cache is, in turn, connected to the off-chip main memory. There are also local on-chip memories that store the Z-Buffer and Color Buffer for the tile being processed. Note that the Color Buffer directly transfers its content to main memory when all the current tile's primitives have been rendered. However, the content of the Z-Buffer does not need to be written to main memory.

III. LIBRA: Locality-aware Intelligent Balance Rendering Architecture

In this paper we propose LIBRA, a parallel tile rendering architecture that employs a novel temperature-based tile scheduler to make a more effective utilization of the computing resources and improve GPU performance. This is achieved by processing in parallel hot and cold tiles to properly balance the memory accesses and avoid saturating the memory system along the rendering of a frame. In essence, LIBRA ameliorates the congestion in memory due to the increased parallelism by keeping the memory utilization uniformly distributed by combining tiles with high and low memory demands, while not hurting the spatial locality in the L1 and L2 caches.

As mentioned above, to increase the performance and reduce the energy consumption of the GPU, LIBRA relies on rendering multiple tiles in parallel for a given number of shader cores. However, this sole approach substantially increases the pressure over DRAM, so one of the main goals of our proposal is to reduce DRAM contention. This is achieved by smartly choosing the order in which tiles are processed, in such a way that we smooth the DRAM memory bandwidth required throughout the rendering of each frame. This tile order executes tiles with high memory demands (hot tiles) concurrently with other that exhibit low memory demands (cold tiles), but at the same time it is important that the chosen tile order does not penalize the locality in the L1 and L2 caches, otherwise, this would result in further DRAM requests.

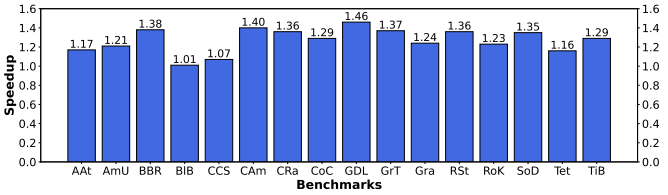


Fig. 4: Speedup when doubling the number of cores in a Raster Unit from 4 to 8.

A. Parallel Tile Rendering

A conventional manner to increase the computing power is to simply add more shader cores, since the Fragment stage is the main bottleneck of the Raster Pipeline. Remember that, to guarantee the semantics set by the programmer, there are barriers between stages, so a tile cannot proceed to a given stage until the preceding tile has completed that stage. Because of that, adding more cores to accelerate the Fragment stage may be ineffective for tiles that do not have enough work to keep all cores busy for most of the time.

To provide a better insight about this issue, Figure 4 shows the speedup when increasing the number of shader cores in the Raster Unit from 4 to 8. It can be observed that doubling the number of cores does not work well for many of the applications in our benchmark suite. For the sake of visibility, the plot only shows the benchmarks whose speedup is lower than 1.50. It must be noted that a considerable number of benchmarks suffer this condition, 16 out of 32, which are the ones reported in Figure 4. Also note that some benchmarks (such as BlB and CCS) exhibit speedups even smaller than 1.10, despite doubling the number of computing cores, quite far from the ideal 2x speedup.

To tackle this poor scalability observed in the more memory-intensive applications, LIBRA seeks to increase the overall GPU performance by exploiting a less conventional approach based on rendering multiple tiles in parallel, to avoid having idle cores. In other words, at the same time we increase the number of cores, we also increase the amount of work (i.e., number of tiles) to be processed concurrently. For illustration purposes, in the rest of the paper we assume that LIBRA will render two tiles in parallel, i.e., it will feature two Raster Units.

Figure 5 depicts the main blocks of the proposed parallel tile rendering (PTR) architecture. Note that one input FIFO queue is required for each Raster Unit to allow them to progress at their own pace. These FIFO queues store a primitive in each entry, taking into account that all the primitives of a given tile must be rendered in the same Raster Unit to maintain the program order among overlapping primitives.

Since the Tile Fetcher of the proposed PTR architecture has several FIFO output queues, a scheduler is needed to select which Raster Unit will process each tile. The most straightforward schedule mechanism is to use an interleaved

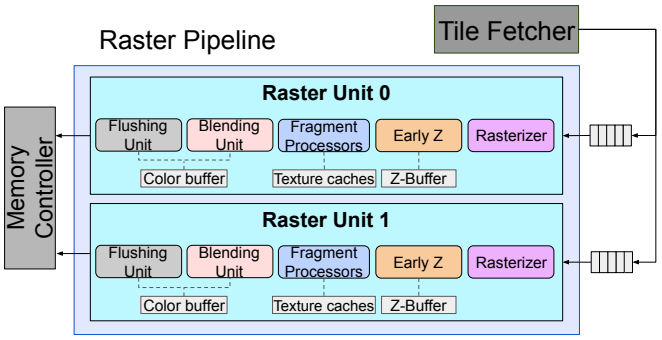
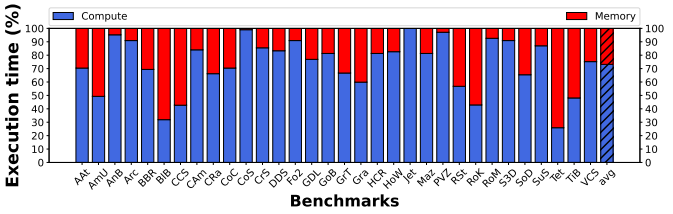
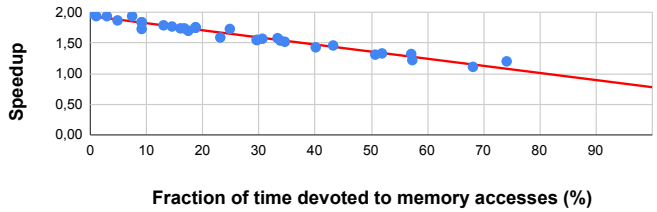


Fig. 5: Architecture of the Raster Pipeline to enable parallel tile rendering. Each Raster Unit has its own private resources.



(a) Breakdown of the execution time between compute and memory phases.



(b) Correlation between speedup and memory intensiveness of the evaluated benchmarks. The X-axis represents the fraction of time spent on memory accesses.

Fig. 6: Execution time breakdown and correlation between the speedup and memory tasks. It can be observed that various benchmarks have a significant memory activity that correlates with the poor speedup obtained.

tile assignment, following the original tile order assumed for our baseline GPU (Z-order). I.e., the Tile Fetcher fetches tiles in the predefined order which are dispatched to a Raster Unit in an alternating manner. This basic scheduling of tiles provides a good workload balancing across the two Raster Units.

Parallel tile rendering has the potential to be more effective than single tile rendering for the same total number of cores. However, due to the increase in memory pressure in the shared L2 and main memory, memory becomes the main bottleneck for some applications, and the performance gain is not even close to the expected 2x for several graphics workloads. Figure 6.a shows the fraction of the execution time that is spent on memory accesses for all applications in our benchmark suite. This graph has been obtained by simulating the application with an ideal memory system (whose accesses always hit in the L1 caches) and simulating it again with a realistic memory configuration. The difference in execution time between the two configurations is due to the memory activity, which is depicted in red in this graph. On the other hand, Figure 6.b shows the speedup of a system with two Raster Units over a system with just one, as a function of the percentage of execution time that is spent on memory. We can see that these two metrics are strongly correlated. The more memory-intensiveness the less speedup, which confirms that memory is the main bottleneck to fully exploit parallel tile rendering. For the rest of this paper, we consider a benchmark as memory-intensive when at least 25% of its execution time is spent on memory accesses.

Tiles are highly heterogeneous in the sense that some have a few simple primitives to render whereas others contain very complex geometry. This translates into very different requirements in terms of computation and memory accesses. Another observation is that tiles with similar characteristics tend to appear spatially clustered (as illustrated in Figure 2). E.g., if there is a screen region where multiple objects overlap, all the neighboring tiles where these objects fall upon will have a similar amount of primitives, fragments, texture accesses, etc. Therefore, if we want to avoid rendering simultaneously two tiles with high memory requirements, it may be good to avoid rendering adjacent tiles at the same time. However, it is true that nearby tiles tend to share more textures than far apart ones, so rendering distant tiles concurrently may hurt locality.

In conclusion, deciding which tiles are rendered in each Raster Unit and in which order has a very important impact on the memory utilization, and thus it is a key component of LIBRA, which is described in detail next.

B. *Temperature-aware Tile Scheduler*

The second main component of LIBRA is a novel tile scheduler that minimizes DRAM pressure while maintaining data locality, and balancing the workload of the Raster Units. Note that in TBR GPU architectures there are four sources of DRAM memory accesses. They correspond to geometry accesses (i.e., to fetch the geometric description of the scene), accesses to the Parameter Buffer (a data structure that enables the tiling of the frame), texture accesses (images mapped onto object surfaces to add high-frequency details), and accesses to the Frame Buffer (which holds the final colors of the entire frame). Since LIBRA aims to reduce the memory pressure during the Raster Pipeline, geometry-related accesses are ignored.

As discussed previously, tiles within a frame are diverse due to varying characteristics of the scene across different regions. To quantitatively illustrate this, Figure 7 shows the number of DRAM requests generated in intervals of 5000 cycles during the execution of a frame of the popular game Candy Crush (CCS). It can be observed that there are certain intervals which are much more memory-intensive than others. To achieve a more homogeneous memory activity across the whole execution, LIBRA tries to overlap the rendering of tiles with the highest memory demands with those with the

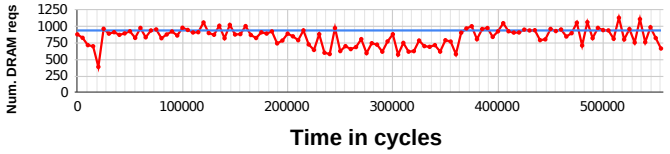


Fig. 7: Number of main memory requests during the execution of a frame of Candy Crush in intervals of 5000 cycles.

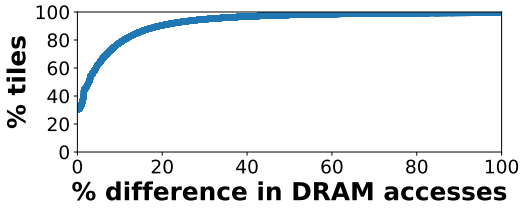


Fig. 8: Cumulative per-tile DRAM accesses difference for consecutive frames.

lowest demands. This way, it avoids intervals with an excessive number of simultaneous DRAM requests. To that end, LIBRA devotes one Raster Unit for processing the hot tiles, and the other one to process the cold tiles. The main goal is to avoid processing two high-demanding tiles simultaneously, since the response time of memory increases exponentially as the utilization factor of it augments. It is important to note that tiles are completely independent and can be processed in any order, unlike primitives that must be processed in program order in each tile for correctness (since they may overlap). This fact is exploited by LIBRA's novel tile scheduler.

Let us describe next how LIBRA is able to determine hot and cold tiles in a given frame. First, recall that animated applications exhibit a high degree of frame-to-frame coherence to enhance user experience, i.e., consecutive frames are normally very similar. Figure 8 shows the cumulative difference in DRAM accesses of the same tile for several consecutive frames averaged over our entire benchmark suite. It can be seen that more than 80% of the tiles have a difference lower than 20%, which confirms the high degree of frame-to-frame coherence between two consecutive frames.

LIBRA exploits this coherence to predict the next frame's behavior. In particular, it counts the number of DRAM accesses and instructions in each tile of a frame and use this information to predict the hot and cold tiles in the next frame. We define the temperature of a tile (a proxy for memory intensity) as the ratio of DRAM accesses over the number of instructions, and arrange the tiles from highest to lowest temperature (i.e., DRAM request frequency). This requires a small table sized to the number of tiles in a frame, which depends on the screen resolution but typically are just a few thousands (see implementation details in Subsection III-E). Once the tiles have been ranked based on their temperature, the Tile Fetcher dispatches them to the different Raster Units (RUs). E.g., if two RUs are used, one RU will be dedicated to processing the hot tiles (whose IDs are obtained from the top of the ranked table) whereas the other RU will process the cold tiles (using IDs from the bottom of the table).

C. Supertiles

Texture data locality is key for GPU efficiency, assuming there are no bottlenecks in other stages of the pipeline. However, scheduling tiles solely based on their memory access temperature may lead to processing the frame by performing jumps through distant tiles, which results in losing the natural locality that exists when processing nearby tiles. To address this divergence, we propose to assemble tiles in squared groups of tiles, which we refer to as *supertiles*. For example, a 4x4 supertile covers a rectangular region of 16 adjacent tiles. The Tile Fetcher assigns a particular supertile to a Raster Unit, so its corresponding tiles will be scheduled to that Raster Unit one after another. This way, LIBRA is capable of retaining texture locality inside a Raster Unit (thanks to using supertiles) while also managing to reduce replication in the rest of Raster Units (since different Raster Units will process distant frame areas). In addition, reducing block replication in the L1 texture caches potentially increases their locality, as the aggregated effective cache capacity is increased. To better illustrate the utilization of supertiles, Figure 9 shows the popular game Hill Climb Racing (HCR) [24] when the Raster Unit is scheduled at tile and supertile level. We can identify that adjacent tiles tend to reuse textures (e.g., the texture of the ground or the coins) and that hot regions tend to cover several adjacent tiles.

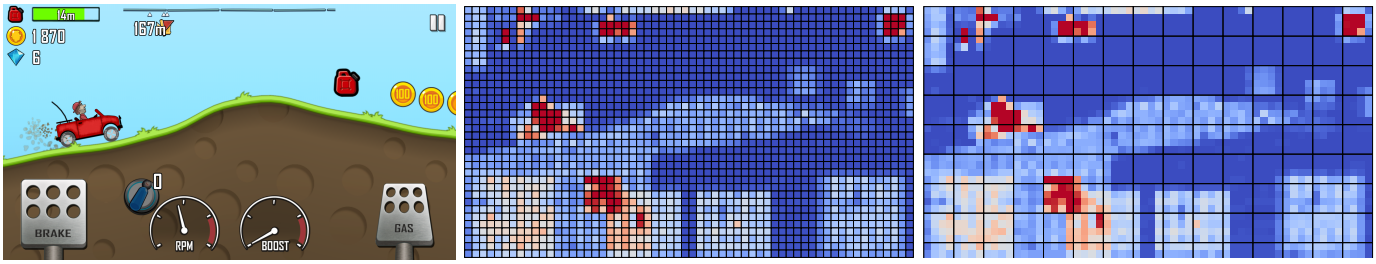
Supertiles can be of any size, providing enormous potential for exploring different alternatives. In this work, we limit the options to supertile sizes that are powers of two: 2x2, 4x4, 8x8, and 16x16. Larger values would cover almost the entire screen and would be ineffective in preventing main memory access peaks. The supertile size is dynamically chosen at run time for each application depending on its characteristics, as described in next subsection.

D. Adaptive per-Frame Scheduling

Each graphics application exhibits its own characteristics, traversing different application phases as the execution progresses. Moreover, even the same application commonly shows different computing demands on different frames. Consequently, an adaptive mechanism is needed to determine the most adequate tile scheduling approach.

Our proposed scheduler leverages frame coherence to implement this adaptability based on the last frame's characteristics. This dynamic scheduler must be able to react to scene changes from one frame to the next. Based on this, it will determine the order in which tiles will be processed in the current frame, either a temperature-aware order or the conventional Z-order; and second, it must determine the supertile size to be used in the current frame (if temperature-aware order is chosen).

Determining the tile traversing order. LIBRA can schedule tiles using two different ordering schemes: following the conventional Z-order, or following the proposed temperature-aware order. The decision to select one ordering or the other



(a) Rendered frame (b) Tile-level scheduling heatmap (c) 4x4 supertile-level scheduling heatmap

Fig. 9: Hill Climb Racing [24] frame and its heatmap of memory accesses when considering either a tile-level or a supertile-level scheduling. Nearby tiles tend to employ similar textures, and hotspots cover a cluster of neighboring tiles.

is based on two metrics: performance (i.e., the number of cycles spent on the Raster Pipeline) and the locality achieved by the L1 caches in the previous frame. For the latter, we use the texture caches' hit ratio as a proxy since texture accesses constitute the main source of DRAM requests. If the hit ratio is sufficiently high, it is unlikely to have congestion in main memory, and the temperature-aware order is disabled. Our experimental evaluation showed that a threshold of 80% provides good results. That is, if the hit ratio of the texture caches in the previous frame exceeded this threshold, the Z-order will be used to rasterize the current frame.

Performance is the other metric used to determine the tile ordering scheme to follow. To do that, LIBRA compares the cycles spent on the Raster Pipeline for a given frame with those of the previous frame. According to frame-to-frame coherence, consecutive frames should account for a similar number of cycles. Note, however, that decisions regarding the tile ordering are only taken when a significant performance variation is detected. For that, we define another threshold value to detect a significant performance variation. Based on our experimental evaluation, this threshold has been set to 3%. Therefore, a performance variation higher than this threshold switches the tile ordering scheme (from temperature-aware to Z-order, or vice versa).

Finally, we experimentally found out that following Z-order when the hit ratio is high, or the temperature-aware order otherwise, does not always achieve the optimal performance. For some benchmarks, a temperature-aware order is more beneficial than Z-order, even if the hit ratio threshold is exceeded. This scenario is detected when both the hit ratio and performance degrade with respect to the previous frame, in which case, the alternative ordering scheme is chosen for the following frames.

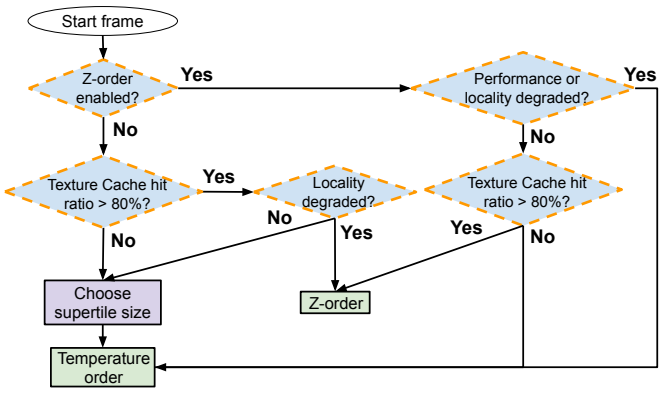


Fig. 10: Dynamic algorithm to determine the tile order.

To provide an overall picture of this adaptive scheme, Figure 10 shows a block diagram of the algorithm used to determine the tile traversing order for the current frame.

Determining the supertile size. As explained in Section III-C, our approach works at a supertile granularity to avoid hurting the locality of textures. The size of these supertiles is dynamically determined on a per-frame basis as well. The resizing policy begins with a predetermined supertile size, which is gradually increased in subsequent frames while performance keeps improving. Otherwise, the supertile size is decreased (in successive frames) until performance is degraded, after which the adaptive policy switches back to increase the supertile size. This way, supertiles are dynamically resized according to the application characteristics for each frame. To avoid unnecessary size changes, a threshold value is used to decide when supertiles must be resized. According to our empirical analysis, a performance variation of more than 0.25% provides the best results. As mentioned before, we have considered supertile sizes of 2x2, 4x4, 8x8 and 16x16 tiles. Note, however, that tiles within a supertile are always traversed in Z-order.

Once the supertile size is set, if the selected tile ordering scheme for the current frame is our temperature-based one, supertiles must be ranked from hottest to coldest based on their average number of DRAM accesses per instruction, as described in Subsection III-B. To do that, the per-tile memory accesses and instruction count metrics of the previous frame are first aggregated at the chosen supertile granularity. Further implementation details can be found in Subsection III-E, but note that the ranking operation is completely done in parallel with the Geometry stages (as we will show) and no timing overhead is introduced by LIBRA. Once the Geometry Pipeline has finished, the Tile Fetcher is ready to dispatch supertiles to the different Raster Units (RU), alternatively assigning a hot supertile (from the top of the ranking) to one

RU, and a cold supertile (from the bottom of the ranking) to the other RU, to ensure that memory accesses and memory bandwidth are properly balanced.

Summarizing, LIBRA introduces an adaptive tile scheduling approach that distributes main memory accesses more evenly across the raterization of each frame. By implementing an adaptive locality-aware scheduler, our approach can effectively manage and prevent a high memory bandwidth demand that could potentially overload the memory system. This is achieved with a negligible overhead, as described next.

E. Hardware Implementation

LIBRA is designed to require minimal hardware overhead. In terms of storage, it only needs a small on-chip buffer to store the number of instructions and the number of DRAM accesses performed per supertile.

Each entry of the buffer holds two additional values to store the accesses per instruction and the supertile ID, used for the ranking operation. Overall, 16 bits are used for the number of memory accesses, 24 bits for the instruction count, 15 bits for the calculated accesses per instruction, and 9 bits for the supertile ID, making a total of 64 bits per entry. Since in our experimental evaluation we employed a FHD screen, a total of 510 2x2 supertiles cover the entire frame. Thus, the buffer only needs at most 510 entries (less for larger supertiles) which corresponds to a storage cost of about 4KB, and represents less than 0.2% of the L2 area.

The dynamic algorithm also incurs minimal storage overhead. It just needs four counters to store the number of cycles and the texture caches hit ratio of the last two frames, while the logic is implemented with a small FSM.

Finally, when the temperature-based tile order is chosen, supertiles must be ranked based on their average number of DRAM accesses per instruction. Next, we provide a timing overhead estimate assuming an ordering cost of $O(n \log n)$. The logic sequentially compares pairs of values which are swapped when needed. For each of the $O(n \log n)$ comparisons (i.e., 4587 for $n = 510$), two reads are needed followed by two potential writes. Assuming a very conservative approach where the two reads to the on-chip buffer take one cycle, the comparison another cycle, and the potential writes another cycle, it leads to an upper bound of $3 \times 4587 = 13761$ cycles to complete the ranking operation. To put this number in context, we have measured that a frame requires 270000 cycles on average just for the Geometry stages and for the evaluated benchmarks. Therefore, the latency of the ranking operation needed by the temperature-based tile order can be totally hidden, as it is done in parallel with the Geometry Pipeline. Similarly, the energy overhead introduced by LIBRA is negligible since only a single comparator and a fixed-point divisor are required.

IV. Evaluation Methodology

A. Simulation Infrastructure

To evaluate our proposal we have employed TEAPOT [10], a cycle-accurate GPU simulation framework that allows running unmodified Android applications and assesses the performance and energy consumption of the modeled GPU and the memory system. In order to do that, TEAPOT relies on well-known tools such as McPAT [48] for the GPU energy estimation, and DRAMsim3 [47] to model the timing and energy consumption of DRAM and memory controllers. Table I shows the parameters employed in our simulations, which model an architecture closely resembling a modern ARM Valhall mobile GPU [6], [7]. The baseline architecture comprises a single Raster Unit with eight cores, while LIBRA distributes the eight cores across two Raster Units, each containing four cores. Each shader core has a private Texture cache.

TABLE I: GPU simulation parameters.

	Global Parameters	
Tech Specs	800 MHz, 1V, 22nm	
Screen Resolution	1920x1080 (Full HD)	
Tile Size	32x32 pixels	
	Main Memory (LPDDR4)	
Tech Specs	1.2 GHz, 1.2V	
Latency	50-100 cycles	
Size	8 GB	
	Caches	
Vertex Cache	64-bytes/line, 4KB, 2-way, 1 cycle	
Tile Cache	64-bytes/line, 32KB, 4-way, 2 cycles	
Texture Cache (per core)	64-bytes/line, 32KB, 4-way, 2 cycles	
L2 Cache (shared)	64-bytes/line, 2MB, 8-way, 18 cycles	
	Baseline	**LIBRA**
Raster Units	1	2
Cores per Raster Unit	8	4

B. Benchmarks

In order to provide confident results for the evaluation of LIBRA, we have selected a wide range of commercial Android graphics applications as benchmarks. The choice criteria for these games is based on variety, to cover a wide diversity of benchmarks, and also on their popularity, determined by the number of downloads in the Google Play Store.

Table II shows the set of benchmarks used to evaluate our proposal. We evaluated sequences of 25 frames, but results remain consistent with larger frame sets. We cover games in 2D (e.g. CCS), 2.5D (e.g. CoC), and 3D (e.g. SuS). In addition, there is a lot of variation when it comes to the average memory footprint per frame in different games. The average footprint for all the benchmarks is more than 4MB, but some of them, such as HoW and RoM, have much higher memory requirements. Conversely, games like CrS and Jet have a more modest memory footprint.

V. Experimental Results

In this section we first evaluate the effects of LIBRA in terms of performance, memory latency and energy with respect to a conventional GPU that has the same number of shader cores but in a single Raster Unit (i.e., does not perform parallel tile rendering). We also provide a detailed analysis of where the benefits of LIBRA come from.

As mentioned previously, we have classified applications into two categories, memory-intensive and compute-intensive

TABLE II: Evaluated benchmarks.

Benchmark	Alias	Genre	Type	Downloads (millions)	Footprint (MB)	Benchmark	Alias	Genre	Type	Downloads (millions)	Footprint (MB)
Air Attack	AAt	Action	2.5D	10	2.1	Gravity Tetris	GrT	Puzzle	3D	5	0.8
Among Us	AmU	Action	2.5D	500	3.4	Gravity: Don't Let Go	Gra	Arcade	3D	1	5.2
Angry Birds	AnB	Puzzle	2D	100	1.1	Hill Climb Racing	HCR	Racing	2D	1000	2.8
Archery Master 3D	Arc	Sports	3D	100	2.3	Hot Wheels: Race Off	HoW	Racing	2.5D	50	8.3
Beach Buggy Racing	BBR	Racing	3D	10	7.5	Jetpack Joyride	Jet	Arcade	2D	100	0.7
Block Blast!	BlB	Puzzle	2D	100	3.6	3D Maze / Labyrinth	Maz	Adventure	3D	10	3.6
Candy Crush Saga	CCS	Casual	2D	1000	2.6	Plants vs. Zombies	PVZ	Strategy	2D	500	2.7
Captain America: Sentinel of Liberty	CAm	Action	2.5D	5	1.5	Real Steel World Robot Boxing	RSt	Action	3D	50	4.9
City Racing 3D	CRa	Racing	3D	50	3.5	Rise of Kingdoms: Lost Crusade	RoK	Strategy	2.5D	50	7.1
Clash of Clans	CoC	Strategy	2.5D	500	2.4	Royal Match	RoM	Puzzle	2.5D	100	27.5
Counter Strike	CoS	Action	3D	50	0.7	Sniper 3D: Gun Shooting Games	S3D	Action	3D	500	6.0
Crazy Snowboard	CrS	Sports	3D	15	0.9	Sonic Dash	SoD	Arcade	3D	100	4.8
Derby Destruction Simulator	DDS	Racing	3D	10	3.2	Subway Surfers	SuS	Arcade	3D	1000	2.9
Forest 2	Fo2	Adventure	3D	1	4.7	Tetris	Tet	Puzzle	2D	10	9.4
Geometry Dash Lite	GDL	Rhythm	2D	100	1.2	Tigerball	TiB	Casual	3D	10	6.5
Golf Battle	GoB	Sports	3D	50	4.9	Vegas Crime Simulator	VCS	Action	3D	10	4.3

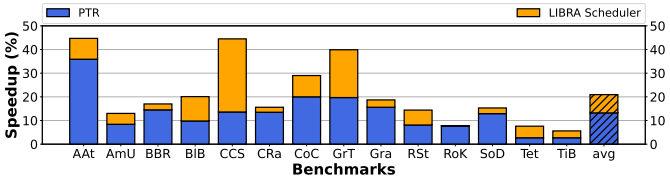


Fig. 11: Speedup of LIBRA w.r.t. the baseline GPU for the memory-intensive applications.

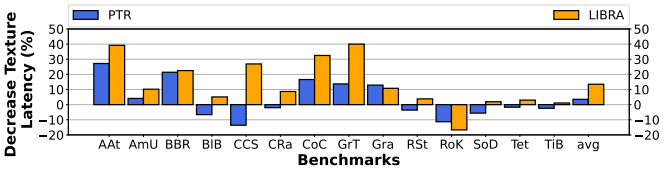


Fig. 12: Decrease in texture latency w.r.t. the baseline.

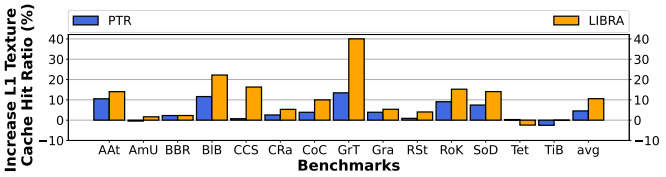


Fig. 13: Increase in overall GPU texture cache hit ratio w.r.t. the baseline.

ones. The former are those applications with important memory activity (at least 25% of the execution time spent on memory accesses); the rest are classified as compute-intensive. We initially focus on memory-intensive applications since LIBRA is specially designed to optimize memory-intensive applications. At the end of this section we discuss the impact of LIBRA on compute-intensive applications.

A. *LIBRA Results*

1) Performance. Figure 11 shows the speedup achieved by LIBRA with respect to the baseline GPU. Overall, we obtain an average speedup of 20.9% for the evaluated benchmarks. To better understand the contribution of each one of the two components of LIBRA, the blue segments represent the speedup achieved solely by using two Raster Units in a parallel tile rendering setup (PTR), while the orange segments correspond to the additional speedup achieved by the novel memory-bandwidth- and locality-aware scheduler. It can be seen that PTR alone obtains an average speedup of 13.2%, whereas the adaptive scheduler contributes with a significant extra 7.7% improvement.

Employing a PTR architecture on its own can significantly improve performance for many applications over doubling the number of cores in a single Raster Unit. For instance, AAt improves its performance by 35.9%. However, by adding the proposed adaptive tile scheduler, LIBRA is capable of boosting performance even more for all the applications, achieving up to an extra 31% for CCS and 20.2% for GrT (leading to a total speedup of 44.5% and 39.9%, respectively). Note that this is a significant improvement since performance is critical in real-time rendering, which leads to an average 11.4% increase in frame rate (FPS) with negligible overhead.

On the other hand, it can be observed that some benchmarks (such as Gra or RoK) do not obtain that much benefit in performance from the adaptive scheduler. This is highly correlated with the poor shader core utilization when processing each tile. Therefore, for some applications it is difficult to hide long latencies due to their low workload and high miss ratio.

2) Texture Latency. As mentioned earlier, the Fragment Stage is usually the bottleneck in the Graphics Pipeline due to the complexity of the shader programs and their memory demands when accessing textures. Long-latency operations like cache misses render a warp blocked in the shader cores. Therefore, reducing texture access latencies is crucial for not slowing down performance.

To provide a better insight of the contribution of LIBRA's adaptive scheduler, in this case we present separate bars to differentiate results from PTR alone and from LIBRA (PTR with the scheduler). Figure 12 shows the decrease in texture access latency compared to the baseline. The blue bars denote the latency decrease obtained by employing just a PTR architecture, whereas the orange bars show the latency decrease achieved by LIBRA. It can be observed an average decrease of 13.5%. However, PTR alone increases latency for some benchmarks since it is not able to properly face memory congestion periods. On the other hand, LIBRA achieves significant reductions compared to both baseline and PTR alone. This shows the effectiveness of the proposed adaptive scheduler which can provide latency reductions of up to 40%.

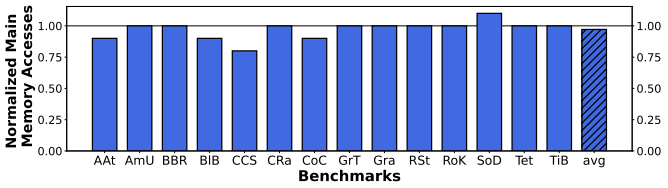


Fig. 14: Normalized main memory accesses w.r.t. PTR alone.

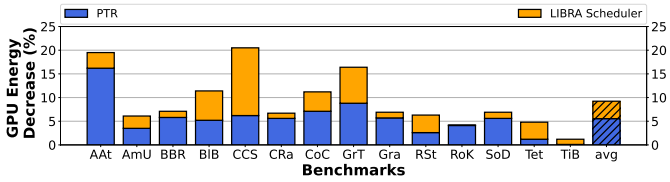


Fig. 15: Decrease in GPU total energy w.r.t. the baseline.

3) Texture Locality. Figure 13 shows the hit ratio increase for the overall texture caches. As before, we present the results in separate bars differentiating PTR alone from LIBRA. It can be seen an average hit ratio increase of 10.6% compared to the baseline, with some benchmarks achieving up to 40%. Regarding block replication within the Texture caches, we have observed average reductions of 32.5% compared to PTR alone, but we have seen that there is no correlation between an increase in the texture cache hit ratio and a reduction in texture block replication. Note also that an increase in the hit ratio does not necessarily translate to an increase in performance since there could be a bottleneck in other stages of the pipeline. We have also observed some decrease in the hit ratio for the Tile cache, but the Tile Fetcher is not a bottleneck and still can sustain the throughput to feed all the Raster Units in the Raster Pipeline.

4) Main Memory Accesses. Figure 14 plots the main memory accesses generated by the Raster Pipeline for a GPU integrating LIBRA normalized with respect to a GPU with PTR alone in order to evaluate the benefits of LIBRA's scheduler. As expected, there is no significant reduction in the number of DRAM accesses as it is not the design goal for the adaptive scheduler. Still, some applications show a noticeable reduction in their main memory accesses (up to 20% for CCS).

Note, however, that the benefit from LIBRA's scheduler does not come from locality improvement but from properly balancing main memory requests over time. For instance, take GrT as an example. We can observe that the adaptive scheduler provides significant benefits (almost 20%) while main memory accesses remain constant. This shows the effectiveness of LIBRA in evenly distributing the memory load.

5) Total GPU Energy. Figure 15 shows the total GPU energy decrease with respect to the baseline GPU. Overall, an average energy decrease of 9.2% is achieved by LIBRA. Again, the blue part represents the decrease achieved by PTR alone, which averages 5.5%. On the other hand, the orange part corresponds to the additional energy savings achieved by the adaptive scheduler. We can observe that the LIBRA's scheduler contributes an additional 3.7% in energy savings. For several benchmarks we observe impressive energy reductions, e.g., AAt and CCS achieve up to 19.5% and 20.5%, respectively. Overall, we can observe that the adaptive scheduler achieves significant savings for many applications. Notice that energy efficiency is crucial for mobile GPUs, and this is achieved with a negligible hardware cost.

B. Performance Provided by Supertiles

This subsection evaluates the effect of only using supertiles while deactivating the temperature-based order, in order to observe how applications behave under this scenario.

Figure 16 shows the speedup with respect to the baseline GPU with two Raster Units (i.e., a conventional PTR architecture) considering static supertile sizes of 2x2, 4x4, 8x8, 16x16 in all the frames, plus the speedup achieved by LIBRA that implements a dynamic supertile resizing mechanism. It can be observed that LIBRA outperforms the static supertiles for most of the cases. In fact, for some applications (such as AmU, BlB, CCs, and GrT) the difference is significant. On average, static supertiles of 2x2, 4x4, 8x8, and 16x16 yield speedups of 0.6%, 2.1%, 2.8% and 3.2%, respectively, while LIBRA achieves about 7%. We have measured that half of the benefit from LIBRA's scheduler comes from employing a dynamic supertile resizing scheme whereas the remainder comes from the tile traversal ordering. We can also observe some benchmarks (e.g., BBR, Gra, RoK) for which a fixed supertile size outperforms LIBRA. For these applications locality matters more than memory congestion.

Summarizing, supertiles allows us to recover the lost locality from the temperature-based scheduling. The combined effect of both mechanisms efficiently balances memory requests along frame execution.

C. Compute-intensive Applications

As mentioned earlier, results reported in the previous section correspond to benchmarks classified as memory intensive (i.e., with at least 25% of their execution time spent on memory accesses) since LIBRA's scheduler can only obtain benefits on applications with some degree of memory activity. For completeness, we analyze here the impact of LIBRA on compute-intensive benchmarks, with low memory activity, to show that our scheduler does not harm their performance. Figure 17 shows the breakdown of the obtained performance. As before, the blue parts indicate the speedup provided by PTR alone, whereas the orange parts show the speedup introduced by the adaptive scheduler.

Overall, it can be observed an average increase in performance of 11.6% for these applications. Most of it (9.9%) corresponds to the benefit achieved by just using a conventional PTR architecture, whereas the remaining 1.7% comes from the proposed tile scheduler. This low performance improvement coming from LIBRA's scheduler is expected, as these applications do not put as much pressure on the memory hierarchy. However, some compute-intensive benchmarks still show

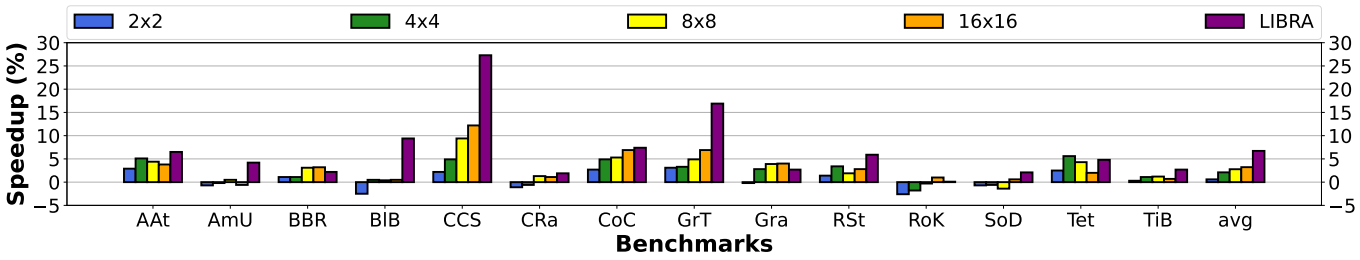


Fig. 16: Speedup obtained by static supertiles and LIBRA w.r.t. PTR alone.

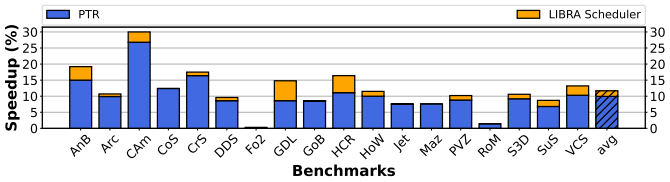


Fig. 17: Speedup obtained w.r.t. the baseline GPU for the compute-intensive applications.

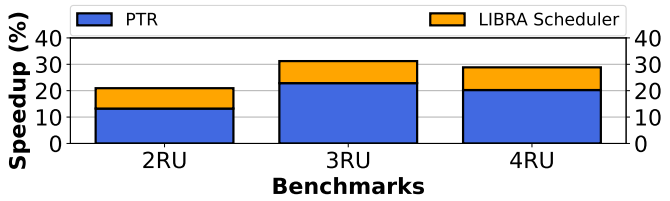
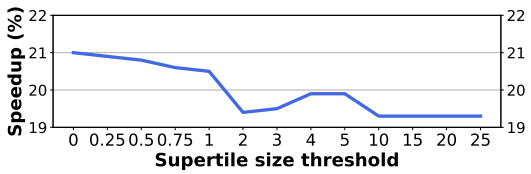


Fig. 18: Speedup of LIBRA w.r.t. a baseline GPU with a single Raster Unit comprising an equal number of cores.



(a) Threshold for supertile size.

(b) Threshold for tile ordering.

Fig. 19: Speedup of LIBRA w.r.t. the baseline GPU when varying the thresholds employed by the LIBRA's scheduler.

considerable performance improvements from our adaptive scheduler (e.g., GDL achieves gains higher than 5%) which shows that our scheduler is capable of smoothing periods of memory congestion even if they are not the common case in these compute-intensive applications.

D. Increasing the Number of Raster Units

In the subsections above we evaluated LIBRA with two Raster Units (i.e., rendering two tiles in parallel). However, LIBRA can be expanded to render more tiles in parallel by including more Raster Units. In this subsection we evaluate the scalability of LIBRA by increasing the number of Raster Units with four cores each compared to a baseline with a single Raster Unit with an equal number of cores in total. For instance, the evaluation with three Raster Units compares a single Raster Unit of twelve cores over LIBRA with three Raster Units of four cores each (i.e., twelve cores in total).

LIBRA allocates one Raster Unit to process hot tiles, while the rest are dedicated to the cold ones. This means that only one Raster Unit handles the hottest tiles at any given time, preventing multiple Raster Units from adding excessive memory pressure.

Figure 18 shows the speedup achieved by LIBRA when increasing the number of Raster Units with respect to a baseline GPU configured with the same number of cores. We can observe that LIBRA is quite effective while increasing the number of Raster Units. In particular, it achieves average speedups of 31.3% and 28.8% with three and four Raster Units, respectively, which are higher than the speedups obtained with two Raster Units (20.9%).

E. Sensitivity Analysis

Supertile size threshold. Figure 19.a shows the average speedup obtained by LIBRA with two Raster Units compared to the baseline GPU when varying the threshold that decides when the supertile must be resized. It can be observed that, in general, increasing this threshold decreases performance since LIBRA takes more time to react to changes in the scene. We chose 0.25% because it is small enough to react fast to changes and it yields slightly better results than 0% across all the benchmark suite (including the compute-intensive applications). Increasing the threshold further is detrimental, and beyond a value of 15% the results do not practically change because such a large threshold behaves as having a fixed supertile size since the size remains almost always the same.

Tile traversal order threshold. Figure 19.b shows the average speedup obtained by LIBRA with two Raster Units compared to the baseline GPU when varying the threshold used to decide when to switch the tile ordering scheme. We can observe

that a threshold of 3% provides the best performance results. Note, however, that other values provide similar results. Values beyond 4% show practically the same speedup since the ordering scheme hardly ever changes and it ends up employing the temperature-based scheme all the time.

VI. Related work

Parallel Rendering. Some works employ PC clusters where an API allocates the workload among machines based on different configurations [1], [19], [20], [29], [30]. To accelerate image composition, [21], [22], [26], [49], [54] implemented application-specific hardware. Note that none of them are GPUs. PFR [9] splits the GPU into two clusters where two consecutive frames are rendered in parallel to exploit inter-frame texture locality. Other works [18], [32], [40], [50], [52], [55], [60], [64], [78]–[80] distribute the workload among different GPUs. To the best of our knowledge, our work is the first one exploring GPU design with multiple Raster Units in a Raster Pipeline.

Tile Scheduling. In the literature there can be found a few works that propose different tile traversal orderings but none of them are for multiple Raster Units belonging to the same Raster Engine. Kerbl et al. [38] explore different tile scheduling traversals among many Raster Units distributed in multiple Graphics Processing Clusters (GPCs). This differs from our work since a GPC is employed in high-end desktop GPUs where each GPC includes a single Raster Unit, and they focus on load balancing threads. DTexL [35] employs a Hilbert tile traversal order to facilitate quad scheduling for texture memory locality. Another work [36] traverses tiles within a frame in the reverse order of the previous frame to enhance L2 texture caching. In [58] it is explored mapping tiles for left and right eyes to the same shader core for VR applications.

Memory Sensitivity. The inability of a program to overlap memory accesses with other useful work underutilizes GPU resources. For GPGPU workloads there are works that address new techniques to reduce this memory sensitivity. In [11] it is proposed a memory controller design that enhances DRAM performance by increasing row-buffer locality through batching requests to the same DRAM row. Other works target warp specialization schemes that overlap memory access and compute [12], [13], [17], [74], [75], prefetching [41]–[43], [51], [57], [68], and improving warp and thread block scheduling [33], [34], [39], [44]–[46], [59], [62], [66], [67], [69], [73]. In [65] it is designed a locality-aware memory hierarchy. Kayiran et al. [37] reduce memory subsystem saturation by throttling the number of CTAs that are active on a shader core. As far as we know, no previous studies have investigated the memory sensitivity of graphics applications. Specifically, our work is the first to explore new policies on mobile GPUs for balancing the memory bandwidth and alleviating DRAM pressure that is not focused on reducing memory accesses.

Locality. Works in [27], [70] apply a similar concept to our supertiles but with entirely different purposes. They create bigger tiles to optimize the accesses generated by the Parameter Buffer. On the other hand, even though one of our main goals is to preserve locality, we can find in the literature some works that focus on improving it, particularly for texture accesses, which are typically the most DRAM bandwidth-consuming. Corbalan et al. [15] propose a NUCA organization for the L1 Texture Caches to increase their effective capacity. Xie et al. [77] explore the use of PIM architectures to reduce the DRAM traffic from texture accesses. Other works prefetch texture memory in the L1 Texture Caches [8], [31] or apply texture compression [3], [23], [61], [71], [76].

VII. Conclusion

Contemporary GPUs require more visually appealing graphics to provide a satisfying user experience through advanced model and screen enhancements. The simplest approach to improve performance is to increase the computing units, but as we have shown in the paper, this method may not always be effective. Therefore, we have explored parallel tile rendering to optimize the use of GPU resources. To the best of our knowledge, this is the first work evaluating parallel tile rendering in mobile GPUs. Unfortunately, applications may experience congestion as pressure increases in the shared memory subsystem.

In this paper, we have introduced LIBRA, which enables parallel tile rendering by employing a novel locality-aware scheduler to keep memory utilization uniformly distributed throughout the execution time. It predicts the memory pressure of a given tile in a frame by exploiting frame-to-frame coherence while not penalizing the memory hierarchy miss ratio. As the scenario is different for each application and varies during runtime, with minor changes in the GPU, we have introduced a mechanism that gathers tile statistics. Besides, it employs a novel tile scheduler that allocates tiles among different Raster Units based on this profiled data. It makes use of a dynamic scheme that takes into account the tile and frame characteristics to minimize the possible memory congestion.

Experimental results show that LIBRA provides 20.9% increase in performance, averaged over a wide range of real-world graphics applications. In addition, we obtain a 9.2% reduction in GPU energy with negligible overhead.

Acknowledgment

We thank the anonymous reviewers for their feedback. 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 FPI research grant PRE2021-100336.

References

[1] K. Akeley, "Reality engine graphics," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 109–116.

[2] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*, 4th ed. USA: A. K. Peters, Ltd., 2018.

[3] T. Akenine-Möller and J. Ström, "Graphics for the masses: a hardware rasterization architecture for mobile phones," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 801–808.

[4] M. Anglada Sánchez, "Exploiting frame coherence in real-time rendering for energy-efficient gpus," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2020.

[5] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Memory bandwidth requirements of tile-based rendering," in *Computer Systems: Architectures, Modeling, and Simulation*, A. D. Pimentel and S. Vassiliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 323–332.

[6] ARM Developer, "Arm GPU Datasheet," https://developer.arm.com/documentation/102849/latest/, accessed June 2024.

[7] ARM Developer, "The Valhall shader core," https://developer.arm.com/documentation/102203/0100/Valhall-shader-core, accessed June 2024.

[8] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. USA: IEEE Computer Society, 2012, p. 84–93.

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

[10] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 37–46.

[11] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. USA: IEEE Computer Society, 2012, p. 416–427.

[12] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011.

[13] M. Bauer, S. Treichler, and A. Aiken, "Singe: leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 119–130.

[14] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power signal processing systems," in *Proceedings of 1994 IEEE Workshop on VLSI Signal Processing*, 1994, pp. 178–187.

[15] 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, pp. 144–151.

[16] P. Cozzi and C. Riccio, *OpenGL insights*. CRC press, 2012.

[17] N. C. Crago, S. Damani, K. Sankaralingam, and S. W. Keckler, "Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1–16.

[18] Y. Dong and C. Peng, "Multi-gpu multi-display rendering of extremely large 3d environments," *The Visual Computer*, vol. 39, no. 12, pp. 6473–6489, 2023.

[19] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436–452, 2009.

[20] S. Eilemann, D. Steiner, and R. Pajarola, "Equalizer 2.0–convergence of a parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 2, pp. 1292–1307, 2018.

[21] M. Eldridge, H. Igehy, and P. Hanrahan, "Pomegranate: a fully scalable graphics architecture," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, p. 443–454.

[22] D. Ellsworth, "A new algorithm for interactive graphics on multicomputers," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 33–40, 1994.

[23] S. Fenney, "Texture compression using low-frequency signal modulation," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '03. Goslar, DEU: Eurographics Association, 2003, p. 84–91.

[24] Fingersoft Ltd, "Hill Climb Racing," https://fingersoft.com/games/hill-climb-racing/, accessed April 2024.

[25] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of iram architectures," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 327–337.

[26] 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," in *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 79–88.

[27] C.-C. Hsiao, C.-P. Chung, and H.-C. Yang, "A hierarchical primitive lists structure for tile-based rendering," in *2009 International Conference on Computational Science and Engineering*, vol. 2, 2009, pp. 408–413.

[28] H. Hubschman and S. W. Zucker, "Frame-to-frame coherence and the hidden surface computation: constraints for a convex world," *ACM Transactions on Graphics (TOG)*, vol. 1, no. 2, p. 129–162, apr 1982.

[29] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "Wiregl: a scalable graphics system for clusters," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 129–140.

[30] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: a stream-processing framework for interactive rendering on clusters," *ACM Trans. Graph.*, vol. 21, no. 3, p. 693–702, jul 2002.

[31] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ser. HWWS '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 133–ff.

[32] M. Jaroš, L. Říha, P. Strakoš, and M. Špeťko, "Gpu accelerated path tracing of massive scenes," *ACM Trans. Graph.*, vol. 40, no. 2, apr 2021.

[33] 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," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 395–406.

[34] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 332–343.

[35] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González, "Dtexl: Decoupled raster pipeline for texture locality," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 213–227.

[36] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González, "Boustrophedonic frames: Quasi-optimal l2 caching for textures in gpus," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023, pp. 124–136.

[37] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 157–166.

[38] B. Kerbl, M. Kenzel, D. Schmalstieg, and M. Steinberger, "Effective static bin patterns for sort-middle rendering," in *Proceedings of High Performance Graphics*, ser. HPG '17. New York, NY, USA: Association for Computing Machinery, 2017.

[39] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A gpu pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 163–175.

[40] Y. Kim, J.-E. Jo, H. Jang, M. Rhu, H. Kim, and J. Kim, "Gpupd: a fast and scalable multi-gpu architecture using cooperative projection and distribution," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 574–586.

[41] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "Cta-aware

prefetching and scheduling for gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 137–148.
[42] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 614–625.
[43] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 213–224.
[44] 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*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 515–527.
[45] 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*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 175–186.
[46] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for gpu kernels," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 14–27.
[47] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
[48] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480.
[49] W.-S. Lin, R. Lau, K. Hwang, X. Lin, and P. Cheung, "Adaptive parallel rendering on multiprocessors and workstation clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 3, pp. 241–258, 2001.
[50] H. Liu, P. Wang, K. Wang, X. Cai, L. Zeng, and S. Li, "Scalable multi-gpu decoupled parallel rendering approach in shared memory architecture," in *2011 International Conference on Virtual Reality and Visualization*, 2011, pp. 172–178.
[51] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 235–246.
[52] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: Numa-aware gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 123–135.
[53] 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.
[54] S. Molnar, J. Eyles, and J. Poulton, "Pixelflow: high-speed rendering using image composition," in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 231–240.
[55] J. R. Monfort and M. Grossman, "Scaling of 3d game engine workloads on modern multi-gpu systems," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 37–46.
[56] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Co. Ltd., Ottawa, Canada, Tech. Rep., 1966.
[57] S. Mostofi, H. Falahati, N. Mahani, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Snake: A variable-length chain-based prefetching for gpus," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 728–741.
[58] J.-H. Nah, Y. Lim, S. Ki, and C. Shin, "Z2 traversal order: An interleaving approach for vr stereo rendering on tile-based gpus," *Computational Visual Media*, vol. 3, pp. 349–357, 2017.
[59] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 308–317.
[60] NVIDIA, "SLI Best Practices," 2011, https://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf.
[61] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, ser. EGGH-HPG'12. Eurographics Association, 2012, p. 105–114.
[62] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram, "Apres: improving cache efficiency by exploiting load characteristics on gpus," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 191–203.
[63] J. Peddie, *The History of the GPU - Eras and Environment*. Springer Nature, 2023.
[64] X. Ren and M. Lis, "Chopin: Scalable graphics rendering in multi-gpu systems via parallel image composition," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 709–722.
[65] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 86–98.
[66] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.
[67] 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*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 99–110.
[68] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 73–82.
[69] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 174–185.
[70] E. Sorgard, B. Ljosland, J. Nystad, M. Blazevic, and F. Langtind, "Method of and apparatus for processing graphics," Jun. 28 2007, uS Patent App. 11/633,647.
[71] J. Ström and T. Akenine-Möller, "ipackman: high-quality, low-complexity texture compression for mobile phones," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 63–70.
[72] SYBO Games, "Subway Surfers," https://subwaysurfers.com/, accessed April 2024.
[73] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, jun 2021.
[74] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 295–306.
[75] H. Wei, E. Liu, Y. Zhao, and H. Yu, "Efficient non-fused winograd on gpus," in *Advances in Computer Graphics: 37th Computer Graphics International Conference, CGI 2020, Geneva, Switzerland, October 20–23, 2020, Proceedings 37*. Springer, 2020, pp. 411–418.
[76] Y. Xiao, C.-S. Leung, P.-M. Lam, and T.-Y. Ho, "Self-organizing map-based color palette for high-dynamic range texture compression," *Neural Computing and Applications*, vol. 21, pp. 639–647, 2012.
[77] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, "Processing-in-memory enabled graphics processors for 3d rendering," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 637–648.
[78] C. Xie, F. Xin, M. Chen, and S. L. Song, "Oo-vr: Numa friendly object-oriented vr rendering framework for future numa-based multi-gpu systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 53–65.
[79] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-

gpu systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 339–351.

[80] H. Zhang, J. Ma, Z. Qiu, J. Yao, M. A. A. Sibahee, Z. A. Abduljabbar, and V. O. Nyangaresi, "Multi-gpu parallel pipeline rendering with splitting frame," in *Computer Graphics International Conference*. Springer, 2023, pp. 223–235.