Behind the Scenes: Memory Analysis of Graphical Workloads on Tile-based GPUs

Germán Ceballos[1], Andreas Sembrant[1], Trevor E. Carlson[2], and David Black-Schaffer[1]

[1]Uppsala University, Department of Information Technology
{german.ceballos, andreas.sembrant, david.black-schaffer}@it.uu.se

[2]National University of Singapore, Department of Computer Science
tcarlson@comp.nus.edu.sg

***Abstract*—Graphics rendering is a complex multi-step process whose data demands typically dominate memory system design in SoCs. GPUs create images by merging many simpler scenes for each frame. For performance, scenes are tiled into parallel tasks which produce different parts of the final output. This execution model results in complex memory behavior with bandwidth demands and data sharing varying over time, and which depends heavily on the structure of the application. To design systems that can efficiently accommodate and schedule these workloads we need to understand their behavior and diversity.**

In this work, we develop a quantitative characterization of the data demands of modern graphics rendering. Our approach uses an architecturally-independent analysis, identifying different types of data sharing present in the applications, independent of their scheduling. From this analysis, we present a limit study into the potential to improve memory system performance by tackling each type of data sharing.

We see that there is the potential to reduce graphics bandwidth by 43% if we can take full advantage of data reuse between tasks and scenes within each frame. For the particularly complex benchmarks, capturing inter-task reuse alone has the potential to reduce bandwidth by 15% (up to 31%), while targeting inter-scene reuse could provide a savings of 60% (up to 75%). These insights provide us the opportunity to understand where we should focus design efforts on graphics memory systems.

I. Introduction

Most resources in today's SoCs (compute logic, memory bandwidth, die area, etc.) are dedicated to rendering graphics. Yet while these graphics workloads dominate today's designs, there has been little work towards understanding their behavior. In this paper we address this through a quantitative analysis of the memory system behavior of modern 2D and 3D workloads.

We first break down how rendering is accomplished by looking at the overall structure of frames, scenes (shader programs), and tasks (individual rendering tiles). Later, we explore the diversity of memory footprints across a range of applications from complex games to simple web page scrolling. This analysis reveals a wide range of memory demands across applications, within applications' scenes and tiles, and over time (Section V), independent of the hardware and scheduling. From there we investigate how much data is shared at the frame, scene, and task level (Section VI), as this sharing can be realized through reduced bandwidth with appropriate caching and scheduling. With this information, we then explore the potential of scheduling and architecture optimizations to take advantage of data reuse (both sharing and private reuse) at the scene and task levels (Section VII). In addition to exposing the diversity of graphics workloads for the first time, this analysis shows the potential for future work in memory system design and scheduling to significantly reduce graphics bandwidth by scheduling scenes and tasks more cleverly.

II. Modern Graphics Rendering

The rendering process of modern graphics applications is a complex series of steps wherein each frame is broken down into multiple smaller rendering passes for different parts of the image and effects. The final image is created by further processing which composes the intermediate results.

Each of these computations is accomplished by a shader program, which uses input buffers or textures and produces output buffers or textures. These *scenes* are executed across the parallel compute resources of the graphics processor by dividing them up into *tasks*, which work on separate portions of each scene in parallel.

- **Buffer/Texture:** A memory resource read or written during rendering to store results of each scene. These can be shared and reused.
- **Scene:** The execution of a graphics shader program consuming buffers and textures as inputs and typically producing one or more as outputs.
- **Task:** Scenes are computed in parallel by tiling the output image into independent tasks.
- **Frame:** a frame is produced by executing a series of scenes to produce the final output. Current systems target to produce 60 to 120 frames per second.

To illustrate the rendering process, we start by looking at how a frame of the Manhattan benchmark is generated (Figure 1). This benchmark represents a complex game scenario by rendering a futuristic cityscape animation. Each frame of Manhattan consists of 60 different *scenes*, over 4,000 *tasks*, and requires more than 95MB of input data. The general flow is shown in Figure 1 (left). The application starts by rendering sequences of scenes that store intermediate results in different output buffers (1). These buffers are merged by following scenes, and used as inputs to later scenes that add special lighting and effects (3 and 4), and produce other details such as texts and 2D overlays (5) for the final frame.

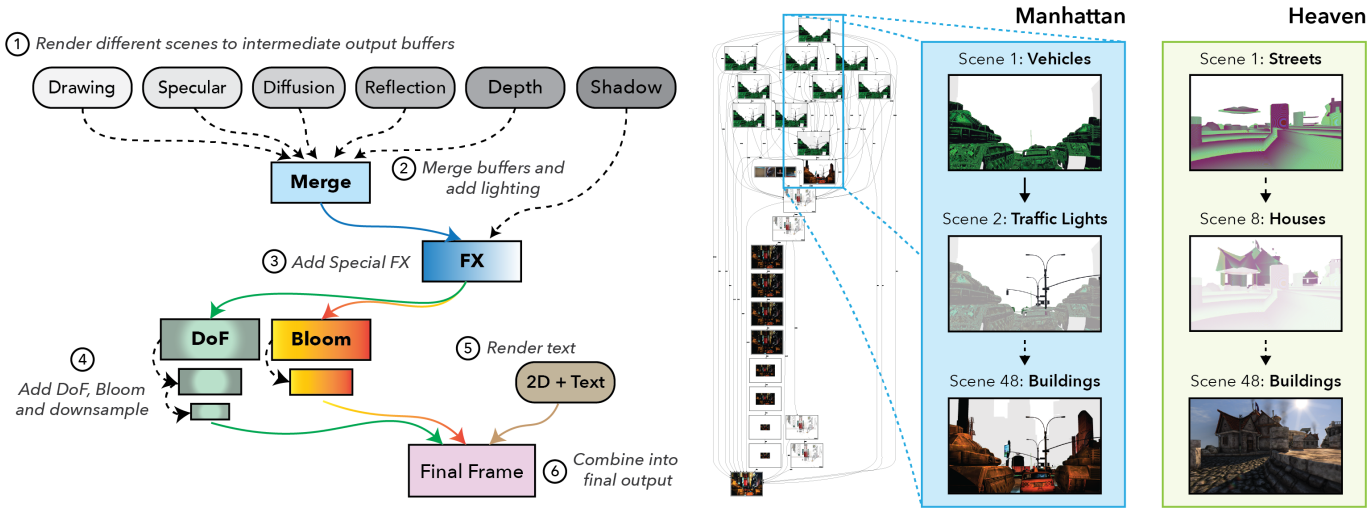

Figure 1. *How Manhattan renders a frame*. Left: schematic overview of intermediate scenes. Right: details of frame 300. A sequence for the application Heaven is also shown. See: http://y2u.be/F1_1Zyq3M7Y

Figure 1 (right) shows the full complexity of frame number 300 for Manhattan. Each box in the graph represents a different scene execution with its corresponding intermediate output images. Arrows between the boxes show how data is shared between scenes, indicating both sharing potential and execution dependencies. Not all scenes use all output buffers, and multiple input buffers may be consumed by the same scene.

A fragment of this is highlighted in Figure 1, where multiple scenes are reading and writing to the same intermediate output buffers. Scene 1 first draws the vehicles. Its result is later used by Scene 2 to overlay traffic lights, and, finally, by Scene 48, which outlines the skyline of the buildings in the background. A similar example from the Heaven benchmark is also shown.

The combinations of scenes for each frame make the rendering process very complex. However, the actual execution is far more intricate, as each scene is tiled into multiple tasks which are then scheduled for parallel execution on the available hardware. The interplay between the scene structure, task parallelism, and scheduling that leads to complex memory system behavior.

Much of this structure comes from the graphics programming model itself: scenes are explicitly data-parallel across the output image, and the explicit definition of input and output data for each scene defines dependencies for the parallel execution. Even though these workloads are ubiquitous today, previous work has not investigated understanding their behavior as it relates to memory system design.

To understand the execution of these workloads we need to first quantitatively characterize their memory behavior in an architecturally independent way. This cannot be done statically, since the memory behavior of each shader program will be determined by the input buffers and textures, which vary over time. To accomplish this, we present a new taxonomy to precisely describe the execution model in terms of data accesses and data reuse.

A. Taxonomy of Memory Accesses

Frames require a significant amount of input data (up to 250MB in the benchmarks studied here), and each piece of data may be used in one (*private* data) or multiple (*shared* data) steps of the rendering process. From Figure 1, we can see how part of the final output image is being created by compositing scenes 1, 2, and 48 to add elements to the image. This process often reads and writes to the same buffers, creating a producer-consumer relationship between the scenes and exposing reuse.

At the same time, most modern GPUs tile scenes into *tasks* for parallel execution[1]. While each task produces an independent portion of the output, tasks can share data from input textures or buffers, which exposes reuse between the tasks inside each scene. Lastly, as frames are rendered at a high rate, there is typically little change between frames, exposing reuse between frames.

To be precise, we define three *different types of sharing*:

- **Inter-Frame Reuse**: memory addresses (data) used by multiple frames. A frame/scene *uses* some data if there is a task in that frame/scene that uses that data.
- **Inter-Scene Reuse**: data used by multiple scenes within *the same* frame, i.e. *sharing* between tasks in *different* scenes.
- **Inter-Task Reuse**: data used by multiple independent tasks within the same scene, i.e. *sharing* of data between tasks within *the same* scene.

This sharing is a fundamental property of how the frames/scenes/tasks use the data and *does not* depend on the dynamic execution order. However, how much of this sharing is realized through the memory system's caches *does* depend on the dynamic scheduling of tasks and the architecture.

Figure 2 illustrates this classification. As we can see, Scene 1 is produced in parallel by multiple tasks, with some of them

[1]The number of tasks is related to how the scenes are tiled for parallel execution. We assume 64x64 pixel tiles/tasks, although some architectures (AMD) can process smaller tiles (32x32).

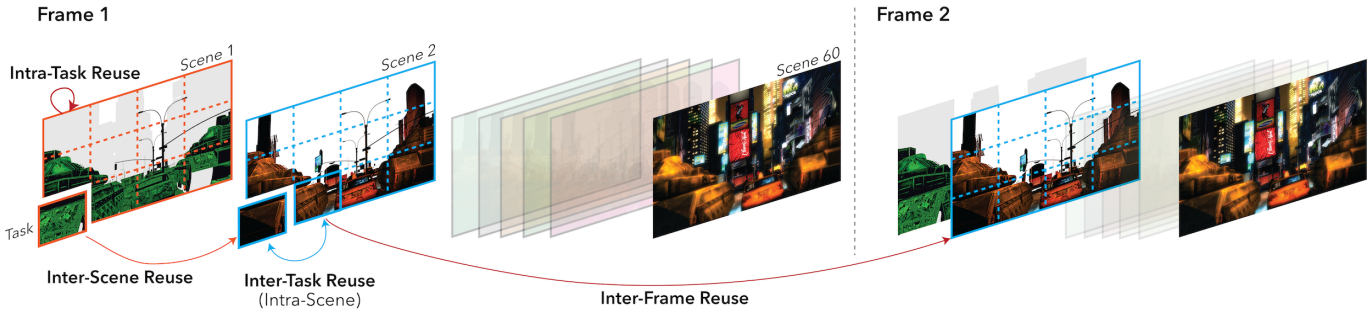

Figure 2. Taxonomy of Memory Accesses.

(top left corner) accessing the same data repeatedly from a texture (intra-task reuse). Next, Scene 2 executes, and some of its tasks (bottom left) use data generated by Scene 1 (inter-scene reuse). At the same time, multiple tasks in Scene 2 are reading the same input texture from memory to overlay it onto the vehicles. This will be treated as inter-task reuse according to our taxonomy. Finally, several of the same input textures and output buffers are used by the following frames, and thus will be considered inter-frame reuse.

Some data reuses are dependencies (read-after-write, RAW), while others are not (read-after-read, RAR). As the graphics programming model enforces that all tasks are independent, dependencies can only happen between scenes, and not between tasks within a given scene.

Understanding data sharing in the execution of an application is critical for developing memory system and scheduler optimizations that are able to maximize reuse through caches and minimize bandwidth consumption. In the following sections we focus on analyzing this behavior in an *architecturally independent* manner, to explore the intrinsic behavior of the applications. The above taxonomy allows us to study sharing properties inherent to the application and not to its execution, as we can identify exactly at which level the data was shared.

III. Setup: Simulator and Benchmarks

We evaluated the workloads using the GLTraceSim [1] tracing and replay framework. This work use GLTraceSim to understand the sharing characteristics of the applications, which has not previously been evaluated. GLTraceSim allows us to model cache miss ratios and bandwidth under different GPU/CPU core and cache hierarchy configurations.

We examined workloads ranging from complex 3D games and animations to simple web-page scrolling. Table in Figure 3 summarizes our applications, along with characteristics such as average achieved frame rate (FPS) on two graphic cards (integrated Intel HD Graphics 4000 and discrete Nvidia GTX1060). For the integrated card, average bandwidth consumption is also shown measured from HW counters. These benchmarks come from three different suites: Phoronix Test Suite [2], Chrome/Telemetry [3] and GFXBench [4].

Workloads can be divided into three categories according to average achieved FPS, as a way to estimate the amount of work (computation) involved to render each frame: Low ($<$ 30/60 fps on integrated GPU/discrete GPU), Medium ($<$ 60/120 fps) and High ($\geq$ 60/120 fps). Low FPS benchmarks such as Manhattan, Heaven, Valley, and Tesseract are intensive applications that render 3D animations with a large number of objects on-screen, high-quality textures, complex reflections, lightning and sophisticated particle interactions. Their frames use over 100MB of input data, and require more than 50 scenes and thousands of different tasks to render. Medium FPS workloads, such as OpenArena, Trex and Xonotic_UD have lower-resoultion textures and simpler environments. Their frames require between 50-70MB of data and fewer than 20 scenes. Finally, the High FPS benchmarks are mainly from the Telemetry suite, where several web pages are rendered and scrolled in the Chrome browser.

The FPS categories give a rough overview of the amount of work required for each frame, but they cannot explain the details of their execution based on scenes and tasks. We need insight into those details to understand how the memory system interacts with these applications and identify particular bottlenecks inside the rendering pipeline. This is crucial for performing memory-related optimizations to achieve higher frame rates, particularly for complex (Low FPS) applications. In the following section we analyze this complexity in detail.

IV. Benchmark Complexity

FPS and bandwidth consumption have been widely used as overall metrics for the amount of work required to render a frame. However, neither are sufficiently detailed to capture the underlying complexity of the rendering process and explain their memory behavior.

We address this by looking (1) at how data is used and shared between *frames*, (2) within and between *scenes* inside each frame, and (3) between and within each *task* in and across scenes. This breakdown allows us to understand the underlying data sharing behavior of the application, and see how those characteristics will map to specific architectures and schedulers.

Table I also shows the number of scenes and tasks per frame. Intuitively, these relate to both the length of the rendering process and the amount of available parallel work. We can see significant diversity in benchmarks of the same category: Valley achieves 20% lower frame rate on average than Heaven, but it generates 40 fewer scenes per frame. On the other hand, Manhattan spawns a larger number of tasks for each frame

Benchmark	Avg FPS (Discrete)	Avg. FPS (Integrated)	Avg. BW (Integrated)	Avg. Scenes/Frame	Max Scenes	Std. Deviation Scenes/Frame	Avg. Tasks per Frame	Std. Dev. Tasks/Frame
manhattan	43	19	629	51	63	18	3,947	2,745
heaven	27	5	1,332	127	130	37	6,520	4,340
valley	18	4	1,652	78	90	23	21,929	13,539
tesseract	32	16	461	64	92	26	3,090	2,889
openarena	113	23	585	10	11	3	1,968	1,424
trex	98	36	235	19	28	8	5,003	2,981
xonotic_UD	162	27	303	15	23	4	3,830	2,509
xonotic_LD	280	112	123	1	1	0	572	474
chrome	461	208	73	1	7	0	621	369

Figure 3. (Table I) Graphic Workloads. Benchmarks are colored by FPS category (Low, Mid, High). Overall statistics for the rendering process are also displayed. Average bandwidth for discrete cards is not included as there is no harwdare support to measure it.

compared to Tesseract, but achieves higher frame rate and consumes more bandwidth. At the same time, TRex runs slower than OpenArena, generating twice as many tasks and scenes on average, but consumes only a third of its bandwidth.

Finally, all Chrome benchmarks have a very similar, and simple, single-scene structure with only around 600 tasks, which is why they achieve a very high frame rate. A particularly interesting benchmark is Xonotic_LD (which is the lower-quality setting of Xonotic_UD), where the number of scenes is similar to the Chrome workloads, but bandwidth consumption is twice as large, and the achieved frame rate is half as high.

From the more detailed statistics, we can see how overall performance is a complex interaction of many factors including the number of scenes, their task parallelism and the amount of data needed and reused. Existing techniques that look at the actual execution, such as cache miss ratio curves ([5], [6]) or analytical performance models ([7], [8]), give results that are highly dependent on the particular scheduling (ordering of tasks and scenes) and memory system (caching effects of shared and privately reused data). Those approaches may give an accurate view of the application on one particular system, but they conflate system-specific effects and intrinsic application behavior. Furthermore, as we have seen with Xonotic_UD and Xonotic_LD, changing the input of what the shader program is rendering (e.g., textures with different qualities) changes the behavior at the memory system level, which prevents us from using static analysis alone (e.g., through compilers).

To address this, we propose to study memory behavior from an execution-independent perspective at three levels: frames, scenes, and tasks. To do so, we adopt the notion of *footprint*, defined as the *total amount of unique data (memory addresses)* used, which is not dependent on the architecture or execution schedule. This would be equivalent to studying the applications running under an infinitely large cache and gives an upper bound for the potential of reuse-aware scheduling heuristics to reduce bandwidth. Removing architectural and scheduling implications allows us to understand memory properties inherent to the application, i.e. the potential value of shared data.

Before examining sharing, we first study the total footprint of the frames, scenes, and tasks, including how the footprints vary across applications and why. This provides the general context for later understanding the sharing, such as how much data is shared out of the total data used and how much is read-only versus read-write. After looking at the total footprint, we will investigate the amount of shared data at each level, how the sharing structure (types of sharing) change depending on the workloads and the main reasons behind those changes.

V. Analyzing Footprints

A. Frame Footprints

Figure 4 compares the graphic workloads based on how much data they use, and in which manner: unique data per frame (footprint) versus data shared both within scenes and within tasks. There is a large diversity in the average footprint per frame across workloads. Applications such as Heaven and Valley use over 250MB per frame, Manhattan, Tesseract, Trex and Openarena range from 50-100MB, and most of the Chrome benchmarks are under 20MB per frame.

Despite this variation, when we examine how the frame footprint varies over time *within* each workload (Figure 5), we see that the frame size remains constant. This implies that the memory requirements across frames of the same application does not change significantly, which motivates a closer look into the behavior of scenes and tasks within frames.

B. Scene Footprints

Figure 5(a) shows box plots for the scene footprints, skipping the initialization/startup phases of the applications (Heaven: 32 frames; Manhattan, Valley, Openarena: 100; all others: >400). Workloads are color coded by their respective FPS categories, as seen in Table I. The plots show the distribution of the scene footprints across all frames. Each dot is a scene of a particular frame, distributed along the x-axis according to its footprint. The box covers the interquartile range (IQR) of the population (50% of the scenes) and the color transition inside the box shows the median size. Whiskers are displayed at 1.5x IQR to highlight 75% of the scenes. The average scene footprint is annotated with gray bars and values.

As we can see from Table I, the number of scenes is not constant for all the frames. Some scenes are *always* executed,

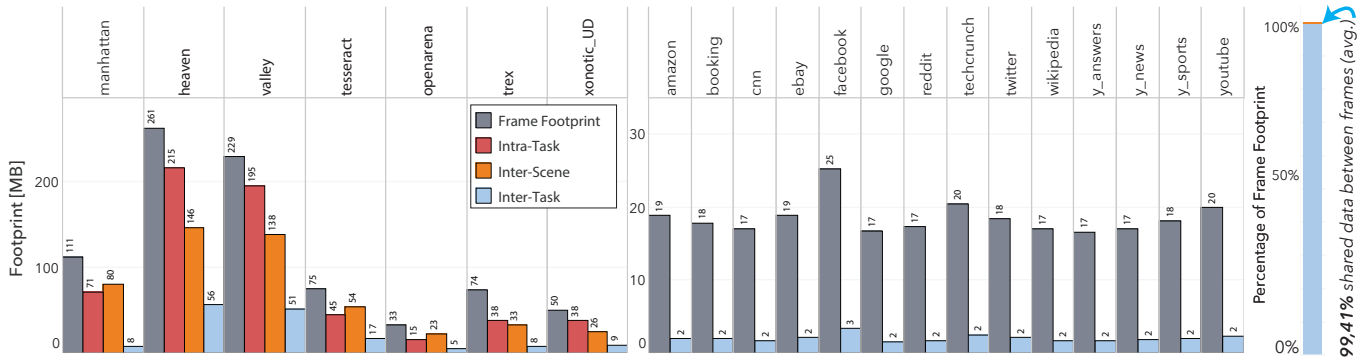

Figure 4. Average per-frame footprint and reuse. The Telemetry workloads (Chrome) do not expose any Intra-task or Inter-scene reuse.

but many others are spawned on demand depending on the content of the final output image. This variability increases the complexity of the rendering process, but, as we will see later, the variable scenes represent less than 2% of the total footprint and less than 4% of the total tasks spawned.

Figure 5(a) shows that some applications present a large diversity in scene footprints. Manhattan has a significant number of scenes of 40 to 60MB each, with the majority between 10 to 20MB. Heaven, on the other hand, achieves lower frame rate, but most scenes are under 10MB, with less than 12% over 30MB. High FPS benchmarks such as Xonotic_LD and Chrome have low variation across scenes, with tight distributions around their averages of 23 and 17MB.

To clearly show the distribution, the cumulative distribution function (CDF) of scene footprints is shown in Figure 5. The CDFs let us quickly identify diversity (by the slope of the curve) and absolute size. From the x-axis, we can see *how many* scenes are under a certain size. For instance, 22% of the scenes in Manhattan are under 10MB, but 25% of the scenes are under 37MB. This means that 3% of the scenes are between 10 and 37MB which is a 3.7X increase in size compared to 22% of the scenes. Similarly, Heaven sees a 1.5x increase in size for 1% of the scenes: between 39MB and 55MB.

The Mid FPS benchmarks (OpenArena, Trex and Xonotic_UD) have fewer changes in their CDFs, but with the opposite effect: All scenes are under 30MB in footprint and compared to the Low FPS workloads, these applications expose only small increases in size for most of the scenes. For example, OpenArena jumps from 20% of the scenes under 19MB, to over 60% of them below 20MB. In Trex the variation in footprint of 60% of the scenes (15%-75%) is only 7MB.

Finally, the High FPS applications show little diversity in scene size, with almost flat CDFs. All Chrome benchmarks have scenes around 18MB, due to the fixed-size textures and outputs for rendering the web pages. Moreover, Xonotic_LD shows how changing the rendering configuration on the same benchmark (from high definition on Xonotic_UD to low definition on Xonotic_LD) can trade the complex multi-scene rendering process into a single-scene process for achieving higher frame rates (with a loss of image quality). This is captured in the figure as we see how the scene diversity changes from Xonotic_UD to Xonotic_LD: In the high-definition execution, there are several extra scenes (lighting, shadows, reflections, etc.) that are small in footprint but add to the workload complexity. When changing to low-definition, those scenes are not executed.

Looking at the scene footprint across a range of benchmarks reveals the diversity in application structure, with low frame rate (more complex) applications typically displaying both larger and more diverse scene footprints.

Scene Variation: The causes behind the workload diversity are related to how the different shader programs are structured, i.e. what they execute, the objects needed to be rendered for the final image, and how those objects change over time (from frame to frame).

a) Shader program structure (what the scenes are doing/drawing): Different scenes perform different computations as they draw different objects. Therefore, scenes can use amounts of data that vary drastically. A simple example can be seen in Figure 1 with some scenes from Manhattan. Scene 1 draws the vehicles on screen. The application is structured such that most of the initial environment is generated during this scene. Later, Scene 2 draws the traffic lights based on that environment, and finally Scene 48 the buildings in the background.

When looking at where these scenes are in the distribution (Figure 5 (a)), we see that Scene 1 is 46.75MB, while Scene 2 is 2.65MB and Scene 48 is 15.02 MB. As expected, drawing the smaller traffic lights is far less data intensive than the buildings, and one order of magnitude lower than the vehicles.

Meanwhile, different scenes can compose different parts of the image simultaneously, creating a tree of producer-consumer relationships. Thus, scenes from different branches might not be dependent on each other, implying flexibility in scheduling, which can affect data reuse for a given architecture. For instance, in the same frame of Manhattan, the sky and the advertisements displayed on the buildings are rendered by independent scenes. All of these independent scenes are merged in Scene 49 (before the lightning, shadows and special effects) which is 54.14MB.

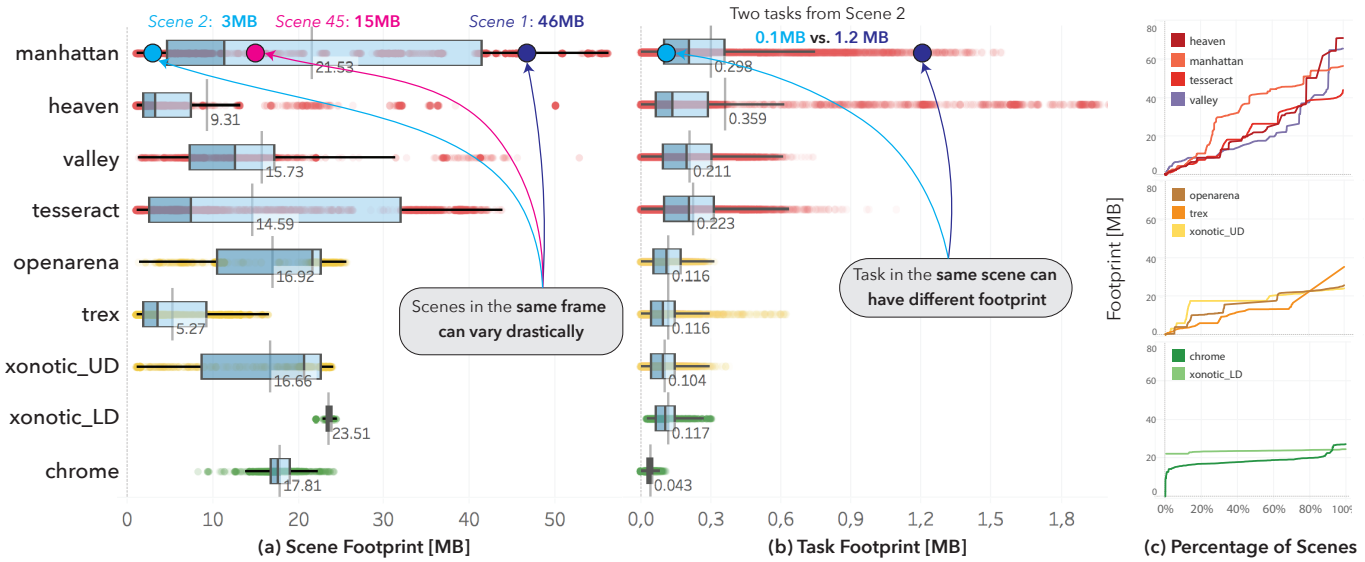

Figure 5. Footprint diversity across scenes **(a)** and tasks **(b)**, and as CDF **(c)**, colored by FPS category (Low FPS=Red, Mid FPS=Yellow, High FPS=Green)

b) Different input/output resolutions: Another reason for footprint diversity is simply that some scenes produce smaller outputs, such as rendering special effects at reduced resolution, down-sampling, and rendering generic graphics or information displays. These scenes will have a smaller footprint as the sizes of their inputs/outputs are naturally smaller.

c) Frames changing over time: Finally, objects in the frame change over time. This means that the same scene in two different frames might be of very different size. For example, Scene 1 drawing the vehicles in Manhattan will increase/reduce its footprint over time, as the vehicles move in and out of the frame.

C. Task Footprints

In addition to diversity in scene footprints, there is a similar degree of diversity among the tasks in a scene. As each task renders a different part of the scene, each may have a different footprint depending on which scene it comes from and what part it is rendering. Understanding inter-scene task footprints is critical for designing schedulers.

Figure 5(b) shows task footprints across all frames, ranging from 0 to 2MB. Low FPS benchmarks (red) show a large diversity in the task footprints, which correlates with their greater scene diversity. Most of the tasks are between 256kB and 512kB (64-128 bytes per pixel). High FPS applications tend to have tasks with smaller footprints (50kB, 12B/pixel).

This diversity can have two different natures, as highlighted in Figure 5(b). First, tasks within *the same* scene can have different footprints. This can be observed in Figure 1, where Scene 1 draws the vehicles. As the vehicles are in the bottom part of the image, the tasks that draw those portions will have large footprints, while the ones towards the top (the sky) will draw nothing and have small footprints.

This diversity is a consequence of the programming model, where graphics applications create parallelism in a brute-force approach: tasks are spawned for pixels even if they do not need to draw anything. Figure 5 highlights how two tasks from Scene 2 in Manhattan can vary in size drastically.

In addition to variations within a scene due to differences in what is being rendered, the overall complexity/footprint of each scene affects its tasks' footprint. This can be seen with larger scenes having tasks that use more data and scenes rendering at lower resolutions typically generate fewer tasks.

D. Insights from Scene and Task Footprint Analysis

There is significant diversity in memory footprints across and within graphics applications caused by factors inherent to the programming model, such as different shader programs, dynamic objects on screen, and brute-force parallelism. This diversity occurs at different granularities (between frames, between scenes, and between tasks within and across scenes) revealing complex application data behavior. Analyzing footprints at these levels is a key step to understanding and characterizing the memory behavior inherent to graphics workloads. This variation will have different architectural implications, but most importantly, analyzing it from an architectural independent perspective (using footprint) allows us to identify trends and characteristics that differentiate workloads beyond simple FPS categories.

Now that we have investigated a key component to characterize memory behavior in graphic workloads, we focus on the portion of the data that is shared to better investigate architectural implications for scheduling and caching. To do so, we present a detailed study of sharing between frames, scenes and tasks (Section VI), and then we explore how their memory behavior is affected by scheduling during execution (Section VII).

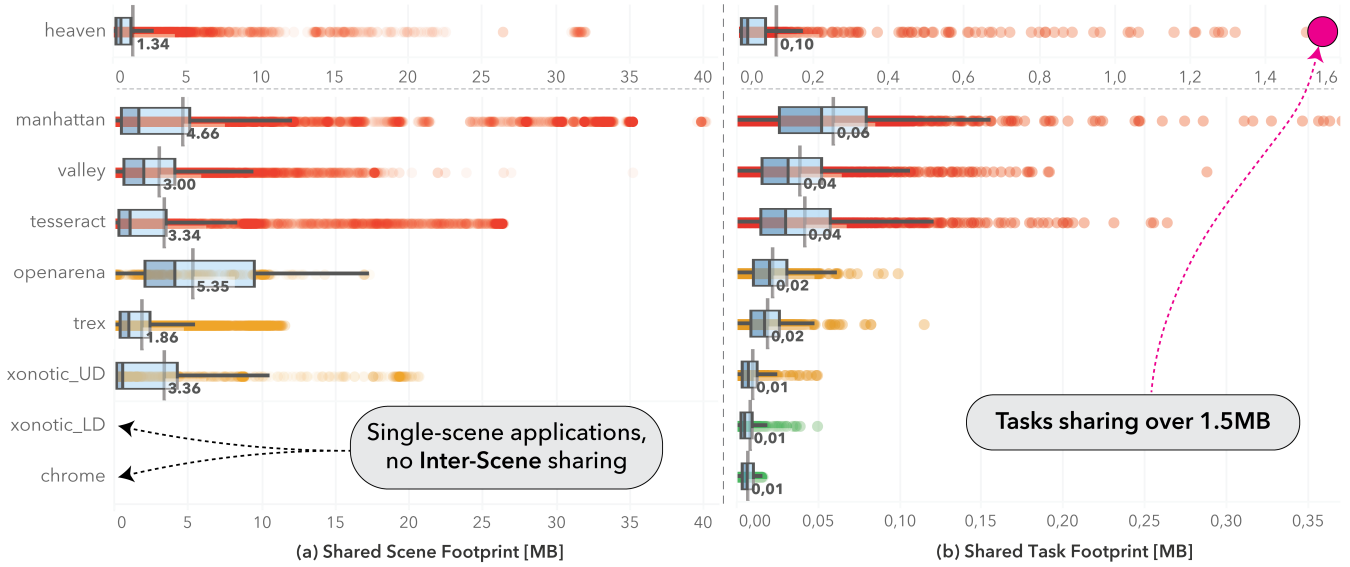

Figure 6. Sharing diversity across scenes **(a)** and tasks **(b)**.

VI. Analyzing Sharing

Understanding sharing in graphics workloads is vital to design schedulers and memory hierarchies that can maximize reuse through caches. We study sharing across all levels (between frames, between scenes and between tasks), and later we use this analysis in Section VII to understand the hardware implications of these sharing properties.

A. Frame Sharing

As frames do not change much over time, we expect a large amount of data to be reused, as is shown in Figure 4. Across frames, so much data is reused (over 99%) that it dwarfs the new data required for each frame. Thus, if the cache can hold one frame's worth of data, then main memory bandwidth will be essentially zero. For these benchmarks, this requires a cache between 20MB and 260MB which is practical with eDRAM caches ([9] shows how multiple frames can be partitioned and rendered at once to take advantage of inter-frame sharing at the cost of increased latency.). However, for systems that cannot afford such a large cache, we need to look within the frame at the scenes and tasks to see what lower-level sharing we can exploit.

B. Scene Sharing

Within frames, we see that there is significant data sharing between scenes (inter-scene). This is shown in Figure 4 with the orange bars, which represent average inter-scene footprint per frame (in MB). Across workloads inter-scene sharing is 51% of the total frame footprint. Low FPS workloads expose average inter-scene reuse of 68% (up to 84% in Heaven). In Mid FPS workloads the average inter-scene reuse per frame is 54% of the total frame footprint (up to 73% in Xonotic_UD), while high FPS workloads have negligible inter-scene reuse as they mostly have one scene.

Figure 6 (left) shows the distribution of sharing between pairs of scenes. From these distributions we can not only identify pairs of scenes with maximum and average sharing potential, but also understand the full sharing diversity of the applications. The Low FPS workloads are particularly interesting (Manhattan, Heaven), with a significant number of scenes sharing a large amount of data (30-40MB). These scenes are also scheduled far apart in time in most of the frames (Scene 1 vs Scene 49), indicating there is a challenge for scheduling and caching systems to take advantage of the shared data. At the opposite extreme are the High FPS workloads. With only one scene per frame, as in Chrome or Xonotic_LD, the inter-scene sharing is zero (only inter-frame sharing). For these applications, the only sharing potential is between tasks.

Furthermore, we can check the type of sharing based on the operations performed by each memory access. From the total scene sharing, on average, 35% of the data shared is done in a read-only fashion (reuse) across workloads, while the remaining is done in a read-write manner. This gives further insight into the potential for maximizing temporal versus spatial locality and vice-versa.

C. Task Sharing

Figure 4 shows how the amount of data per frame that was shared between tasks of the same scene (*inter-task* sharing). The bar displays the *aggregated* data across all scenes, counting how much data was touched by more than one task inside each scene (for all the scenes in that frame). On average, inter-task sharing represents 15% of the total frame footprint. Considering only the Low FPS benchmarks, this changes to 19% (up to 23% on Valley) due to the fact that these benchmarks use more input/output textures. Interestingly, the High FPS benchmarks, which did not expose inter-scene reuse, show a significant 16% inter-task reuse.

To look at inter-task sharing, we repeat the same process from previous section inside each scene, and look at sharing between all task pairs in Figure 6 (right). Tasks from the Low FPS benchmarks Manhattan, Tesseract and Valley share on average 6 times as much data than the Mid FPS workloads. An exciting case in the Low FPS benchmarks is the sharing behavior of Heaven (top right of the figure). In Heaven, we can observe tasks sharing less than the Mid FPS applications on average, but the spread of the population is much larger (more than 1.5 times the IQR), meaning that there are a significant number of tasks that share between 0.5MB and 2MB. This is promising given that the tasks sharing a large amount of data could benefit from being co-executed, or scheduled nearby in time one when running on hardware with small shared caches.

D. Task and Scene Sharing Diversity

A large portion of the diversity in sharing is explained by the overall footprint of the scene and task, studied in Section V. In addition, some scenes share data with others because they are composing results in a producer-consumer fashion, e.g. in Figure 1, Scene 2 draws the traffic lights *over* part of of what Scene 1 drew before, causing those scenes, and the tasks in those regions, to share data. Similarly, Scene 48 draws the buildings on the background after Scene 2 finished with the traffic lights, but since the area for the buildings is much larger, they will share more data. On the other hand, other scenes will have rendered parts that were already consumed by other scenes, and will not share with these scenes.

Task sharing is most common when the tasks render similar objects or multiple instances of the same object. In Manhattan's Scene 1, the vehicles are rendered by multiple tasks, and as they are close together in image coordinates, they are likely to use (and therefore share) similar parts of the underlying texture data. Also, if the same object is drawn multiple times, those tiles will share input data. This can observed in Figure 1, which shows the output of one scene of Heaven in 1 with the two houses. In this case, tasks working on those regions will be using the same inputs. Therefore, sharing occurs regardless of the position of the task in the 2D space, if the same object is shown multiple times on screen.

E. Conclusion on Sharing

There is significant sharing across all levels of the applications: frames, scenes, and tasks. Frames can be handled by large caches, but if such a large cache is not practical or possible, it is necessary to look within the frame at sharing between and within scenes and tasks. At those levels, we have seen that the Low FPS applications have significant sharing at both the scene and task levels, while the High FPS applications show considerable sharing between tasks. With this understanding of where sharing occurs, we can now look at the architectural and scheduling implications.

VII. Scheduling

Traditional graphics scheduling approaches have focused on keeping hardware resources busy [10]. Yet scheduling can have a drastic impact in the data locality properties of the applications as well [11], [12], [13]. Scheduling tasks that share data together can reduce bandwidth and improve performance (frame rate) as they will be able to keep reused data in smaller caches. However, because these reuse effects are dependent on both the schedule and the particular cache hierarchy, our analysis to this point has been focused on architecturally independent metrics, such as footprint, to avoid biasing from a particular schedule and cache design.

To explore the potential for reuse-aware schedulers, we look at the effects of targeting each type of sharing individually and together: intra-task, inter-task and inter-scene. To do so, we present an approach that analyzes the *maximum* sharing we could obtain if we could completely realize sharing for each of those categories, and how that would affect the data accessed by the application. This analysis enables us to understand where it would be profitable to investigate new scheduling and memory system designs.

The two most used graphics scheduling heuristics are First-come First-served (FCFS, simple queue) and the Z-scheduler, however neither of them explicitly look at sharing in their scheduling decisions. In the **Z-scheduler**, tasks are enqueued in a hierarchical Z order. First the entire scene is divided in four quadrants, creating the partial schedule: top-left quadrant, top-right, bottom-left and bottom right (*Z* pattern). The process is repeated recursively in each of the quadrants until the granularity is a task. This scheduler was designed to optimize for tasks working on similar objects.

A. Understanding the limits of scheduling

Scheduling strategies that take advantage of data sharing will try to maximize data reuse through the caches, minimizing costly traffic to main memory. For example, private caches are often able to capture many data reuses local to each task (intra-task reuses), but only if the task footprint is smaller than the private cache. Analogously, a GPU's shared cache can take advantage of reuses between tasks/scenes if they fit given the execution schedule. The minimum bandwidth consumption possible for an application is its footprint. However, that bandwidth will only be realized if all data reuses are perfectly captured by the caches and schedule. To understand which of the different types of sharing affect bandwidth the most (and thus have the most potential for optimization), we will examine our applications running under four different optimizations:

1) **Perfect Intra-Task Sharing:** assuming an infinite size cache for reuses local to each task (e.g., every task fits its own data in the cache).
2) **Perfect Inter-Task Sharing:** assuming an infinite size cache for data shared between tasks of the same scene.
3) **Perfect Inter-Scene Sharing:** assuming an infinite size cache for data shared between scenes *within* a frame.
4) **Perfect Sharing:** Enabling all optimizations above together.

We model these scenarios by classifying memory accesses in GLTraceSim and using this information to determine whether an access goes through the standard cache path or is logically

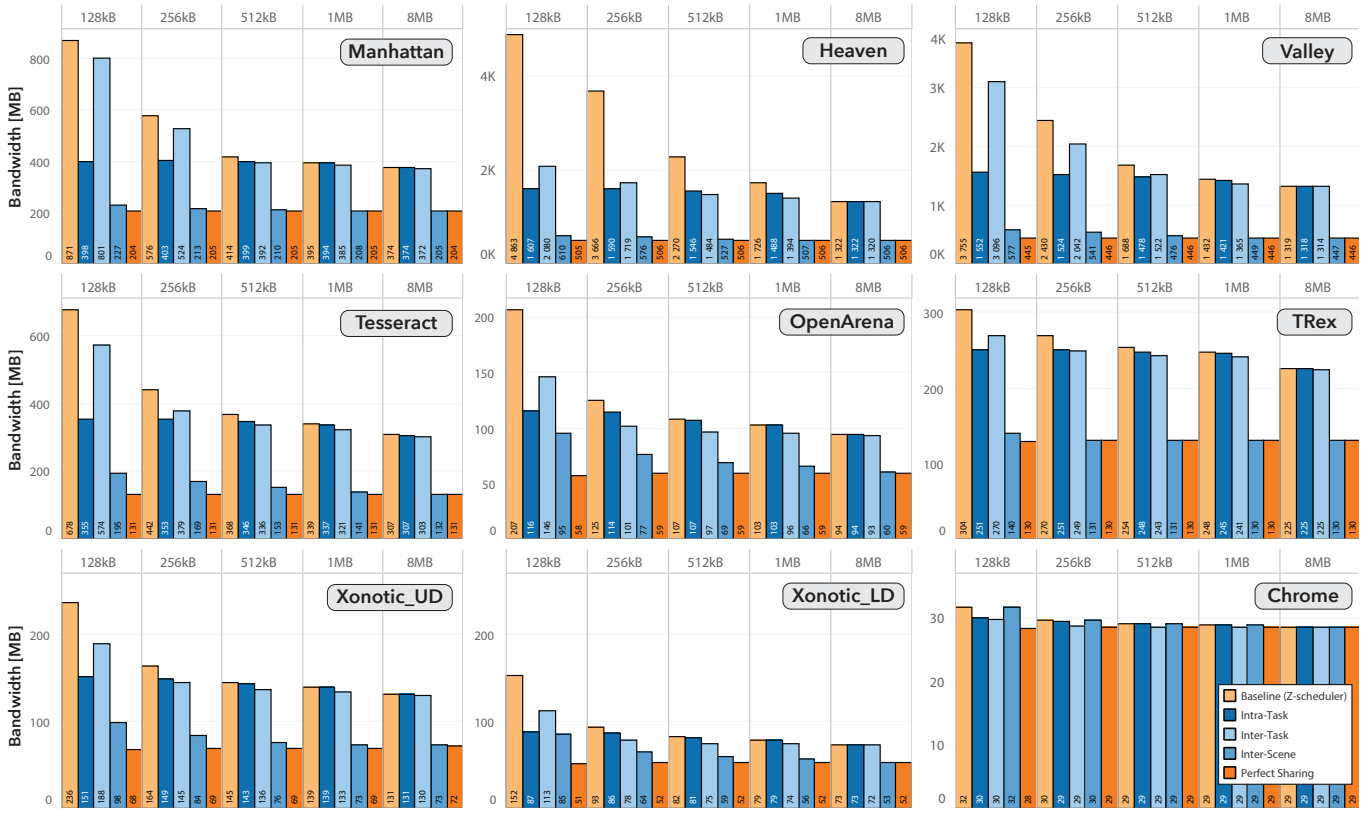

Figure 7. Graphics bandwidth (average MB per frame) as a function of reuse optimizations (perfect inter-scene sharing, perfect inter-task sharing, perfect intra-task sharing, and perfect sharing) and cache size. Assumes 64B cacheline.

placed in an infinite cache for the targeted shared data. As a result, the targeted shared data will always hit after its first access and does not consume space in the regular shared cache, leaving it available for whatever other data is brought in later. In this manner we can evaluate the maximum potential for capturing all of each type of reuse, without being dependent on the particular memory hierarchy or schedule.

B. Evaluation

We evaluate a common GPU configuration with 4-cores, 32kB private caches and a shared last level cache of size ranging from 128kB to 8MB.

Figure 7 shows the average bandwidth consumption per frame for each optimization across a range of shared cache sizes. The baseline is a standard Z-scheduler. By comparing the traditional Z-scheduler with each of the perfect sharing optimizations we can see how good a job the Z-scheduler is doing at capturing the available reuses in the applications.

1) Optimizing Intra-task Reuses: When modeling perfect intra-task reuse, we assume that the data that is accessed multiple times within the same task will always hit in the shared cache, which is equivalent to having the tasks access their data only once. In most applications, small cache sizes (128kB through 512kB) show a significant benefit from perfect intra-task reuse. The largest differences are exposed by the Low FPS applications where bandwidth is reduced by 40% on average and up to 74% in Heaven over a traditional Z-scheduler. This means that for all these workloads, a key requirement is that tasks are able to fit in their local caches in order to significantly minimize bandwidth.

As the cache size is increased, we see how the difference between the Z-scheduler and having perfect intra-task reuses becomes smaller across all applications. This is because the cache is able to capture all intra-task reuses. Larger caches will be able to fit tasks with larger footprints more easily (tasks are often less than 2MB) and thus, the Z-scheduler will see no additional benefit from perfect intra-task reuse. This implies that the upper bound for optimizing reuses local to a task is limited by the number of tasks that fit in the cache. In this particular case, as the cache is shared among four cores, all the tasks running in parallel need to fit in the cache for this to hold, and will be the same when sharing with more cores.

2) Optimizing Inter-task Reuses: Inter-task reuses are interesting because they can be used (1) to schedule tasks one after the other to maximize temporal reuse, or (2) at the same time to maximize spatial reuse. Our modeling of perfect inter-task reuse is equivalent to assume that whenever a task is accessing data previously accessed by another task, it will be present in the cache, missing only on new data.

In all but one of the applications, we see how this optimization is more promising than intra-task sharing, even at the smallest cache size, meaning there is far more potential from

looking at reuse *inside* the scene than that having perfect local reuses.

On average we see a 10% bandwidth reduction with a 512kB cache, and 5.35% on a 1MB cache. The overall improvement is 3.82% across all cache sizes. This optimization is most effective in Heaven (up to 32%), Tesseract (11%) and Valley (10%), all of which are Low FPS benchmarks that exposed the greatest diversity in sharing (Section VI). The only case where perfect intra-task uses more bandwidth than inter-task sharing is Valley (512kB cache), meaning it does not make sense to optimize reuse between tasks if their data does not fit in the cache.

3) Optimizing Inter-scene Reuses: The largest effect is seen with perfect inter-scene sharing, which is not surprising given the significant amount of the footprint that was shared between scenes (Section VI). Modeling perfect inter-scene sharing is analogous to the previous two optimizations, wherein we assume that scenes that are reusing data from previous scenes will hit in the shared cache.

The Low FPS benchmarks had several scenes sharing up to 40MB of data. As this optimization assumes this data can be cached (although this data is still much smaller than caching the entire scene or frame), reuses local to a scene will not impact bandwidth. Heaven, Manhattan, Valley and Tesseract see an average bandwidth reduction of 59% across all cache sizes, up to 74%.

The Mid FPS benchmarks also see a dramatic reduction (41% on average) compared to perfect inter-task sharing. This is remarkable as these benchmarks have fewer scenes, up to 70% smaller in footprint and share up to 60% less than the Low FPS benchmarks.

Finally the High FPS, single-scene applications (Xonotic_LD and Chrome) have no sharing, and thus we see no benefit from the inter-scene optimization. Furthermore, as their tasks are very small, task data sharing is negligible, and none of the optimizations make a significant difference. For these single-scene applications, we would need a cache large enough to capture the whole frame's data reuse to observe an impact on bandwidth, as shown in Section V.

VIII. Related Work

Graphics GPU Workload Analysis: Mitra et al. [14] measured intra-frame texture locality and bandwidth consumption per frame. In addition, Roca et al. [15] use a cycle-level GPU simulator to look at a number of general application characteristics to understand workloads from both a bandwidth and computation perspective. In this work, we explore the sharing potential for modern graphics applications at the intra-frame, intra-scene and intra-task levels to understand bandwidth reduction potential inherent to the applications' sharing characteristics. George [16] proposed graphics GPU workload subsetting as a way to select representative graphics frames for architectural evaluation. Our work goes beyond frame similarity to look at scene and task properties at a system-level. Instead of using frame clustering to make simulation tractable, our approach analyzes traces across hundreds of frames to derive the potential for bandwidth reduction.

Graphics GPU Techniques: Parallel Frame Rendering [17], and the work from Arnau, et al. [9] aim to improve performance by taking advantage of frame-based locality. Our analysis methodology, in contrast, aims to show the potential for savings across many levels of sharing, not only between frames. In addition, techniques like ARM's Transaction Elimination [18] send only final frame-to-frame differences to save bandwidth and Teapot [19] shows how reducing graphics quality can improve energy efficiency. These techniques strive to realize benefits of the memory behavior of the applications, while our work explores the behavior itself in an architecturally-independent manner.

GPGPU Simulation and Analysis: The work from Bakhoda, et al. [20] presents a detailed simulation platform and study of CUDA applications that target general-purpose compute and data processing. This work, in contrast, presents a methodology for analyzing and understanding graphics applications which use a fundamentally different programming model and tend to show significant time-varying behavior.

IX. Conclusion

In this paper, we explored the diversity of memory behavior in modern graphics workloads, which are some of the most popular workloads today, and showed how a simple classification based on performance achieved (fps) or bandwidth is not enough to understand their complexity. To analyze the applications in detail, we presented a taxonomy to describe the memory behavior of graphics programs in an architecturally- and scheduling-independent manner, allowing us to understand fundamental properties of the applications such as footprint and sharing at different levels (frames, scenes, and tasks). In addition, we analyzed a wide range of workloads and explored the sources of the diversity in memory system behavior.

As a use case for this work, we looked into the potential gains for developing schedulers and memory hierarchies that can take advantage of the inherent sharing at the scene and task levels of the application with perfect sharing optimizations. We found that bandwidth can be reduced by 43% on average across all workloads by taking advantage of reuse between tasks and scenes inside a frame. For intensive, low-frame rate benchmarks, bandwidth can be reduced by 40% on average from inter-task reuse (up to 76%) and 60% from inter-scene reuses (up to 75%). This new workload characterization goes beyond previous work to identify promising directions for improving memory system efficiency and runtime graphics scheduling for future systems.

We also explored how these properties are affected by the interplay of scheduling and caching when executing on particular architectures. By simulating four different perfect-sharing optimizations (at scenes- and task-levels) we were able to reveal the limits of memory related optimizations, in particular, how bandwidth would be affected by scheduling heuristics. We show how bandwidth can be reduced by 43% on average across all workloads by taking advantage of reuse

between tasks and scenes inside a frame. For intensive, low frame rate benchmarks, bandwidth can be reduced by 40% on average from inter-task reuse (up to 76%) and 60% from inter-scene reuses (up to 75%).

REFERENCES

[1] A. Sembrant, T. E. Carlson, E. Hagersten, and D. Black-Schaffer, "A graphics tracing framework for exploring cpu+gpu memory systems." in *IEEE IISWC.*, 2017.

[2] Phoronix Test Suite, *www.phoronix-test-suite.com*.

[3] Telemetry, *www.chromium.org/developers/telemetry*.

[4] GFXBench, *www.gfxbench.com*.

[5] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," in *ISPASS*, March 2006, pp. 89–99.

[6] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *ISPASS*, March 2010, pp. 55–65.

[7] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009.

[8] J. C. Huang, J. H. Lee, H. Kim, and H. H. S. Lee, "GPUMech: Gpu performance modeling technique based on interval analysis," in *MICRO*, Dec 2014, pp. 268–279.

[9] J. M. Arnau, J. M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ISCA*, June 2014, pp. 529–540.

[10] J. Ragan-Kelley, "Keeping many cores busy: Scheduling the graphics pipeline," in *SIGGRAPH*, 2010.

[11] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on numa systems," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer Berlin Heidelberg, 2013, pp. 156–170.

[12] G. Ceballos, E. Hagersten, and D. Black-Schaffer, "Formalizing data locality in task parallel applications," in *ICA3PP*, 2016, pp. 43–61.

[13] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," in *ASPLOS*. New York, NY, USA: ACM, 2013, pp. 343–356.

[14] T. Mitra and T.-C. Chiueh, "Dynamic 3D graphics workload characterization and the architectural implications," in *MICRO*, 1999, pp. 62–71.

[15] J. Roca, V. Moya, C. Gonzalez, C. Solis, A. Fernandez, and R. Espasa, "Workload characterization of 3D games," in *IISWC*, Oct 2006, pp. 17–26.

[16] V. M. George, "3D workload subsetting for GPU architecture pathfinding," in *IISWC*, Oct 2015, pp. 130–139.

[17] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Parallel frame rendering: Trading responsiveness for energy on a mobile GPU," in *PACT*, 2013, pp. 83–92.

[18] ARM, "http://www.arm.com/products/graphics-and-multimedia/mali-technologies."

[19] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: A toolset for evaluating performance, power and image quality on mobile graphics systems," in *IISWC*, 2013, pp. 37–46.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.