# A Graphics Tracing Framework for Exploring CPU+GPU Memory Systems

Andreas Sembrant, Trevor E. Carlson, Erik Hagersten, and David Black-Schaffer
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05, Uppsala, Sweden
{andreas.sembrant, trevor.carlson, erik.hagersten, david.black-schaffer}@it.uu.se

***Abstract*—Modern SoCs contain CPU and GPU cores to execute both general purpose and highly-parallel graphics workloads. While the primary use of the GPU is for rendering graphics, the effects of graphics workloads on the overall system have received little attention. The primary reason for this is the lack of efficient tools and simulators for modern graphics applications.**

**In this work, we present GLTraceSim, a new graphics memory tracing and replay framework for studying the memory behavior of graphics workloads and how they interact in heterogeneous CPU/GPU memory systems. GLTraceSim efficiently generates GPU memory access traces and their corresponding, synchronized, CPU render thread memory traces. The resulting traces can then be replayed in both high-level models and detailed full-system simulators.**

**We evaluate GLTraceSim on a range of graphics workloads from browsers to games. Our results show that GLTraceSim can efficiently generate graphics memory traces, and use these traces to study graphics performance in heterogeneous CPU/GPU memory systems. We show that understanding the impact of graphics workloads is essential, as they can cause slowdowns in co-running CPU applications of 26 - 59%, depending on the memory technology.**

## I. Introduction

The majority of today's processors are heterogeneous SoCs consisting of both CPU cores and GPU cores. To understand these systems, we need tools that enable architects to explore both the general-purpose applications that run on the CPU as well as graphics applications that use the GPU for rendering. For studying CPU workloads there are numerous tools available (e.g., Gem5 [7], Sniper [10]). Likewise, there are many tools for studying GPU *compute* (GPGPU) workloads (e.g., GPGPUSim [7], gem5-gpu [29]). However, tools for GPU *graphics* are either out-of-date [14] or not publicly available (e.g., ARM [**?**], [13]). This places the architecture community at a significant disadvantage, as we have little capability to analyze and evaluate the primary workloads run on the majority of these devices.

To address this problem, we present GLTraceSim: a fast and easy-to-maintain graphics tracing and replay framework for studying the impact of graphics workloads on memory systems performance. GLTraceSim generates graphics memory access traces with variable parallelism, and includes the synchronized memory accesses from the CPU render thread. The traces can then be replayed through fast high-level models or detailed simulators. We extend the Gem5 full-systems simulator with GLTraceSim, which allows us to simulate the effects of GPUs of varying sizes (degrees of parallelism). GLTraceSim is built upon well-maintained publicly available tools to ensure its longterm viability.

With GLTraceSim's capability to capture and inject graphics traces, we study the memory behavior of a range of graphics workloads and the interaction between the memory system and GPU performance. We investigate how contention for shared resources (e.g., DRAM and shared caches) in heterogeneous CPU/GPU systems affects performance by co-running graphics applications with CPU-only applications.

Our results show that GLTraceSim enables us to practically study graphics memory behavior across hundreds to thousands of frames. We see that graphics applications have working set sizes from 10s to 100s of MBs. These large working sets flush caches and stress the memory system resulting in CPU application slowdowns from 26 - 59% (on average), depending on memory interface technology (GDDR vs. LPDDR).

The key contributions of this work are:

- A new framework for generating graphics memory traces, that is fast and built upon well-maintained software components, and captures synchronization between the CPU render thread and GPU traffic.
- An investigation of memory system behavior across a wide range of graphics applications and cache sizes.
- The integration of graphics traces into the Gem5 memory system to study the effects of graphics workloads on heterogeneous systems.

## II. Graphics Tracing Framework

The goal of GLTraceSim is to provide a fast and maintainable simulation infrastructure for studying the interaction of graphics workloads with the memory system of heterogeneous CPU/GPU processors. To do so, GLTraceSim leverages and combines several well-maintained publicly available tools into a complete framework.

The GLTraceSim flow consists of four steps (Figure 1):

1) Capture the application's OpenGL graphics calls.
2) Replay the OpenGL calls through a software renderer to generate the graphics and CPU memory accesses, and detect CPU/GPU synchronization points.
3) Replay the generated trace through high-level models or detailed simulation in Gem5.

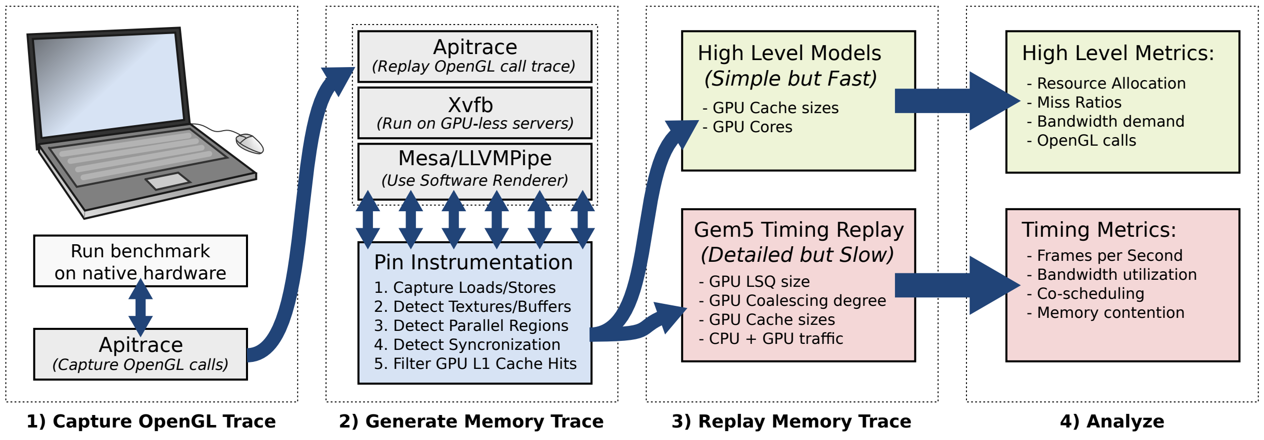

Figure 1. Simulation Flow.

4) After simulation we can then analyze the results for the memory system (high-level models) and performance (detailed simulation).

*Step 1: Capture the OpenGL Call Trace*

GLTraceSim uses the APITrace [2] tool to capture a trace of all OpenGL calls from an unmodified user application. The resulting trace file includes calls and their arguments, and varies in size with the complexity of the application (e.g., from 42 MB for scrolling in Facebook using the Chrome browser to 1.7 GB for rendering a game sequence in Xonotic [HD]). On average, APITrace captures the trace with a 25% overhead.

This approach renders applications on the native graphics hardware, which allows us to capture the trace with correct timing information. As the applications are rendered at high speeds on the native hardware, the application's internal logic for deciding which frames to generate based on the achieved frame rate will generate a trace that is roughly what one would expect from a normal execution. However, by capturing the trace at the particular frame rate of the hardware, we set an upper limit on the speed at which we can inject the resulting trace into our simulations. That is, we can simulate more slowly (by dropping frames) but if we need to simulate a higher frame rate, the trace will not correctly follow the application's behavior (i.e., inserting new frames). Our trace-based approach also provides determinism to the following simulation steps as the input (i.e., trace) does not change.

To be able to replay the OpenGL call trace later through the software renderer to generate the graphics address trace, we must run the application with the OpenGL version that the software renderer supports. Currently, Mesa's LLVMPipe driver is fully compatible with OpenGL 3.3, but it also supports many individual OpenGL 4-4.5 calls. To use the right OpenGL version, we can either run the application on hardware with that particular OpenGL version, or we we can force the correct version with runtime flags (e.g., *MESA_GL_VERSION_OVERRIDE*). In the evaluation, we capture the traces using a Intel HD 4000 GPU (Intel i5-3210M processor) that natively supports OpenGL 3.3.

*Step 2: Generate the Graphics Memory Trace*

GLTraceSim uses the OpenGL call trace file from Step 1, and replays the calls using glretrace from APITrace through Mesa's LLVMPipe [26] to render them in software. GLTraceSim instruments the LLVMPipe with an Intel Pin [9] tool to capture the actual graphics memory trace. We use the Xvfb [35] virtual framebuffer to enable execution on headless servers.

*1. Capturing Loads/Stores.:* The Pin tool uses information from LLVMPipe to determine which memory accesses to keep (graphics related memory operations), and which ones to discard (LLVMPipe specific memory operations).

*2. Detecting Textures/Buffers.:* To simulate memory accesses from a GPU, we are only interested in accesses to framebuffers, textures, and vertex buffers (i.e., only accesses to the GPU's memory). To identify these, we monitor when resources are created (*llvmpipe_resource_create*) and destroyed (*llvmpipe_resource_destroy*) in the software renderer. Any memory accesses to addresses that are mapped to an active graphics resource are classified as accesses to GPU memory and are saved to the trace files. This removes approximately 93% of all of the software renderer's memory accesses as these accesses are software overhead (e.g., spill and fill between registers and the stack).

The resource allocation information is also stored for later analysis. For example, we can use it to determine how much GPU memory is allocated and what type of textures are accessed (e.g., 1D, 2D, mipmap-level, compression, etc.).

*3. Detecting Parallel Regions.:* While GPUs execute thousands of threads in parallel, our software renderer is only capable of executing tens of render threads. To provide the flexibility to choose the degree of parallelism (e.g., the number of GPU cores) in later simulations, we capture information about which portions of the rendering can execute in parallel. We takes advantage of LLVMPipe's tile-based renderer[1],

[1]Tile-based rendering has been traditionally used in mobile graphics GPUs as a way to reduce bandwidth requirements (e.g., ARM [12]). Modern high-performance graphics GPUs have shifted to tile-based rendering (e.g., NVIDIA [20]) and similar caching schemes (e.g., AMD [33]).

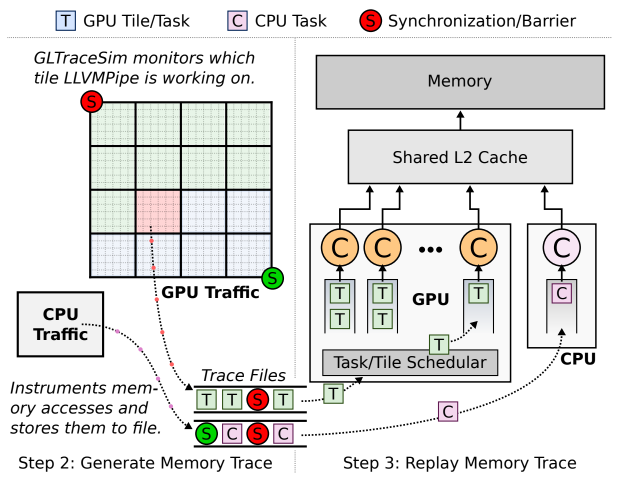
T GPU Tile/Task
C CPU Task
S Synchronization/Barrier
GLTraceSim monitors which tile LLVMPipe is working on.
GPU Traffic
CPU Traffic
Trace Files
Instruments memory accesses and stores them to file.
Memory
Shared L2 Cache
GPU
Task/Tile Schedular
CPU
Step 2: Generate Memory Trace
Step 3: Replay Memory Trace

Figure 2. Capturing tile memory accesses and CPU/GPU synchronization points from the software renderer (Step 2) and replaying the resulting trace into a high-level trace-based cache simulator (Step 3).

where each tile can be executed in parallel.

Figure 2 illustrates how we capture tile memory accesses. We detect when a tile starts (*lp_rast_tile_begin*), and tag each memory access with its tile. The replay engine (Step 3) can then simulate an arbitrary number of GPU cores by scheduling the different parallel tiles to the simulated cores.

This approach captures memory accesses from the back end of the GPU pipeline (e.g., the fragment shader). To detect memory accesses from the front end, we monitor when the software renderer executes vertex shaders and geometry shaders (*llvmpipe_draw_vbo*). This enable us to determine where memory accesses originate and to separate the memory accesses into CPU and GPU access streams by determining if a memory access happens during an LLVMPipe instrumented call (GPU access) or not (CPU access).

*4. Detecting Synchronizations.:* The memory access trace is then split into several streams based on the origin (CPU or GPU) and the parallel render tiles. In order to replay the different streams, we need to determine how they synchronize with each other. There are three main synchronization points: 1) command flush (e.g., glFlush), 2) GPU task synchronization, and 3) CPU/GPU resource transfers (e.g., textures/buffers).

**Command Flush Synchronization.** We insert a global synchronization barrier in all streams when we detect flush commands (*llvmpipe_flush*).

**Tile Synchronization.** We monitor when a scene starts processing tiles and when the scene ends (*lp_rast_begin* and *lp_rast_end*), and insert synchronization barriers in the GPU access stream accordingly. This models the internal GPU synchronization to ensure that all tasks have been scheduled from the current scene.

**Resource Transfer Synchronization.** The GPU can not read a resource (e.g., texture/buffer) before the resource has been created by the CPU. To account for this, we monitor resource usage and insert a resource barrier in the CPU and the GPU streams when a resource that was last used by the CPU is read/written to by the GPU, and vise versa.

| Name | Type | Suite | BW | FPS |
|---|---|---|---|---|
| Trex | Scene | GFXBench | Med | $\approx 60$ |
| Manhattan | Scene | GFXBench | Med | $< 60$ |
| Heaven (Unigine) | Scene | Phoronix | High | $< 60$ |
| Valley (Unigine) | Scene | Phoronix | High | $< 60$ |
| Tesseract | Game | Phoronix | Med | $\approx 60$ |
| OpenArena | Game | Phoronix | Med | $< 60$ |
| Xonotic [LD] | Game | Phoronix | Low | $> 60$ |
| Xonotic [HD] | Game | Phoronix | Med | $\approx 60$ |
| Chrome | Browser | Telemetry | Low | $> 60$ |

Table I
GRAPHICS BENCHMARKS.

*5. Filtering GPU L1 Cache Hits.:* GLTraceSim does not model the low-level details of the GPU's functional ALU units, SIMD lanes, etc., as we are only interested in the GPU's off-core memory traffic. We therefore have to remove the short reuses in the memory access stream which would be typically filtered by the local L1 caches in a hardware GPU. To do so, each renderer thread simulates a 32 kB 4-way filter cache, and removes any memory accesses that hit in this cache from the memory access trace. This filters out about 70% of all the memory accesses to GPU memory/resources.

Despite L1 filtering, and the use of a binary format and gzip compression, the resulting trace files can be quite large (e.g., 82 GB for GFXBench/Manhattan). Step 2 is able to generate about 100k filtered memory accesses per second and store them to the trace file. For Manhattan, it takes on average 2.6 minutes to process one frame, whereas it only takes 6.5 seconds to process one frame for the Chrome browser.

### *Step 3: Replaying Memory Traces*

GLTraceSim replays the memory access traces to either a fast high-level model or to the detailed Gem5 full-system simulator. The high-level replay models a CPU core, a number of GPU cores, and a shared LLC cache (see Step 3 in Figure 2). It reads the trace file, and schedules tiles to the GPU cores, which then issue memory requests to the shared cache. By modeling at this abstraction level, we are able to study the memory behavior of thousands of frames (0.13-16 seconds of simulation time per frame), which is not possible with the more detailed simulation model in Gem5, due to its low simulation speed (85-9574 seconds per frame). The detailed replay model in Gem5, however, allows us to simulate full-system timing effects, which are essential for analyzing the performance impact of the GPU memory traffic on other cores.

### *Step 4: Analyze*

The final step is to analyze the results. From the high-level model, we can obtain cache miss ratios and bandwidth demands over long periods of time (hundreds of frames). For the detailed Gem5 model, we can measure performance effects, including the number of frames rendered per second, DRAM-page hit rates, and the effects of co-running graphics workloads with CPU-only applications (Section IV).

## III. GRAPHICS CHARACTERIZATION

### *A. Methodology*

To evaluate accuracy, we use high-level modeling to compare the memory system behavior reported by GLTraceSim

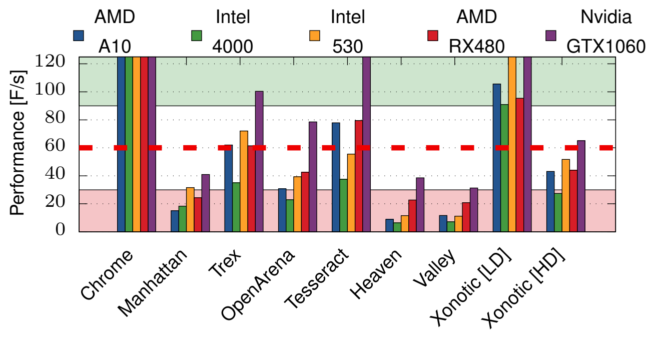

Figure 3. Graphics Performance Overview: Achievable frame rates vary dramatically by application complexity and GPU performance.

over thousands of frames to data captured from three integrated GPUs: Intel's HD4000 and HD530, and AMD's A10/R7. We use integrated GPUs as they provide direct memory bandwidth measurements from the shared CPU/GPU memory controller's performance counters. We also use two discrete GPUs (AMD RX480 and NVIDIA GTX1060) to characterize the graphics applications' performance. We use the average of ten benchmark runs in each case to reduce sampling noise.

GLTraceSim's OpenGL call trace capture (Step 1) was run on an Intel HD4000 system, while generating the memory traces (Step 2) and replaying them (Step 3) were executed on an Intel Xeon compute cluster. All traces are captured at a resolution of 1920×1200.

## *B. Graphics Benchmarks*

Our graphics workloads come from three graphics benchmarks suites: GFXBench [17], Phoronix Test Suite [28], and Chrome/Telemetry [31]. Table I gives an overview of the different benchmarks, their bandwidth, and the achieved frame rates on our OpenGL call trace capture machine. Manhattan, Trex, Heaven and Valley render complex scenes, while Tesseract, OpenArena, and Xonotic are portions of games. We use two versions of Xonotic, one with low definition (LD) graphics settings and one with high definition (HD) settings. Finally, we render several web pages with the Chrome browser, as it uses OpenGL textures to display and scroll its content, and use Telemetry to control the application's behavior.

Figure 3 shows the wide range of performance (in frames per second) achieved by the different benchmarks across the GPUs. The benchmarks are further classified by their bandwidth (BW) and Frame per Second (FPS) in Table I. For example, the Chrome web browsing benchmark *Amazon* has a very low bandwidth requirement, and all graphics cards can deliver more than 60 frames per second for it. *Heaven*, on the other hand, has a high bandwidth demand and delivers very low performance on even the most powerful GPU.

## *C. Accuracy: Memory Bandwidth*

To compare the results from GLTraceSim, we examine DRAM bandwidth usage over time in MBs per Frame in Figure 4. We use the frame number as a proxy for time in order to compare the different GPUs with the simulator, since this enables us to align the statistics from the GPUs for precise comparisons. The light colors in the background show the measured bandwidth from sampling the memory controller's performance counters. The performance counters measure all DRAM activity in the system, including traffic from the CPU. For the Chrome web browser benchmarks (a), which use as little as 20MB/frame, the impact of the background memory system activity is particularly noticeable. To highlight the bandwidth trend, we plot a smoothed version of the measured performance counter data in darker lines.

We compare the three baseline integrated GPUs with three GLTraceSim configurations: **2C-256kB**: a low-end GPU that has only two GPU cores (i.e., it can execute two tiles in parallel), and a shared last-level 256 kB cache, **16C-512kB**: a medium-sized GPU with 16 GPU cores and a 512 kB last-level cache, and finally, **32C-4MB**: a large GPU with 64 GPU cores and a 4 MB last-level cache.

**Low Bandwidth Applications: Chrome (a).** The Chrome web browser uses OpenGL to display web pages by splitting the web page into several tiles, and drawing each tile to a GPU texture. The GPU renders the page by piecing together the individual textures into a full web page. To scroll the web page, the GPU only has to move the textures up. Chrome uses tiles that are 256x256 pixels large, which is 16 times larger than our simulated GPU's tile size.

Figure 4a) shows Chrome loading and scrolling a wide variety of different web pages. As expected, the bandwidth demand is fairly low since the GPU only performs simple 2D graphics operations. The AMD-A10 uses less bandwidth than the two GPUs from Intel, but all show similar time-varying behavior. For example, we see a noticeable bandwidth increase when browsing Facebook compared to Ebay.

GLTraceSim only simulates accesses to graphics memory, whereas the baseline systems include accesses from the whole system (CPU, GPU, and IO devices). We therefore expect GLTraceSim to report lower bandwidth compared to the hardware baselines. The figure shows that GLTraceSim exhibit similar bandwidth trends over time as the baselines, particularly with regards to activity spikes. Of the three baselines, GLTraceSim tracks AMD's A10 bandwidth the closest.

All three GLTraceSim configurations have roughly the same bandwidth requirements because the graphics memory footprint is larger than the last-level GPU cache size, and the textures are not reused within frames (i.e., the last-level GPU cache is flushed every frame, and there is no temporal reuse).

**Medium Bandwidth Applications: Manhattan (b).** Manhattan renders a complex action scene with many special effects. As with modern games, Manhattan does not render everything in one pass, but instead breaks up the rendering into many separate steps [1]. This process is shown in Figure 5, where the application first renders partial shading information into different buffers (1), which are then merged (2), and shadows and special effects such at lens flares are added (3). Bloom and Depth of Field and other post-processing are then added (4). Finally, 2D graphics overlays, such as maps and text, are generated (5), and merge into the final buffer (6) that is displayed on the screen.

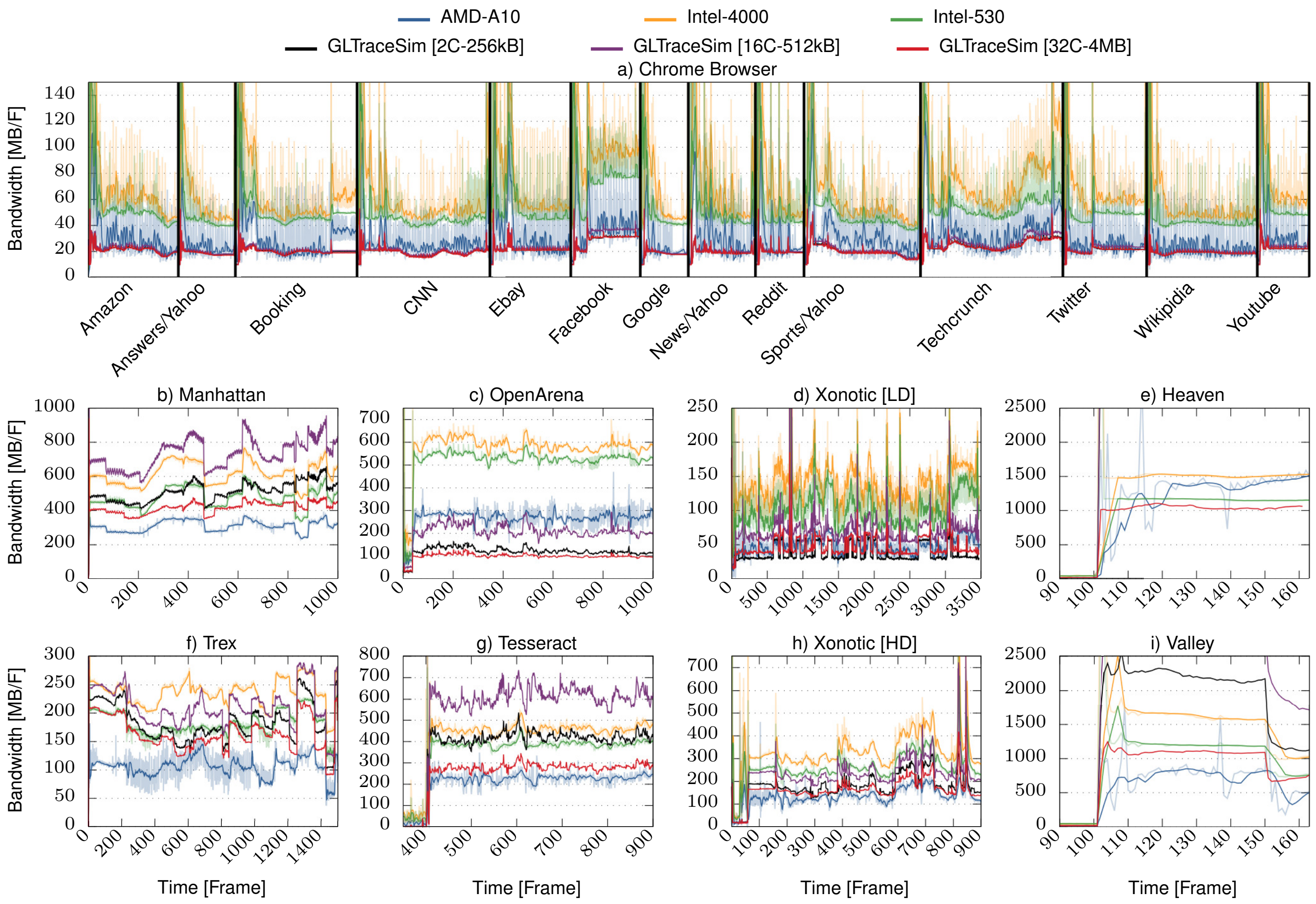

Figure 4. Bandwidth consumption per frame for hardware measured using performance counters and the GLTraceSim framework. Above, Google Chrome rendering 14 popular web pages. Below, 8 games and walk-throughs demonstrate significant variation in bandwidth consumption over-time and across benchmarks. Results from GLTraceSim closely track the bandwidth consumption trends seen in hardware.

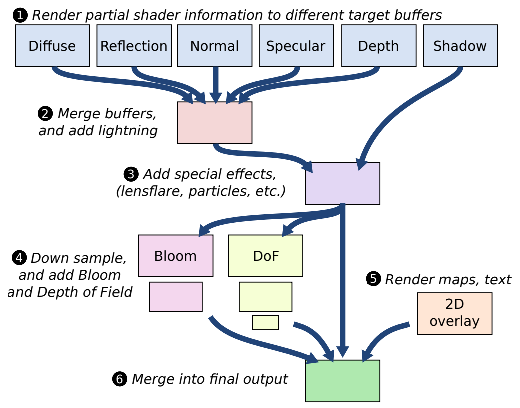
❶ Render partial shader information to different target buffers
Diffuse
Reflection
Normal
Specular
Depth
Shadow
❷ Merge buffers, and add lightning
❸ Add special effects, (lensflare, particles, etc.)
❹ Down sample, and add Bloom and Depth of Field
Bloom
DoF
❺ Render maps, text
2D overlay
❻ Merge into final output

Figure 5. The complex rendering pipeline of modern graphics applications [15], [1]. Each box represents a texture rendered as part of the creation of the final frame.

The more complex Manhattan benchmark requires much higher bandwidth than Chrome. On average, the AMD-A10, Intel-4000, and Intel-530 need 311, 620, 473 MB per Frame, respectively. To render the scene at 60 FPS, the memory system needs to be able to transfer 18, 36, 28 GB/s of data.

However, this behavior varies over 10s to 100s of frames. For example, as the camera moves through the scene new objects become visible and others are occluded. This is visible as a large drop in bandwidth demand for the Intel-4000's and Intel-530's bandwidth demand at frame 463, and then a big jump at frame 618. This bandwidth change is captured by GLTraceSim, particularly in the 2C-256 kB and 16-512 kB configurations, which both show the drop and increase. The 32C-4 MB configuration has a larger cache than 2C-256 kB and 16-512 kB, which filters out much of those bandwidth spikes. AMD-A10 has similar performance to 32C-4 MB, and shows a much smaller performance variation. It is important to note that to see this one needs to simulate over 100 frames between those two points, which requires a very efficient simulation infrastructure given each frame's complexity.

**High Bandwidth Applications: Heaven (e).** Heaven is the highest bandwidth benchmark. The first 100 frames are initialization, wherein it loads the required textures etc., but the bandwidth demand increases dramatically when it finally starts to render the real scenes at frame 102. On average, Heaven requires 1.3, 1.5, and 1.2 GB per Frame for AMD-A10, Intel-4000, and Intel-530, respectively.

The 2C-256 kB and 16-512 kB configurations require 2.3 and 3.2 GB per Frame, respectively, and exceed the y-axis. The 32C-4 MB configuration has more cache capacity, and has the lowest bandwidth demand at about 1 GB per Frame.

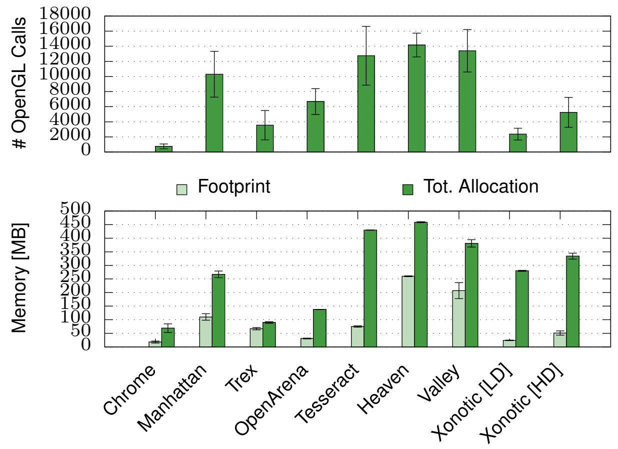

Figure 6. OpenGL Calls (top) and Memory Usage (bottom) per Frame.

This shows that GLTraceSim can effectively model a wide range of memory behavior, and shows how the increase in cache capacity can effectively reduce GPU graphics bandwidth requirements (See Section III-E for a more detailed analysis on the cache behavior of Heaven).

**Game Settings: Xonotic (d, h).** We ran Xonotic (a first person shooter) with both low definition (LD) graphics quality settings (Figure d) and high definition (HD) settings (Figure h). LD has many more frames (3500 vs. 900). This is because GLTraceSim is able capture OpenGL call traces at a much higher rate at the lower quality as the native hardware can render the frames more quickly.

In Xonotic, the player runs through a building. We therefore see large changes in bandwidth demand as the player moves between rooms and encounters different opponents. If we compare the two plots we see that HD requires about $3\times$ more bandwidth than LD. On average, (137 vs. 49), (329 vs. 136), and (266 vs. 103) MB per frame for AMD-A10, Intel-4000, and Intel-530, respectively. Again, this benchmark shows significant variation in behavior across hundreds of frames, which underlines the importance of a simulation methodology that can efficiently reproduce such long executions.

**Summary.** The comparison of graphics bandwidth between GLTraceSim and hardware GPUs in shows that: 1) graphics applications have a wide variety of memory behaviors over time spans of 10s to 100s of frames, 2) modern GPUs have noticeably different bandwidth requirements, 3) GLTraceSim can model different types of GPUs by varying number of GPU cores and shared cache capacity, and finally 4) GLTraceSim is able to realistically model the bandwidth demand and runtime performance changes in applications' memory behavior.

## *D. OpenGL Graphics Programming*

Our benchmarks use the OpenGL API to communicate with the GPU. In this section, we study how they use OpenGL, including the number of library calls, and memory usage. The graphs in the rest of this section do not include data from initiations phases (e.g., frame 0 to 100 in Heaven).

**Graphics Calls.** Figure 6 (top) shows average number of OpenGL calls per frame, and the error bars show the standard deviation. Chrome calls OpenGL the least with an average of 738 library calls per frame. Heaven calls OpenGL the most with over 14 thousand times per frame. This large number of API calls from the CPU underlines the importance of including CPU performance and synchronization between the GPU's and CPU's memory access streams.

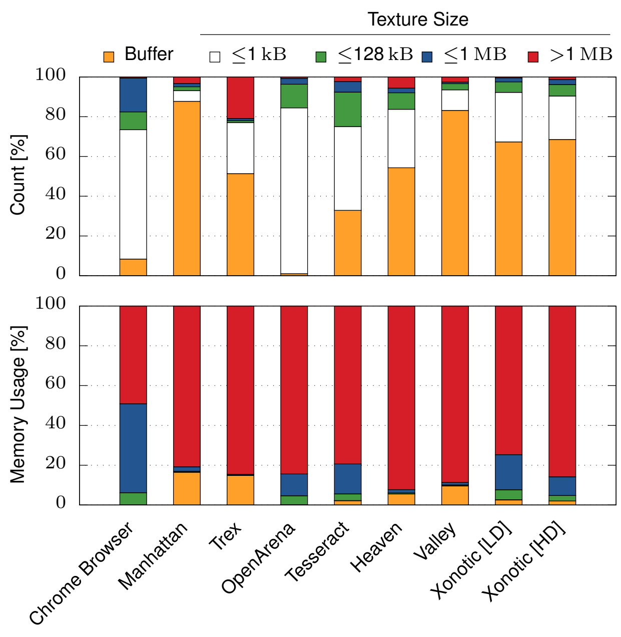

Figure 7. Memory Breakdown.

**Memory Usage.** Figure 6 (bottom) shows the data footprint (the amount of data touched) per frame, and the total amount of allocated GPU memory. Most applications allocate all textures that will be used in the near future in advance to avoid later stalls. The result is that only a fraction of the allocated textures are actually used in most frames. For example, in Tesseract, only 17% of the allocated data is actually used.

**Memory Breakdown.** Figure 7 shows memory usage by Buffers and Textures, with textures displayed based on their size: Small (≤1 kB), Medium (≤128 kB), Large (≤1 MB), and Huge (>1 MB).

The top figure shows the number of resources used during a frame for each category as percent of the total number of used resources. For example, 65% of Chrome's resources per frame are textures with a memory footprint less than 1 kB. In Manhattan, 88% of the resources are buffers. The bottom figure shows the total footprint of each category. While there are few Huge textures, they have the largest footprint. The exception is Chrome, which has several Large textures (the 256x256 pixels web page tiles). The remaining footprint comes from the framebuffer, that is categorized as a Huge texture.

## *E. Cache Requirements*

We have seen that graphics bandwidth varies significantly over time and that benchmarks have different distributions of resource sizes. To understand the impact of this on bandwidth, we now evaluate how varying the cache size affects the GPU's

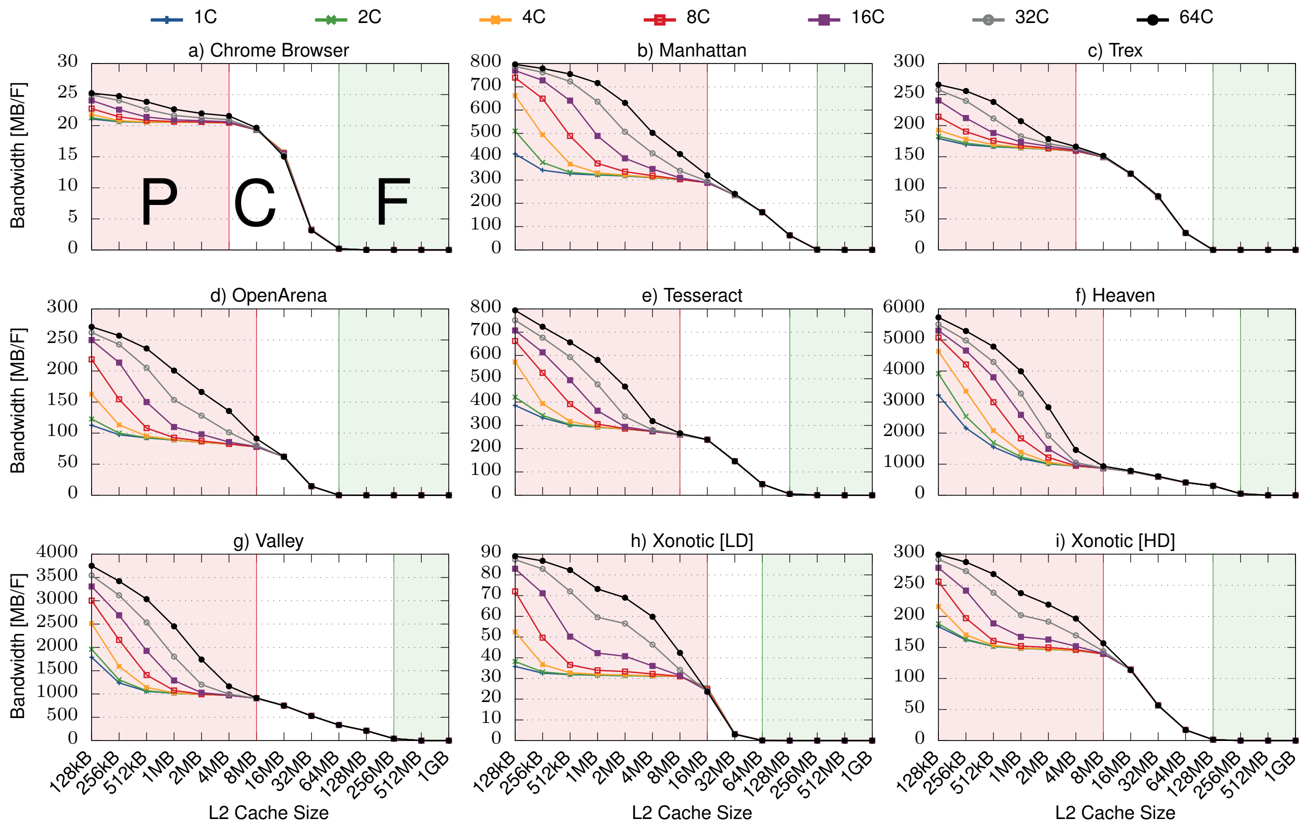

Figure 8. DRAM bandwidth demand for a variety of cache sizes and GPU core counts. The three regions, Pollution (red, area P), Capacity (white, area C) and Footprint (green, area F), show the three distinct phases present in graphics rendering. Larger core counts can significantly benefit from an increasingly larger cache (transition from P to C and F areas).

bandwidth need. Figure 8 shows the bandwidth ( MB/F) as a function of cache size for GPUs with 1, 2, 4, 8, 16, 32, and 64 cores (i.e., the 64-core version can process 64 tiles in parallel). The graphs are divided into three areas to highlight different cache behavior: intra-GPU core Pollution (P, left area), cache Capacity (C, middle area), and application Footprint (F, right area). Note that in all cases the individual GPU cores are modeled to include private L1 caches.

**Pollution (Area P).** The P area shows that as the number of cores increases, more pressure is put on the cache, which increases the miss ratio and thereby the required memory bandwidth. For Manhattan (b), we see that a 64-core GPU needs a 16 MB cache to maintain the same bandwidth demand as a 1-core GPU with a 256 kB cache. That is, the cache requirement scales linearly with the number of cores. However, this is not always the case. For Trex (c), the 64-core GPU only needs a 4 MB cache instead of a 16 MB cache to avoid the extra bandwidth over the 256 kB cache (i.e., 16:1 ratio). Trex has a smaller footprint than Manhattan (Figure 6), which reduces the pressure and can result in constructive interference when the same texture is used in multiple tiles at the same time.

**Capacity (Area C).** The C area shows the effect of how well the cache captures temporal reuse. For example, when the same texture is used multiple times during a frame to render different objects. OpenArena (d) and Xonitic [LD] (h) have curves that drop rapidly. This results in a a bi-modal behavior where the data suddenly fits in the cache. Heaven (f) and Valley (g), however, have steadily decreasing curves. This indicates that adding more cache will not necessarily fit the whole working set size, but it will improve performance.

**Footprint (Area F).** The F area shows where the whole footprint fits in the cache (the footprint in Figure 6). For example, OpenArena (d) needs 64 MB, and Heaven (f) needs 256 MB. In order to fit the whole footprint for all applications, we need a cache capacity of 256 MB, which is far greater than traditional SRAM-based caches can provide, but within the scope of DRAM caches. Note that this capacity reflects running the applications at a resolution of 1920×1200. For 4k resolution, the bandwidth requirements are expected to increase by 5× [22], requiring a much larger cache to sustain this performance with a similar memory bandwidth.

## IV. CPU/GPU Memory System Performance

The previous section explored the behavior of graphics workloads in isolation. To study how graphics traffic affects whole system performance, we now integrate GLTraceSim with the Gem5 [7] full-system simulator by providing detailed replay of the graphics memory trace through the Gem5 memory hierarchy, together with Gem5's own simulated CPU core traffic.

Figure 9 shows an overview of the Gem5-simulated system.

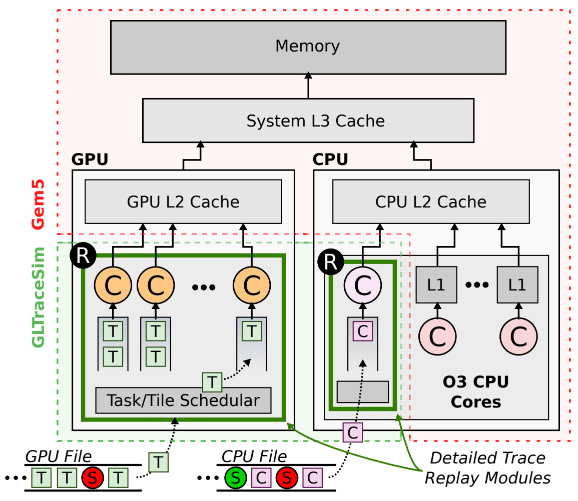

Figure 9. Detailed Replay with Gem5.

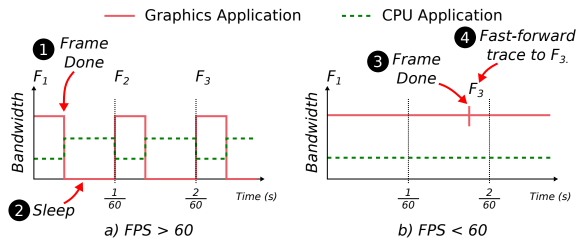

Figure 10. The replay module synchronizes the GPU's effective execution rate (how quickly it can generate memory accesses) with the target frame rate (60 FPS).

GLTraceSim components (GPU tile scheduler and trace replay, and synchronized CPU render thread trace replay) are in green and the Gem5 components (memory system and simulated CPU cores) in red. The simulated system has a system-level L3 cache that is shared between the GPU and the CPU. Both the GPU and the CPU have their own private L2 caches. The GPU consists of the GLTraceSim *GPU replay module* that replays the GPU traffic into the Gem5 memory system R. The CPU side has one GLTraceSim *CPU replay module* for the CPU traffic generated by the graphics application, which is synchronized to the GPU memory trace as discussed earlier, as well as several standard Gem5 out-of-order cores that simulate CPU workloads in detail. The Gem5 cores have their own L1 instruction and data caches. The replay modules do not have L1 caches since the traces have already been filtered with a L1 filter cache during trace generation (Step 2).

## A. *Gem5 Extensions: A Detailed Replay Module*

The detailed replay module for Gem5 consists of four primary components: 1) a load/store queue that controls the speed of the GPU trace replay into the Gem5 memory system, 2) a coalescer that groups neighboring memory accesses together to reduce the DRAM-page miss rate, 3) a rate controller that drops frames or puts the GPU to sleep depending on the target frame rate and what the system can achieve, and finally, 4) a synchronization controller that synchronizes the GLTraceSim GPU replay module with the GLTraceSim CPU replay module.

**Load and Store Queue.** The LSQ limits the replay model's number of outstanding memory requests. When the queue becomes full, the replay module stalls until its requests are processed. This back-pressure from the memory system allows us to model the interaction between the GPU-injected memory stream and those of the simulated CPU cores. The user can adjust the performance level of the modeled GPU by changing the LSQ depth and number of GPU replay cores.

**Memory Coalescer.** The GPU can generate memory accesses from many different accesses stream in parallel depending on its core count. This results in many DRAM page misses since each stream may accesses a different page. To improve the DRAM performance, the coalescer groups requests to 2 kB regions (i.e., DRAM page size) together, such that they are issued after each other if possible. This is typically handled directly in hardware for GPUs.

**Rate Controller.** We model a GPU that targets the frame rate of the captured trace (typically 60 FPS). If the system is unable to meet this rate (e.g., it cannot inject GPU traffic at a high enough rate due to memory system bottlenecks/contention or too few parallel tiles) then GLTraceSim will skip GPU frames, as would a real GPU. If the modeled GPU can render faster than this, then we “sleep" the GPU for the time available time between frames. As discussed earlier, this is an artifact of having only captured GPU data for a certain maximum frame rate. This interaction is handled by the *Rate Controller*, and is different from typical GPGPU simulations, which run at their maximum compute or memory capability at all times.

Figure 10a) shows the case when the GPU can render more than 60 FPS. The red line shows graphics application's bandwidth and the green line shows the CPU application's bandwidth. Here the GPU finish rendering the frame within $1/60$ s ❶, and goes to sleep until the next frame starts ❷. This results in fluctuating bandwidth demand which affects the CPU applications performance. The replay module must therefore pause the trace replay (i.e., sleep).

Figure 10b) shows the case when the GPU is not able to render 60 FPS. Here the GPU finishes rendering the frame after more than $1/60$ s ❸. The replay module must then drop a frame to keep up. To do so, it drops $F_2$ by skipping ahead in the trace replay to $F_3$ ❹.

**CPU $\leftrightarrow$ GPU Synchronization.** The OpenGL API provides an asynchronous queue for sending work from the CPU to the GPU. In many situations, this provides sufficient slack such that the GPU is the main performance bottleneck. However, as mentioned in the previous sections, we need to synchronize the CPU and the GPU replay modules with each other at particular synchronization points, which can limit the potential performance. This synchronization happens in two ways: 1) through the trace files using synchronization messages that guarantee that both modules are at the same point in time before they are allowed to send new memory requests (e.g., when creating a texture, we pause the GPU until the textures is moved from the CPU to the GPU), and 2) through the Rate Controller when the GPU goes to sleep and when the GPU has to drop frames. This synchronization enables us to model

| | | |
|---|---|---|
| CPU Gem5 Frequency / Cores | 2.5 GHz, 4 cores | |
| – Width: F / D / R / I / W / C | 3 / 3 / 3 / 8 / 8 / 8 | |
| – Size: ROB / IQ / LQ / SQ / IR. / FPR. | 128 / 64 / 16 / 16 / 128 / 128 | |
| CPU Replay Core Freq. (Little \| Big) | 1.5 GHz | 2.5 GHz |
| – LSQ Size (Little \| Big) | 4 entries | 16 entries |
| CPU L1 Instruction / Data Caches | 32kB, 64B, 8-way, Stride PF (D:8) | |
| CPU L2 Unified Cache | 256kB, 64B, 8-way | |
| GPU Frequency / Cores | 2.5 GHz, 16 cores | |
| GPU LSQ Size | 1024 entries | |
| GPU L2 Cache | 512kB, 64B, 8-way, 1024 MSHRs | |
| System L3 Cache | 2MB, 64B, 16-way | |
| System DRAM (LPDDR \| GDDR) | LPDDR3-1600 | GDDR5-4000 |

Table II
PROCESSOR CONFIGURATION.

the effects of how the CPU performance of the render thread affects and is affected by overall system performance.

## B. Experimental Setup

We configured Gem5 (ARM ISA) to simulate a contemporary energy-efficient processor. The CPU has 4 Gem5 OoO cores, a GLTraceSim CPU replay core, a shared 256 kB L2, and runs at 2.5 GHz. We use two GLTraceSim CPU replay core configurations (Little/Big) to evaluate how the CPU affects graphics performance. The GPU has 16 cores and a shared 512 kB L2. There is a system-level 2 MB L3 cache (Figure 8 shows that this is enough cache capacity to avoid cache pollution). We evaluate both LPDDR3 and GDDR5 memory to study the effect of memory bandwidth. The Gem5 caches are non-inclusive/non-exclusive (i.e., no back invalidations), meaning that the GPU will not flush the CPU's private caches if it uses all of the L3 cache.

To evaluate the impact of the graphics workloads on CPU performance, we co-run SPEC CPU2006 applications [18] with their reference inputs together with the the graphics workloads Chrome/Facebook and Heaven with the GLTraceSim GPU trace replay and synchronized GLTraceSim CPU trace replay. The Chrome web browser can easily meet the 60 FPS target, resulting in the behavior similar to Figure 10a, whereas Heaven illustrates what happens when the GPU cannot finish rendering the frame within the 60 FPS time frame (Figure 10b).

## C. CPU Performance

Figure 11 shows the SPEC applications' *CPU performance* (normalized IPC) when they are paired with the two GPU applications (Chrome/Facebook and Heaven). In this section, we discuss the *CPU* performance changes we see with respect to the GPU application intensity, memory bandwidth and GPU/CPU synchronization.

**GPU Application Intensity (Chrome vs. Heaven).** The two graphics applications demonstrate the differences between high GPU intensity (Heaven, red) and low intensity (Chrome, green) behavior. Low intensity graphics applications (Figure 10) cause the GPU to periodically sleep allowing exclusive access to shared resources by the CPU applications. On the other hand, high-intensity benchmarks are continuously consuming cache and memory resources, hurting CPU application performance by up to 59% on average (and more than 5× for soplex) for the LPDDR/Big configuration.

Surprisingly, hmmer see a 50% performance degradation for the LPDDR configuration despite having a very low 1.0% L1D miss ratio. This happens because each L1 miss is extremely costly: the L1D misses also miss in both the L2 (58.8%) and L3 (26.1%) caches due to cache pollution, and end up competing for DRAM bandwidth with the GPU.

One exception to this behavior is povray, which is an L2-resident application. Misses in the L1 cache (1.4%) successfully hit in the L2 cache (>99.9%) and therefore do not compete with the GPU for memory bandwidth. As a result, povray shows <10% performance degradation with GPU traffic.

**Memory Bandwidth (LPDDR vs. GDDR).** The LPDDR and the GDDR results show that increasing the memory bandwidth provides significant performance speedups for the SPEC applications when they are co-executed with Heaven (30% to almost 100% on average). The higher bandwidth is better able to accommodate both the GPU and SPEC applications, reducing memory contention and improving performance. Chrome stresses the memory system less than Heaven, and we therefore only see less than a 25% performance improvement, even with GDDR's increased bandwidth.

**Synchronization (Little vs. Big core).** The top (a, Little) and the bottom (b, Big) show the effect of the graphics CPU replay core's performance. For the SPEC CPU applications, we see an average performance drop of 40% when paired with the Little graphics CPU replay core, and almost 60% when paired with the Big CPU replay core (on average).

The dealii benchmark is not affected by Heaven when we use the Little CPU replay core, but the performance drops by 40% when we switch to the Big CPU replay core. The soplex benchmark is affected even more, where the 3× performance degradation drops to 5× with the Big core. Both of these applications are heavily influenced by the resulting increase in DDR memory traffic from the Big replay core, which increase the time it takes to access DRAM for the CPU applications.

## D. GPU Performance

Figure 12 shows the *graphics performance* in frames per second (FPS) normalized to LPDDR3 with a Big graphics CPU replay core while running in isolation.

**GPU Application Intensity (Chrome vs. Heaven)** While GPU performance can be affected by co-running applications, the single-threaded SPEC applications do not significantly reduce the available memory bandwidth, nor do they slow the render thread's performance enough to reduce the frame rate. This is clearly seen by the lack of any significant impact on GPU performance in Figure 12.

**Memory Bandwidth (LPDDR vs. GDDR).** Heaven's maximum frame rate increases by almost 25% across all applications for both Big and Little graphics replay cores with GDDR5 memory. The Chrome browser application improves even more (50%) on the Big graphics CPU core. As this application is limited by the speed of its GLTraceSim CPU replay core with the Little core, it is unable to exploit the additional GDDR memory bandwidth, resulting in a small performance improvement.

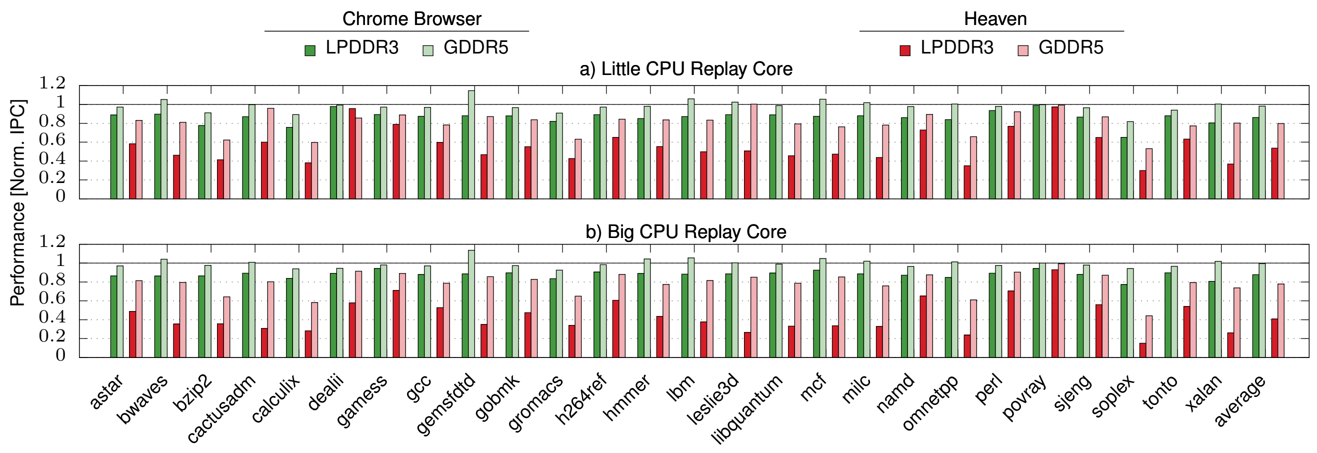

Figure 11. CPU application performance in instructions per cycle (IPC) with graphics workloads, normalized to an LPDDR3 configuration with the application running in isolation (no graphics workload).

**Synchronization (Little vs. Big core).** The graphics CPU replay core's performance affects the system in two ways: it limits the GPU's bandwidth (as the GPU cannot proceed past CPU/GPU synchronization points until the graphics CPU replay core catches up), and it puts pressure on the shared L2 cache due to its own memory access stream. We see a slight performance reduction for Heaven when using the Little graphics CPU replay core instead of the Big CPU replay core (red, Figure 12 bottom vs. top). Heaven is GPU bandwidth-bound and is mostly unaffected by a slower graphics CPU replay core. The Chrome browser, however, sees almost a 50% performance loss with the Little graphics core compared to the Big core as it is much more CPU-dependent.

### *E. Summary*

GLTraceSim's simulated GPU graphics subsystem, with both the GPU and CPU replay cores, interacts in a variety of ways with Gem5's regular execution-driven CPU application simulations. The GPU, given its high bandwidth demands, reduces both the available DRAM bandwidth and cache capacity, as seen by its impact on the SPEC CPU2006 applications. This has a varying impact on the applications depending on their cache/bandwidth characteristics and data footprint. GPU graphics applications, in this study, are not greatly affected by the CPU2006 applications, but instead show performance differences with hardware changes, such as increased bandwidth or a more powerful CPU replay core. Only with a global view of both GPU, render thread, and traditional applications can we evaluate the system-level impact of co-running applications.

## V. Related Work

**GPGPU Simulators.** There has been significant work developing infrastructure for studying general purpose compute on GPUs (e.g., GPGPUSim [5], gem5-gpu [29], Multi2Sim [32], GPUTejas [25], Parallel GPU Simulation [23], MIAOW [6], Nyami [8], and Barra [11]). GPGPUSim is a detailed simulation model, whereas MIAOW is an open-source hardware implementation of a GPGPU. These approaches have been instrumental in understanding how to optimize GPUs for compute. However, they are not able to run graphics workloads. The goal of GLTraceSim is to deliver insights about the memory system interaction between GPUs running graphics workloads and the rest of the system. Unlike these simulators, GLTraceSim does not model the graphics rendering in detail, but rather uses a functional simulation to rapidly generate memory traces over hundreds of frames.

**Dynamic GPGPU Instrumentation.** GPU-Ocelot [16] and SASSI [30] enable the instrumentation of CUDA applications and GT-Pin [21] enables the instrumentation of OpenCL applications in a similar vein to Intel's Pin for CPUs. However, as with GPGPU simulators, these tools only allow the instrumentation of compute applications, and not graphics.

**GPGPU Modeling.** Hong et al. [19] and Baghsorkhi et al. [4] propose analytical models and Wu et al. [34] propose a machine learning model for performance. However, these models do not address graphics.

**GPU Graphics Simulators.** The closest related work is the ATTILA [14] simulator and TEAPOT [3]. ATTILA is an detailed micro-architectural simulator for studying the GPU rendering pipeline. However it studies much older graphics pipelines (OpenGL 1-2 and DirectX 9), which are no longer representative. The goal of GLTraceSim is to study the memory system in heterogeneous CPU/GPU systems, and not low-level GPU pipeline details. This enables us to simulate at a higher abstraction level resulting in a faster simulation, which allows us to study long-running applications with significant memory footprints. TEAPOT focus on Android applications (OpenGL ES), and is like many other GPU simulators not publicly available.

**GPU-less Graphics Simulators.** Two techniques for simulating the CPU component of graphics applications are NoMali [13] and a Simics-based [24] simulation technique [27]. These methodologies allow the analysis and simulation of the CPU portion of a workload by ignoring or offloading the details of the GPU execution. However, as we have seen

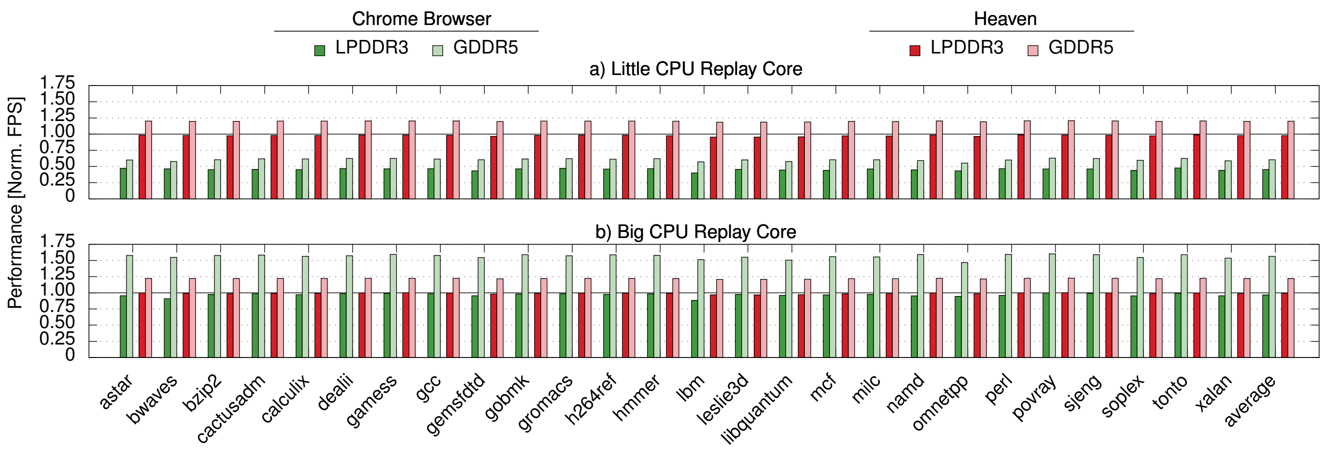

Figure 12. GPU performance in frames per second (FPS) normalized to the GPU application running in isolation with the Big graphics CPU replay core.

in this work, most of the DRAM traffic is generated by the GPU itself. Taking into account the GPU memory traffic is necessary for accurate system-level studies that involve both the GPU graphics and CPU cores.

## VI. Conclusions

This papers presented GLTraceSim, a new graphics tracing and replay framework for exploring heterogeneous CPU/GPU memory systems (the same techniques can readily be applied to discrete graphics as well by modeling PCI traffic). GLTraceSim efficiently generates GPU memory access traces for modern graphics applications, which can then be replayed in high-level models or in detailed simulators. GLTraceSim is built upon well-maintained publicly available tools, and we plan to release it in the same vein.

These capabilities enabled us to study the memory behavior of graphics applications over hundreds to thousands of frames, and explore how GPU core count and last level cache capacity affect memory bandwidth requirements. We further extended the Gem5 full-system simulator to integrate GLTraceSim memory traces with simulated CPU modules to explore how graphics applications affect co-running CPU applications. We found that GPU-heavy applications can slow down CPU applications by as much as 26 - 59% depending on memory technology used (GDDR vs. LPDDR).

## References

[1] Adrian Courreges, "GTA V - Graphics Study," in *www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/*, 2015.
[2] Apitrace, "www.github.com/apitrace/apitrace/."
[3] 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.
[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," in *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.
[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2009.
[6] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, Jun. 2015.
[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.
[8] J. Bush, P. Dexter, and T. N. Miller, "Nyami: a Synthesizable GPU Architectural Model for General-purpose and Graphics-specific Workloads," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2015.
[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," 2005.
[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
[11] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," in *Proc. Int. Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.
[12] J. Davies, "Bifrost, the new GPU architecture and its initial implementation, Mali-G71," in *Hot Chips: A Symposium on High Performance Chips (HotChips)*, 2016.
[13] R. de Jong and A. Sandberg, "NoMali: Simulating a Realistic Graphics Driver Stack Using a Stub GPU," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2016.
[14] V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "ATTILA: a Cycle-level Execution-driven Simulator for Modern GPU Architectures," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, March 2006.
[15] Dorian Black, "GFXBench 3.0: A Fresh Look At Mobile Benchmarking," in *www.tomshardware.com/reviews/gfxbench-3-graphics-performance,3743-2.html*, 2014.
[16] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot," 2011.
[17] GFXBench, "gfxbench.com."
[18] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006.
[19] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2009.
[20] J.-H. Huang, "GeForce is PC gaming," in *GDC 2017 Keynote*, 2011.
[21] M. Kambadur, S. Hong, J. Cabral, H. Patil, C. K. Luk, S. Sajid, and M. A. Kim, "Fast computational gpu design with gt-pin," in *Proc. Int. Symposium on Workload Characterization (IISWC)*, 2015.

[22] J. Kim, "The future of graphic and mobile memory for new applications," in *Hot Chips: A Symposium on High Performance Chips (HotChips)*, 2016.

[23] S. Lee and W. W. Ro, "Parallel GPU Architecture Simulation Framework Exploiting Work Allocation Unit Parallelism," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2013.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[25] G. Malhotra, S. Goel, and S. R. Sarangi, "GpuTejas: A Parallel Simulator for GPU Architectures," in *Proc. Int. Conference on High Performance Computing (HiPC)*, 2014.

[26] Mesa, "www.mesa3d.org."

[27] E. Nilsson, D. Aarno, E. Carstensen, and H. Grahn, "Accelerating Graphics in the Simics Full-System Simulator," in *Proc. Int. Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

[28] Phoronix Test Suite, "phoronix-test-suite.com."

[29] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, Jan 2015.

[30] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible Software Profiling of GPU Architectures," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2015.

[31] Telemetry, *https://www.chromium.org/developers/telemetry*.

[32] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[33] Vega: AMD's New Graphics Architecture for Virtually Unlimited Workloads, *http://www.amd.com/en-us/press-releases/Pages/vega-amds-new-2017jan05.aspx*.

[34] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU Performance and Power Estimation Using Machine Learning," in *Proc. Int. Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[35] Xvfb, "www.x.org/wiki/."