2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Improving CPU Performance through Dynamic GPU Access Throttling in CPU-GPU Heterogeneous Processors

Siddharth Rai and Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur, INDIA
{*sidrai, mainakc*}@*cse.iitk.ac.in*

***Abstract*—Heterogeneous chip-multiprocessors with integrated CPU and GPU cores on the same die allow sharing of critical memory system resources among the applications executing on the two types of cores. In this paper, we explore memory system management driven by the quality of service (QoS) requirement of the GPU applications executing simultaneously with CPU applications in such heterogeneous platforms. Our proposal dynamically estimates the level of QoS (e.g., frame rate in 3D scene rendering) of the GPU application. Unlike the prior proposals, our algorithm does not require any profile information and does not assume tile-based deferred rendering. If the estimated quality of service meets the minimum acceptable QoS level, our proposal employs a light-weight mechanism to dynamically adjust the GPU memory access rate so that the GPU is able to just meet the required QoS level. This frees up memory system resources which can be shifted to the co-running CPU applications. Detailed simulations done on a heterogeneous chip-multiprocessor with one GPU and four CPU cores running heterogeneous mixes of DirectX, OpenGL, and CPU applications show that our proposal improves the CPU performance by 18% on average.**

***Keywords*-CPU-GPU heterogeneous processors; 3D scene rendering; shared last-level cache; DRAM bandwidth; access throttling;**

I. Introduction

The drivers of the microprocessor evolution, such as, shrinking transistor size, improvement in clock frequency, and increase in high-performance core count are approaching a practical limit [8], [43]. Heterogeneous processing has emerged as one of the new paradigms for extracting performance from the processors in an energy-efficient manner. A CPU-GPU heterogeneous chip-multiprocessor (CMP) implements one such design where a set of latency-optimized CPU cores along with a throughput-optimized GPU having a large number of shader pipelines are integrated on one die. Moreover, to maximize the on-chip resource utilization, the CPU and GPU cores share a substantial part of the memory system. For example, AMD's accelerated processing unit (APU) architecture shares everything beyond the on-chip cache hierarchy (including the coherent request buffers) between the CPU and the GPU cores [4], [17], [41]. On the other hand, Intel's Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Kabylake series of processors share even the on-die last-level cache (LLC) in addition to sharing the on-chip interconnect, memory controllers, and in-package eDRAM cache (available in Haswell onward processors) [6], [9], [15], [16], [30], [35], [42].

A heterogeneous processor can be used in various different computing scenarios. For example, only the CPU cores can be used to carry out traditional general-purpose computing. Only the GPU can be used to execute 3D animation utilizing the entire rendering pipeline or to do general-purpose computing utilizing only the shader cores of the GPU (typically known as GPU computing or GPGPU computing). However, to maximize the computational use of a heterogeneous processor, one needs to simultaneously utilize the CPU cores and the GPU. The focus of this paper is one such scenario where the GPU of a heterogeneous processor is used to execute 3D animation utilizing the entire rendering pipeline and at the same time the CPU cores are used to carry out general-purpose computing. Such a scenario arises in various real-world situations. For example, when the GPU is rendering a 3D animation frame, the CPU cores are typically engaged in preparing the geometry of the next frame requiring AI and physics computation. Also, in a high-performance computing facility, while the CPU cores do the heavy-lifting of scientific simulation of a certain time step, the GPU can be engaged in rendering the output of the last few time steps for visualization purpose [25], [38].

In this paper, we execute 3D animation workloads drawn from DirectX and OpenGL games along with general-purpose CPU computation workloads drawn from the SPEC CPU 2006 suite[1] on a heterogeneous architecture akin to Intel's design, where, the LLC, the on-chip interconnect, the memory controllers, and the DRAM banks are shared between the CPU and the GPU cores. We motivate our study by showing that tightly coupled sharing of memory system resources in such architectures leads to significant degradation in performance of co-running CPU and GPU applications compared to when they are run standalone (Section II). However, we find that there are several GPU

[1] The SPEC CPU workloads are identified to have similarity with the PC games [1].

applications that deliver performance much higher than the minimum quality of service (QoS) level required for visual satisfaction of the end-user utilizing the GPU for running 3D animation. This observation opens up opportunities for improving the CPU performance by dynamically shifting the memory system resources to the CPU workloads from the GPU workloads in the phases where the GPU delivers a level of performance that is higher than necessary. More specifically, our proposal (Section III) involves an accurate algorithm for dynamically estimating the frame rate of the GPU application without relying on any profile information or assumption related to the rendering algorithm implemented by the GPU. Based on the estimated frame rate, our proposal employs a light-weight mechanism to throttle the shared LLC access rate from the GPU so that the GPU is just able to deliver the required level of performance. Throttling the LLC access rate from the GPU frees up LLC capacity as well as DRAM bandwidth which can be dynamically shifted to the co-running CPU workloads. Detailed simulations of a heterogeneous CMP (Section V) having four CPU cores and a GPU show that our proposal is able to improve the performance of the CPU workloads by 18% on average (Section VI). Our two major contributions are summarized in the following.

1) We propose a highly accurate dynamic frame rate estimation algorithm that does not require any profile information and can work with both tile-based deferred rendering and immediate mode rendering pipelines.
2) We propose a simple and low-overhead memory access throttling mechanism for GPUs that can effectively and accurately maintain a given QoS target while freeing up LLC capacity and DRAM bandwidth for the co-running CPU applications.

II. Motivation

In this section, we first motivate the necessity to study memory system resource management techniques in CPU-GPU heterogeneous processors by showing that the co-running CPU and GPU applications in such systems suffer from significant performance degradation compared to when they are run standalone. However, we observe that several GPU applications continue to deliver a level of performance that is higher than necessary. When GPUs are used for rendering 3D scenes, it is sufficient for them to achieve a minimum acceptable frame rate. This relaxation is guided by the fact that due to the persistence of vision, human eyes cannot perceive a frame rate that is above a limit. This observation motivates us to explore mechanisms to dynamically shift memory system resources from the GPU to the CPU cores so that the GPU can just meet the minimum frame rate requirement. Prior studies have explored LLC bypassing as a technique to dynamically shift cache capacity from the GPU to the CPU cores in heterogeneous processors [24]. In this section, we argue that throttling the GPU access rate to the memory system is a more effective technique for shifting memory system resources to the co-running CPU applications.

To understand the implication of heterogeneous execution on the performance of the CPU and the GPU applications when they execute together and contend for the memory system resources, we conduct a set of experiments. In these experiments, the heterogeneous CMP has one CPU core and a GPU clocked at 4 GHz and 1 GHz, respectively. The shared LLC is of 16 MB capacity and there are two on-die single-channel DDR3-2133 memory controllers.[2] In the first of these experiments, we run a CPU job (SPEC CPU 2006 application) on the CPU core and keep the GPU free (standalone CPU workload execution). In the second experiment, we run a 3D animation job (drawn from DirectX and OpenGL games) on the GPU and keep the CPU free (standalone GPU workload execution). Finally, we run both jobs together to simulate a heterogeneous execution scenario. Figure 1 shows the performance of the CPU job and the GPU job in the heterogeneous mode normalized to the standalone mode for fourteen such heterogeneous workload mixes. Each workload mix contains one DirectX or OpenGL application and a SPEC CPU 2006 application. On average, both the CPU and the GPU lose 22% of performance when going from the standalone mode to the heterogeneous mode (see the GMEAN group of bars).[3] This loss in performance results from the contention for LLC capacity and DRAM bandwidth between the two types of applications running simultaneously. Prior studies have also observed large losses in performance due to memory system resource interference between the co-running CPU and GPU applications [3], [11], [18], [23], [24], [28], [31], [36], [37], [40].

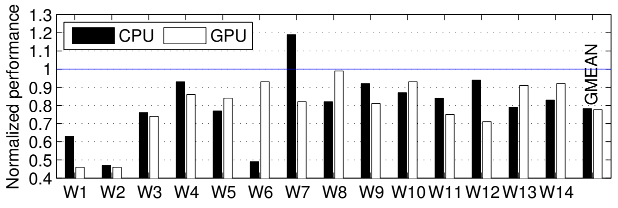


Figure 1. Performance of CPU and GPU in heterogeneous execution normalized to standalone execution. The y-axis shows the ratio of the standalone execution time to the heterogeneous execution time.

Even though a 3D scene rendering workload suffers from a large loss in performance when going from the standalone mode to the heterogeneous mode of execution, this loss may not be noticed by an end-user if the frame

[2] Section V discusses our simulation environment in more detail.

[3] The CPU application in W7 enjoys a 20% improvement in performance in the heterogeneous mode compared to the standalone mode due to an unpredictable improvement in DRAM row-buffer locality.

rate continues to be above the level required for visual satisfaction. Figure 2 shows the frame rates of the individual GPU applications belonging to the fourteen heterogeneous mixes for both standalone and heterogeneous modes of execution. We observe that even in the heterogeneous mode several GPU applications continue to deliver a frame rate that is comfortably above the 30 frames per second (FPS) mark, which is generally considered to be the acceptable frame rate for visual satisfaction. Ideally, such applications should relinquish part of the memory system resources so that they can be utilized by the CPU applications. The challenge in designing such a dynamic memory system resource allocation scheme is two-fold. First, one needs to estimate and accurately project the frame rate of a GPU application. Second, based on this projection, one needs to design a memory system resource shifting algorithm that moves an appropriate amount of memory system resources from the GPU to the CPU cores so that the GPU continues to perform just around the target QoS threshold. These two algorithms form the crux of our proposal. In the rest of the study, we consider 40 FPS to be the target QoS threshold for 3D scene rendering leaving a 10 FPS cushion for handling any momentary dip.

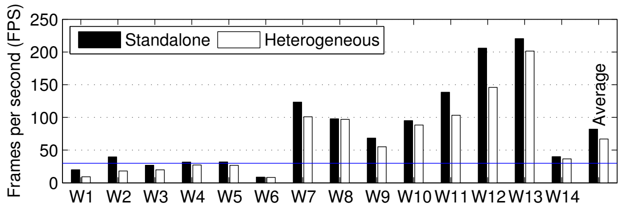


Figure 2. Comparison of GPU frame rate in standalone and heterogeneous execution. The reference line shows 30 FPS mark.

Two primary memory system resources that are shared between the GPU and the CPU cores are LLC capacity and DRAM bandwidth. Therefore, for best outcome, an algorithm that dynamically shifts memory system resources from the GPU to the CPU applications based on estimated performance levels would try to focus on both LLC capacity and DRAM bandwidth. Prior studies have explored bypassing the LLC for GPU read misses thereby targeting only the LLC capacity [24]. An ideal LLC bypass algorithm for GPU applications would only free up portion of the LLC for CPU applications while leaving the DRAM bandwidth consumption of the GPU unchanged. However, since designing an ideal bypass algorithm is difficult, it is expected that the DRAM bandwidth consumption of the GPU will increase when LLC bypassing for GPU read misses is enabled. Since the GPU is designed to have high latency-tolerance, these additional LLC misses may not hurt the GPU performance. However, the extra DRAM bandwidth consumed by these additional GPU LLC misses can lead to significant drop in CPU performance.

Figure 3 shows the impact on CPU workload performance when all GPU read misses are forced to bypass the LLC. On average, compared to the heterogeneous mode of execution without LLC bypass for GPU read misses, the CPU applications lose 2% performance. While there are CPU applications that gain as much as 10% (W4), there are also applications that lose as much as 14% (W9). The CPU applications which fail to utilize the additional LLC capacity created through GPU read miss bypass start suffering due to increased DRAM bandwidth contention. The GPU applications enjoy a significant volume of reuses from the LLC in the baseline. When all GPU fills bypass the LLC, the GPU applications lose these reuses and significantly increase the DRAM traffic. This increased DRAM bandwidth pressure hurts the performance of both CPU and GPU. We revisit this aspect in Section VI when we evaluate HeLM, the state-of-the-art LLC management policy that relies on selective LLC bypass of GPU fills [24]. In summary, any algorithm that dynamically shifts memory system resources from the GPU to the CPU applications must be able to create LLC capacity *as well as* DRAM bandwidth for the CPU applications. In our proposal, we use the important insight that throttling the GPU access rate to the LLC can achieve both the goals. A slowed down GPU access rate automatically ages the GPU blocks faster in the LLC leading to their eviction from the LLC and creating more LLC capacity for the CPU applications. Also, a slowed down GPU access rate to the LLC naturally lowers the GPU's demand on the DRAM bandwidth; even though the volume of GPU LLC misses increases, these misses are sent to the DRAM at a dynamically controlled rate that is just enough for the GPU application to meet the target QoS threshold.

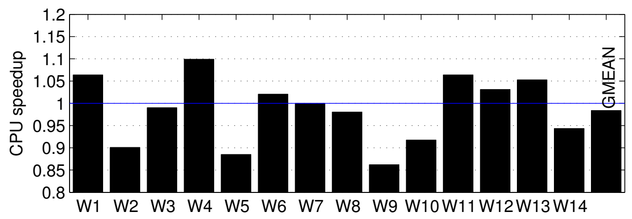


Figure 3. CPU speedup when all GPU read miss fills are forced to bypass the LLC.

III. Memory Access Management

In this section, we present our proposal on memory access management for CPU-GPU heterogeneous processors. Our proposal involves a three-step algorithm. In the first step, we dynamically estimate the frame rate of the GPU application. In the second step, depending on the estimated frame rate, we determine an appropriate rate at which GPU accesses are sent to the LLC. In the third step, depending on the estimated frame rate, the CPU access priority is altered in the DRAM access scheduler. If the estimated frame rate falls below the target QoS threshold, the second and the third steps are not invoked and the GPU application continues to run in the baseline heterogeneous mode without any change in the LLC access rate. The predictive model for online frame rate estimation used in the first step is discussed in Section III-A. The access throttling mechanism is discussed in Section III-B. The changes in the DRAM access scheduler are discussed in Section III-C. Section III-D quantifies the storage overhead of our proposal.

A. Dynamic Frame Rate Computation

For the 3D scene rendering workloads, our proposal requires knowledge of the frame rate ahead of the actual completion of the frame so that the GPU access rate and the DRAM access scheduler can be adapted accordingly. Such dynamic prediction of the frame rate requires answering the following two questions.

1. How much is the amount of work per frame?

2. Given the rendering speed and the amount of work per frame, how to predict the frame rate?

To answer these questions, we divide the entire rendering process into two phases, namely, learning phase and prediction phase. In the learning phase, we monitor rendering of a frame and measure the amount of work done and the time it takes to complete. Once this information is obtained for one complete frame, we switch to the prediction phase. In this phase, the data collected in the learning phase is used to predict the frame rate. To ensure that the observed data follows the collected data, new observations are cross-verified against the learned data. If it is found that the observed values differ from the learned values by more than a threshold amount, the learned data is discarded and we switch back to the learning phase. Figure 4 demonstrates a sequence of phase transitions that happen in a hypothetical rendering job. Rendering begins in the learning phase and continues to remain in that phase till point A, where it is determined that the data for one complete frame has been collected. Thus, at this point we transition into the prediction phase. Now, rendering continues in this phase up to point B, where it is found that the learned data is no more valid and thus, we transition back to the learning phase to collect fresh data. At point C, we again transition to the prediction phase. This cycle is repeated until the entire rendering job completes. Since we would like to maximize the number of frames in the prediction phase for having good prediction coverage, the success of this scheme improves if the amount of work in each frame within a set of consecutive frames remains more or less constant.

1) Learning Phase: Rendering of a frame in a rudimentary sense boils down to computing the color values of all pixels from the input geometry and updating these values into a buffer commonly known as the render target (RT). In general, a single pixel in the RT can get overdrawn multiple

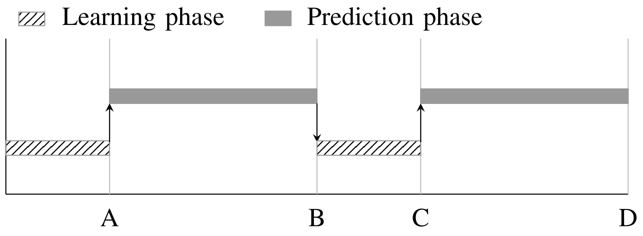


Figure 4. Rendering phases

times depending on the order in which the geometry primitives arrive for rendering and their depth. This complicates the estimation of the amount of work involved in rendering a frame. We divide the RT into equal sized $t \times t$ render target tiles (RTT). We divide the entire rendering of a frame into render target planes (RTP). Each RTP represents a batch of updates that cover all tiles of the RT. Therefore, the number of RTPs is the number of updates that cover all tiles of the RT. This arrangement is shown in Figure 5.

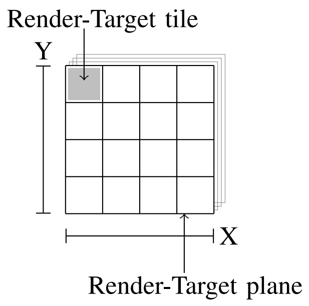


Figure 5. Render-target plane and tile

We maintain a 64-entry RTP information table in the GPU. For a frame, each entry of this table maintains a valid bit and four pieces of information about a distinct RTP. These four pieces of information are: (i) total number of updates to the RTP, (ii) the number of cycles to finish the RTP, (iii) the number of RTTs in the RTP, and (iv) the number of shared LLC accesses (i.e., GPU render cache misses) made by the GPU for the entire RTP. Our implementation assumes each of the four fields to be four bytes in size. The first three fields are used in the prediction phase. The LLC access count is passed on to the access throttling algorithm (see Section III-B) for computing the maximum throttling rate. If the number of RTPs in a frame exceeds 64, the last entry of the table is used to accumulate the data for all subsequent RTPs.

2) Prediction Phase: Our frame rate prediction model uses the RTP count and cycles per RTP recorded during the learning phase to predict the current number of cycles per frame. If the number of RTPs in a frame i is N^i_{rtp} and the average number of cycles per RTP is C^i_{rtp}, then the number of estimated cycles F_i required to render frame i is given by Equation 1.

$$F_i = C^i_{rtp} \times N^i_{rtp} \tag{1}$$

Although N^i_{rtp} is obtained directly from the data collected during the learning phase, C^i_{rtp} for the frame being rendered currently has to be extrapolated using the number of cycles the frame has taken so far and the cycles recorded in the RTP table, so that the current rendering speed of the frame can be taken into account for obtaining the full frame cycle count. Suppose the fraction of a frame that has been rendered so far is λ, the average number of cycles per RTP seen in the current frame is C^i_{inter}, and the average number of cycles per RTP recorded during the learning phase is C^i_{avg}. Therefore, C^i_{rtp} can be computed using Equation 2.

$$C^i_{rtp} = \lambda \times C^i_{inter} + (1 - \lambda) \times C^i_{avg} \tag{2}$$

If we substitute Equation 2 into Equation 1, we obtain the final expression for the predicted number of cycles per frame as shown in Equation 3.

$$F_i = (\lambda \times C^i_{inter} + (1 - \lambda) \times C^i_{avg}) \times N^i_{rtp} \tag{3}$$

We note that λ is computed as the ratio of the number of RTPs completed so far in the frame being currently rendered to the total number of RTPs observed during the learning phase.

B. Access Throttling Mechanism

In this section, we discuss our GPU access throttling mechanism. This mechanism is invoked only if the GPU is found to be meeting a target frame rate. For the 3D scene rendering workloads that are predicted to meet a target frame rate, the rate with which the GPU can send access requests to the shared LLC is throttled down. Throttling GPU accesses before the LLC has two implications. First, the GPU blocks in the LLC are accessed less frequently causing them to age faster compared to the CPU blocks. This replaces the GPU blocks early increasing the number of GPU misses and creating more space for the CPU blocks in the LLC. Second, the GPU accesses that miss in the LLC are seen by the DRAM at a slower rate automatically shifting a bigger proportion of the DRAM bandwidth to the CPU workloads. Our access throttling proposal allows N_G GPU accesses within a window of W_G GPU cycles, thereby enforcing an average GPU access rate of N_G/W_G. This is implemented as follows.

Every GPU access must go through a translation from the GPU address to the global physical address before the access can be routed to the correct LLC bank. This translation is accomplished by looking up a GPU translation table (GTT) resident in the GPU. At the beginning of a window, N_G and W_G are initialized. W_G is decremented on every GPU cycle and N_G is decremented on every GPU access to the GTT. As soon as N_G reaches zero, the GTT ports are disabled until W_G reaches zero. As a result, during this period the GPU is denied access to the LLC. When the GPU requests are denied access to the LLC, they are held back inside the GPU and occupy GPU resources such as request buffers and MSHRs attached to the caches internal to the GPU. This resource contention is modeled in detail in our evaluation. Any negative influence that this resource contention may have on performance automatically gets reflected in the progress of rendering and is captured by our frame rate estimation algorithm. The estimation, in turn, feeds back into the throttling mechanism.

Choosing Values for W_G and N_G. We need an algorithm that automatically adjusts N_G and W_G based on the estimated and the target frame rates. Let the number of cycles per frame at the current predicted frame rate be C_P, the number of cycles per frame at the target frame rate be C_T, and the number of LLC accesses per frame be A (this is recorded during the learning phase of the frame rate prediction model as discussed in Section III-A1). Therefore, if $C_P < C_T$ meaning that the GPU is delivering better frame rate than the target, we would like to increase W_G by $(C_T - C_P)/A$ while holding N_G at one. This increment is done gradually at a step of two in each window. On the other hand, if $C_P \geq C_T$, access throttling is disabled by setting W_G to zero and N_G to one. Small oscillation around the target frame rate can be avoided by disabling throttling within a small guard-band around the target frame rate. Figure 6 shows the flow of the throttling mechanism.

C. DRAM Access Scheduler

The goal of our DRAM scheduling policy is to shift bandwidth to the CPU if the GPU is able to meet target QoS. This is implemented using a simple scheme. If the GPU is currently predicted to meet the target frame rate, the CPU requests are prioritized over the GPU requests. Within a bank, among the requests that can enjoy row buffer hits, the scheduling policy first schedules the CPU accesses in FCFS order and then considers the rest. When a new row needs to be activated in a bank, the oldest CPU access is given priority over the global oldest access. The scheduler follows the baseline FR-FCFS algorithm if the GPU fails to meet the target frame rate. Figure 7 summarizes our entire proposal.

D. Storage Overhead

The storage overhead of our proposal is small. It involves the RTPi table having 64 entries, each entry being 129 bits leading to a total investment of just over 1 KB. Our proposal also requires two short registers to maintain W_G and N_G. One state bit is required to indicate whether the DRAM access scheduler should invoke the baseline policy or the

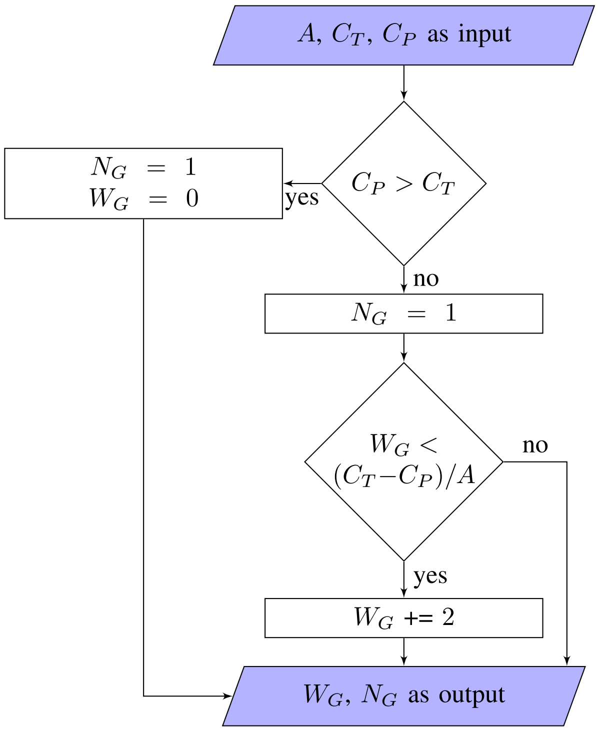
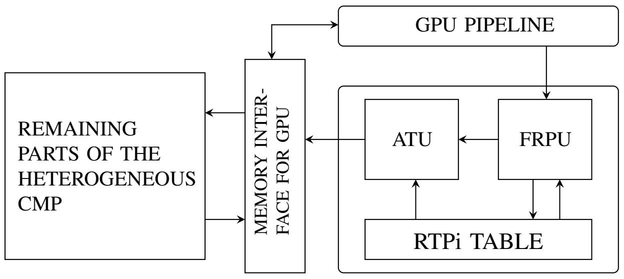


Figure 6. Flow of the algorithm that throttles LLC accesses from the GPU. A, C_T, and C_P are input parameters. W_G and N_G are outputs.



GPU PIPELINE
REMAINING PARTS OF THE HETEROGENEOUS CMP
MEMORY INTER-FACE FOR GPU
ATU
FRPU
RTPi TABLE

Figure 7. Architecture of the frame rate prediction and access throttling mechanism. FRPU is the frame rate prediction unit. ATU is the access throttling unit.

policy with enhanced CPU priority. The primary storage overhead arises from the RTPi table and it is important to note that the accesses to this table are not on the critical path of the GPU accesses. Updates to this table happen off the critical path and this table is read only periodically at a certain interval for computing the projected frame rate.

IV. Related Work

Dynamic frame rate estimation has been studied in the context of mobile SoCs with GPUs implementing tile-based deferred rendering (TBDR) [11], commonly found in mobile GPUs [29], [32]. Dynamic progress estimation of GPUs has also been explored in the presence of prior profile information such as the number of memory accesses issued by the GPU application [40]. In contrast, our proposal does not make any assumption about the implementation of the rendering pipeline and nor does it require a profile pass.

Request throttling to reduce unfairness in the memory system of CPU-based CMPs has been explored [7]. The mechanism and goal of our proposed GPU access throttling are, however, entirely different. Also, in the context of GPGPU applications, there have been studies to throttle up/down the number of active warps and active thread blocks in the shader cores based on memory system congestion and GPU idle cycles [18], [19]. These are primarily shader core-centric proposals and are not effective for the 3D scene rendering workloads that generate a large volume of memory accesses from the fixed function units such as the texture samplers, color blenders, depth test units, etc..

To gain better understanding of this class of shader core-centric throttling mechanisms, we explore the performance of the balanced concurrency management (CM-BAL) proposal [18] in detail for our heterogeneous mixes. CM-BAL scales up or down the maximum number of ready shader threads based on the average stalls observed with different thread configurations. We observe that CM-BAL fails to adequately throttle the GPU frame rate for three reasons. First, throttling the shader threads has a first order effect on the texture access rate to the LLC because the texture samplers are directly attached to the shader cores and the texture accesses are triggered by texture filtering instructions of the shader program. However, in the 3D scene rendering workloads considered by us, the texture accesses, on average, constitute only 25% of all LLC accesses coming from the GPU. As a result, throttling only the texture access rate is not enough. The render output pipeline (ROP) consisting of the color blenders, depth test units, and color writers receives shaded and textured fragments from the shader cores and is responsible for writing out the final pixel colors to the render buffer. It is necessary to drive the usually over-utilized ROP to an under-utilized region of operation to be able to see an effect of shader core throttling on the LLC access rate from the ROP. In practice, this is impossible to achieve through shader core throttling alone. While it is feasible to throttle individual units of the 3D rendering pipeline at appropriate rates, this leads to a design that is far more complex than what we propose. Our proposal does not focus on any particular unit in the rendering pipeline; it throttles the collective rate at which the GPU can access the LLC. Second, different applications show different performance sensitivity toward texture access rate, which experiences the first order impact of shader core throttling. So, throttling the texture access rate is not guaranteed to have a significant performance influence on the GPU. Third, at run-time when the CM-BAL policy is applied, only a fraction of the texture accesses undergo throttling further diminishing the overall performance impact. We do not consider these shader core-centric GPU throttling proposals in the rest of this study.

DRAM access scheduling has been explored for CPU-based platforms, discrete GPU parts, and heterogeneous CMPs. In the following, we discuss the contributions relevant to the discrete GPU parts and heterogeneous CMPs. The memory access scheduling studies for the discrete GPU parts have been done with the GPGPU applications. These studies have explored memory access scheduling to minimize the latency variance among the threads within a warp [5], to accelerate the critical shader cores that do not have enough short-latency warps which could hide long memory latencies [14], and to minimize a potential function so that an appropriate mix of shortest-job-first and FR-FCFS can be selected with the overall goal of accelerating the less latency-tolerant shader cores [20]. There have been studies on warp and thread block schedulers for improving the memory system performance [2], [12], [13], [19], [21], [22].

Motivated by the bandwidth-sensitive nature of the massively threaded GPU workloads and the deadline-bound nature of the 3D scene rendering workloads executed on the GPUs, prior proposals have explored specialized memory access schedulers for heterogeneous systems [3], [11], [28], [37]. The staged memory scheduler (SMS) clubs the memory requests from each source (CPU or GPU) into source-specific batches based on DRAM row locality [3]. Each batch is next scheduled with a probabilistic mix of shortest-batch-first (favoring latency-sensitive CPU jobs) and round-robin (enforcing fairness among bandwidth-sensitive CPU and GPU jobs). The dynamic priority (DynPrio) scheduler [11], proposed for mobile heterogeneous platforms, employs dynamic progress estimation of tile-based deferred rendering (TBDR) and offers the GPU accesses equal priority as the CPU accesses if the GPU progress lags behind the target frame rendering time. Also, during the last 10% of the left time to render a frame, the GPU accesses are given higher priority than the CPU accesses. The progress estimation algorithm used by DynPrio is designed specifically for the GPUs employing TBDR, typically supported only in mobile GPUs such as ARM Mali [29], Kyro, Kyro II, and PowerVR from Imagination Technologies, and Imageon 2380, Xenos, Z430, and Z460 from AMD [32]. The DynPrio scheduler study, however, shows the inefficiency of a previously proposed static priority scheduler that always offers higher priority to the CPU accesses [37]. The option of statically partitioning the physical address space between the CPU and GPU datasets and assigning two independent memory controllers to handle accesses to the two datasets has been explored [28]. A subsequent study has shown that such static partitioning of memory resources can lead to sub-optimal performance in heterogeneous systems [18]. In Section VI, we present a quantitative comparison of our proposal with DynPrio and two variants of SMS.

There have been studies on managing the shared LLC in a CPU-GPU heterogeneous processor. Two of these studies (thread-aware policy [23] and dynamic reuse probability-aware policy [31]) propose new insertion, promotion, and replacement policies for LLC blocks. Another study (HeLM) proposes to selectively bypass the LLC for GPU read misses coming from the shader cores that are dynamically identified to be latency-tolerant, thereby opportunistically shifting LLC capacity to the co-executing CPU workloads [24]. In Section II, we have already pointed out the shortcomings of such a mechanism that is based purely on LLC bypass techniques. Nonetheless, since our proposal is closer to HeLM in its goal, we present a quantitative comparison of our proposal with HeLM in Section VI.

V. Simulation Environment

In the following, we discuss the simulated heterogeneous CMP model and the workloads used in this study.

A. Heterogeneous CMP Model

The details of the simulated environment are presented in Table I. We use a modified version of the Multi2Sim simulator [39] to model the CPU cores of the simulated heterogeneous CMP. Each dynamically scheduled out-of-order issue x86 core is clocked at 4 GHz. The GPU is modeled with an upgraded version of the Attila GPU simulator [26]. The simulator has enough details to capture all the phases of the entire rendering pipeline. The simulated GPU uses a unified shader model where the same set of shader cores is used to carry out vertex shading as well as pixel (or fragment) shading. The overall single-precision floating-point operation throughput of the GPU is one tera-FLOPS. The shared LLC of the heterogeneous CMP receives requests that miss in the CPU cores' L2 caches or GPU's vertex cache, hierarchical depth cache, shader caches, L2 texture cache, L2 depth cache, or L2 color cache. The DRAM modules are modeled using DRAMSim2 [33]. The CPU cores along with their private caches, the GPU, the LLC, and the memory controllers are arranged on a bidirectional ring interconnect having a single-cycle hop time. We will evaluate our proposal for a configuration with four CPU cores and one GPU.

B. Workload Selection

We target computational scenarios where the GPU in a heterogeneous processor renders 3D animation while the CPU is busy with general-purpose computing. Such computational scenarios are encountered in high-performance computing facilities where the CPU is busy with simulating the current time-step of some scientific phenomenon, while the GPU renders the output of the last few time-steps for visualization purpose [25], [38]. Also, in general, when the

Table I
SIMULATION ENVIRONMENT

CPU cache hierarchy
Per-core iL1 and dL1 caches: 32 KB, 8-way, 64B blocks, 2 cycles, LRU replacement policy
Per-core unified L2 cache: 256 KB, 8-way, 64B blocks, 3 cycles, LRU replacement policy
GPU model
Shader cores: 64, 1 GHz, four ALUs per core, each ALU has a 4-way SIMD unit and a scalar unit
Shader threads (same as warps/wavefronts): 4096 in-flight contexts, each thread context is scheduled on a shader core, each instruction of a context is a 16-way vector operation
Thread context scheduler: round-robin among ready contexts, an executing context is blocked on issuing a branch or texture load
Texture samplers: two per shader core, total fill rate 128 GTexel/s
Render output pipelines (ROPs): 16, total fill rate 64 GPixel/s, each ROP has a depth test unit, a color blender, and a color writer
Texture caches: three-level non-inclusive hierarchy, L0: 2 KB, fully-associative, 64B blocks, private per texture sampler, L1: 64 KB, 16-way set-associative, 64B blocks, shared by samplers, L2: 384 KB, 48-way set-associative, 64B blocks, shared by samplers
Depth caches: two-level non-inclusive hierarchy, L1: 2 KB, fully-associative, 256B blocks, private per ROP, L2: 32 KB, 32-way set-associative, 64B blocks, shared by all ROPs
Color caches: two-level non-inclusive hierarchy, L1: 2 KB, fully-associative, 256B blocks, private per ROP, L2: 32 KB, 32-way set-associative, 64B blocks, shared by all ROPs
Vertex cache: 16 KB fully-associative
Hierarchical depth cache: 16 KB, 16-way
Shader instruction cache: 32 KB, 8-way
Shared LLC and interconnect
Shared LLC: 16 MB, 16-way, 64B blocks, lookup latency 10 cycles, inclusive for CPU blocks, non-inclusive for GPU blocks, two-bit SRRIP insertion/replacement policy [10], a replacement sends back-invalidation to CPUs, but not to GPU
Interconnect: bi-directional ring, single-cycle hop
Memory controllers and DRAM
Memory controllers: two on-die single-channel, DDR3-2133, FR-FCFS access scheduling in baseline
DRAM modules: 14-14-14, 64-bit channels, BL=8, open-page policy, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices

GPU renders the current frame of an animation sequence, some of the CPU cores are busy computing the physics and AI of the next frame leading to the input geometry to the GPU; completely unrelated jobs can get scheduled on the rest of the cores of the CPU. The heterogeneous workloads used in this study are built by mixing CPU applications drawn from the SPEC CPU 2006 suite and 3D scene rendering jobs drawn from fourteen popular DirectX 9 and OpenGL game titles. The DirectX and OpenGL API traces for the selected 3D animation frames are obtained from the Attila simulator distribution and the 3DMark06 suite [44]. The simulated game regions (i.e., sequences of multiple consecutive frames) are selected at random after skipping over the initial sequence and detailed in Table II. The "Frames" column of the table shows the sequence of selected frames for each application. The sequence length ranges from two to eighteen frames. The "Res" column shows the frame resolution for each application. The last column of this table lists the baseline average frames per second (FPS) achieved by each GPU application for a configuration having four CPU cores and one GPU running heterogeneous mixes consisting of four CPU applications and the GPU application.

Table II
GRAPHICS FRAME DETAILS

Application, DX/OGL[4]	Frames	Res.[5]	FPS
3DMark06 GT1, DX	670–671	R1	6.0
3DMark06 GT2, DX	500–501	R1	13.8
3DMark06 HDR1, DX	600–601	R1	16.0
3DMark06 HDR2, DX	550–551	R1	20.8
Call of Duty 2 (COD2), DX	208–209	R2	18.1
Crysis, DX	400–401	R2	6.6
DOOM3, OGL	300–314	R3	81.0
Half Life 2 (HL2), DX	25–33	R3	75.9
Left for Dead (L4D), DX	601–605	R1	32.5
Need for Speed (NFS), DX	10–17	R1	62.3
Quake4, OGL	300–309	R3	80.8
Chronicles of Riddick (COR), OGL	253–267	R1	111.0
Unreal Tournament 2004 (UT2004), OGL	200–217	R3	130.7
Unreal Tournament 3 (UT3), DX	955–956	R1	26.8

[4] DX=DirectX, OGL=OpenGL
[5] R1=1280×1024, R2=1920×1200, R3=1600×1200

Each heterogeneous mix used for evaluating our proposal on a configuration with four CPU cores and one GPU consists of four SPEC CPU 2006 applications and one GPU application. There are a total of fourteen mixes, one corresponding to each GPU application. These mixes, denoted by M1-M14, are shown in Table III. These mixes are prepared by drawing fourteen distinct sets of four SPEC CPU 2006 applications at random and combining each of these sets with a distinct GPU application. Each CPU application is identified by its SPEC id. For completeness, we also list the workloads W1-W14 each having one CPU application and one GPU application. These mixes were used in Section II for a configuration with one CPU core and one GPU.

Table III
HETEROGENEOUS WORKLOAD MIXES

GPU application	CPU workload mix
3DMark06GT1	M1: 403,450,481,482; W1: 481
3DMark06GT2	M2: 403,429,434,462; W2: 471
3DMark06HDR1	M3: 401,437,450,470; W3: 470
3DMark06HDR2	M4: 401,462,470,471; W4: 482
COD2	M5: 401,437,450,470; W5: 470
Crysis	M6: 429,433,434,482; W6: 429
DOOM3	M7: 410,433,462,471; W7: 462
HL2	M8: 410,429,433,434; W8: 403
L4D	M9: 410,433,462,471; W9: 462
NFS	M10: 410,429,433,471; W10: 437
Quake4	M11: 401,437,450,481; W11: 410
COR	M12: 403,437,450,481; W12: 434
UT2004	M13: 401,437,462,470; W13: 450
UT3	M14: 403,437,450,481; W14: 434

Within each mix, the first 200M instructions retired by each CPU core are used to warm up the caches. After the warm-up, each CPU application in a mix commits at least 450M representative dynamic instructions [34]. Early-finishing applications continue to run until each CPU application commits its representative set of dynamic instructions and the GPU completes rendering the set of 3D frames assigned to it. The performance of the CPU mixes is measured in terms of weighted speedup. The GPU performance is measured in terms of average frame rate.

VI. Simulation Results

In this section, we evaluate our proposal on a simulated heterogeneous CMP with four CPU cores and one GPU. With each GPU workload, we co-execute a mix of four CPU applications. We divide the discussion into evaluation of the individual components that constitute our proposal.

Accuracy of Dynamic Frame Rate Estimation. Figure 8 shows the percent error observed in our dynamic frame rate estimation technique. A positive error means over-estimation and a negative error means under-estimation. Several applications have zero error. Among the applications that have non-zero error, the maximum over-estimation error is 6% (UT2004) and the maximum under-estimation error is 4% (COR). The average error across all applications is less than 1%.

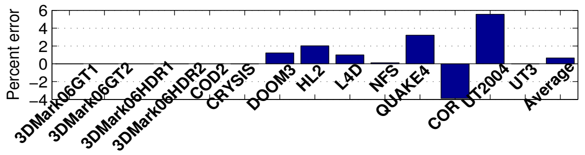


Figure 8. Percent error in dynamic frame rate estimation.

Evaluation of Access Throttling. Figure 9 quantifies the performance of our GPU access throttling mechanism. In this evaluation, we set the target frame rate as 40 FPS for the GPU applications. Referring to Table II, we see that there are six applications that have frame rates higher than this target. The rest of the applications never go above 40 FPS for the selected frame sequence. Therefore, our proposal will be able to apply access throttling to these six applications. The left panel of Figure 9 quantifies average FPS for the GPU applications. For each application, we show three bars. The leftmost bar corresponds to baseline. The middle bar corresponds to a system with access throttling enabled. The rightmost bar corresponds to a system with access throttling enabled and the CPU applications given higher priority over the GPU in the DRAM access scheduler. The right panel of this figure shows the weighted speedup (normalized to the baseline, which is at 1.0) achieved by the four-way multi-programmed CPU workloads when the corresponding GPU workload in the mix is undergoing access throttling. We identify each CPU application in a mix by its SPEC id. The GPU application results confirm that the six applications operate just around 40 FPS when access throttling is enabled. While this represents the average frame rate over the multi-frame sequence for each application, we also verified that each frame within the sequence meets the target frame rate. For the CPU applications, the mixes improve significantly offering an average 11% speedup with GPU access throttling alone; the average speedup improves to 18% when higher CPU priority is enabled in the DRAM access scheduler.

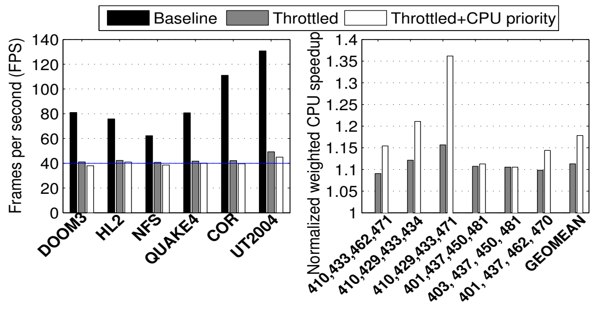


Figure 9. Left: FPS of GPU applications that are amenable to access throttling. Right: weighted CPU speedup for the mixes when the GPU application in the mix is throttled. The CPU application mixes are shown in terms of the combination of the SPEC application ids along the x-axis of the right panel.

To further understand the sources of CPU performance improvement, Figure 10 quantifies the LLC miss count of the GPU applications (left panel) and the corresponding CPU workload mixes (right panel) normalized to the baseline. The GPU applications suffer from an average 39% increase in LLC miss count when GPU access throttling is enabled. This number further increases to 42% when CPU priority in the DRAM access scheduler is boosted in addition to GPU access throttling. As already explained, this is an expected behavior resulting from faster aging of the GPU blocks in the LLC due to lowered LLC access rate of the GPU application. In addition to GPU access throttling, when the CPU priority in the DRAM access scheduler is boosted, the CPU fills return faster to the LLC, thereby evicting the aged GPU blocks even more quickly. The right panel shows that the additional LLC space created due to this leads to a 4% and 4.5% average reduction in CPU LLC miss counts for the throttled and throttled+CPUpriority configurations, respectively.

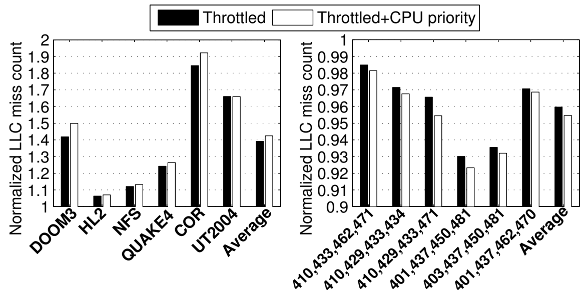


Figure 10. Left: normalized LLC miss count of GPU applications that are amenable to access throttling. Right: normalized LLC miss count of CPU workloads when the GPU application in the mix is throttled.

The significant increase in the LLC miss count of the GPU applications can be of concern because this may lead to higher DRAM bandwidth consumption defeating the very purpose of GPU access throttling. However, it is important to note that these misses occur over a much longer period of time due to a lowered frame rate. Our access throttling algorithm automatically adjusts the throttling rate taking all these into consideration so that the frame rate hovers close to the target level. Figure 11 substantiates this fact by quantifying the average DRAM bandwidth (read and write separately shown) consumed by the GPU applications normalized to the baseline.[6] On average, the GPU bandwidth demand reduces by 35% and 37% for the throttled and throttled+CPUpriority configurations, respectively. Both read and write bandwidth demands go down by significant amounts, across the board. In summary, our proposal frees up more than one-third GPU bandwidth for the CPU workloads to consume. A comparison with the right panel of Figure 10 reveals that DRAM bandwidth shifting is a bigger benefit of our proposal than LLC miss saving for CPU workloads.

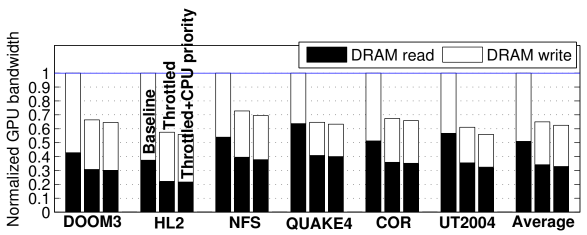


Figure 11. Normalized DRAM bandwidth (read and write) consumed by GPU applications that are amenable to access throttling.

Comparison to Related Proposals. Several DRAM scheduling policies have been proposed for heterogeneous CMPs and evaluated on mixes of 3D scene rendering workloads and CPU workloads. In the following, we compare our proposal against staged memory scheduling (SMS) [3] and dynamic priority scheduler (DynPrio) [11]. Additionally, we present a comparison with HeLM, the state-of-the-art LLC management policy for CPU-GPU heterogeneous processors [24]. These proposals were briefly introduced in Section IV. We evaluate two versions of SMS, namely, one with a probability of 0.9 of using shortest-job-first (SMS-0.9) and the other with this probability zero i.e., it always selects a round-robin policy (SMS-0). SMS-0.9 is expected to favor latency-sensitive CPU jobs while SMS-0 is expected to favor GPU jobs. DynPrio makes use of our frame rate estimation technique to compute the time left in a frame. Figure 12 compares the proposals for the heterogeneous mixes containing GPU applications that meet the target 40 FPS. The upper panel shows that all proposals deliver higher than 40 FPS. Our proposal (ThrotCPUprio) opportunistically applies GPU access throttling and CPU prioritization in the DRAM scheduler to deliver an FPS that is just around the target. As a result, our proposal is able to improve the CPU mixes most (lower panel of Figure 12). On average, SMS-0.9, SMS-0, DynPrio, HeLM, and our proposal improve the performance of the CPU mixes by 4%, 4%, 10%, 3%, and 18%, respectively. The performance improvement achieved by HeLM is low because it suffers from an increased DRAM bandwidth consumption arising from the additional GPU misses that result from aggressive LLC bypass of GPU fills.

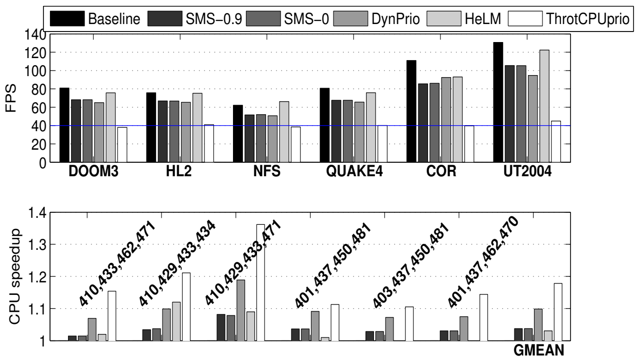


Figure 12. FPS of GPU applications (top panel) and normalized weighted CPU speedup (bottom panel) for the mixes with high frame rate GPU applications.

Our proposal remains disabled in the remaining mixes containing the GPU applications that fail to meet the target FPS. For completeness, Figure 13 shows the comparison for these mixes. The upper panel evaluates the proposals for the GPU applications. SMS suffers from large losses in FPS due to the delay in batch formation. DynPrio fails to observe any overall benefit because it offers express bandwidth to the GPU application only during the last 10% of a frame time and otherwise the CPU and GPU are offered equal priority as in the baseline FR-FCFS. HeLM suffers from an average loss of 7% in FPS due to DRAM bandwidth shortage resulting from the additional GPU misses that arise due to aggressive LLC bypass. SMS-0.9 and SMS-0 improve CPU mix performance (lower panel of Figure 13) by 7% and 6%, respectively, while suffering large losses in GPU performance. DynPrio delivers the same level of performance as the baseline for both GPU and CPU workloads. HeLM enjoys a 4% average improvement in CPU mix performance. In summary, these results clearly indicate that the GPU performance can be traded off to improve CPU performance and vice-versa in such heterogeneous platforms. To understand the overall performance in such scenarios, we assign equal weightage to the CPU and GPU performance and derive the overall performance of the heterogeneous processor. Figure 14 shows these results for the mixes containing the GPU applications that fail to meet

[6] In a few GPU applications (e.g., DOOM3 and HL2), the volume of writes can be more than the volume of reads because the rendering pipeline can create fully dirty color or depth lines in the internal ROP caches and later flush them out to the LLC for allocation without doing a DRAM read.

the target FPS.[7] On average, our proposal and DynPrio both deliver baseline performance for these mixes, while both variants of SMS suffer from large losses. HeLM performs 1% worse than the baseline on average.

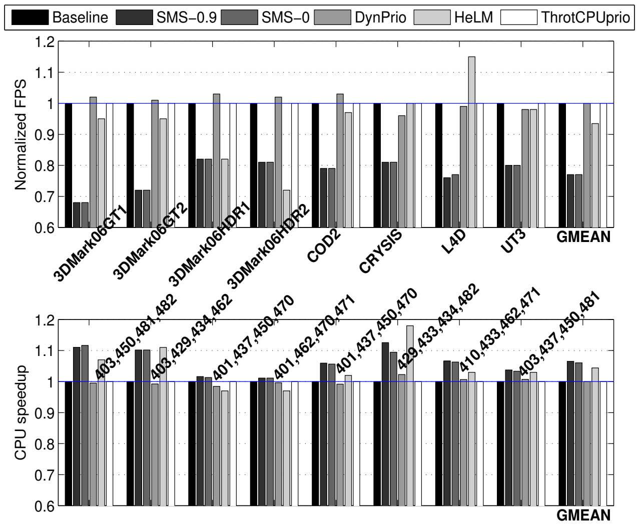


Figure 13. FPS speedup (top panel) and weighted CPU speedup (bottom panel) over the baseline for the mixes with low frame rate GPU applications.

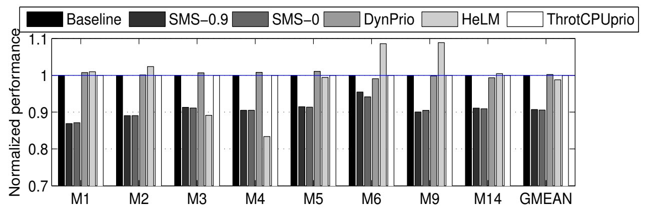


Figure 14. Normalized performance of the heterogeneous processor for the mixes with low frame rate GPU applications when the CPU and GPU are given equal weight in terms of performance.

VII. Summary

We have presented a novel memory access management mechanism for heterogeneous CMPs. The proposed mechanism dynamically shifts LLC capacity and DRAM bandwidth to CPU applications from the co-executing GPU application whenever the GPU application meets the target frame rate. Two light-weight, yet highly effective, algorithms form the crux of our proposal. The first algorithm accurately estimates the projected frame rate of a GPU application. Based on this estimation, the second algorithm computes the effective GPU access rate to the LLC and assists the DRAM access scheduler to decide if CPU priority should be boosted. Detailed simulation studies show that our proposal achieves its goal of offering a bigger share of the memory system resources to the CPU when the GPU does not need it. For the heterogeneous mixes containing GPU applications that meet the target frame rate, our proposal improves the CPU performance by 18% on average while requiring just over a kilobyte of additional storage.

[7] For the mixes where the GPU application already meets the target FPS, this kind of a combined CPU-GPU performance metric is irrelevant because the GPU performance goal is already satisfied and an evaluation of the CPU performance improvement, as shown in Figure 12, is sufficient.

References

[1] K. Asanovic et al. A View of the Parallel Computing Landscape. In *Communications of the ACM*, **52**(10): 56–67, October 2009.

[2] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, October 2015.

[3] R. Ausavarungnirun, K. K-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 416–427, June 2012.

[4] D. Bouvier, B. Cohen, W. Fry, S. Godey, and M. Mantor. Kabini: An AMD Accelerated Processing Unit System on a Chip. In *IEEE Micro*, **34**(2):22–33, March/April 2014.

[5] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 128–139, November 2014.

[6] M. Demler. Iris Pro Takes On Discrete GPUs. In *Microprocessor Report*, September 9, 2013.

[7] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–346, March 2010.

[8] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 365–376, June 2011.

[9] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, J. Hong, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth Generation Intel Core Processor. In *IEEE Micro*, **34**(2):6–20, March/April 2014.

[10] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[11] M. K. Jeong, M. Erez, C. Sudanthi, and N. C. Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference*, pages 850–855, June 2012.

[12] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 332–343, June 2013.

[13] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–406, March 2013.

[14] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*, pages 351–363, June 2016.

[15] D. Kanter. Intel's Ivy Bridge Graphics Architecture. April 2012. Available at http://www.realworldtech.com/ivy-bridge-gpu/.

[16] D. Kanter. Intel's Sandy Bridge Graphics Architecture. August 2011. Available at http://www.realworldtech.com/sandy-bridge-gpu/.

[17] D. Kanter. AMD Fusion Architecture and Llano. June 2011. Available at http://www.realworldtech.com/fusion-llano/.

[18] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the 47th International Symposium on Microarchitecture*, pages 114–126, December 2014.

[19] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, September 2013.

[20] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. In *IEEE Computer Architecture Letters*, **11**(2): 33–36, July 2012.

[21] S-Y. Lee, A. Arunkumar, and C-J. Wu. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the 42nd International Symposium on Computer Architecture*, pages 515–527, June 2015.

[22] S-Y. Lee and C-J. Wu. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 175–186, August 2014.

[23] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 91–102, February 2012.

[24] V. Mekkat, A. Holey, P-C. Yew, and A. Zhai. Managing Shared Last-level Cache in a Heterogeneous Multicore Processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 225–234, September 2013.

[25] P. Messmer. Interactive Supercomputing with In-Situ Visualization on Tesla GPUs. Available at https://devblogs.nvidia.com/parallelforall/interactive-supercomputing-in-situ-visualization-tesla-gpus/.

[26] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 231–241, March 2006. Source and traces available at http://attila.ac.upc.edu/wiki/index.php/Main_Page.

[27] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. T. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 374–385, December 2011.

[28] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. GemDroid: A Framework to Evaluate Mobile Platforms. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 355–366, June 2014.

[29] T. Olson. Mali 400 MP: A Scalable GPU for Mobile and Embedded Devices. In *Symposium on High-Performance Graphics*, June 2010.

[30] T. Piazza. Intel Processor Graphics. In *Symposium on High-Performance Graphics*, August 2012.

[31] S. Rai and M. Chaudhuri. Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors. In *Proceedings of the 30th International Conference on Supercomputing*, June 2016.

[32] M. Ribble. Next-gen Tile-based GPUs. In *Game Developers' Conference*, 2008.

[33] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, **10**(1): 16–19, January-June 2011.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[35] A. L. Shimpi. Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested. June 2013. Available at http://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested.

[36] D. Shingari, A. Arunkumar, and C-J. Wu. Characterization and Throttling-Based Mitigation of Memory Interference for Heterogeneous Smartphones. In *Proceedings of the International Symposium on Workload Characterization*, pages 22–33, October 2015.

[37] A. Stevens. QoS for High-performance and Power-efficient HD Multimedia. *ARM White Paper*, 2010.

[38] J. Stone. HPC Visualization on Nvidia Tesla GPUs. Available at https://devblogs.nvidia.com/parallelforall/hpc-visualization-nvidia-tesla-gpus/.

[39] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 335–344, September 2012.

[40] H. Usui, L. Subramanian, K. K-W. Chang, and O. Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. In *ACM Transactions on Architecture and Code Optimization*, **12**(4), January 2016.

[41] J. Walton. The AMD Trinity Review (A10-4600M): A New Hope. May 2012. Available at http://www.anandtech.com/show/5831/amd-trinity-review-a10-4600m-a-new-hope/.

[42] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. In *Proceedings of the International Solid-State Circuits Conference*, pages 264–266, February 2011.

[43] 2015 International Technology Roadmap for Semiconductors (ITRS). http://www.semiconductors.org.

[44] 3D Mark Benchmark. http://www.3dmark.com/.