Pipelined CPU-GPU Scheduling for Caches

Daniel Gerzhoy and Donald Yeung
Department of Electrical & Computer Engineering
University of Maryland at College Park

ABSTRACT

Heterogeneous microprocessors integrate a CPU and GPU with a shared cache hierarchy on the same chip, affording low-overhead communication between the CPU and GPU's cores. Often times, large array data structures are communicated from the CPU to the GPU and back. While the on-chip cache hierarchy can support such CPU-GPU producer-consumer sharing, this almost never happens due to poor temporal reuse. Because the data structures can be quite large, by the time the consumer reads the data, it has been evicted from cache even though the producer had brought it on-chip when it originally wrote the data. As a result, the CPU-GPU communication happens through main memory instead of the cache, hurting performance and energy.

This paper exploits the on-chip caches in a heterogeneous microprocessor to improve CPU-GPU communication efficiency. We divide streaming computations executed by the CPU and GPU that exhibit producer-consumer sharing into chunks, and overlap the execution of CPU chunks with GPU chunks in a software pipeline. To enforce data dependences, the producer executes one chunk ahead of the consumer at all times. We also propose a low-overhead synchronization mechanism in which the CPU directly controls thread-block scheduling in the GPU to maintain the producer's "run-ahead distance" relative to the consumer. By adjusting the chunk size or run-ahead distance, we can make the CPU-GPU working set fit in the last-level cache, thus permitting the producer-consumer sharing to occur through the LLC. We show through simulation that our technique reduces the number of DRAM accesses by 30.4%, improves performance by 26.8%, and lowers memory system energy by 27.4% averaged across 7 benchmarks.

1. INTRODUCTION

Heterogeneous microprocessors integrate a CPU and GPU onto the same chip, providing physical proximity between the two. Compared to discrete GPUs, the physical proximity allows for significantly lower-latency CPU-GPU communication. Not only can the CPU and GPU communicate through a shared main memory system, but many heterogeneous microprocessors also integrate shared caches and support cache coherence between the CPU and GPU as well, permitting communication to remain entirely on-chip when access patterns permit.

Enabled by these efficient communication mechanisms, a few researchers have recently developed parallelization techniques that utilize the GPUs in heterogeneous microprocessors to speedup more complex and irregular codes. Traditionally, discrete GPUs have been used to accelerate massively parallel kernels which amortize the high cost of kernel off-loads on these systems. But, the efficient communication mechanisms associated with integrated GPUs permit more frequent off-loads of smaller loops to exploit finer granularities of parallelism. This means a wider variety of SIMD loops, possibly contained within larger non-SIMD computations, can be gainfully off-loaded onto the integrated GPUs of heterogeneous microprocessors. At the same time, the CPU cores can be used to execute parallel non-SIMD computations, perhaps themselves overlapped with GPU execution. Such *heterogeneous parallelization of MIMD and SIMD code regions* has been demonstrated for distributed loops [1] as well as nested loops [2].

Besides enabling acceleration of more complex codes, the fast CPU-GPU communication mechanisms available in heterogeneous microprocessors can also benefit massively parallel kernels that have traditionally been accelerated using GPUs. Large GPU kernels move significant amounts of data into and out of the compute units of a GPU. Often, this data is either produced by the CPU immediately prior to kernel launch, or is consumed by the CPU immediately after the GPU finishes execution, or both. Such *producer-consumer sharing* between the CPU and GPU naturally arises as computation migrates from the CPU to the GPU and back.

While heterogeneous microprocessors efficiently support CPU-GPU communication, unfortunately, the on-chip cache mechanisms that afford the greatest levels of efficiency are bypassed for producer-consumer sharing across large kernels. The problem is poor temporal reuse owing to the large volumes of data that are accessed by the CPU and GPU. As a result, any producer-consumer sharing occurring from the CPU to the GPU or vice versa is not supported by the on-chip caches, but instead, occurs entirely through DRAM.

In this paper, we propose *pipelined CPU-GPU scheduling for caches*, a locality transformation supported by a novel CPU-GPU synchronization mechanism that increases temporal reuse between the CPU and GPU. During kernel execution, a GPU does not access the entire dataset associated with the kernel all at once. Instead, it tends to consume and then produce data in a linear streaming fashion. The same is true for the CPU, both when setting up the input data prior to a kernel launch and when consuming the GPU's results after kernel execution. Hence, it is possible to overlap the CPU and GPU execution, creating a software pipeline in which the producer executes just in front of the consumer, feeding it data. A novel synchronization mechanism sequences the producer and consumer through their pipeline stages.

Such pipelined CPU-GPU execution can provide higher performance through increased parallelism. However, *pipelining the CPU and GPU also permits tuning the degree of tem-*

```
__global__ void GPU_Producer(int *A) {
  int i = blockSize() * blockId() + threadId();
  A[i] = ...; //Write to A (produce)
}

void CPU_Consumer(int *A, int iters) {
  for ( int i = 0; i < iters; i++ ) {
      ... = A[i]; //Read from A (consume)
  }
}

int main() {
  int nBlocks = nThreads/threads_per_block;
  int *A = malloc(nThreads * sizeof(int));

  GPU_Producer<<<nBlocks, threads_per_block>>>(A);
  deviceSynchronize();

  CPU_Consumer(A);
}
```

Figure 1: Kernel off-load with producer-consumer (GPU to CPU) sharing of a large array.

poral reuse associated with the producer-consumer sharing pattern simply by controlling how far ahead of the consumer the producer is allowed to execute. If a sufficiently small "run-ahead distance" is maintained, then the size of the data communicated from the producer to the consumer can be made to fit in the heterogeneous microprocessor's caches. Thus, the communication occurs entirely on-chip. This not only improves performance, but it also benefits energy and efficiency by significantly reducing the number of accesses to main memory.

Our work makes several contributions in the context of pipelined CPU-GPU scheduling for caches:

- We propose to overlap CPU and GPU execution in a software pipeline to improve temporal reuse of shared between the producer and consumer.
- We propose a novel hardware synchronization mechanism, called *thread-block throttling*, that permits the CPU to directly control the rate of execution in the GPU, and use this mechanism to maintain the producer's run-ahead distance relative to the consumer.
- We undertake a simulation-based evaluation using seven benchmarks that shows our technique reduces memory system energy by 27.4% and increases performance by 26.8% on average.

The rest of this paper is organized as follows. Section 2 presents our pipelined CPU-GPU scheduling technique with the novel thread-block throttling mechanism. Then, Section 3 describes the experimental methodology used for our quantitative evaluation. Next, Section 4 presents the results. Finally, Section 5 discusses related work, and Section 6 concludes the paper.

2. PIPELINED CPU-GPU SCHEDULING

GPU kernel offloads are a form of computation migration, moving computations from the CPU to the GPU to take advantage of more parallel hardware. This computation migration is accompanied by data movement in a producer-consumer fashion: the CPU provides input values to the GPU upon kernel initiation, while the GPU provides results back to the CPU upon kernel completion. Moreover, the amount of data movement can be quite significant if large array data structures are involved.

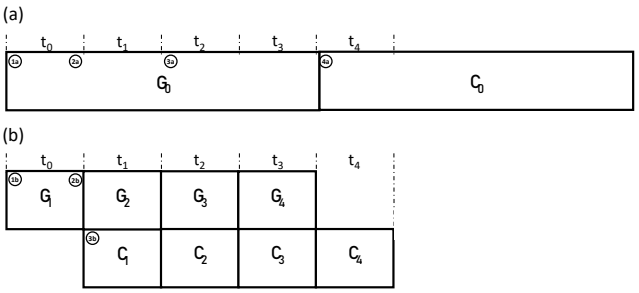


Figure 2: (a) Serial schedule. (b) Pipelined schedule.

The goal of our technique is to exploit the on-chip cache hierarchy of heterogeneous microprocessors to support such producer-consumer data movement between the CPU and GPU efficiently. This section describes the program transformation and GPU support needed by our technique.

2.1 Scheduling for Cache Locality

Figure 1 shows a working code example that we assume runs on a heterogeneous microprocessor. In the figure, the *GPU_Producer* kernel writes an integer array, *A*, while the *CPU_Consumer* function reads it. The GPU kernel and CPU function are separated by a synchronization operation (*deviceSynchronize*); hence, the two execute serially such that the GPU kernel produces the entire array before the CPU function begins consuming it. Figure 2(a) illustrates this serial execution of the GPU kernel and the CPU function over time.

If the integer array, *A*, is large compared to the microprocessor's on-chip caches, then the producer-consumer sharing will occur through DRAM, as illustrated in Figure 3. In Figure 3, we assume the GPU and CPU both have their own private caches, but a last-level cache (LLC) is shared between the two. As the *GPU_Producer* kernel executes, it streams the *A* array into the GPU's private cache in order to perform the producer writes (labeled ①ₐ). Assuming the LLC is managed as a victim buffer, the *A* array bypasses the shared cache during the GPU's initial demand fetches, but fills the LLC when it is evicted from the GPU's private cache (labeled ②ₐ). Eventually, the LLC itself becomes full with *A* array elements, and evicts them back to DRAM (labeled ③ₐ). By the time the *CPU_Consumer* function executes, the data it references has left both the GPU's private cache and the LLC, so the CPU misses to DRAM (labeled ④ₐ). Hence, the *A* array is fetched from DRAM twice: once by the GPU and once by the CPU.

To address this inefficiency, we propose to overlap the GPU and CPU execution so that temporal reuse of the *A* array is improved. Our technique creates a software pipeline of the *GPU_Producer* kernel and the *CPU_Consumer* function. (We also propose a hardware mechanism for efficiently synchronizing the software pipeline, which will be presented in Section 2.2). Instead of serializing the GPU and CPU, we

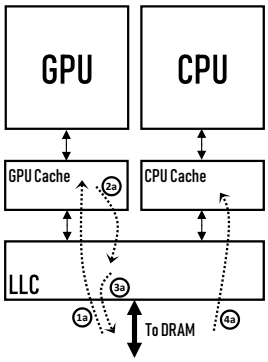


Figure 3: Shared data evicted from LLC before rereference.

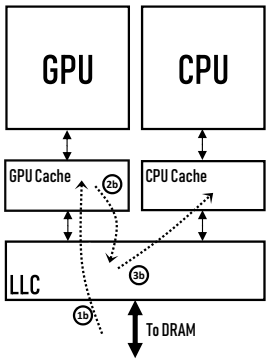


Figure 4: Shared data Hits in LLC at rereference.

chunk the *GPU_Producer* kernel and *CPU_Consumer* function, and execute chunks from the GPU and CPU simultaneously. To enforce data dependences, we stagger the chunks such that the GPU always runs one chunk ahead of the CPU: as the CPU consumes the portion of the *A* array corresponding to chunk *i*, the GPU produces the portion of the *A* array corresponding to chunk *i+1*. Figure 2(b) illustrates this pipelined execution of the GPU kernel and the CPU function assuming each is divided into 4 chunks.

Comparing Figures 2(a) and 2(b), we can see that software pipelining improves performance in part because of parallel execution of the *GPU_Producer* kernel and *CPU_Consumer* function. Rather than executing all 8 chunks in series as shown in Figure 2(a), the pipeline overlaps the execution of 3 chunks, reducing the execution time by $\frac{3}{8}$. (This assumes all chunks run for the same amount of time. Depending on the application, it is also possible for per-chunk execution times to vary which could result in load imbalance and less speedup.) In general, for a chunking factor N, the execution is reduced from $2N$ chunks down to $N+1$ chunks.

But in addition to increased parallelism, software pipelining also reduces the liveness of the *A* array. Rather than wait until the GPU completes the entire kernel to begin execution, the CPU starts consuming the *A* array right after the GPU completes the first chunk. If the GPU and CPU remain synchronized such that the GPU runs ahead of the CPU by 1 chunk, then the CPU will always consume the chunk just produced by the GPU as the GPU produces the next chunk. This means that the producer-consumer sharing can occur through the on-chip cache hierarchy *if two chunks can fit simultaneously in the LLC*. By choosing a sufficiently small chunking factor, or "run-ahead distance," the combined GPU-CPU working set of 2 chunks can be made to fit in cache and enable on-chip communication of the *A* array. The run-ahead distance (RAD) that achieves this for each benchmark can be determined either analytically or experimentally, which we will show in Section 4.1.

Figure 4 illustrates the case when producer-consumer sharing occurs through the on-chip cache. Similar to Figure 3, Figure 4 shows the *GPU_Producer* kernel filling the LLC by way of the GPU's private cache (labeled ①b and ②b). Thanks to the improved temporal reuse afforded by software pipelining, the *CPU_Consumer* function references the *A* array data before it has a chance to leave the LLC, resulting in an LLC hit (labeled ②b). With software pipelining, the *A* array is only fetched from DRAM once.

Although our running example in Figures 1 through 4 involves the specific case of a single GPU producer and a single CPU consumer, our technique generalizes to many other cases. First, the direction of communication can be reversed: it is possible for a CPU producer to feed a GPU consumer. Second, there can be multiple producer / consumer stages working at the same time. Specifically, a chain of 3 or more stages could execute back-to-back. (For example, a CPU producer feeds a GPU consumer which becomes a GPU producer that feeds a CPU consumer). Rather than ensure that each stage takes up 1/2 the LLC, with more simultaneous stages, the fraction of the LLC allocated to each stage goes down proportionally. Section 3.3 will present our workloads and discuss the different software pipelines that are possible.

2.2 Thread-Block Throttling

In addition to chunking the *GPU_Producer* kernel and the *CPU_Consumer* function and executing the chunks in an overlapped fashion, it is also necessary to synchronize the GPU and CPU so that neither one gets ahead of the other in the software pipeline and violate data dependences. Such CPU-GPU synchronization can be challenging, though, given the massive parallelism in the GPU. A critical issue is the amount of computation per synchronization operation. In particular, the smaller the per-synch computation, the more efficient the synchronization mechanisms need to be, and potentially, the greater the coordination that will be necessary with the GPU's massively parallel threads.

As shown in Figure 2(b), a synchronization operation is performed after every chunk is executed by the GPU and CPU. Chunks are sized according to their cache footprint, with the requirement that two chunks must fit in the LLC simultaneously. The relationship between chunk size–say, in terms of GPU threads–and cache footprint size can be highly application dependent. However, in our benchmarks, *we find that each GPU chunk can have several times the number of hardware threads resident in the GPU, and yet still exhibit a cache footprint that fits within the LLC.*

For example, assume the code from Figure 1 runs on a heterogeneous microprocessor with a 4MB LLC. Each chunk of the *GPU_Producer* kernel should not use more than half the LLC, or 2MB. Given 4-byte elements, this implies the GPU can produce 512K elements of the *A* array each time it executes a chunk. In Figure 1, each *A* array element is produced by a single GPU thread, so this translates into 512K threads per chunk, which is about 70x more than the number of threads in the GPU from our experiments (7,680).

This per-synch computation granularity has implications for the kind of synchronization mechanisms we require. For instance, it is likely that purely software approaches will be inadequate. The simplist software approach is to divide the original GPU kernel in Figure 2(a), labeled "GPU_0," into multiple sub-kernels corresponding to the chunks shown in Figure 2(b), labeled "GPU_1"–"GPU_4," and to perform a launch and deviceSynchronize() operation for each sub-kernel. Unfortunately, kernel launches are heavy weight operations with high associated overheads. Although the per-

CPU		GPU	
Number of cores	4	Number of CUs	3
CPU Clock rate	2.95 GHz	GPU Clock rate	1100 MHz
Issue width	8	Number of SIMD Units per CU	4
Issue queue size	64	SIMD size	16
Reorder buffer size	192	Wavefront size	64
L1-I cache (private per core)	32 KB	Wavefront slots (max WFs per CU)	10
L1-D cache (private per core)	64 KB	L1 (TCP) Size (Private per CU)	128 KB
L2 cache (shared per core-pair)	2 MB	L2 (TCC) Size	256 KB
L3 Cache (LLC)			4MB
Main Memory			8 GB DDR4 16x4 (64 bit) @ 2400 MHz

Table 1: Simulation parameters used in the experiments. The modeled heterogeneous microprocessor resembles a Ryzen 2XXXU series APU.

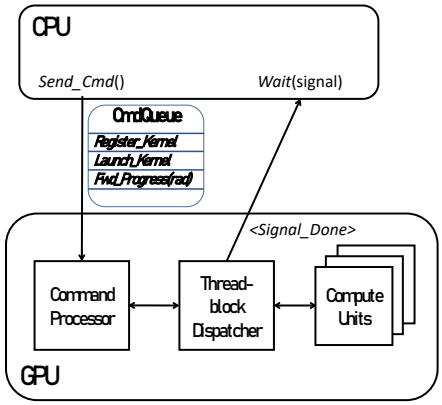


Figure 5: Thread-block throttling mechanism.

synch computation exceeds the number of GPU hardware threads, performing a kernel launch for every chunk would result in significant performance degradation.

Rather than rely on purely software approaches, we propose a hardware-assisted mechanism to mitigate the overheads. Our approach launches a single GPU kernel ("GPU_0" in Figure 2(a)) to amortize the launch overhead, but allows the CPU to control the progress of the kernel's execution. To do this, we expose the scheduling of thread blocks within the GPU's command processor / thread-block dispatcher shown in Figure 5. Normally, the command processor / thread-block dispatcher schedules as much of a kernel's pool of thread blocks as will fit onto the GPU. We modified the dispatcher to schedule thread blocks in groups equal to the per-synch computation size. (The group size is communicated to the command processor during kernel launch). The dispatcher schedules thread blocks one group at a time, with each group released only after a signal from the CPU. (This CPU signal is much lighter weight than a kernel launch). Upon completion of each group, the GPU also signals to the CPU so that it may coordinate the release of subsequent groups. We call this synchronization mechanism *thread-block throttling*. Section 3.2 will provide more details on its implementation.

3. METHODOLOGY

This section describes our experimental methodology for evaluating pipelined CPU-GPU scheduling for caches. Recently, a new Gem5 simulator [3] was developed to include a realistic integrated GPU model from AMD based on the Graphics Core Next 3 (GCN3) architecture, and to support the HSA standard [4]. This new Gem5 simulator better reflects how the hardware-software stack in a real GPU works compared to older simulators, like the original Gem5-gpu, so we use it in our evaluation. Section 3.1 discusses the simulation parameters we use with the new Gem5 simulator, and describes the cache hierarchy architecture we model. Next, Section 3.2 presents the software architecture of the new Gem5's GPU driver system, and the customizations that we created within that driver system to support our technique. Finally, Section 3.3 discusses the workloads used in the quantitative evaluation of our technique.

3.1 Model Configuration

Table 1 lists the configuration parameters we used in the evaluation of our technique on the Gem5 simulator. (The terminology for the GPU attributes in this table is from AMD). We based the configuration off of the Ryzen 3 2XXXU series of APUs. The modeled chip has 4 out-of-order CPU cores integrated with a modestly sized GPU. Since gem5 does not model Dynamic Voltage and Frequency Scaling (DVFS), we chose clock speeds for both the CPU and GPU in the middle of the range for the Ryzen 2200U chip.

Cache Hierarchy. Unlike the original Gem5-gpu simulator which did not model a shared last-level cache (LLC), the new Gem5 simulator does implement an LLC that can support fast CPU-GPU data sharing. Each CPU has private L1 I/D caches, with every two CPUs grouped together in "core pairs" sharing an L2 cache. The GPU's Compute Units (CUs) share an L1 instruction cache known as a Sequencer Cache (SQC), while each CU has a private L1 data cache known as a Texture Cache per Pipe (TCP). The CUs share the GPU's L2 cache known as a Texture Cache per Channel (TCC) [4]. Since our configuration has 4 CPU cores and 3 CUs, there are 3 L2 caches in the system.

In the new Gem5 simulator, the LLC is managed as an exclusive victim cache for the GPU and CPU L2s, controlled by a stateless directory-based controller that implements a coherence protocol called GPU_VIPER. In this protocol, read requests from the L2s check the LLC for the requested block, and if a hit occurs, removes that block from the LLC and sends it to the requesting L2. If a miss occurs, the controller sends a request to main memory while at the same time prob-

Benchmark	Suite	Input	Stage Order
CEDT	Chai	2146 x 3826 video	GGCC
BE	Hetero-Mark	1080p video	CG
EP	Hetero-Mark	8192 Creatures	CGC
DWT2D	Rodinia	1125x2436 image	CG
Kmeans	Rodinia	512K Objects, 34 Features	GC
LavaMD	Rodinia	1000 boxes, 100 Particles per box	CG
SmithWa	OMP2012	ref - 1048576	GC

Table 2: Benchmarks used in the experimental evaluation of Pipeline Scheduling for Shared Data Cache Locality.

ing the other L2s for the requested block. One implication of this is that even if the data is in one of the L2s, DRAM is still read and the energy used for this access is wasted. Presumably, the protocol was designed this way to minimize read latency: speculatively reading DRAM without waiting for the L2 probes to come back. Thus, we are motivated further to service requests from the LLC, regardless of the block's location in the cache hierarchy.

Writes for the CPU and GPU behave differently from each other. The CPU caches write back evicted blocks whether they are clean or dirty. The blocks are then stored in the LLC by the controller. The GPU's caches, on the other hand, are write-through caches. Stores to blocks automatically update all levels of the cache hierarchy, and any evictions from the GPU's L2 cache are considered to be write throughs as well. By default, the LLC is bypassed during a write through, but a pre-existing option exists to fill the LLC during a write through. Our technique relies on this option to keep producer-consumer data in the cache system, and we keep the option turned on save for experiments aimed at removing the benefits of our technique which we will describe in Section 4.

When an eviction is necessary in the GPU's L2, the GPU only writes dirty blocks through to the LLC. This makes sense for the GPU which could be reading massive amounts of data, to not thrash the LLC. However, it creates a problem for workloads that exhibit Read-Read sharing with a GPU kernel reading the data first. Since reads by default are not cached in the LLC and the GPU does not evict clean blocks to the LLC, the CPU has to read main memory a second time for the same data. To alleviate this problem, we modified the GPU_VIPER protocol to be able cache reads in the LLC when it receives a read request.

3.2 Driver Stack Architecture

Along with AMD's GCN3 architecture, the new Gem5 simulator also supports AMD's Radeon Open Compute Platform (ROCm), which serves as the hardware-software interface between the workloads and the GPU. ROCm enables communication from user space to the emulated Kernel Fusion Driver in kernel space (ROCk) by sending command packets conforming to the HSA specification through software queues that map to hardware queues on the GPU. The emulated kernel receives the packets and sends them to the GPU's command processor which executes various functions according to the packet type and sends back a completion signal when the task has been completed.

For instance, when a user performs a kernel launch through ROCm, it sends a *kernel dispatch packet* containing the location of the kernel's code in memory along with its parameters and an additional completion signal to the GPU command processor. The command processor then instructs the hardware scheduler to schedule the kernel's thread blocks to the GPU's compute units, and signals that the kernel has been launched. Finally, when the last thread block of the kernel is completed, the GPU sends the kernel completion signal back to user space via the kernel driver.

Custom Scheduling Controller. In order to implement our thread-block throttling mechanism from Section 2.2, we exploit a type of HSA command packet, known as an *agent dispatch packet*, which contains fields set by the application that the command processor can read. We customized the command processor and hardware dispatcher to respond to two new commands from the agent packet: INJECT_SIGNAL and FWD_PROGRESS. The INJECT_SIGNAL command injects a custom HIP signal created by the software interface and associates that signal with a kernel id. If the hardware dispatcher sees that a custom signal has been injected for a particular kernel id, when that kernel is launched with a normal kernel launch packet, it will not schedule any thread blocks for execution on the GPU. Instead, when the application desires threads blocks to execute on the GPU, it sends the second type of command, FWD_PROGRESS. This command instructs the dispatcher to execute a given number of thread blocks rather than all of the kernel's thread blocks. The number of thread blocks executed can be varied by the application in user space to control the cache footprint of the GPU. When the last of these thread blocks is completed, the command processor sends back the custom signal given to it from the injection command packet.

Using the HSA API significantly reduces the synchronization overhead. Across our benchmarks, we find that our custom synchronization mechanism is roughly an order of magnitude faster than a kernel launch. However, the HSA API also introduces complexity to programmers who want to use our optimization technique. To mitigate the complexity associated with the underlying control scheme, we created a software interface that abstracts much of the complexity away from the programmer. The programmer need only wrap an interface class around the producer-consumer stages in their application code, and then call the pipeline.

3.3 Benchmarks

We use seven benchmarks, shown in Table 2, to evaluate our technique. *CEDT* is the task partitioning version of Canny Edge Detection; *BE* performs background extraction in which a video is passed frame by frame from the CPU to the GPU; *EP* is a genetic algorithm that simulates the evolution of creatures on an island; *DWT2D* performs a popular digital signal processing technique called discrete wavelet transform; *Kmeans* computes the well-known k-means clustering algorithm; *LavaMD* performs a 3D molecular dynamics

Benchmark	PTAS	TPB	RAD-A	RAD-E
CEDT	10	256	1639	956
BE	12	64	5462	8192
EP	12,064	256	3	3
DWT2D	12	256	911	256
Kmeans	284	256	58	32
LavaMD	157.04	100	268	500
SmithWa	73	512	112	65

Table 3: Run-ahead distance determined analytically (RAD-A) and through experimental sweeps (RAD-E).

simulation; and SmithWa implements the Smith-Waterman sequence alignment algorithm. Each of these benchmarks comes from one of four benchmark suites, as indicated in the second column of Table 2: Chai [5], Hetero-Mark [6], Rodinia [7], or SPEC OMP 2012 [8]. (Although SmithWa is originally from SPEC OMP, we use a version of this benchmark parallelized for integrated CPU-GPU chips performed in our prior work [2]).

The third column of Table 2 reports the program inputs we used for each benchmark. In most cases, these are the standard inputs that come with the benchmarks. However, in CEDT and Kmeans, the standard inputs result in small cache footprints that fit in the LLC we simulated, so we increased these input sizes. (For CEDT, we used a higher-resolution video, and for Kmeans, we generated a larger input using the dataset generator that comes with the benchmark). We simulated each benchmark by fast-forwarding past its initialization code, and then turning on detailed models to simulate its main compute code. The one exception is DWT2D. For this benchmark, we performed detailed simulation of the file I/O and pre-processing steps that preceed the main computation. These initialization steps occur in many image processing workloads, and give rise to significant CPU-GPU communication. Although the main computation in DWT2D dominates execution time, its initialization code could be found in many different applications, so we believe studying it is still worthwhile.

Finally, the last column in Table 2 shows the structure of the software pipelines we created for each benchmark. In most cases, there is a single pipeline from the CPU to the GPU, or from the GPU to the CPU (where the latter is the case illustrated in Figure 2 and discussed in Section 2.1). But we also created more complex pipelines, too. For EP, there are two back-to-back pipelines: one from the CPU to the GPU and than another from the GPU back to the CPU. And for CEDT, there is an even longer chain of 4 pipelines: two GPU stages followed by two CPU stages.

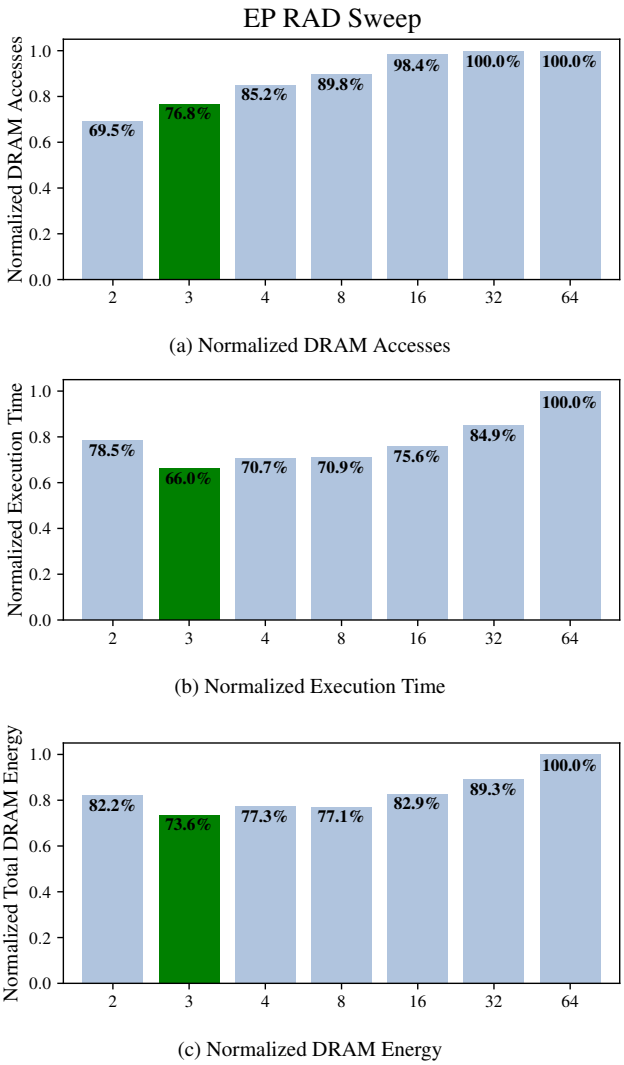
4. EXPERIMENTAL EVALUATION

This section presents our experimental results that demonstrate the effectiveness of pipelined CPU-GPU scheduling for caches. We begin in Section 4.1 with results on determining the best run-ahead distance for each of our benchmarks. Then, we present the memory and performance results in Section 4.2.

4.1 Run-Ahead Distance

As discussed in Section 2.1, our technique requires determining the run-ahead distance (RAD). In particular, our technique determines the RAD in terms of GPU thread blocks since the mechansim from Section 2.2 controls the run-ahead distance by throttling thread blocks. Because the number of threads in each thread block and the amount of data accessed by each thread is application specific, each benchmark will have a different RAD that permits its working set to fit in the LLC (which is fixed at 4MB, as shown in Table 1).

One way to determine the RAD is through analysis of a benchmark's code to identify how much data each thread accesses, and then compute the number of thread blocks that could be accommodated in the LLC given the analyzed per-thread data access size. In Table 3, the column labeled "PTAS" reports this per-thread access size in bytes for each benchmark which we acquired manually. Multiplying this value by the number of threads per thread block (reported in the column labeled "TPB") yields the data footprint for a single thread block from each benchmark. Dividing the LLC capacity by this value yields the analytical RAD, which we report in the column labeled "RAD-A" in Table 3.



(a) Normalized DRAM Accesses

(b) Normalized Execution Time

(c) Normalized DRAM Energy

Figure 6: Run-ahead distance (RAD) sweep for the EP benchmark. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD shown in green.

While the RAD-A results in Table 3 are relatively easy to compute, they may be inaccurate since the analytical approach does not take into consideration factors such as limited LLC associativity nor runtime overhead. To quantify the impact of such real-world effects on the RAD value, we also ran our benchmarks on the simulator multiple times, sweeping the RAD value around the analytically computed values.

For example, Figure 6 shows our RAD sweep experiments for the EP benchmark. Since Table 3 reports a RAD-A value of 3 for EP, in Figure 6, we sweep RAD from 2 to 64 (X-axis), and graph three metrics reported by the simulator: number of DRAM accesses, execution time, and memory system energy. (For all three metrics, lower is better). Figure 6a shows that a smaller RAD value of 2 results in even fewer cache misses and DRAM accesses; however, greater runtime overhead occurs with the smaller RAD value, causing execution time and memory system energy to get worse, as shown in Figures 6b and 6c. In our work, we use energy as the determiner for the best RAD value. Based on energy, Figure 6c shows 3 is indeed the best RAD value for EP.

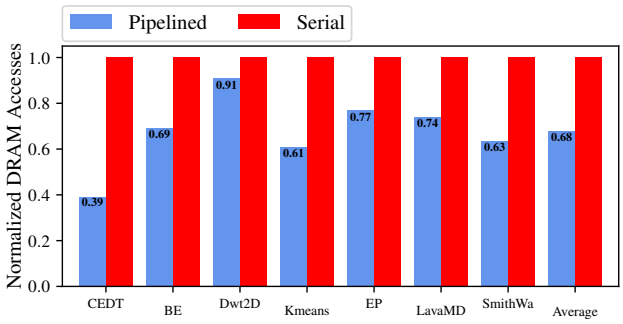
We performed similar RAD sweep experiments for all the benchmarks, and identified the best RAD value experimentally. The column labeled "RAD-E" in Table 3 reports these results. Although the analytically and experimentally computed RAD values for EP are identical, Table 3 shows that RAD-A and RAD-E are not the same in the other benchmarks. In some cases they are similar, but in other cases, there can be a noticeable discrepancy. In our main results reported next, we use the RAD-E values from Table 3.

4.2 Results

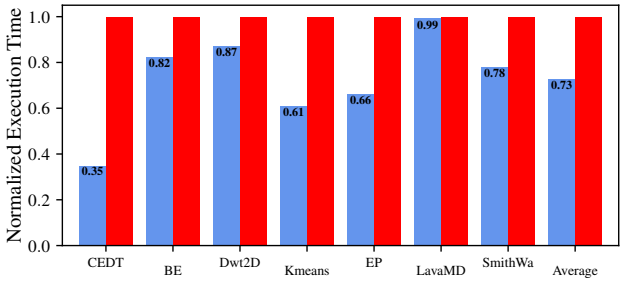
Figure 7 presents the main results of our evaluation. It reports the simulated results for our technique, pipelined CPU-GPU scheduling for caches (blue bars), normalized to the default serial execution (the "1.0" red bars). In the simulations of our technique, the run-ahead distance maintained within software pipelines is the experimentally determined RAD-E values from Table 3. As in Figure 6, results are shown for three separate metrics: number of DRAM accesses, execution time, and memory system energy.

In Figure 7a, we see that our technique significantly reduces LLC misses and their subsequent DRAM accesses across all of the benchmarks. At least 9% (DWT2D), and as much as 61% (CEDT), of the DRAM accesses are eliminated by our technique. Averaged across all benchmarks, the number of DRAM accesses goes down by 30.4% compared to serial execution. This directly quantifies the benefit of keeping producer-consumer communication within the on-chip cache hierarchy.

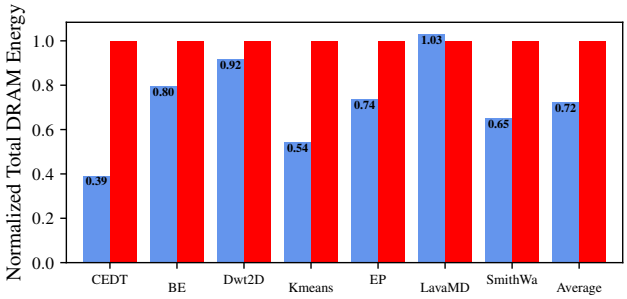
These main memory access savings translate into performance gains. On average, Figure 7b shows our benchmarks enjoy a 26.8% reduction in execution time. Workloads with CPU consumer stages (CEDT, EP, Kmeans, and SmithWa) received the largest performance gains, achieving an average 38.8% execution time reduction. Our technique keeps data meant for CPU consumer stages in the LLC, reducing access latency which can significantly benefit the latency-sensitive CPU cores. On the other hand, those workloads with GPU consumers (BE, DWT2D, LavaMD) did not receive as much performance gain, achieving a less substantial



(a) Normalized DRAM accesses



(b) Normalized execution time



(c) Normalized DRAM energy

Figure 7: Results for all seven benchmarks. Blue bars show pipelined CPU-GPU scheduling for caches using the experimentally determined RAD. Red bars show serial execution.

10.9% reduction in execution time. This is to be expected since the GPU cores are more latency tolerant. For benchmarks with GPU consumers, the speedups are primarily due to software pipelining overlap, and not to locality improvement. The benchmark in Figure 7b with the smallest performance gain, LavaMD, only achieves a 0.74% reduction in execution time. Not only does LavaMD exhibit GPU consumers, but the GPU stage's execution time is much larger than the CPU stage's execution time, leaving little opportunity for overlapped pipeline execution.

Finally, Figure 7c shows that the DRAM access reductions and performance gains from our technique afford memory system energy savings. Averaged across all the benchmarks, we achieve a 27.4% reduction in total DRAM energy. This includes access energy savings as well as reductions in refresh, pre-charge, and associated background energies. Again, we see that the CPU consumer patterns perform better than their GPU consumer counterparts: a 41.3%

reduction in energy on average compared to only 8.8% on average. Notably, LavaMD actually receives an *increase* in total DRAM energy compared to the serial case. While the access energy goes down proportionally to the savings in accesses, the background energies, namely precharge and activation background energies, increase when we apply our technique. LavaMD performs a stencil computation where each block within the calculation accesses its neighbors and has non-contiguous data structures to keep track of details about its neighbors. When our technique is applied, data structures for non-contiguous blocks are accessed, leading to banks waiting in activated and precharged states for a longer amount of time.

5. RELATED WORK

Hestness *et al.* [9] were the first to recognize that pipelining GPU kernels can improve temporal locality and make use of the on-chip caches within heterogeneous microprocessors. However, they do not conduct a detailed study of such locality transformations. Compared to their work, ours is the first to present a synchronization mechanism that permits the CPU to have direct control over GPU execution at an intra-kernel granularity for the purposes of software pipelining. We are also the first to present detailed results on the efficacy of CPU-GPU locality transformations.

Work by Kim *et al.* [10] recognizes that GPGPU workloads may consist of multiple dependent stages that include CPU, GPU kernels, I/O, and copies that constitute pipeline parallelism. They introduce several optimizations in the hardware and virtual memory system to automatically schedule GPU thread blocks based on their dependence relationships with other stages. Rather than study integrated heterogeneous microprocessors, they investigate these pipeline optimizations for discrete GPGPU platforms. In contrast, our work studies integrated CPU-GPU chips, and focuses specifically on saving energy by reducing superfluous DRAM accesses.

Kayi *et al.* [12] and Cheng *et al.* [11] both dynamically detect producer-consumer sharing in chip multiprocessors and come up with coherence protocol optimizations to programs exhibiting producer-consumer sharing. They do not examine GPUs and the complexities they introduce to coherence and producer-consumer sharing.

Finally, several benchmark suites [6, 13, 5] have been developed in recent years to provide suitable programs to test heterogeneous chips. Previously, researchers needed to adapt CPU and traditional GPU benchmarks to glean insights about heterogeneous chips. These suites contain benchmarks which exhibit sharing, producer-consumer relationships, synchronization and more. Our research exploits the examples from these benchmark suites.

6. CONCLUSION

Heterogeneous microprocessors support efficient communication between the CPU and GPU via shared on-chip caches. This paper presents pipelined CPU-GPU scheduling for caches, a technique that improves temporal locality on data shared between the CPU and GPU so that communication can happen through the on-chip caches rather than main memory. Our technique breaks the computations performed in the CPU and GPU into chunks, and executes multiple chunks simultaneously in a software pipeline such that the producer of data executes one chunk ahead of the consumer of the data. Chunks are sized so that the aggregate CPU-GPU working set fits in the last-level cache. We develop a novel synchronization mechanism that permits the CPU to directly control the rate of thread-block scheduling in the GPU in order to maintain the producer's run-ahead distance relative to the consumer. We show through simulation that our technique reduces the number of DRAM accesses by 30.4%, improves performance by 26.8%, and lower memory system energy by 27.4% on average.

7. REFERENCES

[1] F. Wende, F. Cordes, and T. Steinke, "On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering," in *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing*, July 2012.

[2] D. Gerzhoy and D. Yeung, "Nested MIMD-SIMD parallelization for Heterogeneous Microprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, December 2019.

[3] J. Lowe-Power and M. Sinclair, "Re-gem5: Building sustainable research infrastructure," Sep 2019.

[4] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 608–619, 2018.

[5] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, IEEE, 2017.

[6] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, 2016.

[7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the International Symposium on Workload Characterization*, December 2010.

[8] "SPEC OMP 2012. *https://www.spec.org/omp2012/*." 2012.

[9] J. Hestness, S. Keckler, and D. Wood, "Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors," pp. 87–97, 10 2015.

[10] G. Kim, J. Jeong, J. Kim, and M. Stephenson, "Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pp. 341–352, 2016.

[11] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 328–339, 2007.

[12] A. Kayi, O. Serres, and T. El-Ghazawi, "Adaptive cache coherence mechanisms with producer–consumer sharing optimization for chip multiprocessors," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 316–328, 2015.

[13] M. D. Sinclair, J. Alsop, and S. V. Adve, "Heterosync: A benchmark suite for fine-grained synchronization on tightly coupled gpus," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 239–249, 2017.