A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior

Joel Hestness [1], Stephen W. Keckler [2], and David A. Wood [1]

[1]Department of Computer Sciences, The University of Wisconsin at Madison
[2]NVIDIA and Department of Computer Science, The University of Texas at Austin

***Abstract*—While heterogeneous CPU/GPU systems have been traditionally implemented on separate chips, each with their own private DRAM, heterogeneous processors are integrating these different core types on the same die with access to a common physical memory. Further, emerging heterogeneous CPU-GPU processors promise to offer tighter coupling between core types via a unified virtual address space and cache coherence. To adequately address the potential opportunities and pitfalls that may arise from this tighter coupling, it is important to have a deep understanding of application- and memory-level demands from both CPU and GPU cores. This paper presents a detailed comparison of memory access behavior for parallel applications executing on each core type in tightly-controlled heterogeneous CPU-GPU processor simulation. This characterization indicates that applications are typically designed with similar algorithmic structures for CPU and GPU cores, and each core type's memory access path has a similar locality filtering role. However, the different core and cache microarchitectures expose substantially different memory-level parallelism (MLP), which results in different instantaneous memory access rates and sensitivity to memory hierarchy architecture.**

I. Introduction

Desktops, servers and high-performance computing systems are rapidly incorporating graphics processors (GPUs) as a means of accelerating computation. Today, most of these systems employ general-purpose processors (CPUs) and GPUs on separate chips, each with their own DRAM memory systems. In the last couple years, these heterogeneous cores have been integrated on the same chip to provide both fast graphics rendering and accelerated computing. Recent product announcements by AMD [13], Intel [14], and NVIDIA [26] also indicate that emerging heterogeneous CPU-GPU processors will provide a unified virtual address space and cache coherence across these cores. These fused architectures promise to avoid many of the overheads associated with discrete GPUs, including interaction latency, and the time and energy to copy data between the different memories. When located on the same chip, these heterogeneous cores will share system resources such as the on-chip cache hierarchy, interconnect, and the DRAM. This integration raises the importance of understanding how each core type uses the memory system.

This paper presents a complete quantitative analysis of the memory access behavior of multithreaded CPU applications compared to their GPU counterparts with the aim of illuminating the application and microarchitectural causes of memory behavior differences, as well as the common effects of these differences. We focus on a memory system characterization to understand how each core type exposes parallel memory accesses, exploits locality, and leverages capabilities of the memory hierarchy architecture.

The most significant observations include:

- Memory access vectorization and coalescing reduce the number of spatially local accesses to cache lines. Both are fundamental means for exposing MLP to lower levels of the memory hierarchy.
- For data-parallel workloads, CPU and GPU cache hierarchies play substantially similar roles in filtering memory accesses to the off-chip interface.
- Due to threading and cache filtering differences, memory access patterns show substantially different time-distributions: CPU applications exhibit regularly time-distributed off-chip accesses, while GPU applications produce large access bursts.
- The bursty memory accesses of GPU cores makes them more sensitive to off-chip bandwidth while the MLP from multithreading makes them less sensitive to off-chip access latency than CPU cores.

Overall, CPU cores must extract very wide ILP in order to expose MLP to the memory hierarchy, and this MLP can be limited to lower levels of the memory hierarchy due to L1 cache locality. On the other hand, GPU cores and caches aim to mitigate MLP limitations, allowing the programmer to focus their efforts on leveraging the available MLP.

The rest of this paper is organized as follows. Section II describes the characterization methodology. Section III measures application-level characteristics including algorithm structure, operation counts, and memory footprint. Section IV details our measurements of memory access characteristics, including access counts, locality, and bandwidth demand. Section V quantifies and describes the performance effects of these different memory access characteristics. We discuss implications in Section VI and related work in Section VII. Section VIII concludes.

II. Methodology

To adjust many various system parameters and to collect detailed statistics, we simulate and compare the heterogeneous CPU-GPU processors as described in this section. This section also describes the benchmarks and simulation environment used for this comparison.

A. Simulated Systems

Figure 1 diagrams the heterogeneous CPU-GPU processor architecture that we simulate in each of our tests. The baseline parameters of this processor are included in Table I. This basic architecture incorporates CPU and GPU cores on a single chip, and the different core types are allowed to communicate through a unified address space shared memory.

TABLE I. HETEROGENEOUS CPU-GPU PROCESSOR PARAMETERS.

Core Type	ISA	Key Parameters	GFLOP/s Per Core	Baseline Count	Approx. Area (mm^2)
CPU cores	x86	5-wide OoO, 4.5GHz, 256 instruction window	22.5	4	5.3
GPU cores	PTX	8 CTAs, 48 warps of 32 threads each, 700MHz	22.4	4	6.1

Component	Parameters
CPU Caches	Per-core 32kB L1I + 64kB L1D and exclusive private L2 cache with aggregate capacity 1MB
GPU Caches	64kB L1, 48kB scratch per-core. Shared, banked, non-inclusive L2 cache 1MB
Interconnect	Peak 63 GB/s data transfer between any two endpoints
Memory	2 GDDR5-like DIMMs per memory channel, 32GB/s peak

Cores: CPU and GPU cores are at the top of the hierarchies, and we list their configurations in Table I. To control for many of the differences between CPU and GPU cores, we model an aggressive CPU core with peak theoretical FLOP rate comparable to the GPU cores for direct comparison of core efficiency. This core operates at a very high frequency, has a deep instruction window, and 5-wide issue width as a way to observe instruction-level parallelism (ILP) limitations in the per-thread instruction stream. This core is a traditional superscalar architecture and contains a 4-wide SIMD functional unit, for which we compiled benchmarks with automatic vectorization.

We compare these CPU cores against GPU cores configured similarly to NVIDIA's Fermi GTX 500 series streaming multiprocessor (SM) with up to 48 warps and a total of 1536 threads per core. With 32 SIMD lanes per core and a frequency lower than the CPU cores, this GPU core is capable of 22.4 GFLOP/s for single-precision computations, consistent with the CPU core.

While core count scalability will be of interest, this study aims to understand the different effects of core microarchitecture on memory system behavior, so these baseline systems model fixed core counts for each type. In particular, we compare 4 CPU cores against 4 GPU cores. For multithreaded CPU applications, we execute a single thread per core, while GPU applications are able to concurrently execute up to the full 1536 threads per core.

Using a modified version of McPAT [21] with CACTI 6.5 [25], we estimate the die area required to implement each core in a 22nm technology. From this data, we observe that the GPU core has a slightly larger size of $6.1mm^2$, which would require it to achieve about 16% higher functional unit occupancy than the CPU core in order to achieve the same compute density.

In addition to the results presented here, we tested varying CPU and GPU core counts and a broader range of cache hierarchy and memory organizations while maintaining similar peak FLOP rate controls. In general, the results from these tests showed expected changes in the magnitudes of memory access demands, but they did not appreciably change the relative or qualitative access characteristics. For presentation and analysis simplicity, we chose to limit our results to the above core configurations.

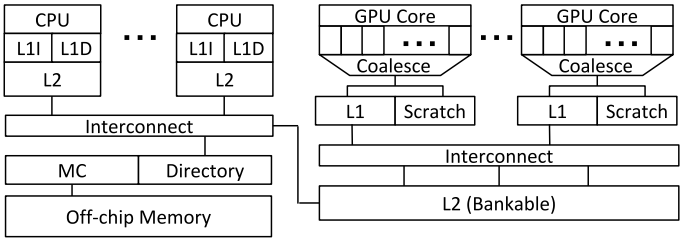

Fig. 1. Heterogeneous CPU-GPU architecture.

Cache Hierarchies: CPU and GPU cores pump memory accesses into the top of the cache hierarchy, where each core is connected to L1 caches. The CPU cache hierarchy includes private, 64kB L1 data and 32kB instruction caches. Each core also has a private L2 cache. To maintain a reasonable comparison of the use of caches between core types, we limit the aggregate L2 cache size to 1MB for both core types. Full coherence is maintained over all CPU caches.

The GPU memory hierarchy includes some key differences compared to the CPU hierarchy. First, each GPU core contains a scratch memory that is explicitly accessible by the programmer. Data is often moved into and out of this memory from the global memory space through the caches. Second, GPU memory requests from threads within a single warp are coalesced into accesses that go to either the scratch or cache memories. Each GPU core has an L1 cache, which is allowed to contain incoherent/stale data. If a line is present in an L1 cache when written to, it is invalidated in the L1 before the write is forwarded to the GPU L2. Finally, GPU cores share a unified L2 cache of 1MB. This GPU L2 cache participates in the coherence protocol with the CPU caches.

Interconnects and Memory: We model two simple interconnects for inter-cache communication. The first connects GPU L1 caches and the GPU L2 cache, and the second connects all L2 caches and the directory. These interconnects faithfully model latency, back-pressure and bandwidth limitations consistent with existing GPU interconnects and multicore CPU crossbars, respectively. In all configurations we test, each L2 cache is capable of driving more requested bandwidth than the peak off-chip bandwidth to ensure that achieved bandwidth to any single core can fully utilize off-chip memory resources.

Finally, we model a shared directory controller that manages the coherence protocol and off-chip accesses through the memory controller. To test a spectra of memory parameters, we use an abstract memory controller design that models first reads, then first-come-first-served memory access scheduling (FR-FCFS) [28], and in appropriate tests, we vary the memory frequency to modulate peak off-chip bandwidth. The memory technology modeled has timing, channel width, and banking parameters similar to GDDR5, but without prefetch buffers, similar to DDR3. Finally, we simulate a spectra of memory bandwidths in certain tests, but we chose 32GB/s as a baseline as it is representative of bandwidth-to-compute ratios in currently available systems.

B. Benchmarks

Selected from the Rodinia benchmark suite [7], we use 9 benchmarks to compare CPU and GPU performance and memory behavior. The Rodinia suite includes applications from image processing, scientific workloads, and numerical algorithms, and all benchmarks are designed to exercise heterogeneous computing systems. The suite includes OpenMP multithreaded versions of the benchmarks that we run in multicore CPU systems, and CUDA versions that run their kernels on the GPU. Table II lists the benchmarks and details about their structure, which we describe in the next section.

In a few cases, we found that the publicly-available versions of Rodinia benchmarks lacked algorithmic mapping and tuning to the target architectures for which they were written. To address this, we refactored and optimized where these transformations were simple to ensure that the behavior we see from the application can be mostly attributed to the underlying core and memory system microarchitecture. Specific optimizations included transposing matrices for reasonable cache strided accesses and GPU coalescing, adding multithreading to unparallelized portions of OpenMP benchmarks, and compiling with optimizations such as loop unrolling and automatic SIMD vectorization. Later, we show that these optimizations result in benchmark performance either near peak FLOPs or limited by some aspect of the memory hierarchy – what we aim to study.

Further, we chose tuning configurations and benchmark input sets to avoid unfair negative performance impacts for either the CPU or GPU versions. We ensured memory access alignment, and avoided excessive bank contention and unoccupied compute units that can arise from parallel portion tail effects [24]. For most applications, the memory access character changes minimally when increasing input set sizes.

Finally, to compare the same portion of benchmark run time when run on the CPU or GPU, we annotate each benchmark to collect simulated system statistics over the portion of the benchmark that includes thread or kernel launches and the parallel portion of work executed by these launches. We denote this portion of the benchmark as the region of interest (ROI). Portions of each benchmark that are common to both CPU and GPU versions, including data setup for the parallel portion of the benchmark and clean-up, are not included in the ROI.

C. gem5-gpu Simulator

To evaluate various heterogeneous CPU-GPU processor designs, we use the gem5-gpu simulator [27], which integrates the GPU core model from GPGPU-Sim [2] into the gem5 simulator [5]. gem5-gpu uses gem5's Ruby memory hierarchy to model various cache protocols with support for coherence and shared components between the CPU and GPU cores. For our CUDA tests, the GPGPU-Sim cores execute PTX instructions rather than an intermediate instruction set. We have validated that using the gem5-gpu memory hierarchy more accurately models NVIDIA Fermi hardware than stand-alone GPGPU-Sim executing PTX.

III. Application-level Characteristics

Before delving into memory access behavior, this section briefly discusses the algorithmic structure of the Rodinia benchmarks and their mapping to each core type. In the aggregate, application-level characteristics and statistics for CPU and GPU benchmarks tend to be very similar, but we describe a few notable differences.

A. Algorithm Structure

The structure of the algorithms that are mapped to each core architecture drive all of the benchmark behavior that we present. Here, we describe a taxonomy of these algorithm structures and how they are mapped to CPU or GPU hardware. Table II lists the benchmarks considered in our characterization, and columns under "Static Structure" list their algorithm classifications in this taxonomy.

First, we describe two classes of applications that employ iterative data processing. The first class of iterative algorithms are listed as "Iterative" and they stream heap data a number of times proportional to I, the number of iterations ($O(I)$). This structure is generally needed for applications in which either data is transformed over progressive time steps, such as heartwall and hotspot, or separate iterations successively refine a solution in search of an optimum, as in kmeans and strmcluster. These algorithms typically spin off parallel computation once or a constant number of times per iteration.

A second class of iterative algorithms divides heap data into chunks, either based on the number of parallel compute units or based on the portions of data that can be efficiently processed per iteration. This iteration structure can be employed when the data for each computation is local to small portions of overall heap. Wider data dependencies make it difficult to chunk the data to be efficiently processed across separate iterations. These algorithms access heap data roughly a constant number of times, so we denote these applications as having constant-order ($O(c)$) heap reads.

The structure of work per iteration in iterative algorithms can be variable or fixed. If data dependencies may change across iterations, the amount of available parallel work may also change. We list these algorithms as having "Variable" iteration size, while algorithms that have unchanging parallel work over iterations are denoted as "Constant". Varying-work iterative algorithms are typically the hardest to parallelize to leverage data-parallel microarchitecture constructs such as vectorization and coalescing, because the data-level parallelism may not be regular to fit nicely into these constructs.

We distinguish one other iterative-like algorithm structure, "Incremental", from iterative algorithms due to the differences in synchronization. The two classes of iterative algorithms described above commonly employ barrier-like synchronization of worker threads between iteration epochs, since the data touched by each thread can typically be isolated from the data of other threads during parallel regions. In contrast, our modified OpenMP version of nw employs fine-grained locking to communicate small chunks of data between threads. Compared to the GPU version, this structure elides the barrier synchronization to reduce thread launch overheads and the locking reduces per-thread working set size to reduce cache contention and load imbalance.

The final class of algorithms lacks any iterative nature, and instead, can be parallelized with one or a constant number of parallel work phases. When results from one parallel phase are consumed by a subsequent phase, these applications are

TABLE II. RODINIA BENCHMARK APPLICATION CHARACTERISTICS.

	Static Structure				Run Time Characteristics		
Benchmark	Algorithm Structure CPU / GPU	Heap Reads	Iteration Size CPU / GPU	Add'l Structure CPU / GPU	Comp. Op (M)	Mem. Op (M)	Heap (MB)
backprop	Pipeline	$O(c)$	Constant	Reduction	155	34	35.0
bfs	Iterative	$O(c)$	Variable	—	395	39	6.7
heartwall	Iterative	$O(I)$	Constant	—	12,750	1,924	41.4
hotspot	Iterative	$O(I)$	Constant	— / Pyramid	1,474	242	7.0
kmeans	Iterative	$O(I)$	Constant	Reduction	2,485	571	7.5
nw	Incremental / Iterative	$O(c)$	Constant / Variable	—	303	50	81.0
pathfinder	Iterative	$O(c)$	Constant	Pyramid	305	76	36.6
srad	Pipeline	$O(c)$	Constant	—	780	171	96.0
strmcluster	Iterative	$O(I)$	Constant	Reduction	1,074	248	2.8

referred to as "Pipeline" parallel. Backprop and srad are both pipeline parallel and access heap data a constant number of times, typically proportional to the number of pipeline stages. Note that both these benchmarks employ barrier-like synchronization between pipeline stages, but other pipeline-parallel applications could use finer-grained synchronization within pipeline stages.

We note two additional algorithm structures that are employed during or between parallel work epochs: data pyramiding and reduction operations. These are common structures and tend to have substantial effects on memory access behavior. Data pyramiding is a common technique employed in image processing (iterative) algorithms that access a small neighborhood of data for each computation [29]. By gathering a slightly larger neighborhood of data, a parallel thread is able to compute the result of multiple outer loop iterations, while avoiding the need to stream that data neighborhood multiple times. The "pyramid" term refers to the way that the data neighborhood grows as the number of merged outer loop iterations grows.

Reduction operations are employed in cases where a large set of data, typically produced by multiple threads, must be inspected to find one or a small number of results. Common cases include reduction sums and searches for extrema. These operations often come with extra computation and memory overhead to store intermediate values as the number of parallel threads is increased. In general, as more threads participate in a reduction operation, there is more overhead to synchronizing the handling of intermediately reduced data. While many efficient constructs do exist, for GPU applications, we will see that reduction operations are tricky to coordinate between CPU and GPU cores, and can lead to large data communication and run time overheads.

B. Dynamic Run Time Characteristics

Table II also includes region of interest dynamic run time statistics for each benchmark executed with a single thread on a CPU core. These statistics include the count of dynamic compute and memory operations, and the size of the heap data accessed during the region of interest. In general, these statistics are often similar across core types, so we only briefly touch on the similarities and pay more attention to specific cases that cause the stats to vary across the core types.

For CPU compute operations, we count integer and floating-point micro-instructions that occupy an execution unit rather than the x86 macro-instructions they comprise, and we compare these counts to PTX instructions executed by GPU cores. Memory operations under x86 count micro-instructions that involve a cache request, and we compare these counts to PTX memory instructions. PTX does not provide a memory-indirect addressing mode, so the count of memory instructions is equal to the number of cache requests. Note that GPU request coalescing reduces the actual number of GPU cache accesses and we describe this in the next section.

The benchmarks, nw, srad, and strmcluster show compute op count differences of at most 10% across the core types. The benchmarks, heartwall, kmeans, nw, and pathfinder show memory op count differences of at most 15%. In the geometric mean across all benchmarks, we find that the number of compute and memory ops varies by at most 38% and memory footprint varies by less than 6% across the system configurations. The large geometric mean differences for compute and memory ops indicates that on a per-benchmark basis, large differences can arise. We find this is due to use of registers, number of threads, and coordinating work between core types.

A fairly common factor in compute and memory op count differences between system configurations is due to register handling. For x86 CPU applications, the small architected register set (16) can cause register spilling to the stack and recomputation of previously computed values. In contrast, GPU cores have some flexibility in register use due to their core multithreading. By running fewer GPU threads per core and late binding register specifiers to physical registers, there is more flexibility for each thread to access more registers, which can avoid spilling and recomputation.

The CPU versions of bfs, kmeans, and strmcluster require numerous calls to small functions or pointer-chasing in their inner loops, which results in an elevated number of address calculations, memory requests, and register spills. This causes up to 1.33× more cache requests compared to their GPU counterparts. Also on the CPU, heartwall is written with a large, flat function that heavily overloads architected register specifiers, causing the CPU cores to execute almost twice as many compute ops to recompute values. Both of these differences result in a performance disadvantage for CPU cores.

A less common factor in compute and memory op count differences is the width of multithreading employed by an application. Specifically, in cases such as backprop, where each thread must incur some fixed number of ops for setup before performing a share of computation, linearly increasing the number of threads also linearly increases the total number of these fixed ops that must be performed.

For most Rodinia applications, care was taken to ensure that thread counts are kept small to avoid these fixed ops, and the extra threading op counts are often within 8% across the different platforms. However, compared to the single-threaded

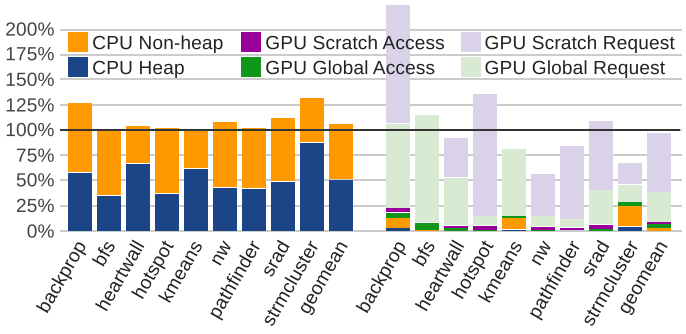

Fig. 2. Breakdown of memory requests and accesses normalized to single thread CPU execution.

CPU version, backprop employs numerous GPU threads, which increases op counts by approximately 1.5×. Strmcluster's numerous CPU thread launches cause approximately 1.7× extra ops. Often, these extra ops can be hidden off the application critical path, because they are executed in parallel by execution units that may have been mostly idle with fewer threads.

Finally, for benchmarks that coordinate reduction operations between CPU and GPU cores, there is elevated communication and synchronization overhead compared to running the application only on CPU cores. In general, the late stages of reduction operations tend to perform poorly on GPUs, because the limited data-level parallelism causes poor occupancy of the GPU's numerous thread contexts. This limitation suggests that the late stages of the reductions should be performed by a small number of threads, perhaps on CPU cores. However, by transferring control to CPU cores, the GPU versions of backprop, kmeans, and strmcluster incur up to 1.5× as many reduction memory ops to move the intermediately-reduced data from the GPU to the CPU. Reduction performance ends up being a primary factor in performance for these applications.

IV. Memory Access Characteristics

In this section, we characterize the memory behavior effects of the microarchitectural differences between CPU and GPU cores and cache hierarchies. While we noted in the last section that CPU and GPU applications show many similarities, the different core types expose and leverage MLP in very different ways; CPU cores use a small set of deep per-thread instruction windows, and high-frequency pipelines and caches to expose parallel memory accesses. In contrast, GPU cores expose parallel memory accesses by executing 100s–1000s more threads at lower frequencies, and threads are grouped for smaller per-thread instruction windows and memory request coalescing.

The results here reveal the primary differences in CPU and GPU core memory access. Specifically, while CPU cores rely heavily on L1 caches to capture locality, GPU cores capture most locality with coalescing and lessen the L1 cache responsibilities by providing scratch memory. Beyond the L1 caches, the memory systems tend to capture very similar locality. Further, we see that different core threading and cache filtering result in extreme differences in instantaneous memory access rates; CPU caches tend to filter accesses down to regular intervals, while GPU cores tend to issue bursts of accesses.

A. Access Counts

Despite large CPU and GPU core threading differences, two core-microarchitecture design characteristics account for the majority of the difference between CPU and GPU cache hierarchy access counts and sizes: scratch memory and address coalescing. We describe their effects here.

Figure 2 plots a breakdown of cache access counts for the CPU version (left group of bars) and the GPU version (right group of bars), normalized to a single-threaded execution of each benchmark on a CPU core. We see that the multithread CPU version has small additional memory accesses compared to a single core CPU due to extra threads. Next, for the GPU, the plot separates memory requests (the absolute number of load/store instructions on the GPU) from cache hierarchy accesses, each of which may be a coalesced set of memory requests. The ghosted portions of the GPU bars represent the number of requests saved by coalescing down to a small number of memory accesses.

Scratch memory: GPU cores provide scratch memory, which can function as local storage for groups of threads to expand the space of local storage with register-like accessibility. In CUDA benchmarks that use the GPU scratch memory, kernels are typically organized into three stages: (1) read a small portion of data from global memory into the scratch memory, (2) compute on the data in scratch memory, and (3) write results back to global memory. Since numerous threads are executing these same stages – often in lock-step – stages 1 and 3 appear as stream operations to the memory hierarchy, and we describe this behavior more deeply below. Besides register files, CPU cores do not have an analogous scratch-like memory, so instead, they typically spill local variables to the L1 caches temporarily.

In the common case, CPU stack accesses and GPU scratch memory accesses account for similar portions of memory requests. These similarities are reflected in the proportion of heap versus non-heap (stack or scratch) memory requests depicted in Figure 2. Most GPU applications employ scratch memory for local data handling, and these requests account for 25–80% of all requests. Moving these requests to scratch memory instead of caches, as on the CPU, reduces the number of requests to the cache hierarchy by a geometric mean of 3.9× before coalescing. Similarly on the CPU, local memory requests account for upwards of 50% of total requests. We also note that CPU stack accesses generally hit in a small number of cache lines in the L1 caches. The high rate of access to CPU stack memory and GPU scratch memory suggests that even a small scratch memory capacity can be useful to mitigate cache hierarchy memory accesses.

A couple benchmarks, bfs and kmeans, do not use scratch memory and opt instead to access all data from the global memory space through the caches. In both cases, global data and access patterns are such that it would be complex to use scratch memory. We will see later that both of these applications are more sensitive to cache and memory capabilities as a result.

Request coalescing: Second, in contrast to CPUs, which execute up to tens of separate concurrent threads, GPUs maintain contexts for thousands of threads through hierarchical grouping. These groups of threads can exploit spatial locality by coalescing requests from separate threads into a small number of cache accesses when the requests are to neighboring addresses.

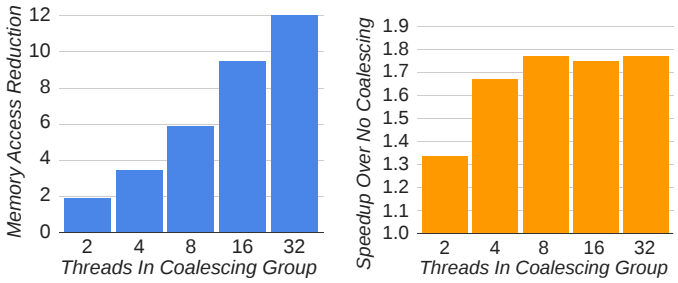

Fig. 3. Coalescing: memory access reductions and resulting ROI speedup, normalized to no coalescing.

To understand the impacts of request coalescing, we ran tests that vary the coalescing degree from no GPU coalescing to full 32-thread group coalescing. Figure 3 shows the geometric mean reduction in number of cache accesses and the ROI speedup for varying coalescing degree normalized to the case with no coalescing. In the common case, full 32-thread coalescing reduces the number of global memory accesses by 10–16× as the GPU is frequently able to coalesce 16 or 32 integer or floating-point requests to a single cache line access.

This data also shows that most of the performance gains come from coalescing across sets of 4–8 threads, as further gains are limited to less than 3% run time in the common case. With wider coalescing, any existing memory bottlenecks would manifest at cache hierarchy levels below the core-L1 interface. Despite the small potential for performance gains, coalescing beyond 8 threads continues to decrease the total number of memory accesses (2× in the geometric mean), which may result in a decrease in cache access power/energy.

Since GPU request coalescing behaves similarly to CPU single-instruction, multiple-data (SIMD) vectorization, we also ran tests to find the effect of SIMD vectorization on memory access counts. We use the gcc compiler automatic SSE 4 vectorization, which allows the CPU to load, operate on, and store data on up to 4-wide integer or single-precision floating point vectors with single instructions, and has a similar effect to coalescing in terms of the width of memory accesses to the cache hierarchy. Gcc is able to vectorize ROI portions of four benchmarks from our set: backprop, hotspot, srad and strmcluster. For these four benchmarks, vectorization reduces the total number of memory accesses by 1.32–1.69× (1.44× geometric mean), and that most of the eliminated accesses are to heap data. Compared to GPU 4-thread coalescing, automatically vectorizable CPU code sees more limited reduction in memory access counts.

Overall, GPU scratch memory and request coalescing reduce the number of global memory accesses by 18–100× compared to CPU applications (27× in the geometric mean). Compared to CPU cores, this reduction alleviates pressure on caches, which in turn allows GPU cores to operate at lower frequencies while still serving data to threads at rates comparable to or greater than CPU cores.

B. Spatial Locality

As we observe memory accesses flowing through the cache hierarchy, we see that L1 caches play different locality filtering roles for the CPU and GPU sides of the architecture. Specifically, in contrast to mostly-scalar CPU cache accesses which must hit in cache to provide strong performance, GPU request coalescing dramatically cuts the accesses to each cache line, which in turn alleviates L1 cache pressure and can expose more MLP to lower levels of the hierarchy.

TABLE III. MEMORY ACCESS LOCALITY METRICS BY CORE TYPE.

	OpenMP:		Access	CUDA:		Access
Benchmark	% L1 Hits	% L2 Hits	Per Line	% L1 Hits	% L2 Hits	Per Line
backprop	95.6	68.1	42.2	39.6	75.7	5.6
bfs	95.5	18.1	4.0	37.6	62.2	3.7
heartwall	99.7	89.3	273	87.7	93.2	19.1
hotspot	98.9	16.2	57.4	37.1	65.3	4.5
kmeans	99.4	0.5	132	44.1	18.9	4.9
nw	99.0	0.6	74.0	0.5	74.6	1.3
pathfinder	98.8	45.8	35.6	46.6	17.0	2.3
srad	97.7	61.0	54.0	19.7	57.3	2.5
strmcluster	97.4	17.0	5.3	25.5	66.5	2.8

For these data-parallel applications, CPU threads have extremely high spatial locality, typically striding through all elements in a heap cache line in subsequent algorithm loop iterations. These access patterns, which also include accesses to stack/local memory that is persistent over many loop iterations, result in high L1 cache hit rates that even exceed those expected by simple strided read memory access. Table III lists these hit rates and the average number of accesses per heap cache line per algorithm pipeline stage.

CPU vectorization and GPU coalescing are designed to capture address spatial locality before memory requests are sent to the caches. Thus, these techniques cause a reduction in the available spatial locality to caches by a factor equal to the effective access width (i.e. up to 1.69× for 4-wide vectorization and more than 14× for 32-wide coalescing in common cases).

To get a sense for the remaining spatial locality in the GPU cache access stream, we observe counts of the number of accesses to each unique global memory cache line during GPU kernels, and we find that lines are typically accessed between 2 and 5 times. Note that most GPU kernels move data from global memory into scratch for local handling, so the small number of remaining spatially local accesses is likely due to separate thread groups accessing the same data rather than thread groups being unable to fully coalesce accesses. In either case, there is little spatial locality left to exploit within each kernel without increasing cache line sizes, and GPU thread group and coalescing widths.

Since vectorization and coalescing cut down the number of spatially local accesses to each cache line, they have the effect of exposing wider access parallelism below the L1 caches. This is a subtle effect in lock-up free caches: Fewer accesses to each cache line reduces the number of accesses that may occupy miss-status handling registers (MSHRs) queued for a small set of outstanding accesses to lower levels of the cache hierarchy. We find that GPU 32-wide coalescing increases the number of concurrent memory accesses to the L2 cache by 1.3–3× over uncoalesced GPU memory accesses. Hence, vectorization and coalescing are both fundamental means for better exposing MLP to lower levels of the memory hierarchy by decreasing MSHR pressure caused by accesses that queue for a small set of lines.

C. Temporal Locality

The prior subsection described how core microarchitecture and L1 caches capture a significant portion of access spatial

locality for both CPU and GPU applications. This suggests that little spatial locality is left for L2 caches to capture. Instead, they serve to extract access temporal locality from separate data sharers.

In all OpenMP benchmarks, the CPU L1 hit rate is above 95%, as each heap cache line is accessed numerous times, and many local variables are accessed some number of times across loop iterations. By comparing the L1 cache hit rates with the common number of memory accesses per cache line, we can see that in all benchmarks, the L1 caches must be capturing nearly all of the intrathread spatially local accesses to each line. For example, strided accesses to 32 consecutive data elements in each cache line would result in a 31/32 = 96.9% hit rate in the absence of intervening accesses. This suggests that the per-thread working set of most benchmarks fits in the CPU 64kB L1 cache. Given this observation, this leaves the L2 caches mostly responsible for capturing temporally local accesses to data shared across cores rather than temporally or spatially local accesses to data previously evicted from the L1 caches due to limited capacity.

Since GPUs typically only access cache lines a small number of times during a kernel compared to CPU cores, there is diminished importance to ensuring temporally local accesses to cache lines. Observing the GPU L1 cache hit rates and common access counts per line, we note that the L1 caches capture fewer than half of the multiple accesses to each cache line in the common case, and more accesses go to the GPU L2 cache than in CPU applications. To establish whether this is a result of contention for GPU L1 cache capacity or data sharing across GPU cores, we ran tests that vary the GPU L1 cache capacity up to 256kB, and we found that L1 hit rates improve by at most 5% with the extra capacity. This indicates that instead of competing for L1 capacity, GPU threads from separate cores are generating most of the temporally local accesses to single cache lines, similar to the CPU L2.

Based on the above observations, we find that CPU and GPU L1 caches have very different importance, though their filtering roles are similar. In the aggregate for data-parallel workloads, CPU L1 caches have many responsibilities; they must be designed to capture both the spatial locality for heap data accesses and the temporal locality of stack accesses. Fortunately for data-parallel workloads, these responsibilities rarely conflict given sufficient L1 capacity, so CPU L1s are quite effective and important for capturing locality.

For GPU applications, register and scratch memory can shift local variable accesses away from the caches, which eliminates the L1 responsibility for capturing temporally local stack requests. Further, GPU coalescing greatly reduces the importance of spatial locality across separate heap accesses, so the L1 caches are mostly responsible for capturing the small number of temporally local accesses from separate GPU threads on the same core, diminishing the overall responsibility of the GPU L1s compared to CPU L1s.

In contrast to L1 caches, L2 caches play a similar role for both CPU and GPU cores. For both core types, the majority of spatial and temporal request locality is captured by components at higher levels of the memory hierarchy, which usually leaves the L2 caches responsible for capturing access locality to data that may be shared among separate cores. We did not find any circumstances in which the L2s function to capture significant temporal request locality from L1 capacity spills.

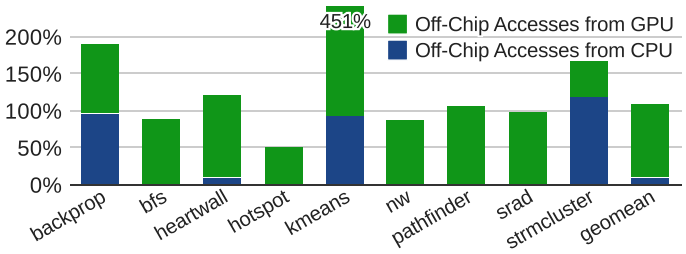

Fig. 4. Number of off-chip memory accesses, normalized to CPU version.

D. Number of Off-Chip Accesses

When observing memory request locality, we noted that L2 caches play a similar role in filtering memory requests for both CPU and GPU cores. This suggests that there may be strong similarities between the off-chip memory accesses for both CPU and GPU applications. Here, we observe off-chip access counts in support of this hypothesis. Figure 4 plots the number of off-chip memory accesses for all GPU benchmarks normalized to the multithreaded CPU version.

We start by noting some important similarities between CPU and GPU applications. In particular, for applications that execute similar code per output data element (backprop, bfs, heartwall, pathfinder, srad and strmcluster), the number of off-chip memory accesses from the GPU is nearly identical to the analogous portions of the CPU version. In addition, we note that the GPU version of hotspot uses a pyramid iterative algorithm that completes two timesteps per GPU kernel launch, which cuts the number of times that the GPU streams data on-chip by a factor of precisely two compared to the CPU version. While a non-trivial transformation, the CPU version could also implement pyramiding to gain potential benefits of reduced off-chip data access.

The major differences between CPU and GPU off-chip access counts typically arise due to the overheads of off-loading computation to the GPU cores. The CPU-GPU coordinated reduction operations in backprop, kmeans, and strmcluster require that the CPU stream GPU-generated intermediate data, which is too large to fit in on-chip caches. In each case, these accesses account for roughly as many off-chip accesses as the OpenMP versions of the applications. As we will see later, this extra data streaming, rather than elevated op counts, accounts for the performance overhead in these applications.

Second, we note that the way that the kmeans algorithm is mapped to the GPU, it does not take advantage of local memory in its inner loop as does the CPU version. This results in the GPU streaming all data points once for each of the k centers that are being considered in a single iteration. The input set we consider in this work has $k = 5$, and indeed, we see that kmeans on the GPU must stream data on-chip nearly 5 times more than the CPU version. While the transformation would be non-trivial, the CUDA version could be modified to store the same local variables as the CPU version to eliminate these extra off-chip accesses.

Finally, we find that memory footprint can contribute to second order effects on off-chip accesses. In particular, heartwall and nw double-buffer data in their GPU and CPU versions,

TABLE IV. ROI REQUESTED OFF-CHIP BANDWIDTH (GB/S).

	CPU cores			GPU cores		
	Avg	Stdev	Max	Avg	Stdev	Max
backprop	7.0	4.1	20.3	11.1	10.4	83.9
bfs	7.0	6.8	30.0	13.8	12.0	94.0
heartwall	0.3	0.9	20.8	1.1	4.8	77.3
hotspot	3.6	1.8	16.3	2.9	4.0	35.6
kmeans	1.0	0.6	11.7	23.8	6.0	80.8
nw	4.3	1.7	16.8	7.5	7.8	82.4
pathfinder	5.0	1.6	16.3	6.2	4.2	79.3
srad	4.9	3.0	20.3	10.1	8.1	87.9
strmcluster	5.2	3.0	16.8	16.5	13.1	114.9

respectively. These extra buffers result in up to 10% differences in memory footprint for their respective cores, and we find that access counts are analogously affected.

We also note that we find little difference in the volume of off-chip memory accesses ($< 1\%$) for GPU applications as we vary the coalescing degree. This indicates that, like their CPU counterparts, the GPU cache hierarchy is able to capture the majority of spatially local accesses to heap cache lines when coalescing is unable to, albeit with possible overheads in performance or power.

E. Bandwidth Demands

While we have noted that CPU and GPU L2 caches play similar roles in capturing memory access locality, here, we demonstrate that CPU and GPU cores expose very different MLP over time that results in a substantial difference in their requested memory bandwidth rates.

To evaluate differences in off-chip memory bandwidth demand, we simulated both benchmark versions with a 32GB/s off-chip memory interface and collected memory access inter-arrival times at the memory controller. From these interarrivals, we calculated instantaneous requested bandwidth over time intervals of 500 memory controller cycles and used them to estimate the average, standard deviation, and maximum requested bandwidth as listed in Table IV.

This data shows that GPU cores nearly always demand greater average bandwidth and greater variance over time than the CPU. Compared to the CPU versions, all GPU applications here show a greater frequency of high instantaneous requested bandwidth. Further, the GPU's instantaneous requested bandwidth regularly exceeds the peak theoretical limit of the off-chip interface by more than 2.5×, while CPU cores only request up to the peak.

To tie requested bandwidth back to application-level characteristics, we observe requested bandwidth over time for a representative portion of the nw benchmark. Figure 5 plots the requested off-chip bandwidth from both core types for this region. The blue line shows the aggregate requested bandwidth from the CPU cores over time. The behavior in this plot is common to most CPU applications, namely CPU cache accesses miss in fairly regular intervals, which cause regular accesses through time and consistent requested bandwidth through each phase of the benchmark.

By contrast, GPU cores expose very bursty memory accesses, frequently exceeding the peak available bandwidth. The green line plots the GPU requested bandwidth with annotations for the start/end of GPU kernels ("Kernel Launch"), and it depicts the common GPU kernel stages we described previously.

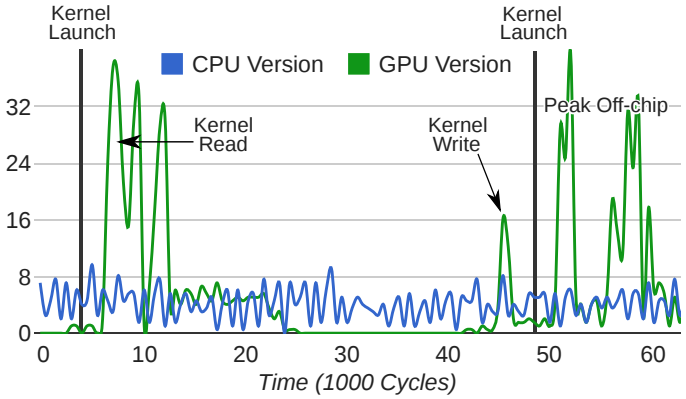

Fig. 5. NW requested off-chip bandwidth (GB/s) over time.

Shortly after a kernel launch, numerous thread blocks issue parallel memory accesses that begin reading data from the global memory space into the core's scratch memory. These reads are issued en masse, causing the cache hierarchy and memory system to fill with buffered accesses. After reading data, the GPU often accesses data in scratch - a time period with few or no global accesses from each thread block - and then the data is written back to global memory as a burst ("Kernel Write").

Burst access behavior is common to all GPU benchmarks we tested, and the character tends to be largely similar to nw. We chose to plot nw because it clearly demonstrates the read-compute-write character of GPU thread blocks. However, nw has low GPU thread occupancy, which limits its ability to further push bandwidth limitations while thread blocks are computing on data. Many benchmarks execute more concurrent thread blocks/groups or have lower ops-to-byte ratios, which often result in larger or more frequent access bursts. When near bandwidth saturation, the distinction of kernel read/write bursts can blur as buffers fill and dependent memory accesses modulate the issue of further outstanding accesses.

The key takeaway here is that GPU burst access behavior results from the way that GPUs group and launch threads. Specifically, at the beginning of a kernel, all capable thread block contexts begin executing at roughly the same time, so this can cause very large bursts of independent accesses. Following this initial burst, smaller but still significant access bursts occur each time a new thread block begins executing or when thread groups pass synchronization events. By contrast, CPU cache access filtering tends to modulate the core's ability to issue nearly as many parallel accesses to off-chip memory.

V. APPLICATION PERFORMANCE IMPLICATIONS

The last section showed CPU and GPU memory access similarities and differences, and we want to understand their application-level effects. Here, we show that the majority of performance differences between CPU and GPU versions is, in fact, directly attributable to the memory access behavior differences. Specifically, the CPU versions struggle to keep up with GPU versions due to MLP limitations which cause memory stalling. This memory stalling results in elevated memory access latency sensitivity for CPU cores, while GPU cores are better able to leverage available bandwidth.

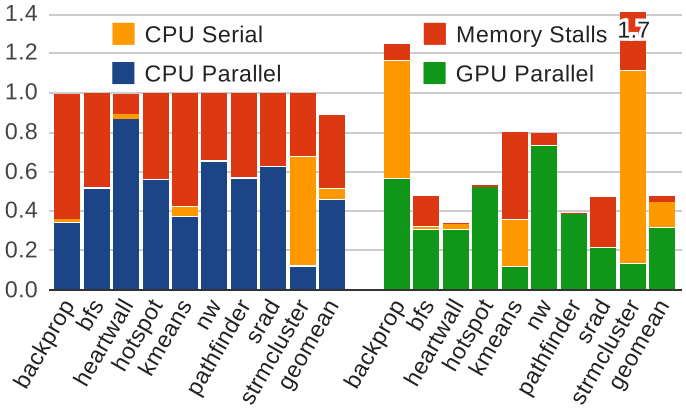

Fig. 6. ROI run time normalized CPU version.

A. *Performance Depends on Memory Stalling*

First, we show that the primary difference in performance between CPU and GPU applications is a result of memory stalling. Figure 6 plots the ROI run times of applications run on CPU cores (left cluster of bars) and GPU cores (right cluster) normalized to the CPU's run time. The figure indicates that CPU versions tend to struggle to keep up with the GPU versions despite comparable peak FLOP rates.

To establish that memory access is the substantial portion of the difference in run time between CPU and GPU versions, we ran tests which cut the memory hierarchy latency to nearly zero cycles as a way to estimate the portion of run time attributable to memory hierarchy performance. Cache and memory bank conflicts were minimal, so we describe that the resulting stalls come largely from each core's ability to expose MLP. The run time gains shifting from the realistic memory hierarchy design to the optimal hierarchy are reflected in the "Memory Stalls" portion of each run time bar.

These memory stall estimates indicate that in the common case, CPU applications suffer memory stalls for 30–50% of run time. If these stalls could be removed from the CPU version run times, four CPU applications - bfs, hotspot, nw, and srad - would come within 25% of their respective GPU versions, and the geometric mean across all applications would be within 8%. Remaining, second-order performance differences are attributable to data communication overhead (backprop, kmeans, streamcluster), elevated CPU compute ops (heartwall), and control-flow ILP limitations (pathfinder).

B. *Latency Sensitivity*

Ultimately, memory request dependencies in CPU thread instruction streams regulate their ability to issue many concurrent outstanding memory requests below the L1 caches and to hide memory access latency. In contrast, GPU cores leverage their deep multithreading to expose wide MLP and issue access bursts to lower levels of the cache hierarchy. It is a common belief that this difference allows GPU cores to hide very long memory access latency.

We confirm this belief by running simulations with additional no-load, off-chip access latencies of 200, 400 and 600 memory cycles, which bump the baseline off-chip access latency roughly 3–7×. Figure 7 presents, for both versions, the geometric mean ROI slowdowns over all benchmarks normalized to the baseline memory. The CPU benchmarks show immense sensitivity (>2×) to additional latency compared to the GPU versions. Even in the GPU benchmarks, the vast majority of latency sensitivity comes from the CPU portion of execution time, while the GPU portions of execution show a maximum slowdown of 5% across all benchmarks.

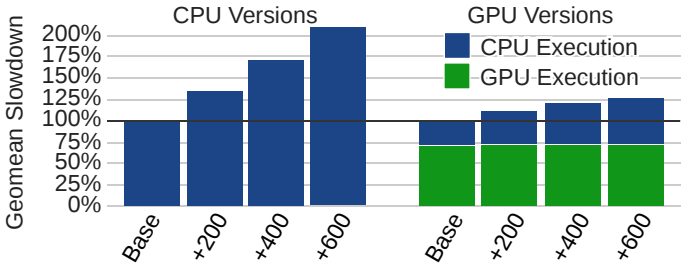

Fig. 7. Geometric mean slowdown over all benchmarks from 200, 400, and 600 additional off-chip memory access cycles.

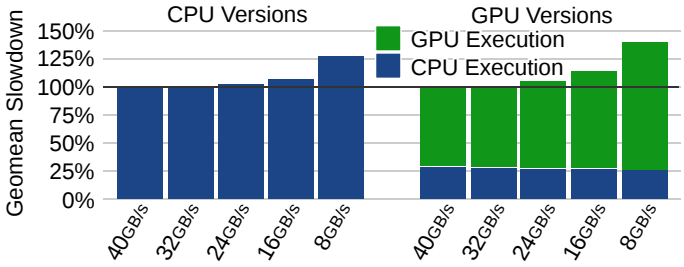

Fig. 8. Geometric mean slowdown over all benchmarks from limiting bandwidth, normalized to 40 GB/s off-chip.

C. *Bandwidth Sensitivity*

Since GPU applications tend to demand greater instantaneous bandwidth from off-chip memory than CPU applications, we would expect that GPU cores should be more sensitive to changes in the peak theoretical bandwidth of the off-chip interface. We quantify the difference in bandwidth sensitivity by running tests that vary off-chip bandwidth from 8 to 40 GB/s. Figure 8 plots, for each system, the geometric mean ROI slowdown over all benchmarks normalized to a system with 40 GB/s peak off-chip bandwidth. These tests confirm that for the complete ROI, GPU applications are 6–10% more sensitive to available memory bandwidth. Further, the GPU parallel portion of these applications is up to 60% more sensitive to bandwidth than the CPU benchmarks.

D. *Discussion*

The performance and memory sensitivity of these applications is directly related to a core's ability to expose MLP, and if we follow CPU and GPU memory access paths, we see where constraints on MLP arise. CPU cores executing fewer threads must extract most MLP from ILP, which poses challenges for parallel accesses below L1 caches, while GPU cores tend to extract greater MLP across a large set of concurrent grouped threads, and to spread memory accesses over a larger instantaneous set of cache lines.

Given the latency sensitivity of CPU cores, there is performance incentive for programmers to aim for high L1 cache hit rates as we see with these applications. These high hit rates typically result from tight code loops and strided memory access, which result in many sequential cache hits to each cache line. Unfortunately, these program structures limit the core's

ability to expose MLP to lower levels of the cache hierarchy, because L1 cache spatial access locality triggers infrequent cache misses to cause accesses to lower levels. As our tests show, this is even the case with the aggressive out-of-order CPU cores, which are capable of continuing execution well beyond waiting cache misses.

By restructuring loop iterations as separate threads and gathering spatial locality across threads, GPU applications tend to avoid the sequential accesses to single cache lines. GPU request coalescing and the use of scratch memory substantially reduce the number of memory accesses that go to the caches, and we described how this reduced number of accesses can increase MLP. Specifically, fewer, less-local accesses reduce pressure on MSHR queuing and increase the number of concurrent parallel memory accesses that can proceed to lower levels of the memory hierarchy.

There are a few options for multicore CPUs to mitigate the effects of low-exposed MLP. First, programmers can parallelize the memory access portion of instruction streams by managing multiple strided access streams in each loop iteration. This tends to be difficult, so other hardware techniques have been developed. Simultaneous multithreading gives the perspective that there are more CPU threads, and indeed, our test show that adding threads can help hide some of the memory stalls. However, this elevates contention for the sequential access to L1 caches. CPU SIMD vectorization can mitigate the number of L1 cache accesses, but does not reduce the number of accesses as dramatically as GPU request coalescing. Finally, CPU cache access prefetching can eliminate much of the latency spent waiting for regular or strided memory access. However, we expect that applications, such as bfs, with irregular memory access will still tend to perform better on GPU cores by being able to issue many parallel accesses to hide latency.

VI. Key Implications

We expect that most of the challenges and opportunities in heterogeneous system design will arise in shared components that will need to balance GPU bursty access behavior against CPU access latency sensitivity. It is likely that applications will find it useful to share data between CPU and GPU cores, and this could mean sharing caches, on-chip interconnect and off-chip memory.

A. Cache Hierarchy Sharing

Trying to share caches between CPU and GPU cores will likely require techniques to ensure latency-critical CPU data remains on-chip as appropriate. GPU cache filtering behavior and bursty memory access results in high volumes and high instantaneous rates of data invalidations and writebacks. If shared caches give equal capacity priority to CPU and GPU data, it is likely that CPU data will be turned over very quickly and erratically when the GPU is operating. Assuming there are applications that could benefit from sharing caches, we expect that new techniques will look to more advanced partitioning schemes, capacity prioritization, and replacement policies to address this challenge.

B. Interconnect and Off-chip Memory Scheduling

Sharing interconnect and off-chip memory among CPU and GPU cores will require quality-of-service (QoS) techniques to appropriately balance bandwidth and prioritize memory accesses. Many prior QoS techniques (e.g. [9, 18]) consider longer-run average statistics as proxies to approximate the application-level performance impact of bandwidth allocation options or delaying memory accesses to prioritize others. However, given the substantial differences in CPU and GPU core architectures and instantaneous memory bandwidth demands, these existing proxies may be insufficient to adequately predict and compare CPU and GPU application performance impacts. Further, most prior techniques are applied in the context of symmetric multicore processors in which there is an assumption that even bandwidth allocation is likely to impact application performance evenly. Our results show that this assumption is not likely to hold up in the context of general-purpose heterogeneous processors, because GPUs tend to be more sensitive to effective bandwidth, while CPUs tend to be more sensitive to access latency. Prior work does investigate scheduling techniques to meet real-time graphics processing constraints in heterogeneous processors [1, 15], but it is likely that estimating appropriate bandwidth balance and finer-grained access prioritization for general-purpose heterogeneous processors will be a broad area of future work.

C. Emerging Memory Technologies

In addition to the challenges posed above, we see opportunity to leverage emerging memory technologies to improve memory latency, bandwidth, and power efficiency. Stacked and on-package DRAM options promise small latency and power improvements, and are likely to provide bandwidth comparable to existing discrete GPUs [22]. We estimate that these improvements will increase CPU benchmark performance by upwards of 5% and GPU performance by as much as 15%.

A shortcoming of available stacked and on-package DRAM options is their small capacity, which is limited by area and thermal constraints. To effectively use these memories, it will be important to store latency- and bandwidth-critical data in them. System designers will need to consider methods for allowing data to be mapped and migrated between memories, while programmers may desire means to intelligently control data placement.

VII. Related Work

To the best of our knowledge, this work is the first to present a quantitative characterization comparing the application and memory system behavior of applications mapped to both CPU and GPU cores with the express aim of illuminating their microarchitectural similarities and differences. We perform this analysis for data-parallel applications.

Memory System Characterizations: A wide range of papers have characterized memory system behavior for parallel applications [4, 30], cloud/server applications [3, 23], GPU applications [6, 16], and mobile/embedded applications [8, 11, 12]. While these prior studies characterized the applications on a single type of platform, our work examines different implementations of the same applications for two different core types with the express goal of understanding the differences in architectural mapping.

Performance Comparisons: Another class of related work seeks to compare the performance of applications implemented for different architectures or using different parallelization

techniques [10, 17, 19]. Of particular significance, Lee et al. compare the performance of a set of applications parallelized to run on either CPU or GPU hardware [20]. That work focuses on the application tuning required to push hard performance limitations: peak FLOP rates and bandwidth limitations. Our work echoes many of the findings of [20], but goes beyond by using tightly controlled simulations to illuminate the precise memory system microarchitecture behavior differences that can cause performance differences.

VIII. Conclusion

As heterogeneous cores become more tightly integrated onto the same die and share the same system resources, understanding the memory system requirements of the cores becomes more critical. In this paper, we presented the first detailed analysis of memory system behavior and effects for applications mapped to both CPU and GPU cores.

Our results show that while the applications are designed with similar algorithmic structures, their mapping to different core types can result in dramatically different memory system characteristics. Specifically, GPU coalescing and scratch memory greatly reduce the importance of L1 caches compared to CPU L1s, which must capture immense spatial locality to ensure performance. GPUs, with their deep multithreading, more readily expose wide bursts of parallel memory accesses to the off-chip interface, which results in greater sensitivity to bandwidth and diminished sensitivity to memory access latency. These memory behaviors are the primary factor in their performance differences.

References

[1] 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 *39th International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 416–427.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 3–14.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[5] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, pp. 1–7, August 2011.

[6] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IEEE International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[8] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "MEVBench: A Mobile Computer Vision Benchmarking Suite," in *IEEE International Symposium on Workload Characterization (IISWC)*, November 2011, pp. 91–102.

[9] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2011, pp. 362–373.

[10] C. Gregg and K. Hazelwood, "Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011, pp. 134–144.

[11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE International Workshop on Workload Characterization (WWC)*, December 2001, pp. 3–14.

[12] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *IEEE International Symposium on Workload Characterization (IISWC)*, November 2011, pp. 81–90.

[13] "HSA Foundation Presented Deeper Detail on HSA and HSAIL," HotChips, August 2013.

[14] "OpenCL Programmability on 4th Generation Intel Core Processors," http://software.intel.com/sites/billboard/article/opencl-programmability-4th-generation-intel-core-processors?wapkw=gpu%20opencl, June 2013.

[15] M. K. Jeong, C. Sudanthi, N. Paver, and M. Erez, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *2012 Design Automation Conference (DAC)*, June 2012.

[16] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-Fetched Caches in GPUs," in *26th ACM International Conference on Supercomputing (ICS)*, June 2012, pp. 15–24.

[17] S. Karlsson and M. Brorsson, "A Comparative Characterization of Communication Patterns in Applications Using MPI and Shared Memory on an IBM SP2," in *Network-Based Parallel Computing Communication, Architecture, and Applications*, ser. Lecture Notes in Computer Science, D. Panda and C. Stunkel, Eds. Springer Berlin Heidelberg, 1998, vol. 1362, pp. 189–201.

[18] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 65–76.

[19] G. Krawezik, "Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors," in *15th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 2003, pp. 118–127.

[20] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *37th International Symposium on Computer Architecture (ISCA)*, June 2010, pp. 451–460.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *International Symposium on Microarchitecture (MICRO)*, December 2009, pp. 469–480.

[22] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 453–464.

[23] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers Via Sensible Co-locations," in *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, June 2011, pp. 248–259.

[24] P. Micikevicius, "GPU Performance Analysis and Optimization," in *GPU Technology Conference*, May 2012.

[25] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A Tool to Understand Large Caches," University of Utah and Hewlett Packard Laboratories, Tech. Rep. HPL-2009-85, 2009.

[26] "NVIDIA Brings Kepler, Worlds Most Advanced Graphics Architecture, to Mobile Devices," http://blogs.nvidia.com/blog/2013/07/24/kepler-to-mobile/, July 2013.

[27] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *Computer Architecture Letters*, vol. 13, no. 1, January 2014.

[28] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 128–138.

[29] M. Strengert, M. Kraus, and T. Ertl, "Pyramid Methods in GPU-based Image Processing," in *Vision, Modeling, and Visualization*. IOS Press, November 2006, p. 169.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 24–36.