GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors

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*—Emerging heterogeneous CPU-GPU processors have introduced unified memory spaces and cache coherence. CPU and GPU cores will be able to concurrently access the same memories, eliminating memory copy overheads and potentially changing the application-level optimization targets. To date, little is known about how developers may organize new applications to leverage the available, finer-grained communication in these processors. However, understanding potential application optimizations and adaptations is critical for directing heterogeneous processor programming model and architectural development. This paper quantifies opportunities for applications and architectures to evolve to leverage the new capabilities of heterogeneous processors. To identify these opportunities, we ported and simulated a broad set of benchmarks originally developed for discrete GPUs to remove memory copies, and applied analytical models to quantify their application-level pipeline inefficiencies. For existing benchmarks, GPU bulk-synchronous software pipelines result in considerable core and cache utilization inefficiency. For heterogeneous processors, the results indicate increased opportunity for techniques that provide flexible compute and data granularities, and support for efficient producer-consumer data handling and synchronization within caches.**

I. Introduction

Heterogeneous systems are evolving to allow tighter CPU and GPU interaction. Many new systems allow GPUs to access CPU physical memory and both cores to reference memory with the same virtual addresses. This capability can mitigate the complexity of correctly accessing complex data structures used by both CPU and GPU cores [16, 22]. However, since data must still move across the PCIe bus in discrete GPU systems (as shown in Figure 1), programmers must still carefully manage data movement to achieve good performance.

In addition to unified virtual memory, current heterogeneous processors include integrated GPUs (Figure 2), which share common physical memory. These chips are integrating coherent communication fabrics among CPU and GPU cores [4, 23, 35]. Further, tightly-integrated processors even include cache coherence among cores [15, 30], an area of active research [18, 26]. These architectures can mitigate costly memory transfers and allow CPU and GPU cores to perform fine-grained communication and synchronization in cache.

Currently, it is unclear how programmers may try to use these emerging system features. These new processors are in their infancy and application programming interfaces (APIs like OpenCL 2.0 [17]) to use these features are still solidifying. As a result, few current benchmarks exercise the performance capabilities of unified virtual memory architectures, and no publicly-available benchmarks exercise cache coherent heterogeneous processor capabilities.

To better understand the potential evolution of applications and architectures for heterogeneous CPU-GPU processors, this paper compares a broad set of existing, publicly available GPU computing benchmarks against ported versions that remove memory copies. We then analytically quantify the compute and cache inefficiencies to identify likely optimization targets. Heterogeneous processors are likely to benefit from three major optimizations:

- Reducing the temporal distance between data producer and consumer tasks using finer-grained communication and concurrently executing compute stages.
- Identifying task data-independence and leveraging mechanisms to migrate independent work to under-utilized cores.
- Detecting memory access contention and appropriately modulating access to increase cache efficiency.

Overall, this analysis shows that heterogeneous processors offer greater compute and cache efficiency opportunities compared to discrete GPU systems. While removing copy overheads from current benchmarks results in modest performance improvement, still half of all memory accesses result from cache contention caused by residual GPU kernel-granularity synchronization. Most benchmarks with high cache inefficiency also show bandwidth limitations, so improving cache utilization should directly decrease bandwidth demand and increase benchmark performance.

After reviewing potential gains from these optimizations, this paper discusses software and hardware directions that may improve programmability and performance of applications executing on heterogeneous processors. Software constructs should offer more flexible compute and data granularities. Hardware mechanisms should support lighter-weight thread handling and producer-consumer communication in caches.

The rest of this paper is organized as follows: Section II presents a case study motivating deeper exploration of GPU computing pipeline structures. Section III articulates the simulation methodology used to compare discrete GPU and heterogeneous processor systems, and Section IV quantifies the comparison. Section V describes and quantifies analytical

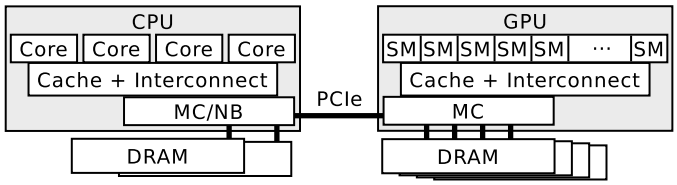

Fig. 1. Discrete GPU system with separate CPU and GPU chips.

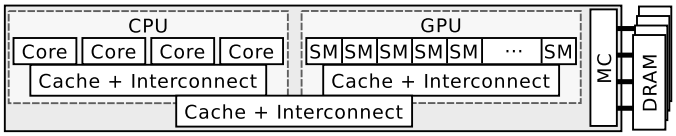

Fig. 2. Heterogeneous processor with integrated GPU on a single chip.

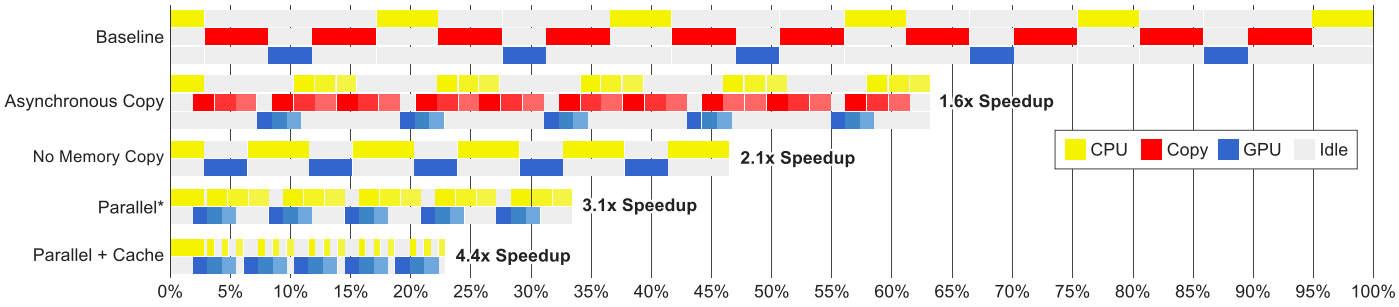

Fig. 3. `Kmeans` simulated and estimated (*) run times for various benchmark organizations.

estimates of run time and memory access improvements in heterogeneous processors. Section VII discusses related work, and Section VIII concludes.

II. Motivating Producer-Consumer Support

To motivate pipeline structure investigation, we begin with a case study of the `kmeans` benchmark running in our simulation environment. `Kmeans` shows significant compute and caching inefficiency due to the bulk-synchronous pipeline structure, and up to 77% of run time can be recovered by restructuring the application to run on a heterogeneous processor. The optimizations tested here on `kmeans` have widely varying potential benefits for other benchmarks and input sets, but further results show they are broadly applicable optimization targets for heterogeneous processors.

A. Example `Kmeans` Benchmark

From the Rodinia benchmark suite, the `kmeans` benchmark iteratively analyzes a set of n-dimensional points to find the k points that characterize clusters of the points. Each iteration involves calculating the distance between each of the points and the current k centers, assigning each point to the closest center, and then replacing poor centers with new candidate centers. Distance calculations and center assignments have wide thread-level parallelism (TLP) and so are performed on the GPU. The center replacement algorithm has limited TLP, so assigned centers are copied back from the GPU memory to perform the center adjustment on the CPU.

Baseline: When using copies in the discrete GPU setting, `kmeans` serializes nearly all of the work and copies. Run time component activity is depicted in Figure 3 as "Baseline". Despite only transferring a small amount of data between CPU and GPU memories in each iteration, over 50% of `kmeans` run time is spent copying data. This is due to the asymmetry of PCIe bandwidth (8GB/s) compared to the CPU and GPU memory bandwidth (24 and 179GB/s), which allow the CPU and GPU to process data substantially faster than a PCIe copy.

Bandwidth asymmetry in discrete GPU systems encourages programmers to minimize data transfers often resulting in wide, bulk-synchronous pipeline stages. For `kmeans`, the GPU sits idle for a substantial portion of run time (82%) though the GPU completes 95% of the compute operations, indicating that `kmeans` incurs very high GPU FLOP opportunity cost[1] for bulk transferring work between CPU and GPU.

[1]We refer to "FLOP opportunity cost" as the portion of compute FLOPs that go unused due to a core being inactive

B. Optimizing `Kmeans`

There are a number of ways that application pipelines can be restructured to improve performance. However, to date, little analysis has compared different inefficiencies and opportunity costs of optimizing GPU application structure for discrete GPUs versus heterogeneous processors. For `kmeans` operating on this particular input set, the programmer's incentive to optimize increases substantially when running on a heterogeneous processor. Specifically, removing memory copies provides a $2\times$ run time improvement, but $2\times$ more improvement can come from further CPU-GPU parallelism and effective cache management. It is difficult or impossible to employ these optimizations in current discrete GPU systems.

Asynchronous Memory Copy Streams: In the discrete GPU system, `kmeans` performance is hamstrung by the need to copy data back and forth between CPU and GPU memories. One option to reduce this overhead is to use kernel fission and asynchronous streams [21, 36]. Kernel fission requires the programmer to explicitly divide independent data and compute chunks of a kernel into separate kernels that can be overlapped with asynchronous memory copies. The "Asynchronous Copy" bars of Figure 3 show the run time activity for a 3-wide asynchronous stream organization.

While a non-trivial code transformation, kernel fission and streams can improve `kmeans` run time by 37%. Memory copies can be overlapped with CPU and GPU execution, though there are data dependencies that the limit overlap. Despite the data dependencies, `kmeans` run time could improve up to the point that the PCIe link is saturated for the full execution.

Emerging unified virtual memory architectures exist that allow coherent data synchronization between CPU and GPU over the PCIe link. However, we expect that the latency to perform these on-demand synchronizations will be too prohibitive to allow data handling as efficiently as streams. For `kmeans`, performance is likely to still bottleneck on copies, because the total data copied would remain the same.

Eliminating Memory Copies: In Figure 3, the "No Memory Copy" bars show the CPU and GPU activity of `kmeans` running on a cache-coherent heterogeneous processor without the need for memory copies. Without the copies, run time can improve over the baseline execution by nearly the total baseline copy time, and GPU utilization improves from 18% to 39%.

Unfortunately, this organization is still quite core and cache inefficient. First, this organization leaves either CPU or GPU cores idle throughout the complete execution. In terms of available compute operations, this organization incurs an opportunity cost of nearly 60% unused FLOPs. Further, this `kmeans` implementation was designed for a discrete GPU

and minimal copy overhead, which encouraged GPU kernel-granularity synchronization. This residual structure results in very inefficient use of cache. Each GPU kernel streams input and output data, and the total size of this data exceeds the size of cache, causing all produced data to spill off-chip before they are consumed. This results in roughly 9.5% more memory accesses than if these results could be passed in cache.

Parallel Producer-Consumer Compute: With a discrete GPU, extracting parallelism requires kernel fission to utilize GPU streams to manage data synchronization. In a heterogeneous processor, however, data synchronization between CPU and GPU can happen in coherent memory, possibly eliminating the need for kernel fission by allowing CPU-GPU communication with simple memory reads and writes.

In Figure 3, we estimate the run time if CPU consumer code runs immediately after GPU producers generate their output ("Parallel"). This estimate assumes similar core execution times as the asynchronous streams version, and analogously, performance improves up to the point that some component bottlenecks run time. In the heterogeneous processor, the CPU becomes the bottleneck, but the overlapped execution results in a 40% run time improvement over the no-copy case, and GPU utilization rises to nearly 65%.

Improved Heterogeneous Processor Caching: While the parallel producer-consumer estimate showed improved core utilizations, in actual simulation of this benchmark organization, performance improves still more due to caching. If the GPU and CPU work is chunked to synchronize small enough intermediate data between cores, the CPU is able to access all of its data out of cache. These cache hits dramatically reduce CPU memory access latency, which dominated the CPU execution time. Figure 3 ("Parallel + Cache") shows that these caching benefits improve `kmeans` run time by another 32%, and GPU utilization reaches 80%.

C. Motivation Summary

Overall, `kmeans` exhibits the major optimization opportunities that may become common in heterogeneous processors. First, overlapping core activity can reduce the opportunity cost of underutilized cores. Such an optimization is likely to be more straightforward when cores can communicate through memory rather than using PCIe transfers. Second, bringing producer and consumer tasks into closer temporal proximity has potential to greatly improve the use of cache, an optimization that is difficult in current discrete GPUs. The final optimized `kmeans` here leaves CPU cores underutilized, and further optimization could focus on migrating computation to those cores.

III. Methodology

A. Simulated System Configurations

This paper compares memory access and performance effects of optimizations for discrete GPUs and forward-looking heterogeneous processors. To control performance capabilities and allow flexible system architectures, we simulate the systems with configuration parameters defined in Table I. Both systems use the same CPU and GPU cores, and their compute capabilities are comparable to current mid-range discrete GPU systems or aggressive heterogeneous processors. Specifically, CPU cores are out-of-order superscalar capable of 14 GFLOP/s peak, and they have access to private L1s and an L2 cache. The GPU contains NVIDIA Fermi-like cores, which can each manage up to 1536 concurrent threads and issue up to 32 SIMT instructions per cycle for a rate of 22.4 GFLOP/s peak. GPU cores each have 48kB of scratch memory and 24kB data + instruction L1 cache, and they share a 1MB L2 cache.

TABLE I. Heterogeneous system parameters.

Component	Parameters
CPU Cores	(4) 4-wide out-of-order, x86 cores, 3.5GHz
CPU Caches	Per-core 32kB L1I + 64kB L1D and exclusive, private 256kB L2 cache, 128B lines
GPU Cores	(16) 8 CTAs, 48 warps of 32 threads, 700MHz 48kB scratch memory, 32k registers, Greedy-then-oldest warp scheduler
GPU Caches	24kB L1 per-core. GPU-shared, banked, non-inclusive L2 cache 1MB, 128B lines
Discrete GPU System	
Interconnects	CPU L2s/MCs: 6-port switch, GPU L1/L2: Dance-hall, GPU L2s/MCs: Direct links
CPU Memory	(2) DDR3-1600 channels, 24 GB/s peak
GPU Memory	(4) GDDR5 channels, 179 GB/s peak
PCI Express	v2.0 x16, 8 GB/s peak
Heterogeneous CPU-GPU Processor	
Interconnects	GPU L1/L2: Dance-hall, All L2s/MCs: High-bandwidth, 12-port switch
Memory	(4) shared GDDR5 channels, 179 GB/s peak

The discrete GPU system models the split CPU and GPU caches and memories. The CPU chip accesses DDR3 memory capable of up to 24 GB/s peak, while the discrete GPU chip has 4 GDDR5 memory channels capable of up to 179 GB/s peak. Consistent with many current discrete GPU systems, memory copies are performed using a PCIe link with peak bandwidth of 8 GB/s between CPU and GPU memories. When data is copied between CPU and GPU memories, any coherent cache lines containing data for the destination addresses are written back or invalidated.

To limit performance effects resulting from memory bandwidth differences, the heterogeneous processor CPU and GPU cores share access to the same GDDR5 memory as the GPU in the discrete system. The bandwidth of GDDR5 is likely to be comparable to emerging memory technologies, such as 2.5D/3D stacked DRAM, which may be used with heterogeneous processors. For the tests in this study, CPU and GPU memory access contention has marginal effect compared to other application-level differences described later.

B. gem5-gpu Simulator

To perform tests with well-controlled and flexible system architectures, we use the gem5-gpu simulator [27]. gem5-gpu offers full-system simulation of discrete GPU systems and heterogeneous processors with flexible memory hierarchies and PCIe configurations. The GPU model is from GPGPU-Sim v3.2.2 [5], and the CPU cores are the out-of-order model in gem5 [6]. All tests use Linux kernel 2.6.28.4.

C. Benchmarks

Depending on their application-level pipeline structure, GPU computing applications can see widely varying memory copy overheads and potential optimization targets. We aim to explore a broad range of these effects, so this study observes benchmarks from four open-source GPU computing benchmark suites. Table II summarizes details about the application-level structures of all benchmarks in these suites.

TABLE II. PRODUCER-CONSUMER RELATIONSHIPS IN BENCHMARKS.

				P-C Constructs		
Suite	Num. Bench	P-C Comm.	Pipe Paral.	Reg-ular	Irreg-ular	SW Queue
Lonestar	14	14	13	14	13	10
Pannotia	10	10	10	10	10	0
Parboil	12	8	8	8	3	1
Rodinia	22	19	18	19	6	0
Total	58	51	49	51	32	11
Portion	100%	88%	84%	88%	55%	19%

The Lonestar GPU suite [7] contains many benchmarks with irregular control flow and memory access behaviors ("Irregular"). Many of these benchmarks operate on graph-like data structures, and many utilize software queues, or "worklists", for tracking available work ("SW Queue"). Similar to Lonestar, the Pannotia benchmarks [8] perform various graph analyses, though each is structured to expose available work without software queues. Pannotia benchmarks are implemented with OpenCL, but ported to CUDA for this study. Representing some more traditional GPU computing workloads, this paper also characterizes the Parboil [31] and Rodinia [9] benchmark suites. These suites contain many image and signal processing, machine learning, and scientific numerical benchmarks, as well as a couple graph handling benchmarks. Of the total 58 benchmarks in these four suites, this paper examines 46 that work fully in gem5-gpu and perform non-trivial computations.

Table II also lists counts of benchmark pipeline characteristics. Most benchmarks (88%) contain multiple producer-consumer pipeline interactions ("P-C Comm."), including CPU execution, GPU kernels, or memory copies between CPU and GPU memories. Of these 51 applications with producer-consumer relationships, all but two could be parallelized to run pipeline stages concurrently or in closer temporal proximity than the unmodified benchmarks ("Pipe Paral."). Section V-A investigates the potential gains from such parallelization.

D. Benchmark Configurations

Memory Copies: To characterize the differences between applications running on discrete GPUs and heterogeneous processors, we run benchmarks with two different memory copy configurations. In discrete GPU simulation, we run benchmarks largely unchanged from their publicly available versions[2], which use CUDA to allocate and copy memory between CPU and GPU memory spaces.

The second benchmark configuration removes data copies between memory spaces. We use a combination of CUDA library and manual benchmark modifications to eliminate separate CPU and GPU copies of data. Specifically, for memory allocations that mirror CPU allocations into the GPU memory space, we allow the GPU to access the CPU allocation directly and eliminate the GPU memory allocation. The CUDA library identifies and eliminates many allocations like this by observing runtime dynamic CUDA calls.

We also manually modify some benchmarks to eliminate memory copies. Many allocations only serve to double-buffer mirrored data. In these cases, the GPU has multiple versions of the same mirrored CPU data, but runtime analysis cannot safely eliminate redundant allocations or copies. Using consistent, explicit copies between CPU and GPU allocations is often sufficient to allow the CUDA library to eliminate copies.

These two means eliminate the substantial majority of memory copy overheads. Specifically, all but one benchmark (Lonestar `bh`) see reduced number of copies, and 24 of the 46 benchmarks have at most 1 remaining memory copy. Because some copies still remain, we refer to this benchmark version as "limited-copy". The next section characterizes improvements from removing copies.

Data Location Assumptions: For each of the benchmarks, we delineate the portion of run time during which data handling and computation occur as the region of interest (ROI). The ROI begins after the CPU sets up all necessary data to be resident in it's physical memory, but before the CPU transfers any data to the GPU memory space or is allowed to launch GPU kernels. The only exception is Rodinia `mummer`, which also reads data from disk while the GPU is executing. Before the ROI begins, the application is allowed to allocate memory to be used by the GPU, but these allocations cannot have been accessed prior to the start of the ROI. We define ROI completion as the time at which all output data produced by the application is once again available in the CPU memory. In the discrete GPU case, this means that resulting outputs are copied back to the CPU memory space within the ROI. In the heterogeneous processor, the ROI can end after the last CPU or GPU activity completes to generate final output.

This ROI definition is important for a couple reasons. Prior work shows that data copies must be accounted for when analyzing application-level performance [13], so especially when directly analyzing memory copy overheads. Further, many open-source GPU computing benchmarks actually perform subsets of full GPU computing applications. Real world applications may shuttle data through other compute kernels rather than reading from disk or randomizing inputs, which are common input methods for benchmarks observed here. In this case, compute kernels executed prior to and following the kernels within our ROIs would require that data end up in a designated location such as CPU-accessible memory.

GPU Memory Management: Besides the architectural differences between discrete GPUs and heterogeneous processors, a notable difference between these systems is GPU memory management. In the discrete GPU setting, GPU memory is not accessible from CPU cores, so the CPU need not have access to GPU address translations. A GPU-specific memory allocator is allowed to map memory for address translations while the PCIe copy engine or GPU are executing, and GPU minor page faults are handled completely by the GPU.

In the heterogeneous processor, however, memory mappings must be consistent across CPU and GPU. This requires that both CPU and GPU address translations access a common page table, and page table updates must be simultaneously visible to both. To achieve this, gem5-gpu implements GPU page faults in a manner similar to IOMMU page faults available in recent Linux kernel versions (e.g. v3.19). Specifically, the GPU raises an interrupt to the CPU, which performs memory page mapping and returns the mapping to the GPU.

Prior studies in GPU address translation [24, 25] have described implementations of efficient GPU address translation, including TLB and page table walk structures. However, they have not explicitly studied the performance impact of

[2]Benchmark versions working in gem5-gpu are available open-source with the simulator: http://gem5-gpu.cs.wisc.edu/repo.

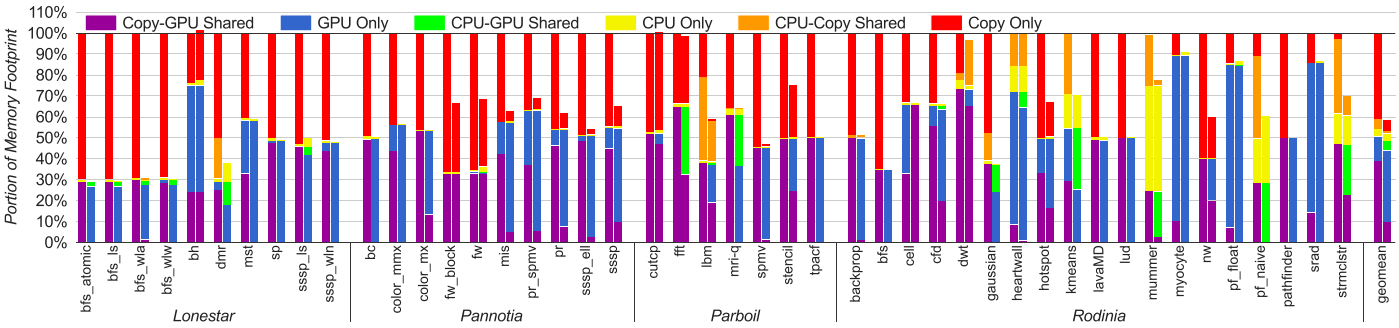

Fig. 4. Breakdown of memory footprint touched by component type for copy (left bars) and limited-copy (right bars) versions normalized to the copy version.

GPU page faults. The results in this paper show that CPU-handled GPU page faults can cause significant performance degradation, and we suggest this as an area of future research.

Input Set Selection: Benchmark input sets were chosen to ensure that the ROIs meet the following criteria. First, all benchmarks execute at least 1B instructions combined for both the CPU and GPU, and total instructions typically exceed 2.1B. The most instructions executed by any benchmark is 90B. Second, since the GPU completes a majority of work, input sets were chosen to ensure that total GPU execution time is at least 5ms and typically more than 30ms. The longest running ROI is 1.535s. Finally, input sets were chosen so that total memory footprint is at least 6MB and usually greater than 42MB for copy benchmark versions. Limited-copy memory footprints are at least 3.5MB and usually greater than 24MB.

IV. Eliminating Memory Copies

Most existing GPU computing applications were developed for discrete GPU systems and thus use explicit memory copies to move data between CPU and GPU memories. As a baseline for further core and cache inefficiency analysis, this section observes how memory copies affect some basic benchmark characteristics: memory footprint, memory access counts, and ROI run time.

The statistics below show that the GPU-kernel-synchronous structure of existing benchmarks limit the immediate use of available cores and cache in heterogeneous processors. Removing memory copies often decreases benchmark memory footprint and total memory accesses, and run times typically decrease by the total time of eliminated memory copies. However, the aggregate number of memory accesses from CPU and GPU cores tends to remain similar after removing copies, and CPU and GPU cores see limited utilization improvements.

A. Memory Footprint

The limited-copy benchmark versions typically just mirror data between CPU and GPU memories, so eliminating mirrored data can significantly reduce the total memory footprint. We measure benchmark memory footprint by observing the addresses of all memory accesses from CPU and GPU cores, and in the discrete GPU system, the PCIe copy engine. Figure 4 breaks down these footprints into mutually exclusive subsets touched by one or more components for the copy and limited-copy versions of the benchmarks. The plot is normalized to the total memory footprint of the copy version of each benchmark (left bar of each pair) to show how the memory footprint decreases when eliminating mirrored data copies in the heterogeneous processor setting (right bars).

First, most copy benchmarks replicate data from the CPU to GPU memory. Copy portions of the bars (red, orange, purple) make up nearly all of each bar, indicating that most of a benchmark's data is copied at some point. In a few other benchmarks, such as Lonestar `bh` and Rodinia `srad`, the GPU uses substantial temporary data that is only ever resident in GPU memory. This GPU-temporary memory often stores large sets of intermediate data that are passed between GPU kernels and cannot be statically bound to GPU scratch memory.

Second, some object-oriented and graph-based computations do not touch their whole data sets though their copy versions need to move that data to GPU memory. For example, in Lonestar `bfs` and Pannotia `fw` benchmarks, the copy engine touches nearly all of the data, but the CPU and GPU combined touch less than one-third of that data. Here, the CPU and GPU traverse the data's structure, but do not necessarily need to touch all data for the desired computations. For benchmarks like this, prior work [1, 2] shows that memory copies can be saved using smart page placement or on-demand page migration to the GPU rather than memory copies.

Of the remaining limited-copy memory footprint, the GPU usually uses more than 70% of the data, suggesting that these benchmarks use the GPU to process the majority of data. In few cases (Lonestar `bh`, Parboil `cutcp` and `fft`, and Rodinia `dwt` and `heartwall`), our memory copy elimination techniques are unable to remove the majority of copied footprint. However, more extensive manual benchmark modification should be able to remove all of these copies.

B. Memory Accesses

As expected, the copy benchmarks also incur excess memory accesses for moving data between CPU and GPU memory spaces. Figure 5 shows each benchmark's total memory accesses broken down by component type for the copy (left bars) and limited-copy benchmark versions (right bars). Most commonly, copy accesses account for 4-10% of total memory accesses, but in a substantial subset of benchmarks, copies account for more than 20% of total memory accesses.

For benchmarks with a small portion of copy accesses, CPU and GPU cores often perform multiple accesses per data element. For most Lonestar and Pannotia benchmarks, memory copies account for at most 5% of total memory accesses, because CPU and GPU work perform multiple traversals of and modifications to irregular data structures, such as graphs. Similarly, benchmarks, such as Rodinia `gaussian` and `lud`, perform iterative refinement to the majority of their data, so copies account for a small portion of memory accesses.

When memory copies are removed, Figure 5 shows that typically all copy memory accesses are eliminated. In the

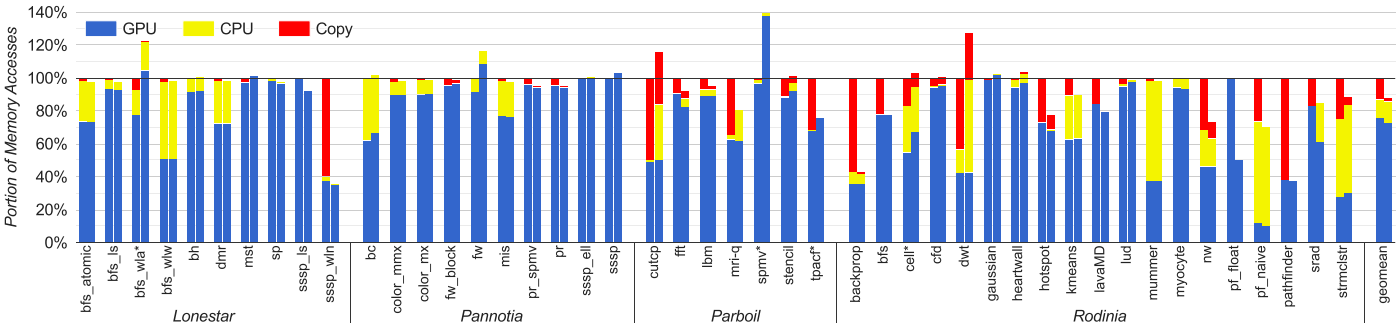

Fig. 5. Memory access breakdown by component type for copy (left bars) and limited-copy (right bars) versions normalized to copy version.

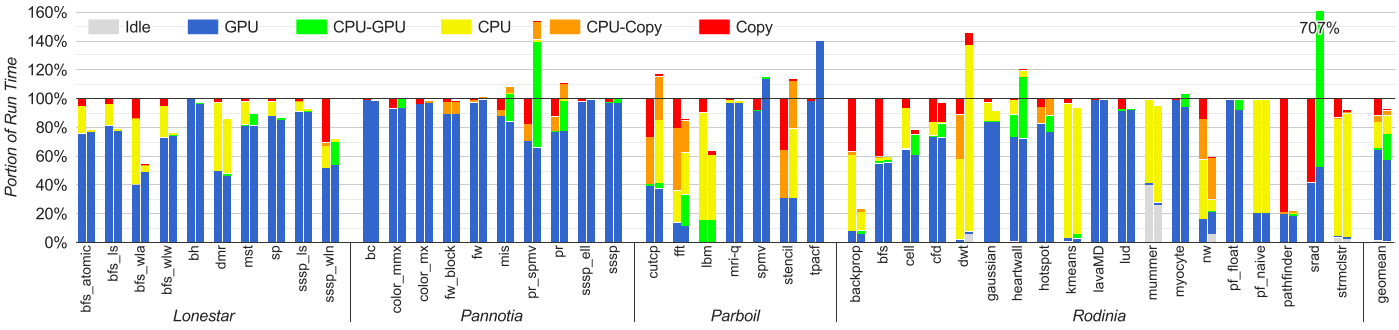

Fig. 6. Run time component activity breakdown for copy (left bars) and limited-copy (right bars) versions normalized to copy run time.

geometric mean, the number of total copy accesses declines by more than 11%. Further, it is common that the CPU and GPU memory access counts remain substantially similar in the limited-copy case. Despite the CPU and GPU being able to share data in caches of heterogeneous processors, the structure of the applications is such that little cache efficiency can be gained simply from removing memory copies. We speak more to this in Section V-C.

Though memory access counts do not indicate any systematic caching improvements when moving to a heterogeneous processor, there are two uncommon conditions under which removing memory copies can significantly change GPU memory access counts. First, when CPU and GPU share memory, the GPU must rely on the CPU for page fault handling, which can upset memory access orderings. For Rodinia `srad`, page fault handling causes accesses to be shifted from the GPU to the CPU, which clears memory pages during page mapping. For Rodinia `pf_float`, the page fault handler causes serialization of some GPU memory accesses, which substantially reduces GPU cache contention and cuts off-chip accesses by 50%. In Pannotia `fw`, GPU access serialization limits L1 cache locality, which results in an increase in off-chip accesses. These behaviors are exceptional, though the affected benchmarks share other characteristics common to other benchmarks.

Another cause of increased GPU memory access counts is memory allocation misalignment, which affects GPU coalescing and cache contention. Specifically, since memory allocations are no longer managed by the CUDA library, which cache-line-aligns GPU allocations, CPU-GPU-shared allocations can lack good alignment. As a result, GPU access coalescing can result in more memory accesses to caches, and stress the ability of the cache to capture temporal locality while streaming data. The benchmarks marked with '$*$' in Figure 5 experience this elevated cache contention. Nearly all of the extra memory accesses result from misalignment and could be avoided by using an aligned memory allocator.

C. Run Time

On average, removing memory copies results in modest performance improvement. Figure 6 shows the run time for copy and limited-copy benchmark versions broken down by the portion of run time during which each component is active. Overall, only CPU-side work can benefit from heterogeneous processor caching and copy removal. In the aggregate, removing memory copies results in a 7% run time improvement, which we break down below.

To the first order, limited-copy benchmark run times improve due to reduced memory copy and CPU execution time. The reduction in memory copies eliminates much of the run time during which the PCIe copy engine is exclusively handling data, as demonstrated by benchmarks like Rodinia `bfs` and `pathfinder`. This is a common gain for many benchmarks and in the geometric mean accounts for an 11% run time improvement.

As a first major caching benefit in heterogeneous processors, benchmarks with significant CPU execution time in their copy versions often see run time improvement due to improved caching and memory copy removal. As exemplified by most Lonestar benchmarks, even small memory copies invalidate data from CPU caches when moving that data between CPU and GPU memories. Since CPU execution is often latency-sensitive, CPU progress is slow as it reads data back into caches from off-chip memory after PCIe transfers. This cache invalidation is often avoided in the limited-copy benchmarks, resulting in a geometric mean 6% run time improvement.

This run time plot also illustrates detrimental first order performance effects of address translation. As mentioned, GPU page faults cause serialization of some GPU memory accesses, which would be executed in parallel if not waiting

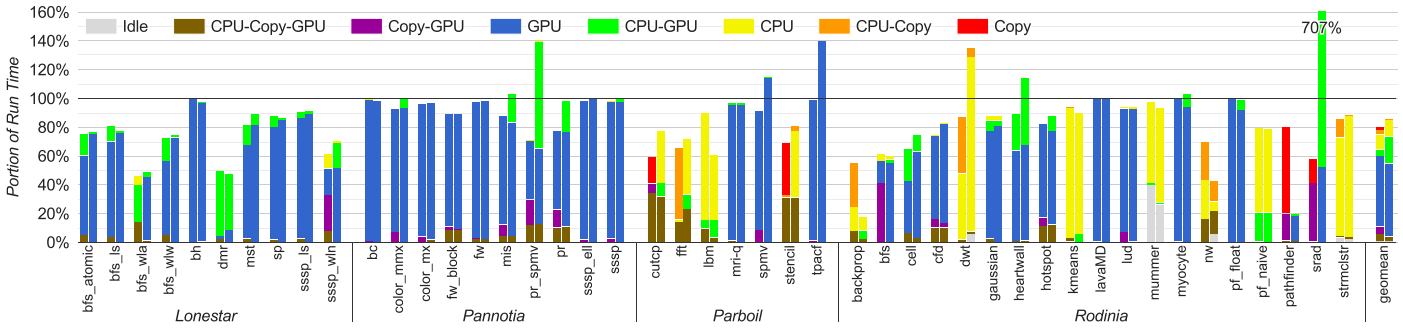

Fig. 7. Estimated: Component-overlap run time breakdown for copy (left bars) and limited-copy (right bars) versions normalized to baseline copy run time.

on the CPU page fault handler. On average, this results in a 9% GPU slowdown, though the majority of this slowdown is experienced by just three benchmarks: Pannotia `pr_spmv`, and Rodinia `heartwall` and `srad` (7× slowdown). In these benchmarks, numerous would-be parallel GPU writes go to unmapped global memory and must wait on the serialized CPU page fault handler for address mappings.

V. Pipeline Optimizations

As noted in the case study, the bulk-synchronous pipeline structure of GPU computing applications leads to core and cache inefficiency. In fact, it is widely believed that most existing GPU computing benchmarks perform serialized CPU and GPU portions of run time. This belief is confirmed by reviewing Figure 6 and noting that most execution time for both copy and limited-copy benchmarks is exclusively running the copy (red), CPU (yellow), or GPU (blue). This indicates that there may be reasonable potential to improve performance by overlapping execution of these components.

In this section, we estimate how eliminating memory copies can change the potential gains of application restructuring to better leverage available parallel resources. We identify three opportunities to improve resource utilization and compare them in the following subsections. The estimates indicate that better core utilization could result in run time gains greater than 20%, and there is significant room to improve cache efficiency in the heterogeneous processor setting.

A. Overlapping Communication and Computation

For both copy and limited-copy benchmark versions, a reasonable first cut at improving performance is to run the same code, but try to expose more overlap of CPU, copy, and GPU activity. For the discrete GPU, such parallelism might be achieved with kernel fission and asynchronous streams as described previously. In the heterogeneous processor, data could be passed in memory between CPU and GPU. Specifically, CPU or GPU consumer threads could be launched, and set to wait for in-memory signals indicating when data is ready to be consumed. The producer threads can set these signal variables as their generated results become available. This software organization could use similar data blocking structure as kernel fission + asynchronous streams, but may avoid the need to split GPU kernels and manage separate kernel streams since threads synchronize in memory.

To test these benchmark transformations broadly would be a tremendous amount of work. However, we can employ analytical modeling to get a sense for their benefits. Using an Amdahl's Law-like calculation, we estimate potential performance gains from overlapping component activity for all benchmarks. The following formula estimates component-overlap run time, R_{co}:

$$R_{co} = C_{serial} + max(C - C_{serial}, P, G) \quad (1)$$

Here, C, P,and G are the CPU, copy, and GPU portions of run time, respectively. Since the CPU acts as the control component, some of its activity strictly cannot be overlapped. C_{serial} accounts for non-overlapped kernel and memory copy launch portions of CPU run time. We estimate this by iterating through pipeline stage statistics and identifying copies and kernel launches that occur while no other kernels or copies are executing to mask the launch latency. C_{serial} can account for up to 9% of benchmark run time, but only for a few benchmarks with numerous, serialized kernels and copies, such as Lonestar `sssp_wln` and Rodinia `bfs`.

We briefly validate the component-overlap model by applying benchmark transformations to three copy and limited-copy benchmark versions: `backprop`, `kmeans`, and `strmclstr`. Each of these applications is structured with wide data-level parallelism per kernel instance, and this structure is common to more than half of the benchmarks in this study. We chunk the kernel input and output data either to apply kernel fission + asynchronous streams in the discrete GPU system, or in-memory "data ready" signal variables in the heterogeneous processor. By chunking the data to execute at least four concurrent streams, the run time of each benchmark improves to within 3.1% of the component-overlap estimate.

While the component-overlap model is accurate for some benchmarks, it can still be optimistic for a couple of reasons. First, the estimate does not account for wide data dependencies from one pipeline stage to the next that may limit parallelism (e.g. Lonestar `dmr` or Parboil `fft`), or memory access contention that may occur from overlapping component activity. Second, it is likely that in order to extract core activity parallelism, more pipeline control code or synchronization primitives may need to be added, possibly increasing run time. If, however, the heterogeneous processor sees improved caching effects, performance could improve beyond this estimate, and we saw this with `kmeans`.

Analysis: Component-overlap run time estimates for copy and limited-copy benchmark versions are depicted in Figure 7, and normalized to the baseline copy run time. These results suggest that overlapping communication and computation has the potential to eliminate much of the performance difference between copy and limited copy versions.

There are two classes of copy benchmarks that are likely to benefit substantially from component-overlap optimizations. First, many benchmarks with regularly structured

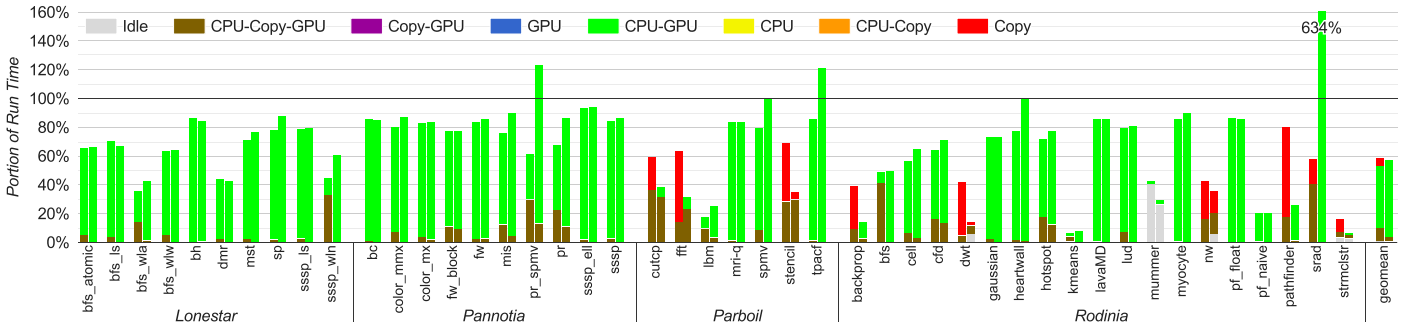

Fig. 8. Estimated: Migrated-compute run time breakdown for copy (left bars) and limited-copy (right bars) versions normalized to baseline copy run time.

data/computation have significant memory copy overheads. Due to their regular structure, it is straightforward to use kernel fission and asynchronous streams to overlap copies with computation, and the potential performance gains often bring them in line with the limited-copy benchmarks. Benchmarks in this class include Parboil `cutcp` and `stencil`, and Rodinia `backprop`, `cell`, `cfd`, `hotspot`, `kmeans`, `nw`, `srad`, and `strmclstr`.

The second class of benchmarks has a common outer-loop structure executed by the CPU. Examples include Lonestar benchmarks and Rodinia `bfs` in which the CPU launches GPU kernels and then waits to decide whether to continue loop execution until kernels complete and results are copied back. Often the deciding factor is whether the GPU generated more work during the last GPU kernel, and this factor can be triggered at any time during the kernel. This structure could be optimized if there were mechanisms to signal the CPU thread as soon as the loop condition became true to allow the CPU to overlap its control code with the running kernel.

B. Migrating Compute Between Cores

Though overlapped execution could improve benchmark run times by 10-15%, there is still significant FLOP opportunity cost for many benchmarks. The majority of this underutilization results from poor balance of work across CPU and GPU cores. Most frequently, CPU cores are left idle for nearly all benchmark run time, but for some benchmarks, GPU cores are left idle for significant portions of run time (e.g. Rodinia `dwt`, `kmeans`, and `strmclstr`).

To quantify the impact of this core underutilization, this subsection estimates potential performance gains from migrating compute between CPU and GPU cores. This can be achieved through a number of mechanisms, including identifying portions of computation that can be hoisted from the beginning/end of one pipeline stage to the prior/next stage, or identifying data-independent portions of a pipeline stage that can be computed on underutilized cores. In discrete GPU systems, work can be migrated between CPU and GPU cores, though such transformations are often unwieldy since they also require migrating data. In heterogeneous processors with shared physical memory, on the other hand, it may be easier to split parallel work across core types to improve core utilizations.

This optimistic migrated-compute estimate assumes that all benchmark compute phases can be effectively distributed across CPU and GPU cores. Since performance would not be able to exceed hard resource limitations, the migrated-compute estimate uses two limiting factors. First, computation cannot exceed the FLOP rate of cores to which it is migrated, so we estimate a peak-FLOP/s-relative run time, $R_{mc_{core}}$. Second, effective memory bandwidth cannot exceed the peak bandwidth to off-chip memory, so we estimate the limit on run time improvement running up against this bound ($R_{mc_{BW}}$). Perfect migrated-compute run time, R_{mc}, is estimated from these as follows:

$$R_{mc_{core}} = \frac{C * F_{cpu} + G * F_{gpu}}{F_{cpu} + F_{gpu}} \quad (2)$$

$$R_{mc_{BW}} = M/BW_{mem} \quad (3)$$

$$R_{mc} = max(P, R_{mc_{core}}, R_{mc_{BW}}) \quad (4)$$

As above, C, P, and G are the CPU, copy, and GPU portions of run time. F_{cpu} and F_{gpu} are the CPU and GPU peak FLOP rates, respectively. M is the total number of memory accesses, and BW_{mem} is the peak achieved memory bandwidth, which generally tops out at about 82% of peak pin bandwidth.

Validating the migrated-compute model is difficult compared to the compute-overlap model, because few programming constructs exist for such transformations. However, we applied manual program transformations to the `kmeans` and `strmclstr` copy benchmark version that indicate that migrated-compute estimated gains are plausible. Specifically, both benchmarks perform matrix-vector and reduction-like operations on CPU cores, and we rewrote these operations to run on GPU cores as part of preceding kernels. Where appropriate, we further utilized GPU atomic operations to bring effective FLOP rates near the GPU theoretical peak. These transformations reduced the amount of data transferred between GPU and CPU memories, and improved run time by more than 2.5× to within 35% of the compute-overlap estimates.

Figure 8 shows the results of the migrated-compute run time estimates for copy (left bar in each pair) and limited-copy (right bar) benchmark versions. In the common cases, such as in Lonestar and Pannotia, the results indicate that fully utilizing compute resources could improve performance by another 4-13% by moving GPU work to idle CPU cores. Such migration may be relatively straightforward if the CPU executes tasks indexed similarly to GPU threads. On the other hand, when CPU execution dominates baseline run time (e.g. Rodinia `dwt`), the potential gains are substantially larger, because shifting more parallelism to the GPU could reduce the substantial, unused GPU FLOPs.

Benchmarks with substantial baseline CPU execution time often contain pipeline organization or memory handling inefficiencies that result from the need to copy data between CPU

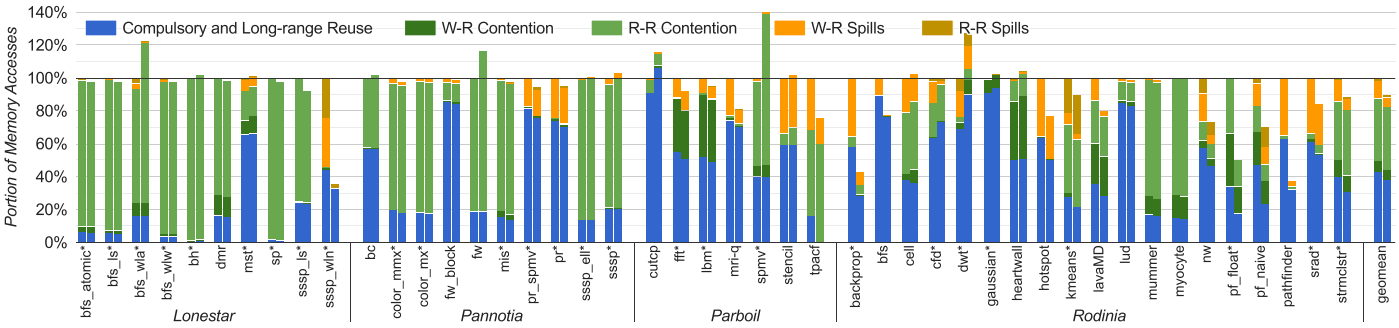

Fig. 9. Memory accesses broken down by cause for copy (left bars) and limited-copy (right bars) versions normalized to copy benchmark versions.

and GPU memory spaces. For example, inefficient pipeline organizations sometimes contain medium-to-high TLP-capable work that the CPU performs single-threaded. Migrating this compute to the GPU would often require copying data to the GPU in the discrete GPU setting. Many of these benchmarks already have high memory copy overheads in the discrete GPU setting, so migrating data may actually negatively impact performance. These inefficiencies indicate that migrating compute is likely to be easier in heterogeneous processors. Parboil `cutcp` and `lbm`, and Rodinia `backprop`, `dwt`, `kmeans`, `pf_naive`, and `strmclstr` contain such inefficiency.

Other benchmarks with substantial CPU execution time have high data movement overheads. Parboil `fft` and `stencil`, and Rodinia `mummer` contain memory copies for double buffering data, or clearing memory regions, which are costly CPU operations. We expect these could each be optimized for heterogeneous processors by improving their data structures to eliminate the need for these memory operations.

Finally, it is likely that remaining copy-dominated benchmarks (20% of benchmarks with red portions in Figure 8 bars) will be difficult to optimize on discrete GPUs. In most of these cases, this performance bottleneck is largely fundamental to the computation being performed: Significant data must be moved relative the amount of computation completed on that data. Further, of these benchmarks, Parboil `fft`, Rodinia `backprop`, `nw`, and `strmclstr` contain many-to-few data dependencies between pipeline stages, suggesting that inter-stage optimization will be difficult in the presence of memory copies.

C. Coordinated Use of Cache Capacity

The performance gain estimates above do not account for an important feature of heterogeneous processors not available in discrete GPU systems: shared CPU-GPU caching. The GPU-kernel-synchronous pipeline structure of existing benchmarks causes large per-pipeline-stage memory footprints, which frequently push data out of cache before it can be reused. Without memory copies to work around, heterogeneous processors have flexibility to improve shared cache management and increase performance.

To quantify opportunities for better cache data reuse, we inspect and categorize off-chip memory accesses based on their relationship to other memory accesses. At the off-chip interface, we record whether a cache block was previously accessed off-chip, the type of the access (read or write), the type of the previous access, and the reuse distance in terms of pipeline stages since previous access. We identify four classes of memory access that may be reduced or eliminated through better benchmark organization or caching. Figure 9 shows the breakdown of these memory accesses, which we describe next.

Required memory accesses: Compulsory memory accesses include the first off-chip read from and last write to a piece of data. Such accesses must occur to complete the computation, so they cannot be eliminated. Grouped with compulsory accesses (blue), long-range reuse accesses occur to data that has been previously accessed, but the time between the accesses spans multiple pipeline stages. Reducing these accesses may be possible, but would probably require substantial benchmark restructuring to improve temporal locality.

Spills: There are two categories of memory accesses caused by cache "spills" from one pipeline stage to the next (orange shades in Figure 9), and they represent about 10% of memory accesses on average. First, in bright orange, "W-R Spills" count memory writes from one stage that are read in the next stage, and thus represent producer-consumer relationships between stages. In darker orange, "R-R Spills" are reads from the same data in consecutive pipeline stages, indicating that the stages share input data. Inter-stage spills commonly result from GPU kernel stages that produce or consume data quantities in excess of cache capacity, another symptom of the GPU-kernel-synchronous structure of these benchmarks. As a result of their pipeline organizations, most benchmarks experience little reduction in cache spills when removing memory copies.

Eliminating inter-stage cache spills can result in significant performance improvement, especially when shifting work between CPU and GPU cores. In the `kmeans` case study, overlapping CPU and GPU execution in the heterogeneous processor eliminated the 9.5% of accesses that resulted from W-R spills, and subsequent cache hits increased CPU performance by 2.6×. CPU cores tend to be more sensitive to memory access latency than GPU cores [14], and shifting accesses from off-chip memory to cache hits can decrease CPU run time proportionally to the reduction in access latency [10].

Contention: When a pipeline stage accesses a large concurrent memory footprint, cache capacity contention can occur causing data to be evicted from cache before it can be reused. Further accesses must pull the data back from off-chip. Most frequently, these repeated contention accesses are reads ("R-R Contention"), which account for 38% of total accesses and upwards of 80% for many benchmarks. Other contention accesses begin with a writeback of the data ("W-R Contention"), but the data is read again during the same pipeline stage, indicating that the writeback occurred before all uses of the data were complete. These accesses can account for up to 36% of a benchmark's accesses. The substantial portion of contention accesses indicates that pipeline stage working sets often greatly exceed the available cache capacity.

Figure 9 also indicates significant potential performance

gain by reducing cache contention. As denoted by '$*$' in the figure, many benchmarks bump against off-chip memory bandwidth limitations during cache contentious pipeline stages. Most of these bandwidth-limited benchmarks (e.g. Lonestar and Pannotia) also show significant cache contention memory accesses. Reducing the excess accesses is likely to proportionally reduce the memory bandwidth demand and run time.

VI. Key Implications

The results in the last section indicate substantial opportunity to optimize core and cache utilization in heterogeneous processors. Potential research directions that will be applicable include producer-consumer parallelism, compute migration, and shared/cooperative cache management.

Software and hardware should provide efficient means to move producer and consumer tasks into closer temporal proximity. To achieve this, prior work has proposed kernel fusion [36] and data pyramiding [32] for GPU-GPU producer-consumer relationships. However, these tend to be complicated program transformations, which can encounter resource limitations, such as GPU register and scratch memory capacity. These methods can still result in cache spills, as experienced by Parboil `stencil`, `cell`, `hotspot`, and `pathfinder`.

More recently, CUDA and OpenCL have added methods to improve producer-to-consumer programmability. CUDA 5.0 introduced dynamic parallelism [20], which allows GPU code to dynamically launch consumer kernels. While this technique can provide programmability benefits for dynamic and irregular applications, it has been found that kernel launch overheads can outweigh performance benefits [34]. OpenCL 2 added support for work queues, which allow producer tasks to queue generated work for other tasks to consume [17]. While queues may also provide producer-to-consumer programming flexibility, programmers will need to carefully pack data to maintain good GPU memory access coalescing for performance and efficiency.

Moving producer and consumer tasks into closer temporal proximity may raise new caching challenges. If producers generate data more quickly than consumers can pull cached data, spills will still occur. Software or hardware techniques could modulate the rate of data production and consumption to keep performance-sensitive data on chip. Further producer-consumer analysis techniques should improve identification of a task's live data and estimation of concurrent memory footprint to aid the programmer in placing data in available cache to avoid existing cache contention. Such techniques could prove very valuable for applications with irregular memory accesses (e.g. most Lonestar and Pannotia benchmarks).

For heterogeneous processors, compute migration could unlock further fusion-like optimization opportunities. When CPU cores provide reasonable compute resources, migrating short-running GPU kernels to CPU cores could increase pipeline compute overlap and increase effective cache capacity when CPU cores have private, non-inclusive cache. Such optimization may work in benchmarks with multiple, varying complexity GPU kernels, as in Lonestar `dmr`, `mst`, or `sp`.

Finally, this paper discusses ways that existing benchmarks may be optimized for heterogeneous processors, and identifies constructs that may be directly used to develop new applications for heterogeneous processors. Forward-looking application development will likely adopt light-weight task handling and data dependency tracking early to ease the effort required to leverage available heterogeneous processor caching, since it is a primary benefit over discrete GPU systems.

VII. Related Work

In addition to the many studies cited throughout the paper, we identify a few classes of related work. First, while prior benchmark characterizations focus mostly on the GPU-side resources and behaviors [7–9, 31], this study focuses on the benchmark pipeline structures and interaction of all components in the systems. We are not aware of any prior studies that compare such results across many benchmark suites.

Prior studies propose methods to manage concurrency and optimize software pipeline structure. These mostly target multicore CPU and SMP systems [12, 33, 37] rather than systems containing GPUs, and few characterize potential optimizations to whole application pipeline structures.

Finally, many prior studies look at GPU application optimization (e.g. [19, 29]) and run time comparisons [11], but we are not aware of any prior studies that quantify potential optimizations focused around the elimination of memory copies in heterogeneous CPU-GPU processors. However, there are studies that describe results from mitigating redundant data and copies in other settings, such as OS data pipes to GPUs [28], and networking [3, 38].

VIII. Conclusion

This paper compares GPU computing application optimization opportunities for discrete GPU systems and heterogeneous CPU-GPU processors. Benchmarks were ported to remove memory copies and run in cache-coherent heterogeneous processors. When comparing core utilization efficiency, the potential gains for improving core parallelism in either system type are similar. However, it is likely that programming models will make it easier to capture parallelism gains in the heterogeneous processor.

A memory access characterization indicates that the bulk-synchronous nature of GPU computing applications causes poor cache efficiency. Data is often spilled off-chip before it can be reused. The layout and use of data indicates that heterogeneous processors are likely to provide caching opportunities that can improve application performance beyond the capabilities of discrete GPU systems. To capture these opportunities is likely to require flexible data/compute granularities and migration, and coordinated/intelligent caching.

IX. Acknowledgement

This work was supported in part with NSF grants CCF-1218323, CNS-1302260, and CCF-1438992. The views expressed herein are not necessarily those of the NSF. Wood has significant financial interests in AMD and Google.

References

[1] N. Agarwal, D. W. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2015, pp. 354–365.

[2] N. Agarwal, D. W. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 607–618.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM Conference*, August 2010, pp. 63–74.

[4] ARM, "CoreLink CCI-400 Cache Coherent Interconnect," 2012.

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

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, August 2011.

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

[8] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, September 2013, pp. 185–195.

[9] 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 *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[10] X. E. Chen and T. M. Aamodt, "Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 3, pp. 10:1–10:28, October 2011.

[11] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 216–225.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006, pp. 151–162.

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

[14] J. Hestness, S. W. Keckler, and D. A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2014.

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

[16] Intel, "The Compute Architecture of Intel Processor Graphics Gen8," https://software.intel.com/en-us/file/compute-architecture-of-intel-processor-graphics-gen8pdf, September 2014, accessed: 2015-04-19.

[17] Khronos Group, "The OpenCL Specification," https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, October 2015, accessed: 2015-04-20.

[18] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 733–745. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750421

[19] NVIDIA, *GPU Gems 3*. NVIDIA Corporation, 2007.

[20] ——, "Dynamic Parallelism in CUDA," http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf, 2012.

[21] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[22] ——, "Unified Memory in CUDA 6," http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, November 2013, accessed: 2015-04-03.

[23] ——, "Summit and Sierra Supercomputers: An Inside Look at the U.S. Department of Energy's New Pre-Exascale Systems," November 2014.

[24] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 743–758.

[25] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 568–578.

[26] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 457–467.

[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] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions To Manage GPUs as Compute Devices," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2011, pp. 233–248.

[29] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008, pp. 73–82.

[30] J. Sell and P. O'Connor, "The Xbox One System on a Chip and Kinect Sensor," *Micro, IEEE*, vol. 34, no. 2, pp. 44–53, March 2014.

[31] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[32] M. Strengert, M. Kraus, and T. Ertl, "Pyramid Methods in GPU-based Image Processing," in *Proceedings of Vision, Modeling, and Visualization*, November 2006, pp. 169–176.

[33] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the International Conference on Compiler Construction (CC)*, April 2002, pp. 179–196.

[34] J. Wang and S. Yalamanchili, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2014.

[35] B. Wile, "Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems," September 2014.

[36] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, "Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission," in *International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, May 2012.

[37] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2010, pp. 86–97.

[38] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling Network Performance for Multi-tier Data Center Applications," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NDSI)*, March 2011, pp. 57–70.