The Journal of Supercomputing (2018) 74:6898–6922
https://doi.org/10.1007/s11227-018-2589-x

Accelerated bulk memory operations on heterogeneous multi-core systems

JongHyuk Lee[1] · Weidong Shi[2] · JoonMin Gil[3]

Published online: 8 September 2018

Abstract

A traditional fixed-function graphics accelerator has evolved into a programmable general-purpose graphics processing unit over the past few years, the general-purpose computing on GPU (GPGPU). Recently, revolutionary measures have been taken along this direction: an integrated GPU, i.e., CPUs and GPUs are integrated into the same package or even into the same die. However, considering a system-on-chip, the GPU takes up considerable silicon resources, but when running non-graphical workloads or non-GPGPU applications it is likely that overall system performance will not be affected. This paper presents a novel approach to accelerate conventional operations that are normally performed on CPUs, which are bulk memory operations such as memcpy or memcmp, using an integrated GPU. Offloading bulk memory operations to the GPU has many benefits: (i) The throughput GPU outperforms the CPU in bulk memory operations; (ii) for on-die GPUs with unified cache between the GPU and the CPU, the CPU can utilize the GPU private cache to store the moved data and reduce the CPU cache bottleneck; (iii) additional lightweight hardware can also support asynchronous offloads; and (iv) unlike the prior art using a dedicated hardware copy engine (e.g., DMA), our approach utilizes as much GPU hardware resources as possible. The performance results based on our solution showed that offloaded bulk memory operations outperform CPU up to 4.3 times faster on micro-benchmarks while using fewer resources. Using eight real-world applications and a cycle-based full-system simulation environment, five of eight applications showed about 30% speedup and two applications showed about 20% speedup.

Keywords Bulk memory operation · GPU · Heterogeneous multi-core system · SIMD

This paper is an extended version of a conference paper that appeared as [1].

✉ JoonMin Gil
jmgil@cu.ac.kr ; joonmin.gil@gmail.com

Extended author information available on the last page of the article

1 Introduction

Increasing needs of custom graphics processing has led a traditional fixed-function graphics accelerator to evolve into a dedicated graphics processing unit (GPU) so that it can execute programmable shader codes. Furthermore, such programmability also allowed general-purpose applications, in addition to 3D graphics rendering, to be executed on a GPU. General-purpose computation on GPUs (GPGPUs) has becoming a widely adopted architectures to build high-performance computing and big data systems, e.g., 86 out of 102 systems in TOP500 were built using GPGPUs as accelerators, as of November 2017 [2].

Despite such rapid evolution, we are facing yet another revolutionary change: an integrated GPU, i.e., CPUs and GPUs are integrated into the same package or even into the same die. Such an integrated processor architecture offers several advantages such as low energy consumption, low platform cost, and low overhead communication between a CPU and a GPU. The design of this integrated architecture is being adopted by both mainstream computing devices ranging from smart phones to entry-level server machine, and high-performance computing systems.

The availability of such an integrated platform has been encouraging application developers to rewrite their applications (in a whole or the computation-intensive parts) that utilize the integrated GPU for further performance and energy efficiency improvement. Unfortunately, the rewriting efforts are often non-trivial, and also only certain computation patterns were able to exhibit performance improvement on GPUs. GPU application domain is still largely limited to 3D graphics rendering and traditional high-performance computing algorithms such as dense linear algebra [3–7]. For a system with integrated CPU–GPU architectures, if few or no GPU-tuned applications are available [8–10], the GPU will often remain idle, not contributing to the overall system performance.

In this paper, we are interested in how to utilize the integrated GPU to accelerate conventional operations that are normally performed by the CPUs, thus to improve application and overall system performance and resource utilizations without changing or rebuilding the application codes. In particular, we are interested in improving the performance of workloads that run on the server machines of commercial data centers. Besides computation requirements, these applications typically process a large volume of data, and devote large percentage of their processing time to I/O and manipulating the data. For example, a memcached server [11] that is heavily used to implement in-memory database for fast query processing in a commercial search engine is also heavily occupied by processing a large volume of data.

For such systems, we propose to offload bulk memory operations such as memcpy or memset to an integrated GPU. We have observed the following characteristics of bulk memory operations and of GPU memory access patterns, and believe that those operations are very good candidates for GPU accelerations:

– Bulk memory operations, e.g., memcpy, if processed in parallel byte by byte (or other size), are embarrassingly parallel and SIMD-friendly. They have no diverging control flow, highly favorable for GPU SIMD style processing [12];

- Bulk memory operations typically access memory in a well-coalesced manner instead of a scatter-gather or random access pattern;
- Unlike a CPU, which has a relatively strong memory ordering restriction, a GPU has much more relaxed memory ordering model. So it is much easier for a GPU to exercise the high memory bandwidth than for a CPU, thus for possible higher resource utilization in a system; and
- Unlike a CPU whose memory requests are throttled by many constraints such as load/store queue size and miss status handling register (MSHR) size, each thread in a GPU can issue memory requests independently, which results in much higher in-flight memory request counts.

In this paper, we demonstrate these characteristics for memory operations by profiling several applications that are commonly used in data center servers. We study different design trade-offs in offloading the bulk memory operations to an integrated GPU. The performance results, using eight CPU benchmark applications, prove the effectiveness of using GPU offloading and caching for the bulk memory operations.

The paper is organized as follows: Section 2 describes background of bulk memory functions, and architectural design of offloading the bulk memory operations to GPU-like SIMD cores integrated with a high-performance CPU. Section 3 presents a simulation environment for the designed solution of accelerating CPU bulk memory operations using integrated GPU-like components. Simulation results and analysis are explained in Sect. 4. Some related works are discussed in Sect. 5, and the final conclusions of the paper are presented in Sect. 6.

2 Related work

The prior arts can be categorized into three categories including, (i) support for bulk memory copy [13–15]; (ii) DMA or a hardware copy engine [16–19]; and (iii) pioneer work of leveraging on-die integrated GPU to run prefetch shader program for improving application performance [20].

Many machine architectures support specific vector or vector-extension instructions that can operate on wide data [13,14,21]. Vector loads/stores can improve memory operation efficiency by reducing code size and access to TLB. It can work on data as wide as the width of the vector registers. However, vector loads/stores are unsuitable for scalar data that are not aligned at vector width. Typically, instruction support for performing a large-region data movement can be implemented using complex instruction set (CISC) instruction format. However, such instructions do not address issues such as parallelizing memory movements, or reducing pressure on cache performance.

Researchers have proposed different approaches to improve memory operations in the past [16–19,22–25]. DMA engines [16] are the most common architectures to accelerate memory movement. Most of the DMA researches focused on copy avoidance and copy acceleration. In the early days, the researchers injected I/O data into processor's cache directly which incurs low efficiency. The recent Intel I/OAT [26] offers an asynchronous copy engine which improves the copy performance with very little startup costs when compared with the traditional DMA engines. By adopting many optimizations, their approach is still limited by the overheads from copy engine, page locking, and synchronization. From the performance perspective, the schemes seem less efficient when dealing with small region of data. Zhao et al. [19] modeled a bulk data copy engine by extending an execution-driven simulator. However, their methods suffer from overhead in communication between the copy engine and CPU. Later, they proposed a new architecture support for bulk memory operation called FastBCI [17]. Compared with their previous architecture, FastBCI does not always require flushing at the blocks before copying starts and does not stall all dependent loads/stores until the operations are completed. Ahn et al. [22] proposed a programmable processing-in-memory (PIM) accelerator called Tesseract for efficiently performing large-scale graph processing in main memory. By putting the computation units into main memory, it efficiently utilizes memory bandwidth and reduces latency and energy overhead due to data movement. Pattnaik et al. [24] proposed a PIM-assisted GPU architecture to reduce performance and energy penalties due to data transfer by placing GPU core close to main memory. This architecture includes a way to automatically identify which code segments are offloaded to the GPU core in memory and how to schedule multiple code segments concurrently. Hsieh et al. [23] proposed a method to enable computation offloading and data mapping without causing difficulties for the programmers in the key challenges in a 3D-stacked memory architecture that solves the bandwidth bottleneck by directly connecting the logic layer and the DRAM layer. This method includes a compiler-based technique that automatically identifies the code to be offloaded to the logic-layer GPU and a technique to predict which memory page the offloaded code accesses. Seshadri et al. [18,25] proposed mechanisms that perform bulk copy, initialization, and bitwise AND/OR operations in DRAM. However, the mechanisms based on hardware have restrictions on the operation extension.

For taking advantage of the on-die programmable GPU, Woo and Lee proposed COMPASS [20], with which one can use an idle GPU as a programmable prefetcher. In this work, a hardware or a low-level software can learn a memory access pattern of a CPU application and launch an appropriate programmable shader to a GPU so that the shader code can predict future access patterns. While both COMPASS and our proposal are aimed to use a GPU to improve the memory performance of a CPU, our proposal is actively offloading bulk memory movement operations in the CPU user and kernel space, which are extremely memory-intensive and embarrassingly parallel, to a GPU.

3 Architecture and design

3.1 Bulk memory operations

3.1.1 Application profiling

Although CPU speeds have been increasing at a steady rate, memory speeds and latencies have not kept up and the gap is widening over time. The problem may become worse in the multi-core scenario because the aggregated memory workload

Table 1 Number of calls to the bulk memory functions

Function call count				
	Clamav	Gzip	Sphinx	Snort
memcpy	2,469,527	9,357,066	593,669	3,026,936
memmove	5266	0	295	21,044
memset	1,227,980	10	9	2,049,679
strcpy	1,116,329	1234	1460	64
strncpy	1,142,766	0	0	71,691
strcmp	31,380	16,288	201,747	7539
bzero	0	0	0	0
k_memcpy	0	0	0	0
k_memset	0	0	0	0
k_memmove	0	0	0	0
k_memcmp	1	2	3	1
k_strcpy	94	0	0	0
k_strncpy	1	0	0	0
k_strcmp	1978	9994	1129	1358
k_ctu	5,963,186	9,358,364	595,974	5,170,014
k_cfu	5,961,964	9,358,383	595,516	5,169,954

demands even higher memory system performance. Consequently, applications that require lots of data movement usually do not scale well with the increased CPU frequency or number of cores. Such applications are common in the server domain.

Our profiling studies of popular memory-bound applications show that functions relying on bulk memory operations are frequently used. Such bulk memory operation functions can be found in both the user and kernel space. They include both functions that support copy of data from one region to another region such as the GNU libc memcpy, memmove, strcpy, kernel function copy_from_user, copy_to_user, and functions that set a memory region to a constant such as the GNU bzero and memset. Our studies indicate that for many applications, execution of those functions is a dominating performance factor, and often the bottleneck for improving performance.

Table 1 shows number of times bulk memory functions [27] are called during execution of four benchmark applications in Ubuntu Linux using an Intel Xeon processor. Detailed explanation of these applications and data inputs can be found in Table 2. As indicated by the results, memcpy, memset, copy_from_user, and copy_to_user are frequently used by the studied benchmarks. In terms of execution time, we measured percentages of execution time spent on both the user space and kernel bulk memory operation functions for eight benchmark applications, see Table 2 for the application details. The results are shown in Fig. 1. The results suggest that in seven of the eight applications, over 50% execution time is for running these bulk memory operation functions. For application Sphinx, although the percentage is about 20%, the amount

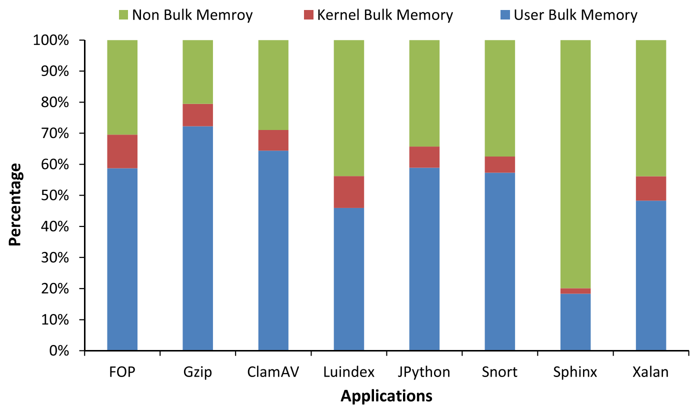


Fig. 1 Profile of bulk memory function execution time (percentage over total execution time)

of data moved is large because majority of the function calls are used to move data larger than 1KB.

3.1.2 Challenges of accelerating the bulk memory operations

Several studies were performed to speedup the bulk memory operations by designing specific vector or vector-extension instructions that can operate on wide data [13, 14,21]. One example is the Intel SSE (version 1 to 4) instruction extension [13]. Designing instructions to support bulk memory operations can address the granularity inefficiency problem by reducing the excessive number of TLB accesses and cache accesses incurred during data movement. Instead of working up the TLB once for each page involved, each load and store instruction incurs one TLB access. However, as pointed out in [17], such approach alone cannot effectively solve the performance bottleneck associated with the bulk memory operations for the following reasons.

Firstly, although bulk memory copy instruction supports the copy to be performed without occupying much processor resources, unfortunately, while the copy instruction itself is executed, the code that follows the copy instruction may not benefit because of stalls in the processor pipeline and reorder buffer, the so-called ROB-clog bottleneck described in [17]. Because the bulk copy instruction may take a long time to complete, it will likely stay in the ROB of the CPU pipeline for a long time. Since instruction retirement is in order, succeeding instructions cannot retire either, and eventually the pipeline will become full, at which point no new instructions can be fetched.

Secondly, studies show that performance of the bulk memory operations is heavily affected by how the codes that follow the copy behave [17]. In particular, cache access patterns to the copied data can play a deterministic role in performance. If the copied data in the cache are not used by the succeeding instructions, the cache will be polluted. On the other hand, if the subsequent instructions access the copied data and result in cache hits, it will benefit from the prefetching effect. Ideally, if the bulk memory movement can produce all the beneficial prefetching effect and at the same time avoid cache pollution, the performance improvement would be dramatic [17]. However, making such intelligent decisions is extremely hard for real-world applications in a realistic setting.

3.1.3 CPU–GPU heterogeneous multi-core

Recent studies and trend show that heterogeneous multi-core architectures provide significant power and performance advantages. For example, AMD's fusion targets at integrating general-purpose CPU and GPU into a single die, which presents an heterogeneous multi-core design with additional graphics-purposed units and other functions of modern GPUs (like GPGPU computation). Intel's Kaby Lake micro-architecture [28] also integrates a CPU with a modern GPU (e.g., Intel HD Graphics 630). Current Kaby Lake GPU supports 24 EUs (execution unit), and the future Kaby Lake may have more EUs.

The two extremes over the spectrum of heterogeneous cores are out-of-order (OOO) cores on conventional CPUs, and simple, massive number of in-order cores on the GPUs. While the OOO cores can dedicate more area to leverage ILP, the GPU cores remain simple. Under equivalent area, the GPU can have more cores to take advantage of the data parallelism in certain applications such as graphics rendering. However, for many non-graphical applications, the GPU, while occupying a substantial part of the silicon, will sit idle and contribute nothing to the overall system performance when running non-graphics workloads or applications lack of data-level parallelism.

Since the introduction of programmable GPUs, many applications have been ported to the GPU to benefit from the massively parallel execution capability of modern GPU. In the early days, discrete GPUs were used for running data-parallel application kernels. However, the long latency between CPUs and a peripheral discrete GPU device restricts certain types of workload to benefit from the GPUs. Heterogeneous CPU–GPU multi-core removes this limitation and opens the door for many exciting new possibilities. To leverage the GPU resource for conventional CPU applications on an integrated system, we propose a novel design and architecture to offload the bulk memory operations to the on-die GPU cores by harnessing the programmability of GPUs and the shared memory architecture between CPU cores and GPU cores. By using the idle GPU for bulk memory operations, we can improve the memory performance of applications that cannot benefit from the GPUs directly. To our best knowledge, our solution is the first one to offload bulk memory operations frequently used in many applications to integrated programmable GPUs.

Different from the DMA-based [17] bulk memory support, offloading bulk memory operations to integrated GPU-like cores has certain advantages. First, in a server environment, the integrated GPU is most likely idle. The solution leverages already available idle resources instead of adding new hardware resources. Second, offloading operations to the integrated programmable GPU is more flexible than using a DMA engine with fixed functionality. For example, the GPU can be programmed to support manipulation during bulk data movement such as endian swap.

As the CPUs and GPUs are integrated in a heterogeneous multi-core to share overlapping workloads, more applications can benefit from offloading computation to the on-die GPU cores. CPU cores and GPU cores can work complimentarily because they perform best under distinct workloads—latency-driven versus throughput-driven workloads, respectively. Additionally, when GPUs and CPUs are increasingly intertwined, a significant portion of CPU and GPU hardware resources can be merged to achieve better area utilization and lower energy consumption. The communications between the GPUs and CPUs can also be improved significantly over time for on-die GPUs. It is envisioned that future integrated CPU–GPU processor will integrate high-performance OOO cores, with throughput-driven GPU-like, lightweighted cores on the same chip. The CPUs and GPU cores can share memory using coherent caches connected through an on-chip interconnect.

3.2 CPU integrated with GPU-like SIMD cores

In this section, we will describe our baseline architecture, which will be useful for readers to understand different design trade-offs that we will describe in the following sections later. Our baseline architecture is a heterogeneous platform that has multiple CPU cores and GPU cores on a chip as shown in Fig. 2. These heterogeneous processing units have different characteristics. First, a CPU core is a traditional, superscalar out-of-order core that is optimized for reducing the latency of each instruction through many architectural techniques such as branch prediction, data forwarding, and speculative instruction scheduling. Such a CPU core has a private instruction L1 cache and a private data L1 cache.

On the other hand, a GPU-like SIMD core is a highly multithreaded, wide SIMD core that is optimized for improving the overall throughput rather than latency. As a result, it does not support fancy speculation techniques. Rather, it is designed to consume less power and less area such as a multi-banked register file (rather than a multi-ported register file of an out-of-order core) to improve the overall throughput

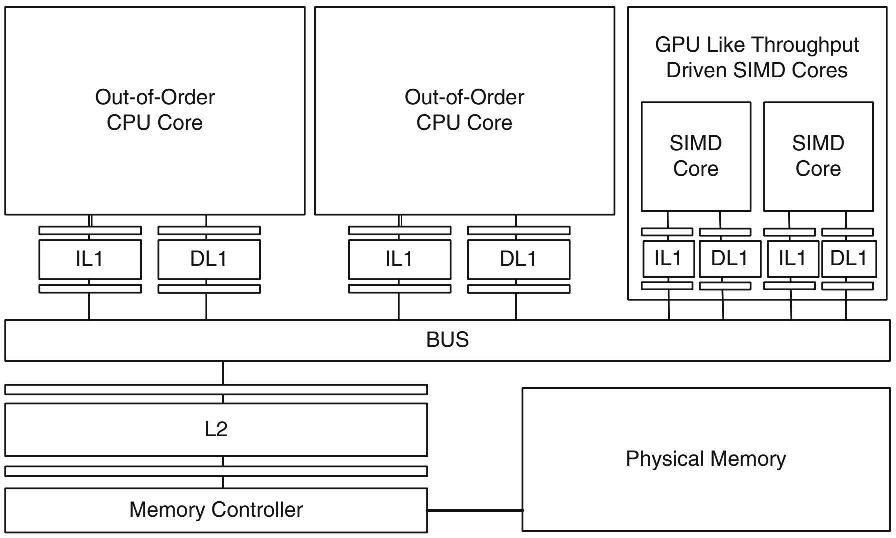


Fig. 2 CPU integrated with throughput-driven GPU-Like SIMD cores

under a given power and area budget. In particular, a GPU core groups a large number of threads (for example, 256 threads) into one scheduling unit, termed a “warp,” and executes the same instruction over those threads simultaneously. Furthermore, because a GPU core supports heavy multithreading among those warps and because it supports many more relaxed consistency models, it can support much more in-flight memory instructions than a CPU core, which goes through a private instruction and data L1 cache as shown in Fig. 2. Note that different threads of a warp can issue non-coalesced memory requests. Our GPU L1 data cache models performs memory coalescing.

In our baseline, CPU cores and GPU cores share several on-die resources. First, private L1 caches of both CPU and GPU cores are connected through an on-die bus sharing its bus bandwidth. Second, the on-die shared bus is connected to a large, shared L2 cache as shown in Fig. 2. Lastly, CPU and GPU memory requests that miss in the L2 cache as well as dirty cache lines that need to be written back from the L2 cache go through a shared memory controller. In addition to such shared resources, both CPU and GPU share the off-chip bus bandwidth and main memory space.

3.3 Bulk memory operation offloading architecture

3.3.1 Parallelizing bulk memory functions

Many applications rely on the GNU C Standard Library (libc) implementation of bulk memory operations such as memcpy, memset, memmove, bzero, and strcpy. The kernel has its own implementation of bulk memory operation functions such as copy_from_user and copy_to_user that are frequently used in I/O control and device drivers. For specific processor model, it is common to provide a customized GNU C Library tailored for achieving the best performance on the targeted machine architecture. For a heterogeneous CPU–GPU processor, vendors can supply customized kernel modules and libc implementation that in runtime offload bulk memory operations to the integrated GPU SIMD cores when the functions are linked with the user space application or kernel codes.

After being called, the GPU aware memory copy function can decide whether the operations should be offloaded to the GPU cores. Often this decision can be based on the size of data movement. When deciding that the data movement operations should be offloaded, the GPU aware library will parallelize the data movement by creating multiple independent GPU threads and dispatch them to the integrated GPU cores. Compared with the CPU threads, GPU threads are much more lightweight and can be created with very little overhead. Furthermore, this process can be accelerated using a hardware engine that takes size, source, and destination addresses of the data movement, and automatically creates the parallel GPU threads.

A large copy must be split up into independent parallel operations, each of which has its own distinct source, destination, and length with contiguous physical memory regions. Since for the targeted CPU–GPU multi-core, we assume that GPU cores do not support virtual memory and TLB for reduced overhead, the source and destination addresses from a virtual address space must be translated by using the TLB before they are sent to the GPU cores. If the translation incurs a TLB miss, it may require

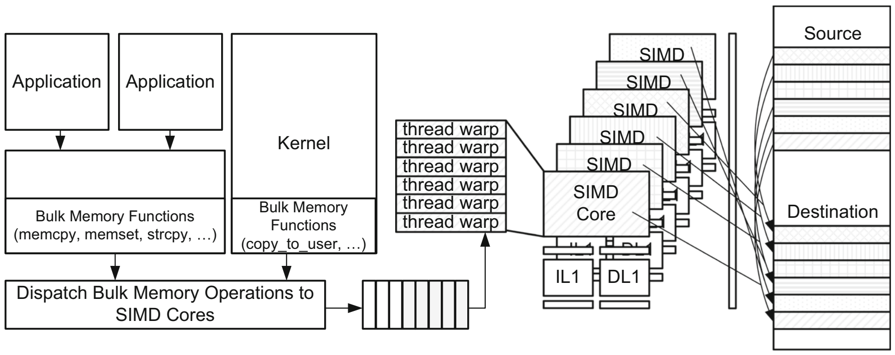


Fig. 3 Offload bulk memory functions to the GPU-like SIMD cores

a page walk that is already supported by the operating system. If the memory copy region crosses a page boundary, then the operation must be broken into smaller sub-data movement operations, each of which only involves maximally one source page and one destination page. As a result, our solution limits the maximum region size of one GPU memory copy thread to be the size of a page.

Once the translation and parallelization processing are completed, each resulting data movement task is dispatched to the GPU cores as shown in Fig. 3. A hardware task queue is used for buffering the created data movement tasks sent from the CPU to the GPU. When there are multiple processes and multiple CPU cores that may send tasks to the GPU simultaneously, task creation and communication between the CPU and GPU can be ordered and/or atomic to ensure that there is no data movement hazard and the operation is thread-safe. A GPU thread can be created for each data movement task. For the multiple GPU cores, the threads can be dispatched to the individual GPU core where they will be bundled together as thread warps and executed. A thread warp can contain multiple threads. The GPU core will schedule the thread warps and dispatch them to the SIMD execution units.

The GPU cores and SIMD engines are designed and best optimized for throughput-driven workload. When threads in a warp access the memory, all threads stall until all of their requests are fulfilled. At the same time, the core switches to another warp to overlap the memory access. Requests from threads in the same warp are sent to the L1 cache at the same time. The GPU cache is non-blocking (MSHR-capable) and multi-banked so that the cache can accept more requests from other warps if the current warp is experiencing cache misses.

According to our observation, different strides for each thread can produce different performance. For different data movement size and thread numbers, we tested different stride sizes in order to find out the optimal value. The threads in the same warp will take consecutively WarpSize*Stride bytes memory space as their workload. The bulk memory movement function in the GNU libc and kernel can be rewritten with the optimal setting for a specific CPU–GPU architecture.

When bulk memory operations are executed by the GPU cores, the CPU has several options. In one option when the data movement size is large, the process can yield CPU resources to another process or thread, and later wake up when the memory operations are completed by the GPU cores. This option can increase the overall throughput of the system in a multi-process and multi-thread workload environment. The second option is to wait for the GPU cores to complete. Though simple to implement, CPU resources will be wasted under this option. A third option is to support asynchronous bulk memory operation functions. After tasks are dispatched to the GPU cores, the function can return immediately to the caller. The caller can continue to execute until it attempts to access the memory space with pending GPU memory operations.

Because a copy operation is broken down into many threads and executed in parallel, it is possible to track whenever a sub-copy task completes by the GPU cores, this will enable asynchronous memory movement at finer granularity. Subsequent instructions in the caller after the memory movement function call can proceed as long as they either access not involved memory space or touch sub-regions of the involved memory space where the movement operations have been completed by the GPU threads.

The overhead associated with dispatching tasks from the CPU to the GPU can be optimized and reduced by using a hardware dispatch engine that carries the aforementioned operations. In the context that the CPU and GPU are integrated on the same die, closely coupled, and share memory space, the time taken for creating and dispatching tasks to the GPU can be made minimal with continued optimization.

3.3.2 Accelerating access to the copied data using SIMD caches

In addition to leveraging the throughput-driven GPU SIMD cores by parallelizing the bulk memory operations and offloading them to the GPU, our solution also explores the GPU caches so that they can be used by the CPU to achieve even higher performance. In a CPU–GPU heterogeneous processor, when the GPU cores are not executing threads, their private caches will stay idle. In the case of 4 on-die GPU SIMD cores, each with its own private cache, integrated with the CPU, the amount of wasted silicon resources is significant. In the envisioned architecture where GPU cores and CPU cores are interconnected through on-chip networking or shared bus, the GPU core caches can be used by the CPU cores. In particular, after GPU cores complete data movement operations for the CPU and stay idle, the copied data can be found in the GPU core private caches. The succeeding instructions on the CPU after the memory movement may access the copied data or the source, and the accesses could result in cache misses. If the data are present in the GPU core private caches, in addition to fetching the data from the next memory hierarchy, the data can also be retrieved from the GPU core private caches. This way, private caches of the GPU SIMD cores are effectively used as the prefetch buffer for the CPU. In the ideal situation, the effective cache size for the CPU will be increased and the memory access latency of the moved data can be reduced as well hitting the GPU private caches.

In [17], when designing a dedicated copy engine for the bulk memory operations, the authors made the observations that the overall performance of bulk memory movement is heavily affected by how the instructions that follow the data movement behave. Sometimes, their impact can be even bigger than the efficiency of the data movement operations themselves. When conditions are met, the subsequent instructions can benefit from the prefetching effect of loading data into the CPU cache assuming that cache pollution is not introduced by storing the data in the CPU cache. In their design,

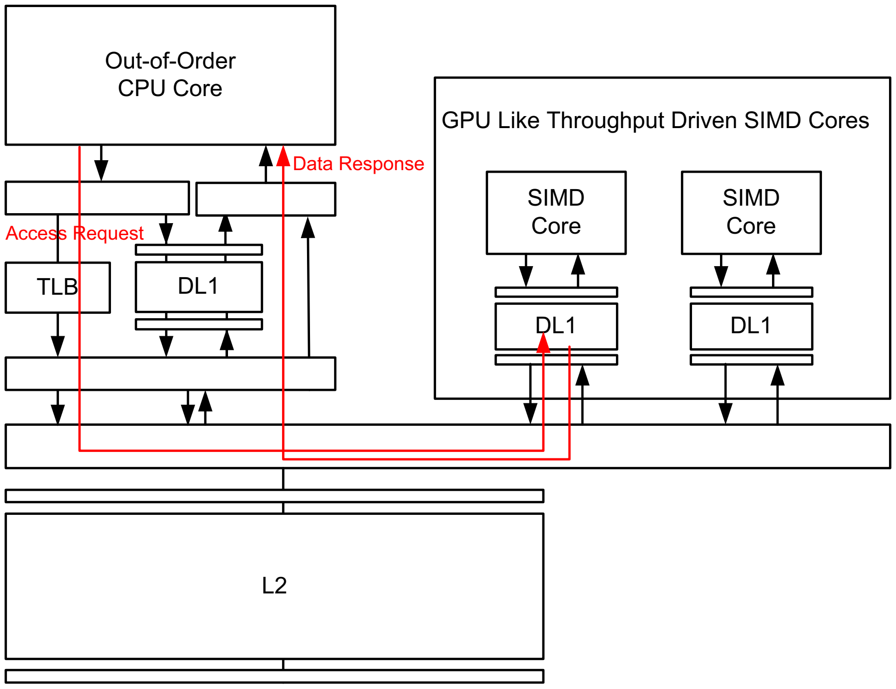


Fig. 4 Leveraging GPU core private caches to buffer bulk movement data for the CPU

because only CPU cache is involved, for each piece of copied data, the copy engine or the user program has to make a delicate decision when the data should be cached so pollution can be avoided and prefetch effect will be achieved. Such a decision is often very hard to make without lots of prior knowledge of the applications. In our design, since private caches of the idle GPU cores are used (the GPU cores will stay idle after completing the data movement tasks) for storing prefetched data, the benefit is almost free from the CPU perspective and the design is much simpler.

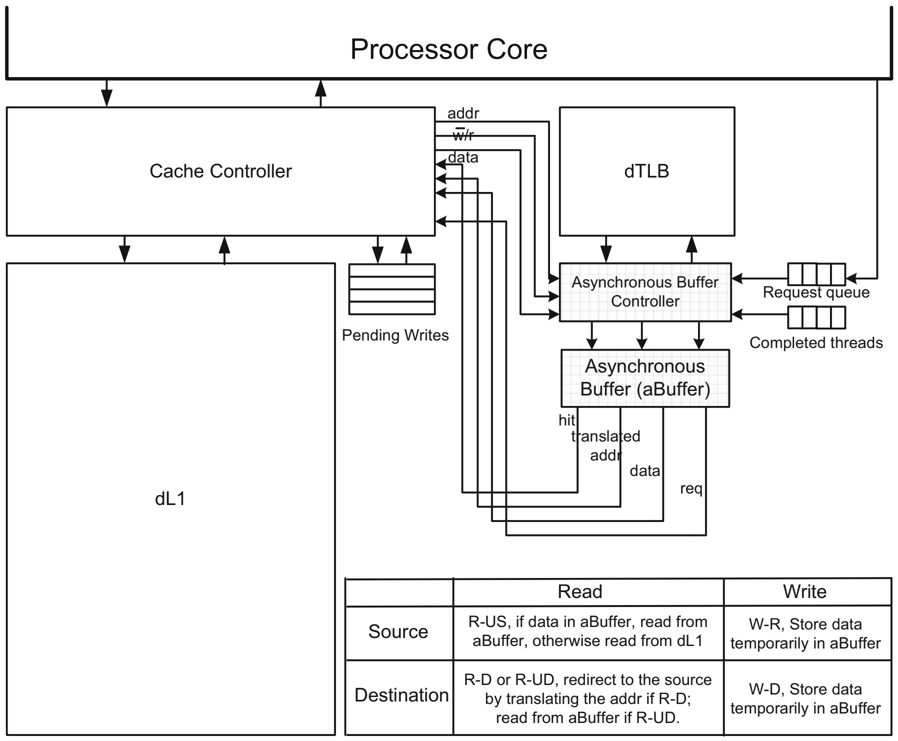
The cache controller of the GPU cores connects to a shared command bus with the CPU cache, see Fig. 4. This shared bus connects with a unified L2 or L3 cache (i.e., in Sandybridge, GPU and CPU share L3 cache). When the CPU fetches data from the unified cache, it will broadcast the request on the shared bus. Cache controllers of the GPU cores snoop the shared bus and can pick up the request from the CPU core. Upon receipt of a request from the CPU core, it will check if the requested data can be found in its private cache by matching the tags. If the data are found, the GPU core cache controller will respond to the request. Accordingly, the CPU core can grab the data from the responding GPU cache. To improve performance further, a CPU core can send the request on the shared bus in parallel with access of the CPU L1 cache. Since the GPU cache uses physical addresses in our design, address of the accessed data has to be translated before the request is posted on the shared bus.

Both small and large data movement can benefit from the GPU caches. In the previous subsection, we descried that small size data movements are not offloaded to the GPU because there is no speed advantage of doing so. However, when private GPU caches can be exploited by the CPU, there is real benefit of offloading small size data movement to the GPU cores so the GPU caches can be used as data buffer by the CPU core. To achieve this benefit, in our solution, when the size of data movement is small, both CPU and GPU will perform the operation. This way, small data movement can be completed in the CPU speed and still be able to benefit from the prefetch effect of the GPU caches. Because the data size is small, the extra memory bandwidth requirement incurred by running these operations is also small. Further, because the CPU and GPU share L2 or L3, the actual front side bus traffic overhead is even less.

3.3.3 Asynchronous offload

It is plausible to support bulk memory operation offload in an asynchronous manner. If designed properly, asynchronous offload of bulk memory operations to the GPU-like SIMD cores can significantly improve performance. In asynchronous offload mode, execution returns to the caller of a bulk memory operation function as soon as possible before the offloaded function completes. Doing so can surely increase performance. However, some kind of interlock or mechanism is required to handle possible data hazards gracefully. Using memcpy as an example, likely hazard scenarios include, read or write access to a location within the destination address range, write access to a location within the source address range. For maximizing performance, asynchronous buffer (aBuffer) as shown in Fig. 5 is designed to keep track hazard occurrence and handle them gracefully. The asynchronous buffer maintains a list of monitored source and destination memory regions. For each memory access to data L1 cache, the cache controller will look up both the data L1 cache and the asynchronous buffer. If the location accessed falls into the range of a source region and it is a write access, a write-to-source (W-S) hazard is detected. If the location accessed falls into the range of a destination region and it is a read access, a read-from-dest (R-D) hazard is detected. If it is a write access, the hazard is write-to-dest (W-D) hazard. Additional hazard cases include read access on source location or destination location with updated contents (read-updated-source, R-US, read-updated-dest, R-UD). The asynchronous buffer can detect all the five hazard scenarios, W-S, R-D, W-D, R-US, and R-UD. To handle the hazard gracefully without undermining performance, the asynchronous buffer does the following, (i) for R-D hazard, the asynchronous buffer translates the address and redirects it to the corresponding location in the source range; (ii) for W-S and W-D hazards, the update is allowed and the updated states are temporarily kept in the asynchronous buffer and flushed to the memory hierarchy after the related bulk memory operation is completed by the GPU-like SIMD cores; and (iii) for R-US and R-UD types of hazards, when detected, the buffered value will be retrieved from the asynchronous buffer and returned to the processor core. If a source or destination region is modified and the new values are kept in the asynchronous buffer, the buffered values will be flushed when the asynchronous buffer controller receives the notification that the related bulk memory operation is completed. The asynchronous buffer controller raises a request to the main cache controller.

For tracking outstanding offloaded bulk memory operations, there is an option of tracking granularity. A larger granularity will allow more aggressive tolerance of



	Read	Write
Source	R-US, if data in aBuffer, read from aBuffer, otherwise read from dL1	W-R, Store data temporarily in aBuffer
Destination	R-D or R-UD, redirect to the source by translating the addr if R-D; read from aBuffer if R-UD.	W-D, Store data temporarily in aBuffer

Fig. 5 Asynchronous offload support

hazards. However, it takes longer time for a monitored address region to stay in the asynchronous buffer, and larger tracking granularity requires more resources as well. Smaller granularity has lower overhead, and the tracked entries do not occupy the asynchronous buffer for too long. In our design, the granularity is set to be the same size of dL1 cache line size. Consequently, as shown in Fig. 6, a large offloaded bulk memory operation will be split into multiple tracking entries in the asynchronous buffer.

Each entry in the asynchronous buffer has a valid bit that indicates if the entry is currently occupied or not. There are two base addresses, one for the source location and the other one for the destination location of a tracked bulk memory operation. Both addresses are aligned and truncated based on the size of dL1 cache line size. For certain bulk memory operations such as bzero, only the destination address is used. For each address, there is a bit vector for indicating the bytes that are tracked within a cache line size memory space. This field is needed because bulk memory operations may not use addresses aligned at cache line boundaries. For an outstanding offloaded bulk memory operation, if a location at the source or destination is modified, the new values are temporarily buffered. For handling R-US and R-UD hazards, there are two additional bit vectors for tracking modified locations for the source and destination memory spaces. Given a memory location to be accessed, two comparators are used to compare it with the base source address and the base destination address. Offset of

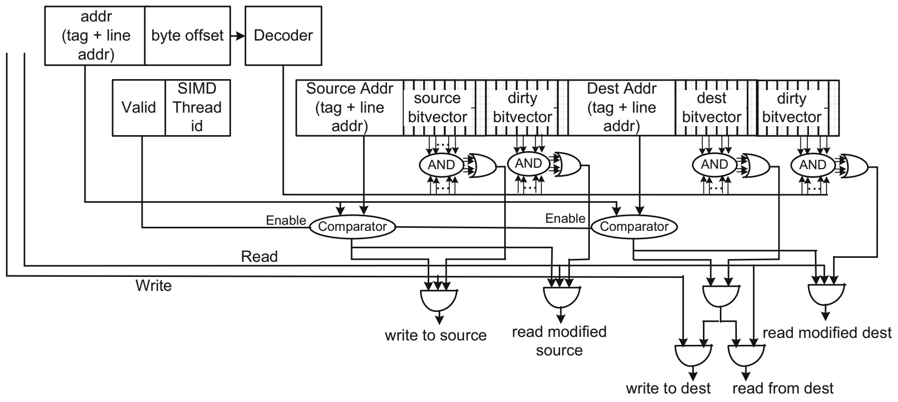


Fig. 6 Asynchronous buffer entry

the accessed memory location is decoded and compared with the bit vectors. Using the mechanism shown in Fig. 6, all the hazards can be detected. For R-D hazard, the address will be translated and redirected to the corresponding source location.

A cache controller does dL1 lookup and asynchronous buffer lookup simultaneously. The asynchronous buffer responds by raising a hit signal if one of the hazard scenarios is detected. The main cache controller will consider the outcome of the asynchronous buffer and act accordingly. For example, if both dL1 and the asynchronous buffer return hit event, and there is a R-D, the main cache controller will issue a new memory access using the translated address to nullify the previous read request. In another example, if R-US or R-UD is detected, the main cache controller will use the data fetched from the asynchronous buffer. The number of entries in the asynchronous buffer is small, typically 4 to 8 entries. So the overhead is very small. For example, assuming 12 bit thread id, 128 bytes for buffering new values for updated source and destination locations, and 8 asynchronous buffer entries, the total storage overhead is 3,720 bits.

3.3.4 Energy efficiency

Offloading bulk memory operations to simple GPU-like cores offers an attractive solution from both performance and energy consumption perspectives. As indicated by the results in Sect. 4, for bulk memory operations, the throughput-driven in-order SIMD cores can outperform OOO CPU cores using less die area. Two GPU-like threaded cores can deliver about the same performance as two OOO CPU cores. Furthermore, the solution does not add new SIMD cores just for the purpose of supporting bulk memory operations. Our targeted scenario is that a decision is already made to integrate a multi-core GPU with a CPU (e.g., AMD fusion). In MV5 simulator, a SIMD core is much simplified OOO core, which means that the SIMD core has less logic units and smaller area.

4 Evaluation method

4.1 Simulation environment

We conducted the experiments on a desktop server with Intel Core i7-7700K processor @ 4.2 GHz, 64GB 2400MHz DDR4, and Ubuntu 16.04 as the Linux distribution. To evaluate our solution, we used a functional full-system emulator, Simics [29], combined with a cycle-based architecture simulator that can model a heterogeneous multi-core. FeS2 is a timing-first, multiprocessor, x86 simulator, implemented as a module for Simics. FeS2 [30] supports accurate execution-driven timing model that includes cache hierarchy, branch predictors, and a superscalar x86 out-of-order core. The memory model is provided by GEMS [31]. FeS2 can decode x86 instructions into uops. Implementation of the uops is based on [32]. MV5 [33] is a cycle-based architecture simulator for heterogeneous multi-core such as a processor integrating OOO CPU cores with large number of GPU-like array-style SIMD cores. MV5 models a cache-coherent memory hierarchy shared by the CPU and GPU cores. In MV5, the GPU SIMD cores can be programmed in an OpenMP-style API implemented in MV5, where data parallelism is expressed in parallel for loops. The GPU version of memory movement operations is programmed using MV5's programming support. The programming model launches multiple copies for SIMD execution. Neighboring tasks are assigned to threads in the same warp in a way that exploits both spatial and temporal data locality [33]. We combined MV5 with FeS2 where the GPU is simulated using MV5's infrastructure and the x86 workload is simulated by FeS2. With modification, FeS2's memory system uses the coherent memory model implemented in MV5. In full-system simulation, we apply the described bulk memory operation offloading to the GNU libc and the kernel. When used for comparison, OOO bulk memory operation performance was collected from optimized GNU libc library.

4.2 Benchmarks

We simulated a set of benchmarks using inputs shown in Table 2. These benchmarks are common, memory-intensive applications. They cover the application domains of scientific computing, document processing, intrusion detection, compression, and data mining. They demonstrate varied data access and communication patterns.

FOP [34] takes an XSL-FO file, parses it, and formats it, generating an encrypted PDF file. It is a part of Dacapo benchmark set, which is a standard test for real-world applications with non-trivial memory loads [35]. JPython [34] interprets the pybench Python benchmark. Pybench is a collection of tests that provides a standardized way to measure the performance of Python implementations. It has been used in the past by several Python developers to track down performance bottlenecks or to demonstrate the impact of optimizations [36]. Luindex [34] uses Lucene to index a set of documents. Lucene is a Java search engine library. Xalan [34] transforms XML documents into HTML. Xalan is also one of the tests used by Dacapo. ClamAV is an open source (GPL) antivirus engine designed for detecting Trojans, viruses, malware, and other malicious threats. It is the de facto standard for mail gateway scanning [37]. Snort is

Table 2 Dataset used for the benchmarks

Benchmark	Description
ClamAV	Official ClamAV test files with infected files [41]
Snort	DARPA training data of weeks 1 and 2 [42]
Sphinx	The OHSUMED test collection of 348,566 references from MEDLINE (an online medical information database) [43]
Xalan	Dacapo Xalan inputs [34]
JPython	Pybench benchmark [34]
Gzip	enwik8 from large text test benchmarks [44]
FOP	Dacapo FOP test inputs [34]
Luindex	Works of Shakespeare and the King James Bible [34]

an open source network intrusion prevention and detection system (IDS/IPS). Snort is the most widely deployed IDS/IPS technology worldwide [38]. GNU zip (gzip) is a popular compression utility [39]. It is based on the DEFLATE algorithm. Although unlike many newer utilities, gzip is single-threaded, we chose to use it as it remains de facto standard for open source platforms. Sphinx [40] is a search engine designed for indexing database content. Sphinx is of particular interest, because it has a significant number of relatively large memcpy operations, with 2454 bytes being average.

4.3 Machine parameters

The simulation is performed with a 4-wide out-of-order superscalar processor with 2 GHz frequency and x86 ISA.

The processor uses a bimodal branch predictor. It has a 32-entry load/store queue, 128-entry reorder buffer, and non-blocking caches with 16-entry MSHR. The I-TLB and D-TLB have 64 fully associative entries. The L1-instruction and L1-data caches are 32KB write-back caches with 64-byte block size, and an access latency of 2 cycles. The L2 cache is unified, non-blocking, 2MB size, 16-way associativity, 128-byte block size, and has an 10-cycle access latency. The memory access latency is 150 cycles, and the memory bus width is 128-bit. The number of GPU in-order SIMD cores is configurable. It can be set from 1 to 4. The maximum number of threads per core is 256. The GPU SIMD core L1 data cache is 16KB, 4-way associative, write-back with 64-byte block size, and an access latency of 2 cycles. The GPU L1 cache has 4 banks. Access latency from the CPU core to the GPU SIMD core L1 caches takes 5 cycles. The asynchronous buffer has 8 entries, and the tracking granularity is 64 bytes. Asynchronous buffer hit can be resolved in 2 clock cycles. A memory location can be translated by the asynchronous buffer in 3 clock cycles. The asynchronous buffer performs address translation and comparison in parallel. All the benchmark applications were tested with the input data described in Table 2. The simulation started when the application passed the initialization stage (using Simics checkpoint support). The cycle-based simulation executed each benchmark application for one billion instructions.

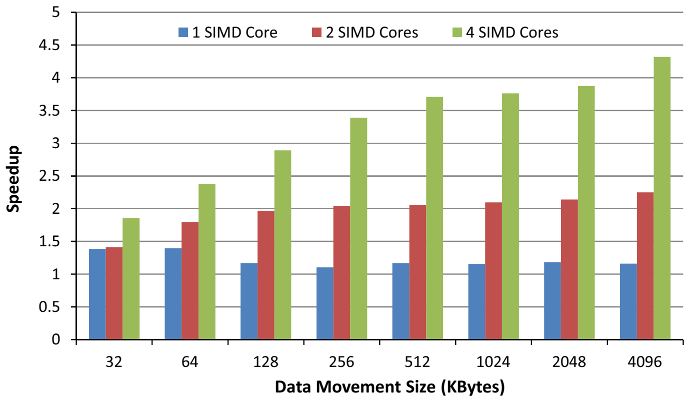


Fig. 7 Speedup with number of cores for different data movement sizes

5 Performance analysis

As first step, we evaluated the performance gain of running memory movement operations on the GPU SIMD cores against the performance of running the same operations on the out-of-order CPU core. For a thorough evaluation, we experimented with configurations of different number of GPU cores. The three settings we studied were, one, two, and four GPU cores.

5.1 GPU versus CPU micro-benchmark

To compare thoroughly the raw performance of memory movement on GPU versus CPU, we explored the design space by varying the number of GPU SIMD cores employed for running bulk memory movement and the size of the data movement. For large size workload, more GPU cores will increase the performance, see Fig. 7. For example, when the data size is 4MB, the speedup using 4 GPU cores is 4.3 times. However, the performance over CPU benefit decreases when the size of workload decreases. For 64KB, the speedup is 2.3 times. Typically, for large size workload, more GPU cores will result in improved performance.

Our experiments suggest that the GPU memory performance is sensitive to the size of stride. Figure 8 shows GPU versus CPU speedup by varying stride sizes and number of GPU cores. The workload size is 256K. As indicated by the results, large stride size usually leads to increased performance. For example, with 4 GPU cores, using 1K stride, the speedup is almost 3.4 times faster than that of 32B stride. The reason behind is how well the cache access pattern of a warp matches with the GPU cache organization. For example, a request of 8 $\times$ 128 bytes from a warp is more efficient than that of requesting 2 $\times$ 128 bytes (stride 32B) 4 times.

When running memory movement operations employing large number of threads on the GPU, not all the execution time is spent on moving the data. There are overheads such as thread queueing and warp scheduling. We studied the overheads, and the results indicate that the overhead percentage in terms of execution time is actually small.

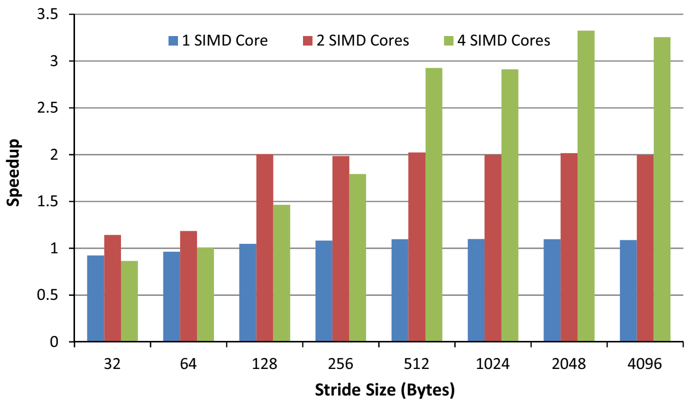


Fig. 8 Speedup under different stride sizes

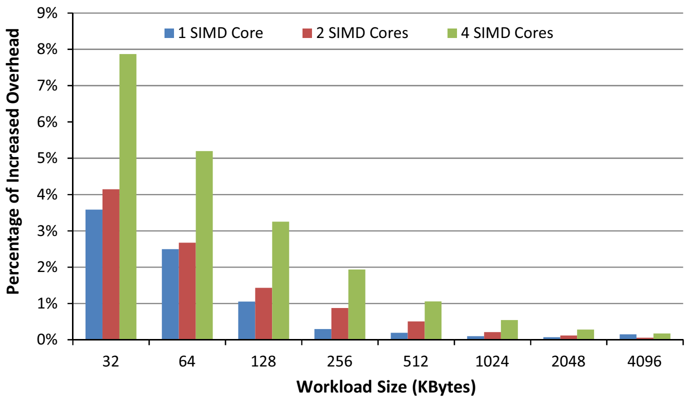


Fig. 9 GPU execution overhead

Figure 9 shows the results under different number of GPU cores and workload sizes. As indicated by the results, more GPU cores will increase the overhead. However, the percentage of increased overhead also depends on the workload size. Our studies also show that increasing stride size can reduce the overhead, especially when the workload size is large.

One interesting exploration is to compare performance between parallelizing bulk memory operations to multiple OOO CPU cores and offloading the operations to throughput-driven GPU-like SIMD cores. We experimented to run bulk memory operations on two OOO CPU cores using SMP middleware API support. The results are compared with the SIMD offloading performance, see Fig. 10. For large size data movement, two OOO CPU cores deliver about the same performance as two throughput-driven in-order SIMD cores. The four SIMD core setting outperforms the two OOO core setting by a very significant margin. It is worth to point out that the in-order threaded SIMD cores are typically much smaller in size than the OOO

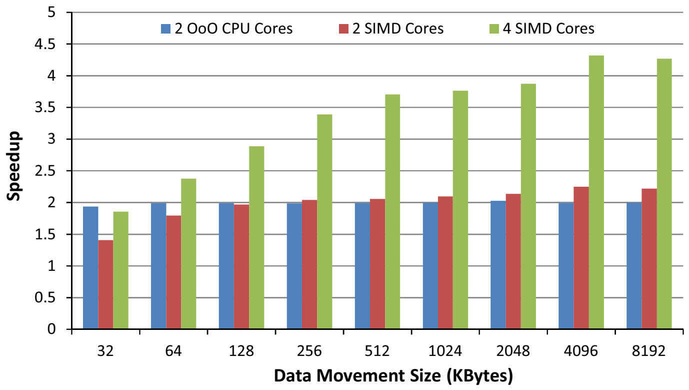


Fig. 10 Comparison between offloading versus parallelization over multiple OoO cores

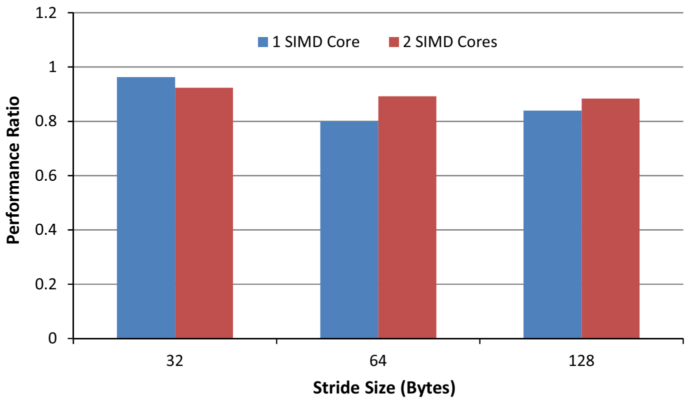


Fig. 11 SIMD clock rate

CPU cores and consequently consume much less energy. The advantage is due to the throughput-driven thread execution model.

Moreover, the GPU-like SIMD cores can run at lower clock frequency. Figure 11 shows performance change for copying 32KB data under three stride settings after the clock speed of the SIMD cores is lowed by one-third. As shown in Fig. 11, performance degradation depends on the number of cores and stride size. For two SIMD cores, the performance reduces to about 90% for the three stride settings. For the single SIMD core case, the performance degradation is slightly larger. However, it is still able to retain 80% of the performance. When more GPU-like SIMD cores are used, it provides more tolerance to frequency drop. Such tolerance is due to the throughput-driven and threaded execution model of the SIMD cores. Therefore, it supports more energy efficient offload of bulk memory operations.

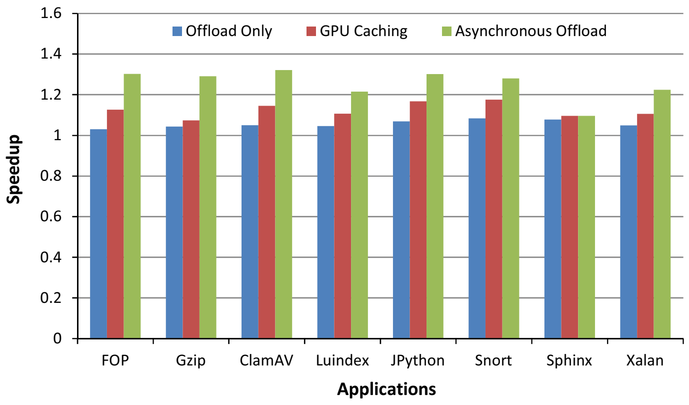


Fig. 12 Speedup of offloading bulk memory operations to the integrated GPU-like cores

5.2 Application performance

We conducted full-system simulations for the eight application benchmarks. In the full-system simulation, we applied the described bulk memory operation offload solution to the GNU libc and the Linux kernel (kernel version 2.6.32.31). All the benchmark applications were tested with the input data described in Table 2. The simulation started when the applications passed the initialization stage (using Simics checkpoint support). The cycle-based simulation executed each benchmark application for one billion instructions. The simulated CPU–GPU processor has four on-die GPU SIMD cores.

Firstly, we evaluated the speedup of offloading the bulk memory operations to the on-die GPU with the GPU caching effect turned off. This scenario evaluates the application performance improvement by only accelerating the bulk memory operations using the GPU. The results are shown in Fig. 12. As indicated by the results, Snort and Sphinx have speedup of 8.3% and 7.8%, respectively. The speedup for JPython is 7%. The rest of applications, FOP, Gzip, Luindex, Xalan, and ClamAv, have speedup from 3 to 5%. Sphinx has more speedup than many other applications because it often invokes copy of large data size.

Secondly, we evaluated the performance effect of both the GPU offloading and GPU caching for the bulk memory operations under the eight benchmark applications. The results are shown in Fig. 12, as the middle speedup bar in results for each application. For ClamAV, JPython, and Snort, the speedup is about 17%. The rest five applications have speedup from 10% to about 13.6%. The average for all the eight applications is more than 13%.

Further, the asynchronous support for offloading bulk memory operations to the GPU-like cores is also evaluated using the described settings. When asynchronous support is turned on, one can observe additional performance improvement. The results are shown as the right speedup bar for each of the eight applications in Fig. 12. For five applications, FOP, Gzip, ClamAV, JPython, and Snort, the overall speedup is more

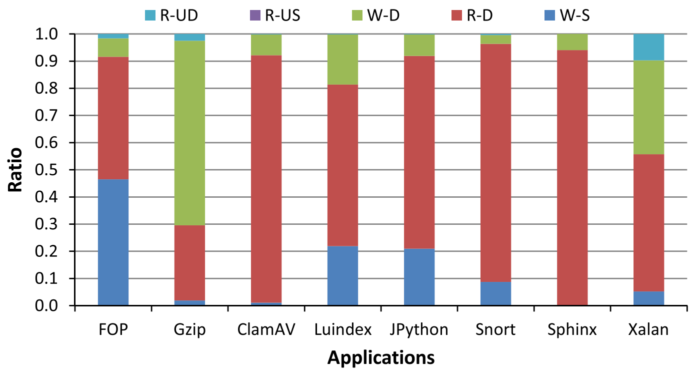


Fig. 13 Profile of accesses in asynchronous mode

than 30%. For Luindex and Xalan, the overall speedup is more than 20%. The only application that does not show additional speedup is Sphinx.

In addition, we collected, during asynchronous offloading mode, distributions of the different hazard cases. The results are plotted in Fig. 13. As indicated by Fig. 13, reading from destination locations (R-D) is the dominating case for almost all the applications. For three applications, reading from destination locations represents close to 90% of all the hazard cases. The second frequent scenario is writing to the source locations. In three applications, Luindex, JPython, and FOP, this scenario occurs more than 20% of the times. Furthermore, we also experimented with a setting of 4 entry asynchronous buffer and compared with the setting of 8 entries, the results have no significant change. Only when the number of entries is reduced further, changes can be observed. Since the hardware overhead of 4–8 entries is already very small, there is no point minimizing the number of asynchronous buffer entries.

6 Conclusions

In this paper, we proposed and presented a novel architectural support to improve the bulk memory operation performance for the CPU applications in a CPU–GPU heterogeneous multi-core processor. With very lightweight system and architectural support, our solution can take advantage of the most likely idle on-die GPU to improve the performance of bulk memory operations.

From our studies, we found that offloading bulk memory operations to the GPU provides three main advantages. Firstly, the throughput-driven GPU with multi-thread support can perform bulk memory operations more efficiently than the CPU. Secondly, for on-die GPU with a unified cache between the GPU cores and CPU, the private caches of GPU cores can be leveraged by the CPU for storing moved data and reducing the CPU cache performance bottleneck. Furthermore, with lightweight hardware support, support for asynchronous offload to the GPU can be enabled.

We evaluated and analyzed our design using a cycle-based full-system simulator that supports a detailed architecture model of a CPU–GPU heterogeneous multi-core system. Our simulation results show that, for memory movement micro-benchmarks, the speedup over CPU can be as much as 4.3 times depending on the workload sizes and the GPU configurations. Evaluation based on eight popular CPU benchmark applications with real-world input dataset shows that we can significantly improve the overall performance of those applications. When offload is combined with GPU caching and asynchronous support, the speedup is more than 30% for five, and more than 20% for two of the eight real-world applications.

Acknowledgements This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1D1A3B03933370).

References

1. Lee J, Liu Z, Tian X, Woo DH, Shi W, Boumber D, Yan Y, Kwon KA (2012) Acceleration of bulk memory operations in a heterogeneous multicore architecture. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. ACM, pp 423–424
2. The 50th TOP500 list (2017). https://www.top500.org/lists/2017/11/. Accessed 4 Sept 2018
3. Benziane SH, Benyettou A (2017) Dorsal hand vein identification based on binary particle swarm optimization. J Inf Process Syst 13(2):268–283
4. Finogeev AG, Parygin DS, Finogeev AA (2017) The convergence computing model for big sensor data mining and knowledge discovery. Hum Centric Comput Inf Sci 7(1):11
5. Ghadekar PP, Chopade NB (2016) Content based dynamic texture analysis and synthesis based on SPIHT with GPU. J Inf Process Syst 12(1):46–56
6. Koo KM, Cha EY (2017) Image recognition performance enhancements using image normalization. Hum Centric Comput Inf Sci 7(1):33
7. Mohd-Hilmi MN, Al-Laila MH, Malim H, Ahamed NH (2016) Accelerating group fusion for ligand-based virtual screening on multi-core and many-core platforms. J Inf Process Syst 12(4):724–740
8. Hao F, Min G, Pei Z, Park DS, Yang LT (2017) k-clique community detection in social networks based on formal concept analysis. IEEE Syst J 11(1):250–259
9. Hao F, Pei Z, Park DS, Yang LT, Jeong YS, Park JH (2017) Iceberg clique queries in large graphs. Neurocomputing 256:101–110
10. Song W, Liu L, Tian Y, Sun G, Fong S, Cho K (2017) A 3D localisation method in indoor environments for virtual reality applications. Hum Centric Comput Inf Sci 7(1):39
11. Memcached—a distributed memory object caching system (2015). http://www.memcached.org/ Accessed 4 Sept 2018
12. Fung W, Sham I, Yuan G, Aamodt T (2007) Dynamic warp formation and scheduling for efficient GPU control flow. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp 407–420
13. Intel streaming SIMD extensions technology (2017). https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html. Accessed 4 Sept 2018
14. Nvidia CUDA (2007). https://developer.nvidia.com/cuda-zone. Accessed 4 Sept 2018
15. Intel advanced vector extensions 512 (AVX-512) (2015). https://www.intel.com/content/www/us/en/architecture-andtechnology/avx-512-overview.html. Accessed 4 Sept 2018
16. Gschwind M (2006) Chip multiprocessing and the cell broadband engine. In: Proceedings of the 3rd Conference on Computing Frontiers, CF ’06. ACM, New York, NY, USA, pp 1–8
17. Jiang X, Solihin Y, Zhao L, Iyer R (2009) Architecture support for improving bulk memory copying and initialization performance. In: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, Washington, DC, USA, pp 169–180
18. Seshadri V, Mutlu O (2017) Simple operations in memory to reduce data movement. In: Hurson AR, Milutinovic V (ed) Advances in computers, vol 106. Elsevier, New York, pp 107–166
19. Zhao L, Bhuyan LN, Iyer R, Makineni S, Newell D (2007) Hardware support for accelerating data movement in server platform. IEEE Trans Comput 56:740–753

20. Woo DH, Lee HHS (2010) Compass: a programmable data prefetcher using idle GPU shaders. In: Hoe JC, Adve VS (eds) ASPLOS. ACM, New York, pp 297–310
21. Abts D, Bataineh A, Scott S, Faanes G, Schwarzmeier J, Lundberg E, Johnson T, Bye M, Schwoerer G (2007) The Cray BlackWidow: a highly scalable vector multiprocessor. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07. ACM, New York, NY, USA, pp 17:1–17:12
22. Ahn J, Hong S, Yoo S, Mutlu O, Choi K (2016) A scalable processing-in-memory accelerator for parallel graph processing. ACM SIGARCH Comput Archit News 43(3):105–117
23. Hsieh K, Ebrahimi E, Kim G, Chatterjee N, O'Connor M, Vijaykumar N, Mutlu O, Keckler SW (2016) Transparent offloading and mapping (tom): enabling programmer-transparent near-data processing in GPU systems. ACM SIGARCH Comput Archit News 44(3):204–216
24. Pattnaik A, Tang X, Jog A, Kayiran O, Mishra AK, Kandemir MT, Mutlu O, Das CR (2016) Scheduling techniques for GPU architectures with processing-in-memory capabilities. In: Proceeedings of the 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT). IEEE, pp 31–44
25. Seshadri V, Lee D, Mullins T, Hassan H, Boroumand A, Kim J, Kozuch MA, Mutlu O, Gibbons PB, Mowry TC (2017) Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, pp 273–287
26. Vaidyanathan K, Chai L, Huang W, Panda DK (2007) Efficient asynchronous memory copy operations on multi-core systems and I/OAT. In: Proceedings of the 2007 IEEE International Conference on Cluster Computing, CLUSTER '07. IEEE Computer Society, Washington, DC, USA, pp 159–168
27. Kernighan BW, Dennis M (1988) The C programming language. Prentice-Hall, Upper Saddle River
28. 7th generation Intel core and Celeron desktop processor families with Intel H110 and Intel Q170 chipsets: platform brief (2017). https://www.intel.com/content/dam/www/public/us/en/documents/platformbriefs/7th-generation-core-processor-deskop-iot-platform-brief.pdf. Accessed 4 Sept 2018
29. Magnusson P, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B (2002) Simics: a full system simulation platform. Computer 35(2):50–58
30. Neelakantam N, Blundell C, Devietti J, Martin MM, Zilles C (2008) FeS2: A full-system execution-driven simulator for x86. In: Proceedings of the Architectural Support for Programming Languages and Operating Systems. ASPLOS 2018
31. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput Archit News 33:92–99
32. Yourst MT (2007) PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2007
33. Meng J, Skadron K (2009) Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In: Proceedings of the 2009 IEEE International Conference on Computer Design, ICCD'09. IEEE Press, Piscataway, NJ, USA, pp 282–288
34. Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B (2006) The dacapo benchmarks: java benchmarking development and analysis. SIGPLAN Not 41:169–190
35. DaCapo benchmark suite. http://dacapobench.org/. Accessed 4 Sept 2018
36. Pybench. http://svn.python.org/. Accessed 4 Sept 2018
37. ClamAV open source antivirus engine. http://www.clamav.net/. Accessed 4 Sept 2018
38. Koziol J (2003) Intrusion detection with Snort, 1st edn. Sams, Indianapolis
39. Gzip. http://www.gzip.org/. Accessed 4 Sept 2018
40. Sphinx text search server. http://sphinxsearch.com/. Accessed 4 Sept 2018
41. ClamAV test files. https://packages.ubuntu.com/xenial-updates/utils/clamav-testfiles. Accessed 4 Sept 2018
42. MIT Lincoln Laboratory 1998/1999 DARPA off-line intrusion detection (1999). https://www.ll.mit.edu/rd/datasets. Accessed 4 Sept 2018
43. TREC-9 filtering track collections (2007). http://trec.nist.gov/data/t9_filtering.html. Accessed 4 Sept 2018
44. Large text compression benchmark (2009). http://cs.fit.edu/~mmahoney/compression/text.html. Accessed 4 Sept 2018

Affiliations

JongHyuk Lee[1] · Weidong Shi[2] · JoonMin Gil[3]

JongHyuk Lee
jonghyuk@cu.ac.kr

Weidong Shi
larryshi@cs.uh.edu

1 Department of Big Data Engineering, Daegu Catholic University, Gyeongsan-si, Republic of Korea

2 Department of Computer Science, University of Houston, Houston, USA

3 School of Information Technology Engineering, Daegu Catholic University, Gyeongsan-si, Republic of Korea