Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems

Tayler H. Hetherington† Timothy G. Rogers† Lisa Hsu§ Mike O'Connor§ Tor M. Aamodt†

†Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
{taylerh,tgrogers,aamodt}@ece.ubc.ca

§Advanced Micro Devices, Inc. (AMD)
{Lisa.Hsu,Mike.OConnor}@amd.com

***Abstract*—The recent use of graphics processing units (GPUs) in several top supercomputers demonstrate their ability to consistently deliver positive results in high-performance computing (HPC). GPU support for significant amounts of parallelism would seem to make them strong candidates for non-HPC applications as well. Server workloads are inherently parallel; however, at first glance they may not seem suitable to run on GPUs due to their irregular control flow and memory access patterns. In this work, we evaluate the performance of a widely used key-value store middleware application, Memcached, on recent integrated and discrete CPU+GPU heterogeneous hardware and characterize the resulting performance. To gain greater insight, we also evaluate Memcached's performance on a GPU simulator. This work explores the challenges in porting Memcached to OpenCL and provides a detailed analysis into Memcached's behavior on a GPU to better explain the performance results observed on physical hardware. On the integrated CPU+GPU systems, we observe up to 7.5X performance increase compared to the CPU when executing the key-value look-up handler on the GPU.**

***Index Terms*—GPGPU, SIMD, OpenCL, key-value store, server**

I. Introduction

With the introduction of programmable functional units, graphics processing units (GPUs) have expanded to general-purpose, high-performance computing (HPC) applications. These types of applications are known to efficiently utilize the underlying GPU hardware to maximize performance. When performing an initial analysis on an application, many programmers may look for characteristics similar to existing applications with proven performance on GPUs. This may lead a programmer to disregard applications that appear to deviate from these characteristics. This bias may eliminate from consideration some applications that actually might perform well on a GPU.

One family of highly parallel and economically appealing applications that initially may not appear to gain performance benefits on GPUs are traditional server applications. HPC applications have been shown to attain good performance on GPUs [11]; however they represent a relatively small segment of the overall computing market[1]. Large-scale server applications represent a larger class of applications, but one that is unstructured. In this paper, we study one such application and show arguably good performance results. These positive results are achieved by taking advantage of the GPU's memory bandwidth and numerous functional units to overcome the application's unstructured behavior. More specifically, we characterize and evaluate Memcached [18], a widely used key-value store application, and show it is a strong candidate for acceleration on certain classes of GPUs.

[1]According to IDC, in 2009 the overall server market had revenues of \$43.2 billion [15] versus \$8.6 billion for HPC servers [14].

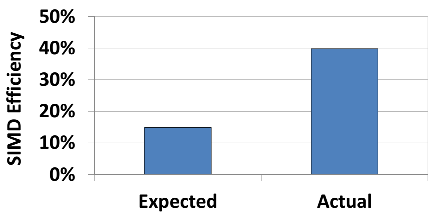


Fig. 1. Memcached SIMD Efficiency: Expected vs. Actual

Memcached was originally designed for LiveJournal, but has since been extended to many other large-scale web applications such as Facebook, YouTube, Wikipedia, Flickr, Twitter, and others [18]. Memcached is a memory-intensive application. On a discrete GPU with isolated memory spaces, large data transfers are required between the CPU and GPU to keep a coherent view of the memory on both the host and device. Assuming Memcached's requests are independent and that each request can be handled by a separate thread, a common expectation might be that each thread exhibits independent and irregular behavior.

Initial analysis into Memcached shows its memory access patterns and execution behavior depend on the input data. In a single-instruction/multiple-data (SIMD) architecture, such as a GPU, groups of threads must execute instructions together in lock-step. This suggests that Memcached might not fit the SIMD model because threads might often execute different instructions based on their input data. SIMD efficiency represents the fraction of scalar threads that execute together in lockstep per cycle. Given these properties, Memcached may appear to have poor SIMD efficiency on a GPU. Figure 1 presents a comparison between Memcached's actual SIMD efficiency ("Actual") and the SIMD efficiency if all code paths were equally likely ("Expected"). On average, Memcached's actual SIMD efficiency is approximately 2.7X higher than a naive assumption about code-path suggests. These results are explained in greater detail in Section V.

The contributions of this paper are:

- It contrasts a programmer's intuition of an application's potential execution behavior on a GPU with the actual behavior. Using Memcached and two other examples, we show that the appearance of irregular control-flow patterns do not directly result in negative performance on a GPU.
- It describes the methodology used to modify Memcached to run on a GPU.
- It analyzes Memcached's performance on AMD Fusion^{TM} CPU-GPU architectures and compares it to discrete GPU architectures.
- To provide deeper insight, it evaluates the performance of Memcached on GPGPU-Sim [1].

The rest of this paper is organized as follows: Section II presents the programming model and baseline architecture used in this study, and discusses the common performance-affecting features on a GPU and how Memcached is impacted, Section III describes how Memcached was ported to the GPU, Section IV describes the methodology and environment used to perform this study, Section V presents a characterization and evaluation of Memcached on hardware and GPGPU-Sim, Section VI describes related works, and Section VII concludes.

II. GPU Architecture and Background

AMD GPU architectures are used in this study, running OpenCL applications. OpenCL is a C-like programming model that provides an application programming interface (API) to interact with GPUs. The compute kernel[2], is a user-defined parallel section of code that runs on the GPU. Execution begins on the host with communication of data and commands on the device taking place in one or more command queues. Here, the host refers to the CPU and the device refers to the GPU. The host then launches a compute kernel on the device, specifying the appropriate hierarchy of threads required by the compute kernel. In OpenCL, this hierarchy consists of an NDRange of work groups of wavefronts of work items. In AMD hardware, each work item is logically equivalent to a scalar thread. The wavefronts group multiple work items, which execute the same instruction on different data streams. The work groups group multiple wavefronts, providing consistency within the work groups. The NDRange provides an indexing space that specifies the breakdown, in terms of size and dimension, of work items into work groups. Figure 2 shows a high-level view of the GPU architecture used in this study [3].

The following sub-sections discuss a few of the main GPU features that need to be considered by a programmer to achieve high performance.

[2]For the remainder of the paper, "kernel" and "compute kernel" are used interchangeably.

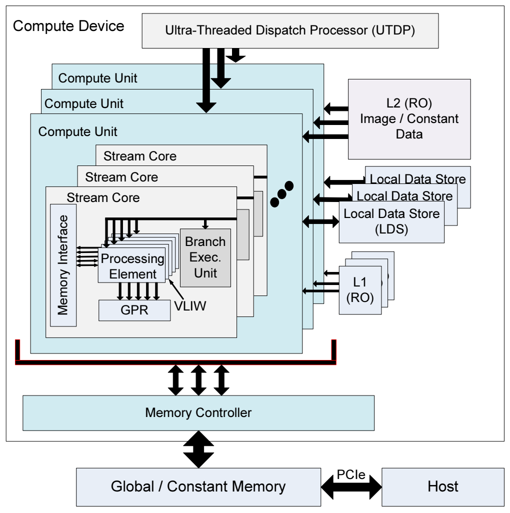


Fig. 2. High-level View of AMD's Baseline Architecture

A. Control Flow

As already mentioned, work items are grouped together into wavefronts. In an application without any conditional control flow operations, such as conditional branches, the wavefront is able to make forward progress throughout the code while executing instructions for each work item in parallel. However, if conditional branches are introduced into the code, it is possible for a sub-set of the work items in a wavefront to take the branch while the remaining work items do not. This is known as branch divergence. To handle branch divergence, a hardware component, similar to the SIMT stack described by Fung et al. [22] [23] can be used to track the active work items at various points throughout the program's execution. An active work item refers to a work item within a wavefront that is currently executing instructions. A work item becomes inactive if it takes a branch that diverges away from the other work items in the wavefront.

Each entry on the SIMT stack contains a bitmask representing the active work items in a wavefront, with the top element in the stack (TOS) signifying the sub-set of work items to execute. The SIMT stack also records the current program counter (CPC) and a re-convergence program counter (RPC). The CPC specifies the instruction that the active work items on a corresponding stack entry will execute once it becomes the TOS. As the work items in the TOS entry execute the instructions, the CPC is incremented accordingly. The other counter, the RPC, specifies the immediate post-dominator (IPDOM) instruction. The IPDOM is defined as the closest point in the program that all paths leaving the branch must go through before exiting the function. The SIMT stack uses the RPC to specify where the active and inactive work items can rejoin.

Figure 3 shows an example of the SIMT execution flow when executing a piece of code taken from the hash function in Memcached. It is also annotated with the actual branch

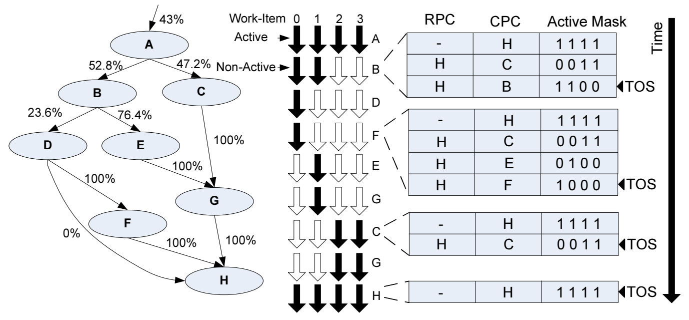


Fig. 3. SIMT Execution Example

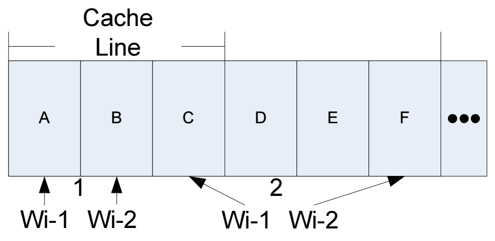


Fig. 4. Memory Request Coalescing

probabilities using the same input data sets as in Section V. In this example, there are four work items per wavefront. Snapshots of the SIMT stack are shown at different points throughout execution on the right.

A work item is represented by a vertical lane in the middle of Figure 3 (labeled 0 through 3). Active work items are the black arrows, and inactive work items are the white arrows. The ovals represent basic blocks, which signify a portion of code with single entry and exit points that ensure no further branch divergence. Without branch divergence, the minimum number of blocks required to reach H from A is four ($A \rightarrow C \rightarrow G \rightarrow H$) and the maximum is five ($A \rightarrow B \rightarrow D/E \rightarrow F/G \rightarrow H$).

At the beginning, all of the work items in the wavefront are set to active and execute the instructions at block A. At the first branch, work items 0-1 go to block B while work items 2-3 go to block C; the corresponding entries are pushed onto the stack. The re-convergence point for each work item is set to the IPDOM block H. This is shown by the top stack expanding from B. The next step is to execute a sub-set of the original wavefront until reaching a re-convergence point. Execution switches to work items 0-1 at B; here, the work items split again, removing its current entry and pushing two new entries onto the stack. Work item 0 executes until reaching the re-convergence point H before allowing work item 2 to execute until reaching H. Work items 2-3 now become active and execute until H. The work items are able to resume concurrent execution once all work items reach the re-convergence point H.

Assuming all the basic blocks have the same number of instructions, the SIMD efficiency in this example is approximately 50% and executes nine blocks in total. Also, block G is executed twice, resulting in more instructions being issued than necessary.

Thus, to achieve the highest performance from the GPU, it is desirable to have as little branch divergence as possible.

B. Memory Access

Memory accesses are another property of GPUs directly affected by grouping of work items in a wavefront. If the instruction being executed by all work items is a memory-access operation, such as a load or a store, each work item will generate a memory request to be handled by the memory system. If each memory request is for a data object in a different region of memory, all requests will be handled separately. This phenomenon is referred to as "memory divergence" for the remainder of this paper. However, if the data being requested from different work items lies within a given range, such as the size of a cache line or the size of data returned on a memory request (i.e. high data locality between work items in a wavefront), an optimization can be applied in which each memory request falling into one of these regions can be coalesced into a single request. This can significantly reduce the amount of memory-request traffic on the memory system.

Consider the example in Figure 4 with two work items, Wi-1 and Wi-2, executing two load instructions. On the first load instruction, labeled 1, Wi-1 loads in A and Wi-2 loads in B. Because both data objects lay in a single cache line, the two memory requests can be coalesced into a single request. However, on the second load instruction, labeled 2, Wi-1 sends a memory request for an object separated further in memory than the size of a cache line from Wi-2's memory request. Thus, two memory requests will be generated. Extending this to real applications, typically containing 64 work items per wavefront and hundredss of wavefronts per kernel, un-coalesced memory requests can generate large amounts of memory traffic to the memory system and significantly affect performance.

C. Data Transfer

Significant losses in performance can occur with large data transfers between the host and device [13] because the time required to transfer data increases linearly with the amount of data needing to be transferred [9]. Discrete GPUs have separate, isolated memory spaces from the host that require explicit data transfer. Data communication takes place over a connection bus between the host and device, typically a PCIe bus. On the discrete card used in this study, the AMD Radeon^{TM} HD 5870, data is transferred between the host and device via a PCIe 2.1 x16 bus, with a maximum transfer rate of 8 GB/s [3].

III. Porting Memcached

A. Memcached

Memcached is a general-purpose, high-performance memory caching system used to improve performance of dynamic distributed databases in server applications by caching the recently queried data in main memory. This alleviates the amount of traffic required to query the database and access non-volatile storage or other external sources. These expensive I/O operations can significantly reduce performance and increase power consumption. Main memory from individual servers is combined to create a large pool of virtual memory that is common across all servers. This aggregation of memory effectively provides a much larger memory space that scales linearly with the number of servers in the system.

A simple interface is provided to store, modify, and retrieve data from the distributed hash tables. All of the write and read operations must perform two hashes of the keys. The first hash selects which server the request should be directed to and the second hash selects the appropriate entry in the hash table. Hash-chaining is used in the event of collisions on write operations and a linear traversal of the linked list is used on read operations when necessary. On a read (GET request) hit, in which the key is found in the hash table, the value corresponding to that key is returned to the requesting client. On a read miss, however, the client is notified of the miss and, if applicable, the database is queried for the missing data, which is then stored into Memcached.

B. Changes Required

This section describes the necessary modifications made to Memcached to offload the GET request handler to the GPU, as well as the corresponding changes made to the host to efficiently interact and communicate with the GPU.

In our implementation of Memcached, we focused on accelerating the read requests on the GPU while leaving the write requests to be handled by the CPU. Berezecki et al. observe that read requests far outnumber write requests in real-world scenarios running Memcached in Facebook [7]. They also conducted experiments showing that write requests have negligible effects on read performance. It is reasonable to assume that the read requests will have the most significant impact on performance, and thus have the greatest benefit – in terms of overall system performance – from being accelerated on the GPU.

To take advantage of the massive amounts of available parallelism provided by the GPU, GET requests are batched on the host, passed to the device on kernel launch, and processed in parallel on the device. Each work item in the Memcached kernel handles a single request, which performs common key-value look-up operations. This consists of computing the hash of the request's key, accessing the appropriate entry in the hash table using the resulting hash value, and then comparing the request's key with the key – or multiple keys in the event of hash collisions – residing at that hash table location. If the keys match, the value corresponding to that key is returned to the requesting object. We assume that the requests have already been directed to the correct server, and thus the hash performed on the GPU corresponds to the second hash mentioned in Section III-A.

When Memcached receives a request from a client via the network, it creates a $connection$ object that contains all information required to process any requests during the lifetime of this connection. The client is then able to send requests through this connection to be handled by the server. These connections contain significant overhead when considering the amount of information required to process the request on the GPU, such the protocol and client information. To reduce the amount of data being sent to the GPU, we created a $payload$ data object that contains a sub-set of the connection information required to process a GET request, such as information about the search key and a pointer to the requested item if found. On each GET request, we allocate and assign a payload object to the requesting connection and batch these payload objects to be transferred to the GPU, reducing the overall amount of data to be transferred.

1) Memory Management: To manage all the data allocated and accessed in Memcached, we implemented a dynamic memory manager on the host. This memory manager is used to store all the data that needs to be visible to both the host and the device; it replaces the *malloc* and *free* system calls originally used in Memcached. Depending on the system being used, the allocated buffers reside in different memory regions on the host or device. On the discrete system, the buffers are allocated in the regular host memory space and transferred to the device when necessary. On the AMD Fusion systems, however, these buffers are allocated in pinned memory to take advantage of the zero-copy memory regions where data can be allocated on either the CPU or the GPU and accessed directly by both with varying bandwidth and latencies.

There are two types of zero-copy memory spaces available: the host-visible device memory and the device-visible host memory. These memory spaces are allocated from pinned host memory, a sub-set of the host's memory space, at system boot time. The device-visible host memory is optimized for access by the host, whereas the host-visible device memory is optimized for the device [3]. To minimize the data access time on the GPU, we used the host-visible device memory.

While current AMD hardware shares a physical memory region between the host and device, it does not share a common address space. The implication of this is that the virtual addresses returned by our *mem_alloc* function, corresponding to the physical location in host-visible device memory, is not the same as the address seen on the device, even though it corresponds to the same physical location. Thus, complex data structures consisting of many multiple-level pointers can not simply be de-referenced on the device.

What is common between the host and device, however, is the offset of each memory object from the start of the allocated memory region. Using this property, we pass the virtual address pointing to the start of the memory region seen by the host as an argument to the Memcached kernel and calculate the offset between the this and the start of the memory region seen by the device. A macro is used to subtract this offset from every memory de-reference on the device: $translate(address, offset)$. The inverse operation is applied to all pointers set on the device, such as the return value in a read request. This ensures that both the host and

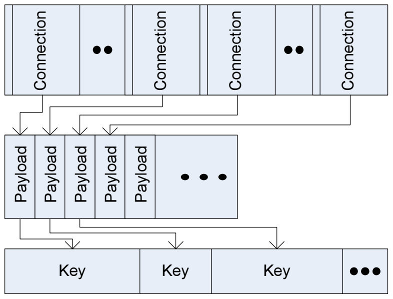


Fig. 5. Contiguous Memory Layout

TABLE I
HARDWARE SPECIFICATIONS

Name	AMD Radeon HD 5870	Llano A8-3850 (AMD Radeon HD 6550D) AMD Fusion	Zacate E-350 (AMD Radeon HD 6310) AMD Fusion
Engine Speed (MHz)	850	600	492
# Compute Units (CU)	20	5	2
# Stream Cores	320	80	16
# Processing Elements	1,600	400	80
Peak Gflops (single-precision)	2,720	480	78.72
# of Vector Registers/CU	16,384	16,384	16,384
LDS Size/CU (kB)	32	32	32
Constant Cache / GPU (kB)	48	16	4
L1 Cache / CU (kB)	8	8	8
L2 Cache / GPU (kB)	512	128	64
DRAM Bandwidth (GB/sec)	153.6	29.9	17.1

device access the same physical memory locations. Gelado et al. [12] implement a similar technique by either ensuring that the virtual pointers returned to the shared memory region are the same or by maintaining address mappings between the host and device. This reduces demands on programmers by eliminating the need to traverse and reconstruct complex data structures that contain multiple pointers on the device, such as linked lists or tree structures.

2) Read-only Data: With the exception of the data structures written to with the result of the GET request, the majority of the data between successive kernel launches is read-only when processing GET requests in Memcached. Current AMD hardware provides various hardware components, such as read-only caches, that can significantly decrease data access time. Where possible, we allocated data in a read-only buffer to take advantage of the buffer's high-bandwidth, low-latency memory access.

3) Memory Layout: Using the dynamic memory manager, we can allocate data in specific layouts to take advantage of the memory-coalescing property discussed in Section II-B. Two data structures guaranteed to be accessed by all work items on the GPU in a known access pattern are the payloads and the keys corresponding to each payload. As introduced in Section III-B, the payload contains a pointer to the requested key, the length of the key, and a pointer to the item being requested. Each work item is assigned a single payload corresponding to a single GET request. Assuming a 32-bit system, the size of each payload is only 12 bytes. With a wavefront size of 64 work items and a cache line size of 128 bytes, these 64 memory requests could be reduced to six to retrieve the same amount of data. Therefore, we ensure payloads are allocated contiguously in memory by allocating a separate dedicated buffer.

To access the payloads, each work item requires only a pointer to the start of the buffer and uses its global work item ID to access the appropriate index in the payload array. This same technique is applied to the keys corresponding to the payloads, such that when each work item dereferences the pointer to the key, it will lie relatively close in memory. Figure 5 shows how the connections, payloads, and keys are allocated in the different buffers.

C. WikiData

We simulated request traffic to our Memcached server using a large input file consisting of read and write requests. Specifically, we used portions of the Wikipedia Workload traces found at [21] to stimulate our application. These workload traces were recorded by Wikipedia's front-end proxy caches and, in total, contain billions of HTTP requests.

Memcached's host code was modified to process the requests from the trace files instead of incoming requests from the network. A configurable number of requests are processed on the CPU, prior to the offloading work to the GPU, to set up the environment and ensure there is a sufficient amount of data stored in Memcached's memory. Once the setup period completes, write requests are handled immediately on the host and read requests are placed into a buffer until the configurable number of read requests are encountered.

IV. METHODOLOGY

A. Hardware

We performed tests on three configurations of GPUs and accelerated processing units (APUs): a high-performance discrete graphics card, a low-power AMD Fusion APU, and a mid-to-high-end AMD Fusion APU. The discrete card was chosen to show potential upper bounds on compute performance, while the low-power AMD Fusion APU provides insight into the performance capabilities of such a system. The mid-to-high-end AMD Fusion APU falls in the middle of these two systems, combining the higher compute performance with the benefits of the APU's shared memory space. The hardware specifications for these GPUs are outlined in Table I.

B. GPGPU-Sim

We used GPGPU-Sim to further analyze Memcached's behavior on a GPU. Although the GPU architecture modeled by GPGPU-Sim differs from the physical hardware analyzed in this study, such as in its use of a VLIW unit, we do not think that this is an issue because the architecture resembles AMD's future architecture [17]. GPGPU-Sim simulates Parallel Thread Execution (PTX) code, a pseudo-assembly language. Table II presents the configurations used in GPGPU-Sim.

TABLE II
GPGPU-SIM CONFIGURATION

# Streaming Multiprocessors	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc.
L2 Unified Cache	512k, 128B line, 8-way assoc.
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	Out of Order (FR-FCFS)
Branch Divergence Method	PDOM [22]
Warp Scheduling Policy	Loose Round Robin
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

C. Control-flow Simulator

We are interested in analyzing an application's control-flow behavior to study: (a) how programmer's intuition compares with reality, and (b) the effect of branch probabilities relative to correlation. To analyze the control-flow for a given application, we designed a stand-alone control-flow simulator that simulates the behavior of a wavefront through an application's control-flow graph. Each branch in the application is annotated with an outcome probability. At each branch, the active work items generate a random number and compare it with the threshold outcome probability at that branch.

This is useful when performing the initial analysis in deciding to port an application to the GPU. Prior to writing any code for the GPU, the programmer can gain better insight into the average SIMD efficiency likely to result on the hardware. The simulator takes three input files: a DOT file containing the information required to generate the control-flow graph, a file containing the number of instructions per basic block (specified in the DOT file), and a file containing the outcome probabilities for each branch in the application. A SIMT stack handles branch divergence and re-convergence, and the simulator measures the overall SIMD efficiency for each iteration through the application's control-flow graph. For a set of given branch probabilities, we ensure SIMD efficiency results converge by averaging a large number of iterations (in this work, 100,000 iterations) through the control-flow graph.

In theory, branch probabilities themselves may not be enough to accurately simulate the SIMD efficiency of an application. Consider an application already ported to run on a GPU. After profiling the execution, it is possible to have a SIMD efficiency of 100% with an outcome probability of 50% at every branch if every work item takes the same path but alternates paths between every execution. This correlation among branches is not implemented in the control-flow simulator and is left to future work. Despite this, by extracting Memcached's actual branch probabilities from GPGPU-Sim and using them as input to the control-flow simulator, we found the estimated SIMD efficiency is within 1.3% of the actual SIMD efficiency of Memcached. One such application that requires branch correlation to estimate the SIMD efficiency accurately is Ray Tracer, which is discussed in Section V-B.

D. Assumptions and Known Limitations

Throughout this study, we assume requests are independent of each other. Thus, all read operations will view the most up-to-date data.

Currently, the size of memory accessible by the GPU and APU is limited. On the APU, each zero-copy buffer can be a maximum of 64 MB, with a system total of 128 MB [3]. This poses various problems for memory-intensive applications, such as Memcached, that require large amounts of memory to be effective. This problem would be eliminated with a larger region of pinned memory available to the GPU and an appropriate interface to allocate and access the additional memory. For example, the industry is moving to address the limited memory capacities available in current graphics cards. AMD announced at the 2011 AMD Fusion Developer Summit (AFDS) that future GPUs and APUs will support accessing CPU virtual memory [10]. We expect future products will allow sharing of arbitrarily large memory spaces between the CPU and GPU cores, eliminating this restriction.

Batching requests inherently adds additional latencies to the system. Offloading requests to the GPU would help reduce system-queuing latency if CPU throughput became the bottleneck when experiencing high incoming request rates. While some applications may not be able to accept the latency impact of batching thousands of requests, we expect we could achieve many of the benefits with smaller batch sizes. Our future work will look at batching fewer requests at a time (e.g., batch a wavefront [or a few] at a time and launch them to some persistent worker threads on the GPU).

We initially profiled Memcached to locate sections of code that bottlenecked performance and would benefit from running in parallel on the GPU. This revealed that the majority of execution time is spent in I/O and handling the network stack. The key-value lookup, although a less significant portion, was the next-highest contributor to the overall execution time of Memcached. Thus, we focused our efforts on porting the key-value lookup handler to the GPU and left optimizing the networking stack for future work.

E. Validation and Metrics

To verify the GPU version of Memcached returns the correct results, we first processed the batch of GET requests on the CPU, and then passed off the same batch of requests to the GPU. On kernel completion, we compared the two results to ensure the same set of items was found. The execution times on the CPU were recorded using a fine-grained time stamp counter (TSC) that records the sequential look-up times for the batch of requests. When timing the CPU, all data was allocated in a cacheable memory region. For each GPU, we recorded the kernel execution times using the AMD APP Profiler tool (v. 2.3). We verified that the comparison with the TSC timer was valid by also timing the kernel execution time on the

TABLE III
CPU HARDWARE SPECIFICATIONS

Name	Llano A8-3850	Zacate E-350
# x86 Cores	4	2
CPU Clock	2.9 GHz	1.6 GHz
TDP	100W	18W
L2 $ / core	1MB	512 KB

Llano A8-3850 system immediately before the kernel launch and immediately after the $clFinish$ synchronization function. Both timing methods output the same values.

V. EXPERIMENTAL RESULTS

At the time of this study, Linux drivers for the AMD Fusion system did not support zero-copy buffers. To access the Windows AMD SDK and the necessary Linux libraries, we used Cygwin [8] to run Memcached on the AMD Fusion systems. One issue with Cygwin is its inability to access all provided hardware counters. This significantly limited the amount and variety of data we were able to collect from Memcached on the hardware. To gain additional information about Memcached's behavior, we profiled Memcached on GPGPU-Sim.

A. Hardware Performance

Memcached was run on the three GPU configurations introduced in the previous section, with the performance measured via hardware counters through AMD App Profiler. Both the AMD Radeon HD 5870 and the Llano A8-3850 (AMD Radeon 6550D) are compared against a single Llano x86 CPU core, while the Zacate E-350 system was compared against a single Zacate x86 CPU core. Table III provides additional information about the CPUs used in this study.

Figure 6(a) presents the average speed-up, in terms of key-value look-ups per second (LPS), for each GPU configuration normalized to the CPU's execution time. Because these results do not include any data transfer times, they explicitly highlight the computational performance benefits when performing a key-value look-up on the GPU relative to the CPU. Even with the irregular control-flow and memory-access patterns present in Memcached, the AMD Radeon HD 5870 is able to perform the key-value look-up on a batch of requests approximately 33X faster, the Llano A8-3850 7.5X faster, and the Zacate E-350 4.5X faster than their CPU counterparts.

Data transfer times result in a large overall performance decrease on the discrete system, as can be seen in Figure 6(b). The APUs, however, have close to 0 transfer time due to the shared memory space. These data transfer times are small but non-zero due to the mapping and unmapping operations. Although the compute power of the APUs is less than the high-performance discrete AMD Radeon GPU, the ability to fully eliminate the transfer of data allows these devices to outperform the AMD Radeon HD 5870.

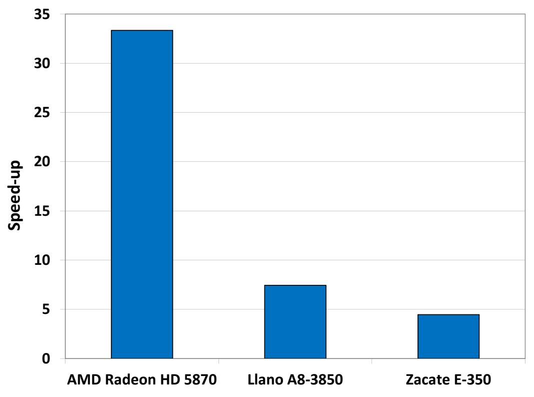
1) Request Batching: Whenever considering batch processing, there is always a trade-off between throughput and latency; as the number of queued requests increases, the time taken to process these requests also increases. We measured these values on the AMD Radeon HD 5870 by varying the batch request size and recording the average time taken to process that batch of requests. Figure 7 presents these results normalized to an initial batch size of 1,024 requests, excluding data transfer times. Also shown in Figure 7(a) is a 0.5-ms latency reference line. Berezecki et al. [7] indicate that a 1-ms delay, including network transfer and TCP processing time, would be the maximum tolerable latency for a request. The large spike in throughput from 1-5X in Figure 7(b) is caused by the minimal increase in latency, approximately 1.3X, while increasing the number of requests/batch by 7.5X (1,024 $\rightarrow$ 7,680). This results in the behavior shown by the throughput, corresponding to the initial increase in latency, which begins to level off and fluctuates between 6X and 7X the throughput at 1,024 requests per batch.

This behaviour suggests that the GPU is underutilized when the request batch size is less than approximately 20,000. Assuming a theoretical incoming request rate can be set to match any level of throughput, selecting a batch size of approximately 8,000 requests per batch qualitatively provides the maximum ratio of throughput to latency.

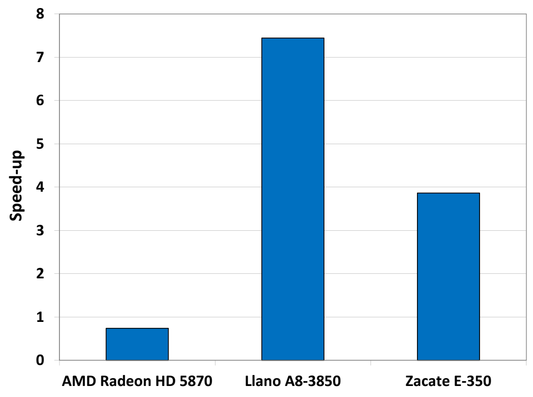
Another property of batch processing to consider is how the performance between the GPU and CPU varies when the request batch size is modified. These results are presented in Figure 8 for both the AMD Radeon HD 5870 and the Llano A8-3850 when compared to a single CPU core on the Llano A8-3850 system. Both architectures show a large initial increase in performance when the batch size is increased between small values. Similar to the throughput behavior seen in Figure 7(b), the speed-up compared to the CPU begins to level out on both architectures at around 40,000 requests/batch.

2) Data Transfer: In applications with large amounts of data needing to be transferred to and from the device, such as Memcached, transfer time can dominate the overall execution time of the kernel. Figure 9 shows the contribution of the execution time and data transfer times as a percentage of the overall execution time for each GPU. We optimistically selected the minimum amount of data that must be transferred to and from the device: the requests to be processed and the results of the requests respectively. Assuming cyclic[3] transfer of data, more than 98% of the overall execution time is spent transferring data for the discrete AMD Radeon HD 5870. These values were recorded assuming that none of the data could have been modified on the host between successive kernel launches, thus ensuring all data in the device memory is valid. Therefore, on kernel launch, the only data that must be transferred are the requests themselves; upon completion of the kernel, all of the results must be transferred back to the host. A more realistic assumption is that an unknown amount of data could have been modified between kernel launches, thus invalidating a portion of the data on the device and requiring explicit tracking and transfers of the modified data on every kernel launch. Tracking which data was modified could be avoided by pessimistically transferring all of the data on every kernel launch, however, this cyclic memory transfer model that

[3]Cyclic refers to transferring data before and after successive kernel launches, whereas acyclic data transfer overlaps data transfer with the kernel execution [19]

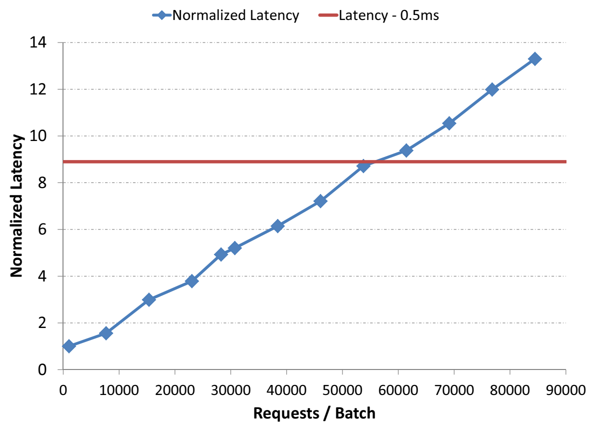


(a) Excluding data transfer

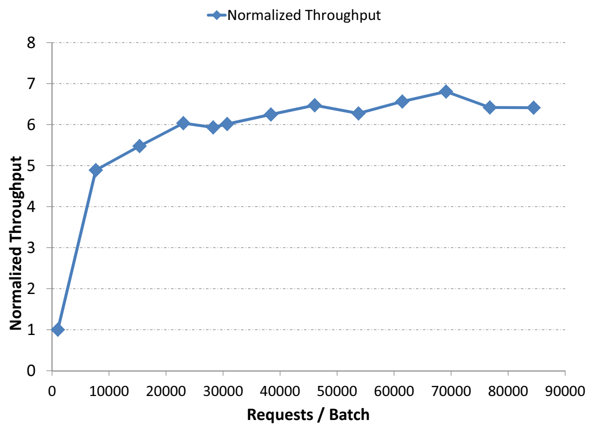


(b) Including data transfer

Fig. 6. Memcached Speed-up vs. CPU on different GPU Architectures (38,400 requests/batch)



(a) Normalized latency



(b) Normalized throughput

Fig. 7. Throughput and Latency While Varying the Request Batch Size on the AMD Radeon HD 5870 (normalized to 1,024 requests/batch)

transfers data regardless of whether it has been modified is sub-optimal.

Others [19], [16], [6], [20] have proposed solutions to this challenge (e.g., implementing frameworks to automatically and acyclicly transfer modified data to the device or requiring programmer annotation of the code to specify memory regions to be explicitly managed) to reduce the impact data transfers have on performance. With the introduction of CPU-GPU architectures that share a physical memory space, such as the AMD Fusion systems, this transfer time can be virtually eliminated. As can be seen in Figure 9, the majority of the overall execution time for the Llano A8-3850 and Zacate E-350 systems is spent performing useful work, rather than waiting for the data transfer to complete. Being able to reduce the time required to transfer data, either by using an architecture with a unified memory space or reducing the transfer overhead by one of the methods proposed by others, is crucial when looking to port an application requiring large data transfers to the GPU.

B. Simulation Performance

Unless otherwise stated, the data presented in this section was collected from the baseline GPGPU-Sim configuration presented in Section IV. This section attempts to gain insight into the performance of Memcached using GPGPU-Sim.

1) SIMD Efficiency of Memcached: If a work item branches away from the other work items in its wavefront, the GPU executes the two sub-groups separately, requiring more cycles than if they were executed together [3]. This reduces overall SIMD efficiency. Combining Memcached's complex control-flow graph, which contains multiple nested conditional branches, with the level of uncertainty in branch outcomes, one can expect Memcached to have very poor SIMD efficiency, directly resulting in poor performance on the GPU. Although pessimistic, an initial view of the system might be that each branch outcome has an equal probability. In many applications, this might be an unreasonable assumption; however, many of the branches in Memcached depend on input data, such as the length of the request key, that varies greatly between requests. This is marginally better than the worst case, in which at each branch the thread grouping deterministically split in half. After further analysis of the system, certain branches may be reasoned to occur rarely or never (e.g., error handling or dead code). These branches can be removed from the analysis by forcing the threads to take a certain path because their inclusion would negatively bias the expected SIMD efficiency

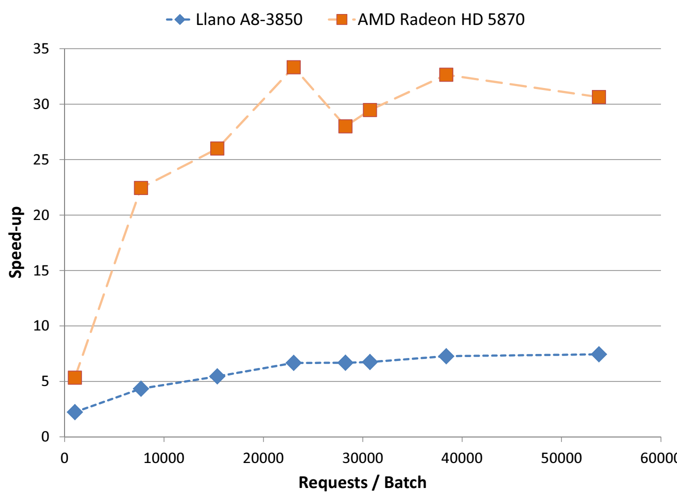


Fig. 8. Speed-up of AMD Radeon HD 5870 and Llano A8-3850 vs. the Llano A8-3850 CPU at Different Request Batch Sizes

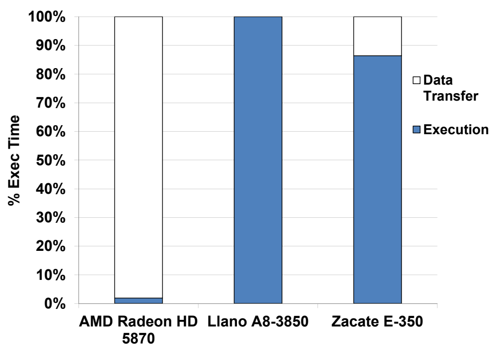


Fig. 9. Memcached Overall Execution Breakdown (23,040 requests/batch)

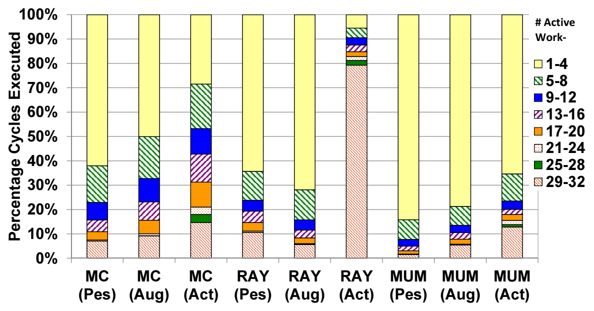


Fig. 10. SIMD Efficiency

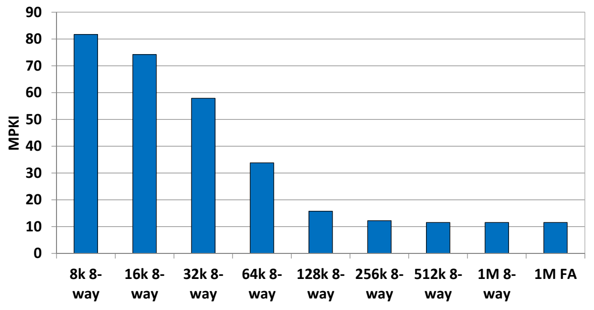


Fig. 11. L1 Data Cache Mmisses per 1,000 Instructions at Various Configurations.

of the system.

Simulating the control-flow behavior of a single thread grouping with the control flow simulator, discussed in Section IV, we can compare Memcached's SIMD efficiency with these initial views of how the system might behave, resulting in the data in Figure 10. This figure compares the overall SIMD efficiency of Memcached's actual execution (Act) with the pessimistic view that all branches have equal outcome probabilities (Pes). In this case, there are 32 work items per wavefront. Each bin in the graph represents the fraction of total program execution in which the specified number of scalar threads was concurrently executing.

We then improve on this pessimistic view by optimizing away all branch paths that are never taken during normal execution (Aug) and compare the recorded SIMD efficiency with the actual execution. Although there is an improvement, the SIMD efficiency of the actual execution still outperforms the theoretical behavior. We extend this analysis to applications known to perform well on the GPU, such as Mummer (MUM) and Raytracer (RAY), and measure how these results compare when similar assumptions are applied, also shown in Figure 10.

Although MUM exhibits a relatively low SIMD efficiency, GPUs tend to have more memory bandwidth than CPUs, which can result in higher throughput on memory-limited applications even in the presence of significant control flow-divergence. MUM follows a similar trend to Memcached; however, the theoretical results in RAY perform significantly worse than the actual results. This is caused by high correlations between work items's branch outcomes within a wavefront. Although each branch outcome may have a relatively random probability, each work item is biased by the results of the other work item within the group.

2) Effect of Memcached on the Memory System: Memcached's key-value retrieval algorithm places a significant amount of stress on the memory system.

Figure 11 shows the misses per 1,000 instructions (MPKI) for Memcached with a variety of L1 data cache configurations. This data shows that Memcached has some exploitable locality and that the working set of our simulated configuration fits in a 256k cache. The remaining 10 MPKI are caused by cold start misses.

Figure 12 shows the performance of Memcached on GPGPU-Sim with a number of L1 global data cache configurations and two variations of an idealized memory system. Performance is presented as a percentage of peak IPC (when every lane is active every cycle). Increasing the cache size results in a continuous performance improvement up to 256k, beyond which it levels off. This result indicates that Memcached is a cache-sensitive workload. Further investigation of the source code reveals that two instructions receive a significant reduction in latency when cache size increases. These instructions are the loads performed inside the key comparison loop which compares the input key and a key found in the

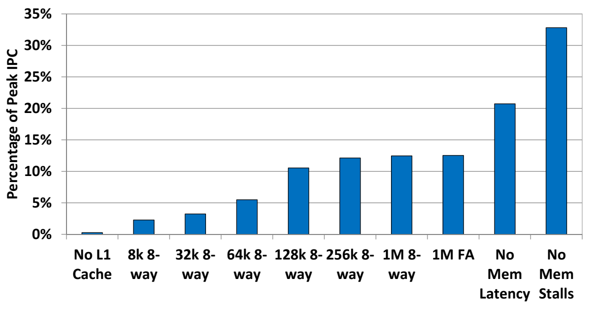


Fig. 12. Performance as Percentage of Peak IPC with Various Realistic L1 Data Cache Configurations and Two Idealized Memory Systems

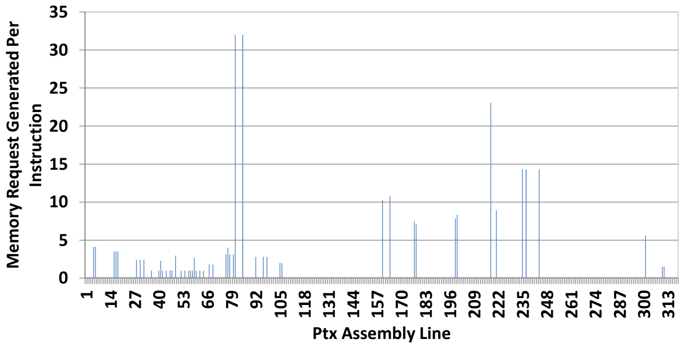


Fig. 13. Memory Requests Generated per Instruction for each Static PTX Instruction

hashtable. This loop accesses memory sequentially, resulting in a high cache hit rate when the cache is large enough to capture the working set.

The 1,024k fully associative (FA) configuration suffers only from cold-start misses. The No Memory Latency data point models a system in which requests can be processed in one cycle, but each compute core can send only one request per cycle to the global memory system. We can see from these two data points the amount of one-touch data loaded by the kernel. Increasing from no cache to a cache that captures all the kernel's locality takes the IPC from less than 1% of the peak to 12%, and removing the cold-start misses provides 21%. This suggests that Memcached contains a high fraction of touched-once data. This was verified by measuring the number of accesses to each L1 cache line prior to eviction in the 1M cache configuration. The No Memory Stalls setup sends memory requests though the pipeline as fast as they are generated. No Memory Stalls show an additional 12% increase in performance over the No Memory Latency system. This result tells us that Memcached spends a large fraction of its execution time with a backed-up queue of memory requests waiting to access the memory system. If wavefronts do not stall on memory then performance is largely limited by SIMD efficiency. The performance of the non-stalling memory system is 33% of peak, while the SIMD efficiency of Memcached is 40%. This 7% discrepancy can be attributed to idle cycles when some cores take longer than others to complete the kernel.

Figure 13 illustrates the amount of memory divergence in Memcached. It presents the average number of global memory

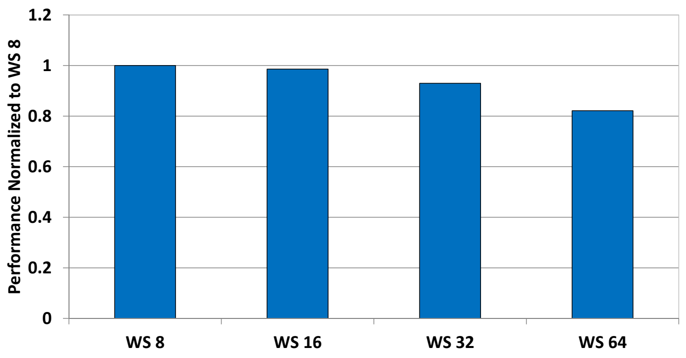


Fig. 14. Performance of Memcached at Various Wavefront Sizes (normalized to a warp size of 8)

requests generated for each static PTX assembly instruction. In our GPGPU-Sim baseline configuration the maximum Y-value for each bar is 32 (all 32 lanes of the wavefront generate a request) and the minimum is two (because requests are coalesced per half-wavefront in GPGPU-Sim). A well-behaved GPU application will attempt to minimize this number and limit the stress on the memory system. From this graph we can see that many memory instructions do not coalesce their accesses into two requests. The bulk of the program's execution time is spent between PTX lines 157 and 253, where the instructions request between seven and 23 cache lines each on average. Further analysis of this code revealed that the only reason these instructions do not request closer to 32 lines is that our SIMD efficiency also drops during this phase, resulting in fewer active lanes. A relatively small amount of code repeatedly generates a large number of memory accesses, which backs up the memory-request queue.

The preceding data indicates that the inclusion of an L1 data cache is critical to the performance of Memcached. Processing more than one memory request per cycle (e.g., through a multi-banked L1 data cache) would also improve performance because it allows the backed-up memory-request queue to empty sooner.

3) Effect of Wavefront Size on Performance: Figure 14 shows the performance of our modified Memcached on the baseline simulator when varying wavefront lengths. The performance is normalized to a wavefront length of eight. This data shows that there is an 18% difference in performance between a wavefront size of eight and 64. This indicates that Memcached's SIMD efficiency is a limiting factor even in the presence of excessive memory stalls.

VI. Related Work

Concurrent work by Berezecki et al. [7] presents a many-core architecture, the Tilera TILEPro64 64-core CPU, used to accelerate Memcached. In their work, different parts of Memcached are modified to run on individual processors, such as network workers, hash-table processes, TCP and UDP cores, and the operating system itself. Although the focus of their work and ours is similar (i.e., accelerating Memcached on a many-core architecture), our work differs in the method of achieving this goal, focuses on the feasibility of running such an application on a GPU, and provides a detailed characterization of Memcached on hardware and on a simulator.

Andersen et al. [4] propose a log-structured datastore system that utilizes lower-power CPUs and flash memory to maintain performance and reduce power consumption. This is effective in key-value store applications, such as Memcached, in which large amounts of computation are replaced with long I/O operations and various network latencies that are not significantly affected by low clock frequencies.

A core operation when processing a GET request is the hash. Because our focus was parallelizing independent requests, the hash algorithm is computed by each work item individually. Massively parallel hashing algorithms, such as the one implemented in StoreGPU [2], provide significant performance increases when the data being hashed is large. However, the keys hashed in this study were all less than 100 bytes and would not benefit from this divide-and-conquer technique.

Others have also exploited the parallel properties of server-resident applications, such as SQL [5], using a similar method of offloading batches of read requests to the GPU. They do not, however, model the data transfer times as they produce significant overheads in the overall execution time. Although we experience the same memory size limitations on the GPU, we are able to include the full data transfer times, which result in negligible times due to the use of the zero-copy shared-memory region on the AMD Fusion systems.

VII. Conclusions

In this paper, we present a characterization and evaluation of Memached on the new AMD Fusion architectures and a discrete AMD GPU architecture. We then present an analysis using GPGPU-Sim to gain additional insight into the behavior of Memcached on a GPU. From this analysis, we conclude that irregular applications, such as Memcached, should not be immediately disregarded when considering porting them to a GPU. We believe the methodology presented in this study of batching user requests for processing on a throughput-efficient device can be generalized so many server applications could take advantage of this framework. We observed that the Llano A8-3850 and Zacate E-350 GPUs outperformed their respective CPUs by factors of 8 and 4, respectively. We also showed that the discrete system was able to outperform the CPU when data transfers are ignored; however, when including the data transfer time, results are hindered by data-transfer overheads.

Acknowledgments

We thank the reviewers for their constructive comments and David Christie for assisting with running simulations.

References

[1] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Int'l Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 163–174, April 2009.

[2] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th Int'l Symp. on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM.

[3] AMD. *AMD Accelerated Parallel Processing, OpenCL - Programming Guide*, 2.5 edition, 2011.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14. ACM, 2009.

[5] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *GPGPU '10 Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, March 2010.

[6] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Gupta, Rajiv, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010.

[7] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core Key-value Store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, July 2011.

[8] C. Faylor, et al. Cygwin. http://www.cygwin.com/.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.

[10] E. Demers. Evolution of AMD Graphics. AMD Fusion Developer Summit, June 2011.

[11] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008.

[12] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 347–358, New York, NY, USA, 2010. ACM.

[13] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS '11*, 2011.

[14] IDC. HPC Server Market Declined 11.6% in 2009, Return to Growth Expected in 2010, March 2010.

[15] IDC. Worldwide Server Market Rebounds Sharply in Fourth Quarter as Demand for Blades and x86 Systems Leads the Way, February 2010.

[16] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A Mapping Path for Multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[17] M. Houston and M. Mantor. AMD Graphic Core Next: Low Power High Performance Graphics & Parallel Compute. AMD Fusion Developer Summit, June 2011.

[18] Memcached. A distributed memory object caching system. http://www.memcached.org.

[19] T. Jablin et al. Automatic CPU-GPU Communication Management and Optimization. In *PLDI*, June 2011.

[20] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In Amaral, Jos, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2008.

[21] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[22] W. Fung, et al. . Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[23] W. Fung, et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.*, 6(2):1–37, 2009.