MemcachedGPU: Scaling-up Scale-out Key-value Stores

Tayler H. Hetherington
The University of British Columbia
taylerh@ece.ubc.ca

Mike O'Connor
NVIDIA & UT-Austin
moconnor@nvidia.com

Tor M. Aamodt
The University of British Columbia
aamodt@ece.ubc.ca

Abstract

This paper tackles the challenges of obtaining more efficient data center computing while maintaining low latency, low cost, programmability, and the potential for workload consolidation. We introduce GNoM, a software framework enabling energy-efficient, latency bandwidth optimized UDP network and application processing on GPUs. GNoM handles the data movement and task management to facilitate the development of high-throughput UDP network services on GPUs. We use GNoM to develop MemcachedGPU, an accelerated key-value store, and evaluate the full system on contemporary hardware.

MemcachedGPU achieves ∼10 GbE line-rate processing of ∼13 million requests per second (MRPS) while delivering an efficiency of 62 thousand RPS per Watt (KRPS/W) on a high-performance GPU and 84.8 KRPS/W on a low-power GPU. This closely matches the throughput of an optimized FPGA implementation while providing up to 79% of the energy-efficiency on the low-power GPU. Additionally, the low-power GPU can potentially improve cost-efficiency (KRPS/$) up to 17% over a state-of-the-art CPU implementation. At 8 MRPS, MemcachedGPU achieves a 95-percentile RTT latency under 300μs on both GPUs. An offline limit study on the low-power GPU suggests that MemcachedGPU may continue scaling throughput and energy-efficiency up to 28.5 MRPS and 127 KRPS/W respectively.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Parallel databases; D.4.4 [*Operating Systems*]: Communications Management—Network communication

Keywords Data center, key-value store, GPU

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA.
 ISBN 978-1-4503-3651-2/15/08…$15.00.
DOI: http://dx.doi.org/10.1145/2806777.2806836

1. Introduction

Data centers contain large numbers of servers, memory, network hardware, non-volatile storage and cooling that, together, can consume tens of megawatts. Reducing energy consumption is therefore a key concern for data center operators [8]. However, typical data center workloads often have strict performance requirements, which makes obtaining energy efficiency through "wimpy" nodes [4] that give up single-thread performance nontrivial [24, 49]. This has increased the focus on specialized hardware accelerators, such as ASICs and FPGAs, to improve the performance of individual servers [47, 49, 58]. Although ASICs can provide higher efficiencies than reprogrammable FPGAs [34], generality is important in the data center to support many constantly changing applications [8]. FPGAs can provide high levels of efficiency for a single application; however, long reconfiguration times on current FPGAs [47, 49] may mitigate some of these benefits if the full application does not fit on the FPGA or when context switching between multiple applications. This is at odds with improving efficiency through higher utilization by consolidating multiple workloads onto a single server [8, 35]. While the ease of programming has improved on recent FPGAs with support for higher-level languages through high-level synthesis (HLS), such as OpenCL [14] and CUDA [47], HLS tends to achieve improvements in developer productivity by trading off the quality of the results [5].

Alternatively, graphics processing units (GPUs) are capable of providing high-throughput and energy-efficient processing on general-purpose architectures. While early GPUs had a reputation for high thermal design power (TDP) [47], recent GPUs have drastically improved energy-efficiency and TDP [46]. GPUs are currently being used in data centers to accelerate highly data-parallel applications, such as machine learning at Google [15]. In this work, we explore the potential of also using GPUs as flexible energy-efficient accelerators for network services in the data center.

Towards this end, we implement and evaluate an end-to-end version of the Memcached [38] distributed, in-memory key-value caching service, MemcachedGPU, on commodity GPU and ethernet hardware. Memcached is a *scale-out* workload, typically partitioned across multiple server nodes. In this work, we focus on using the GPU to *scale-up* the

throughput of an individual server node. We exploit request-level parallelism through batching to process multiple concurrent requests on the massively parallel GPU architecture, and task-level parallelism within a single request to improve request latency. While previous works have evaluated batch processing of network requests on GPUs, such as Memcached [23], HTTP [2], or database queries [7, 57], they focus solely on the application processing, which can be a small subset of the total end-to-end request processing. In contrast, we develop a complete system, *GNoM* (GPU Network Offload Manager), that incorporates UDP network processing on the GPU in-line with the application processing (Figure 1). *GNoM* provides a software layer for efficient management of GPU tasks and network traffic communication directly between the network interface (NIC) and GPU.

This is the first work to perform all of the Memcached read request processing and network processing on the GPU. We address many of the challenges associated with a full system network service implementation on heterogeneous systems, such as efficient data partitioning, data communication, and synchronization. Many of the core Memcached data structures are modified to improve scalability and efficiency, and are partitioned between the CPU and GPU to maximize performance and data storage. This requires synchronization mechanisms to maintain a consistent view of the application's data. The techniques presented in this paper may be relevant to other network services that require both CPU and GPU processing on shared data structures.

This work also tackles the challenges with achieving low-latency network processing on throughput-oriented accelerators. GPUs provide high throughput by running thousands of scalar threads in parallel on many small cores. *GNoM* achieves low latency by constructing fine-grained batches (512 requests) and launching multiple batches concurrently on the GPU through multiple parallel hardware communication channels. At 10 Gbps with the smallest Memcached request size, the smaller batches result in requests being launched on the GPU every $\sim 40\mu$s, keeping the GPU resources occupied to improve throughput while reducing the average request batching delay to under 20μs.

This paper makes the following contributions:

- It presents *GNoM*[1], a software system for UDP network and application processing on GPUs (Section 3), and evaluates the feasibility of achieving low-latency, 10 GbE line-rate processing at all request sizes on commodity ethernet and throughput-oriented hardware (Section 5).
- It describes the design of MemcachedGPU, an accelerated key-value store, which leverages *GNoM* to run efficiently on a GPU (Section 4), and compares MemcachedGPU against prior Memcached implementations (Section 5.6).
- It explores the potential for workload consolidation on GPUs during varying client demands while maintaining a level of QoS for MemcachedGPU (Section 5.4).

2. Background

This section provides a brief overview of the background for Memcached, the network interface (NIC) architecture, and the GPU architectures assumed in this work.

2.1 Memcached

Memcached is a general-purpose, scale-out, in-memory caching system used in distributed database systems to improve performance by reducing the amount of traffic to back-end databases. Memcached is used by many popular network services such as Facebook, YouTube, Twitter, and Wikipedia [38]. Memcached provides a simple key-value store interface to modify (*SET*, *DELETE*, *UPDATE*) and retrieve (*GET*) data from the hash table. The key-value pair and corresponding metadata is referred to as an *item*. To avoid expensive memory allocations for every write request, Memcached uses a custom memory allocator from pre-allocated *memory slabs* to store the items. To reduce memory fragmentation, Memcached allocates multiple memory slabs with different fixed-size entries. The hash table stores pointers to the items stored in the memory slabs. Keys are hashed to lookup item pointers from the hash table.

Facebook Memcached deployments typically perform modify operations over TCP connections, where it is a requirement that the data be successfully stored in the cache, whereas retrieve operations can use the UDP protocol [40]. Since Memcached acts as a look-aside cache, dropped *GET* requests can be classified as cache misses or the client application can replay the Memcached request. However, excessive packet drops mitigate the benefits of using the caching layer, requiring a certain level of reliability of the underlying network for UDP to be effective.

2.2 Network Interface (NIC) Architecture

The NIC connects to the host (CPU) via a PCIe bus. Network packets enter RX (receive) queues at the NIC, which are copied to pre-allocated, DMA-able RX ring buffers typically in CPU memory. The NIC sends interrupts to the CPU for one or more pending RX packets. The NIC driver copies packets to Linux Socket Buffers (SKB) and returns the RX buffer back to the NIC. Optimizations such as direct NIC access (DNA) may reduce memory copies by allowing applications to directly access the RX buffers. In Section 3, we expand on this technique by using NVIDIA's GPUDirect [45] to enable efficient GPU processing on RX packets.

2.3 GPU Architecture

GPUs are throughput-oriented offload accelerators traditionally designed for graphics. Recent GPUs are capable of running non-graphics applications written in C-like languages, such as CUDA [43] or OpenCL [31]. In this work we use CUDA and focus on discrete NVIDIA GPUs connected to

[1] Code for GNoM and MemcachedGPU is available at https://github.com/tayler-hetherington/MemcachedGPU

the host device through a PCIe bus. However, the system presented in this paper is also relevant with AMD and integrated GPUs. CUDA applications are composed of two parts: a host side (CPU) responsible for initializing the GPU's environment and communicating with the GPU, and a device side (GPU) which executes one or more kernels. A CUDA kernel is a user-defined parallel section of code that runs on the GPU. Scalar CUDA threads are grouped into warps typically containing 32 threads. Each thread in a warp executes instructions in a lock-step single-instruction, multiple-thread (SIMT) fashion. Warps are further grouped into cooperative thread arrays (CTAs). A CTA is dispatched as a unit to a SIMT stream multiprocessor (SM) but individual warps within the CTA are scheduled independently within the SM.

When threads in a warp take different execution paths due to branch instructions, each sub-group of threads must be executed serially, causing SIMT lanes to become idle. This is known as branch divergence. Another optimization, referred to as memory coalescing, combines accesses from threads in a warp to adjacent memory locations, improving performance. Accesses to separate memory locations from within a warp is known as memory divergence.

The host communicates with the GPU through a set of hardware managed CUDA *streams*, allowing for concurrently operating asynchronous tasks (e.g., computation and data transfers). AMD hardware provides the same capability through OpenCL command queues. NVIDIA's Hyper-Q [44] enables up to 32 independent hardware streams.

3. GNoM

GNoM provides a CPU software framework that aims to enable a class of high-throughput, low-latency UDP network applications to run efficiently on GPUs. This section addresses some of the challenges with achieving this goal and presents the software architecture of *GNoM* that facilitated the design and implementation of MemcachedGPU.

3.1 Request Batching

Request batching in *GNoM* is done per request type to reduce control flow divergence among GPU threads. Smaller batch sizes minimize batching latency, however, increase GPU kernel launch overhead and lower GPU resource utilization, which reduces throughput. We find that 512 requests per batch provides a good balance of throughput and latency (Section 5.3). While MemcachedGPU contains a single request type for batching, *GET* requests, workloads with many different request types could batch requests at finer granularities. For example, multiple smaller batches could be constructed at the warp-level of 32 requests and launched together on the GPU to perform different tasks [9].

3.2 Software Architecture

GNoM is composed of two main parts that work together to balance throughput and latency for network services on GPUs: *GNoM-host* (CPU), responsible for interaction with the NIC and pre/post-GPU data and task management; and *GNoM-dev* (GPU), responsible for UDP packet and application processing. Figure 1 presents the interactions between the NIC, CPU, and GPU in *GNoM*, as well as the main CUDA kernel for *GNoM-dev*. RX packets are DMA-ed directly to the GPU's global memory using GPUDirect. Only metadata describing the request batch is sent to the CPU. We focus on accelerating the RX path in *GNoM*. While *GNoM* can implement a similar TX (transmit) path using GPUDirect, MemcachedGPU uses a third party CPU Linux network bypass service, *PF_RING* [41], to accelerate the TX path on the CPU. This decision was driven by the design of MemcachedGPU in which the main data structures used to populate the response packets are stored in CPU memory (Section 4).

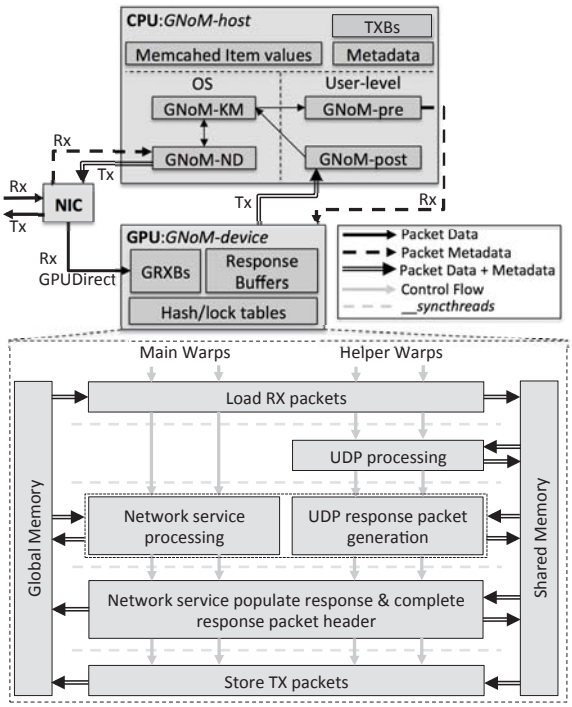


Figure 1: GNoM packet flow and main CUDA kernel.

3.2.1 GNoM-host

GNoM-host is required to provide task management and I/O support for the GPU because current GPUs cannot directly communicate with, or receive interrupts from the NIC. *GNoM-host* is responsible for efficiently managing data movement between the NIC and GPU, managing work tasks on the GPU, and performing any post-processing tasks required prior to sending response packets. To accomplish these tasks, *GNoM-host* is composed of three software components: a modified Intel IXGBE network driver (v3.18.7), *GNoM-ND*, a custom Linux kernel module, *GNoM-KM*, and a user-level software framework, *GNoM-user*.

GNoM-KM and GNoM-ND *GNoM* allocates pinned, unpageable GPU memory using GPUDirect (220MB partitioned into 32 - 2KB buffers per 64KB GPU page) to store

incoming RX packets (GRXB). To reduce memory copies, the GRXBs hold the packets throughout their lifetime on the server, recycling them to the NIC only when processing is complete. This differs from the baseline network driver which recycles the RX buffers immediately to the NIC after copying the packets into Linux SKBs. *GNoM* requires significantly more pinned memory than the baseline network flow to ensure that a GRXB is available for incoming packets. For a given processing latency, a packet drop occurs when GRXBs are not recycled quickly enough to accommodate the new RX packet. If more buffers cannot be allocated, then the throughput must be reduced accordingly. 220 MB is the maximum amount of pin-able GPU memory one can allocate for GPUDirect on the NVIDIA Tesla K20c.

GNoM-ND DMAs RX packet data directly from the NIC to the GRXBs in GPU memory using GPUDirect and constructs metadata in CPU memory describing the batch of GPU packets (Figure 1). *GNoM-ND* passes the batch metadata to *GNoM-KM* once a batch is fully populated and ready for GPU processing. The NIC populates the GRXBs in the order they were registered. To identify a full batch of packets in GPU memory, the metadata only requires a pointer to the first packet and the total number of packets.

GNoM-KM provides an interface between the Linux kernel and *GNoM-user*. It includes hooks for *GNoM-user* to communicate indirectly with the NIC, such as configuring the NIC for use with *GNoM*, reading metadata for request batches, and recycling GRXBs.

GNoM-User *GNoM-user* consists of pre-processing (*GNoM-pre*) and post-processing (*GNoM-post*) user-level *pthreads* (Figure 1). *GNoM-pre* retrieves request batch metadata from *GNoM-KM*, performs application specific memory copies to the GPU, launches CUDA kernels that perform the network service processing on the batch of requests, and constructs CUDA events to detect when the GPU processing completes. *GNoM* uses CUDA streams to overlap processing of multiple small batches to provide a better tradeoff between packet latency and throughput.

GNoM-post polls the CUDA events waiting for the GPU network service processing to complete, populates the response packets with application specific data, and transmits the response packets using *PF_RING*. For MemcachedGPU, this consists of copying the item's value from the Memcached memory slabs in CPU memory to the TXBs (*PF_RING* transmit buffers) in CPU memory for each packet in the batch. Finally, *GNoM-post* recycles the now free GRXBs back to *GNoM-ND* for future RX packets.

Non-GPUDirect (NGD) GPUDirect is currently only supported on the high-performance NVIDIA Tesla and Quadro GPUs. To evaluate MemcachedGPU on lower power, lower cost GPUs, we also implemented a non-GPUDirect (NGD) framework. NGD uses *PF_RING* [41] to receive and batch Memcached packets in host memory before copying the request batches to the GPU. NGD uses the same *GNoM-user* and *GNoM-dev* framework; however, *GNoM-KM* and *GNoM-ND* are replaced by *PF_RING*. Section 5.3 evaluates NGD on the NVIDIA Tesla K20c and GTX 750Ti GPUs.

3.2.2 GNoM-dev

The lower portion of Figure 1 illustrates the CUDA kernel for UDP packet and network service processing (e.g., MemcachedGPU *GET* request processing). Once a network packet has been parsed (UDP processing stage), the network service can operate in parallel with the response packet generation since they are partly independent tasks. The number of threads launched per packet is configurable (MemcachedGPU uses two threads). *GNoM-dev* leverages additional *helper threads* to perform parallel tasks related to a single network service request, exploiting both packet level and task level parallelism to improve response latency.

GNoM-dev groups warps into main and helper warps. Main warps perform the network service processing (e.g., Memcached *GET* request processing) while helper warps perform the UDP processing and response packet header construction. The main and helper warps also cooperatively load RX data and store TX data (e.g., response packet headers and any application specific data, such as pointers to Memcached items in CPU memory) between shared and global memory efficiently through coalesced memory accesses. This requires CUDA synchronization barriers to ensure that the main and helper warps maintain a consistent view of the packet data in shared memory. The UDP processing verifies that the packet is for the network service and verifies the IP checksum. While most of the response packet header can be constructed in parallel with the network service processing, the packet lengths and IP checksum are updated after to include any application dependent values (e.g., the length of the Memcached item value).

4. MemcachedGPU

This section presents the design of MemcachedGPU and discusses the modifications required to achieve low latency and high throughput processing on the GPU.

4.1 Memcached and Data Structures

In typical Memcached deployments [40], *GET* requests may comprise a large fraction of traffic (e.g., 99.8% for Facebook's USR pool [6]) when hit-rates are high. Hence, we focus on accelerating Memcached *GET* requests and leave the majority of *SET* request processing on the CPU.

Memcached data structures accessed by both *GET* and *SET* requests include a hash table to store pointers to Memcached items, memory slabs to store the Memcached items and values, and a least-recently-used (LRU) queue for selecting key-value pairs to evict from the hash table when Memcached runs out of memory on a *SET*. Memcached keys can be an order of magnitude smaller than value sizes (e.g., 31B versus 270B for Facebook's APP pool [6]), placing larger storage requirements on the values.

These data structures need to be efficiently partitioned between the CPU and GPU due to smaller GPU DRAM capacity versus CPUs found on typical Memcached deployments. We place the hash table containing keys and item pointers in GPU memory. Values remain in CPU memory.

4.1.1 GET requests

MemcachedGPU performs the main *GET* request operations on the GPU. This includes parsing the *GET* request, extracting the key, hashing the key, and accessing the hash table to retrieve the item pointer. Aside from the item values, all of the Memcached data structures accessed by *GET* requests are stored in GPU global memory. MemcachedGPU uses the same Bob Jenkin's lookup3 hash function [27] included in the baseline Memcached. *GET* requests bypass the CPU and access the hash table on the GPU as described in Section 4.2. Each *GET* request is handled by a separate GPU thread, resulting in memory divergence on almost every hash table access. However, the small number of active GPU threads and the GPU's high degree of memory-level parallelism mitigates the impact on performance. After the GPU processing is complete, *GNoM-post* copies the corresponding item value for each packet in the batch from CPU memory into a response packet (TXB) to be sent across the network.

4.1.2 SET requests

SET requests require special attention to ensure consistency between CPU and GPU data structures. *SET* requests follow the standard Memcached flow over TCP through the Linux network stack and Memcached code on the CPU. They update the GPU hash table by launching a naive *SET* request handler on the GPU.

SET requests first allocate the item data in the Memcached memory slabs in CPU memory, and then update the GPU hash table. This ordering ensures that subsequent *GET* requests are guaranteed to find valid CPU item pointers in the GPU hash table. Another consequence of this ordering is that both *SET* and UPDATE requests are treated the same since the hash table has not been probed for a hit before allocating the item. An UPDATE simply acts as a *SET* that evicts and replaces the previous item with the new item.

SET requests update the GPU hash table entries, introducing a race condition between *GETs* and other *SETs*. Section 4.2.2 describes a GPU locking mechanism to ensure exclusive access for *SET* requests. As *GETs* typically dominate request distributions, we have used a simple implementation in which each *SET* request is launched as a separate kernel and processed by a single warp. Accelerating *SET* requests through batch processing should not be difficult but was not the focus of this work.

4.2 Hash Table

This section presents the modifications to the baseline Memcached hash table to enable efficient processing on the GPU while minimizing the impact on hit-rate.

4.2.1 Hash Table Design

The baseline Memcached implements a dynamically sized hash table with hash chaining on collisions. This hash table resolves collisions by dynamically allocating new entries and linking them into existing linked lists (chains) at conflicting hash table entries. This ensures that all items will be stored as long as the system has enough memory.

This hash table design is a poor fit for GPUs for two main reasons. First, dynamic memory allocation on current GPUs can significantly degrade performance [25]. Second, hash chaining creates a non-deterministic number of elements to be searched between requests when collisions are high. This can degrade SIMD efficiency when chain lengths vary since each GPU thread handles a separate request.

To address these issues, MemcachedGPU implements a fixed-size *set-associative* hash table, similar to [11, 36]. We select a set size of 16-ways (see Section 5.2). We also evaluated a modified version of *hopscotch hashing* [22] that evicts an entry if the hopscotch bucket is full instead of rearranging entries. This improves the hit-rate over a set-associative hash table by 1-2%; however, the peak *GET* request throughput is lower due to additional synchronization overheads. The hopscotch hash table requires locking on every entry since no hopscotch group is unique, whereas the set-associative hash table only requires locking on each set.

Each hash table entry contains a header and the physical key. The header contains a valid flag, a last accessed timestamp, the length of the key and value, and a pointer to the item in CPU memory. MemcachedGPU also adopts an optimization from [19, 36] that includes a small 8-bit hash of the key in every header. When traversing the hash table set, the 8-bit hashes are first compared to identify potential matches. The full key is compared only if the key hash matches, reducing both control flow and memory divergence.

Hash table collisions and evictions The baseline Memcached uses a global lock to protect access to a global LRU queue for managing item evictions. On the GPU, global locks would require serializing all GPU threads for every Memcached request, resulting in low SIMD efficiency and poor performance. Other works [40, 56] also addressed the bottlenecks associated with global locking in Memcached.

Instead, we manage local LRU per hash table set, such that *GET* and *SET* requests only need to update the timestamp of the hash table entry. The intuition is that the miss rate of a set-associative cache is similar to a fully associative cache for high enough associativity [48]. Instead of allocating a new entry, collisions are resolved by finding a free entry or evicting an existing entry within the set. While the set-associative hash table was also proposed in [11, 36], we expand on these works by evaluating the impact of the additional evictions on hit-rates compared to the baseline Memcached hash table with hash chaining in Section 5.2.

SET requests search a hash table set for a matching, invalid, or expired entry. If the *SET* misses and no free entries

are available, the LRU entry in this hash table set is evicted. *GET* requests traverse the entire set until a match is found or the end of the set is reached. This places an upper bound, the set size, on the worst case number of entries each GPU thread traverses. If the key is found, the CPU value pointer is recorded to later populate the response packet.

Storage limitations The static hash table places an upper bound on the maximum amount of key-value storage. Consider a high-end NVIDIA Tesla K40c with 12GB of GPU memory. *GNoM* and MemcachedGPU consume ∼240MB of GPU memory for data structures such as the GRXBs, response buffers, and *SET* request buffers. This leaves ∼11.75GB for the GPU hash table and the lock table (described in Section 4.2.2). The hash table entry headers are a constant size, however, the key storage can be varied depending on the maximum size. For example, the hash table storage increases from 45 to 208.5 million entries decreasing from a maximum key size of 250B to 32B. From [6], typical key sizes are much smaller than the maximum size, leading to fragmentation in the static hash table.

If a typical key size distribution is known, however, multiple different hash tables with fixed-size keys can be allocated to reduce fragmentation. For example, [6] provides key and value size distributions for the Facebook ETC workload trace. If we create five hash tables with static key entries of 16, 32, 64, 128, and 250B with each size determined by the provided key distribution (0.14%, 44.17%, 52.88%, 2.79%, and 0.02% respectively), this enables a maximum of 157 million entries for a 10GB hash table. Using an average value size of 124B for ETC, this static partitioning on the GPU would enable indexing a maximum of 19.2GB of value storage in CPU memory compared to only 5.5GB when allocating for the worst case key size.

Our results on a low-power GPU (Section 5.3) suggest that integrated GPUs, with access to far more DRAM than discrete GPUs, may be able to achieve high throughputs in MemcachedGPU and are an important alternative to explore.

4.2.2 GPU Concurrency Control

Aside from updating the timestamps, *GET* requests do not modify the hash table entries. Thus, multiple *GET* requests can access the same hash table entry concurrently as they have similar timestamp values. However, *SET* requests require exclusive access since they modify the entries. To handle this, we employ a multiple reader (shared), single writer (exclusive) spin lock for each hash set using CUDA atomic compare and exchange (CAS), increment, and decrement instructions. The shared lock ensures threads performing a *GET* request in a warp will never block each other whereas the exclusive lock ensures exclusive access for *SET* requests to modify a hash table entry.

For *SET* requests, a single thread per warp acquires an exclusive lock for the hash table set. The warp holds on to the lock until the *SET* request hits in one of the hash table entries, locates an empty or expired entry, or evicts the LRU item for this set.

Table 1: Server and Client Configuration.

	Server	Client
Linux kernel	3.11.10	3.13.0
CPU	Intel Core i7-4770K	AMD A10-5800K
Memory	16 GB	16 GB
Network Interface	Intel X520-T2 10Gbps 82599 (mod. driver v3.18.7)	Intel X520-T2 10Gbps 82599 (driver v3.18.20)

Table 2: Server NVIDIA GPUs.

GPU	Tesla K20c	Titan	GTX 750Ti
Architecture (28 nm)	Kepler	Kepler	Maxwell
TDP	225 W	250 W	60 W
Cost	$2700	$1000	$150
# CUDA cores	2496	2688	640
Memory size (GDDR5)	5 GB	6 GB	2 GB
Peak SP throughput	3.52 TFLOPS	4.5 TFLOPS	1.3 TFLOPS
Core frequency	706 MHz	837 MHz	1020 MHz
Memory bandwidth	208 GB/s	288 GB/s	86.4 GB/s

4.3 Post GPU Race Conditions on Eviction

While the CPU item allocation order (Section 4.1.2) and GPU locking mechanism (Section 4.2.2) ensure correct access to valid items in CPU memory, a race condition still exists in *GNoM-post* for *SET* requests that evict items conflicting with concurrent *GET* requests. This race condition exists because separate CPU threads handle post GPU processing for *GET* and *SET* requests. For example, a *GET* request may access stale or garbage data when populating its response packet with an item that a concurrent *SET* request, occurring later in time on the GPU, may have just evicted.

Removing the race condition requires preserving the order seen by the GPU on the CPU. To accomplish this, each *GNoM-post* thread maintains a global completion timestamp (*GCT*), which records the timestamp of the most recent *GET* request batch to complete sending its response packets. This is the same timestamp used to update the hash table entry's last accessed time. If a *SET* needs to evict an item, it records the last accessed timestamp of the to-be evicted item. On the CPU after updating the GPU hash table, the *SET* polls all *GNoM-post* GCT's and stalls until they are all greater than its eviction timestamp before evicting the item. This ensures that all *GETs* prior to the *SET* have completed before the *SET* completes, preserving the order seen by the GPU. This stalling does not impact future *GET* requests since the *SET* allocates a new item prior to updating the hash table. Thus all *GET* requests occurring after the *SET* will correctly access the updated item.

5. Evaluation

This section presents our methodology and evaluation of MemcachedGPU, *GNoM*, and NGD.

5.1 Methodology

All of the experiments are run between a single Memcached server and client directly connected via two CAT6A Ethernet cables. The main system configuration is presented in Table 1. The GPUs evaluated in this study are shown

in Table 2, all using CUDA 5.5. The high-performance NVIDIA Tesla K20c is evaluated using *GNoM* (supports GPUDirect) and the low-power NVIDIA GTX 750Ti is evaluated using NGD. All three GPUs are evaluated in the offline limit study (Section 5.5). MemcachedGPU was implemented from Memcached v1.4.15 and is configured as a 16-way set-associative hash table with 8.3 million entries assuming the maximum key-size. Note that this is only ∼46% of the maximum hash table size on the Tesla K20c.

The client generates Memcached requests through the Memaslap microbenchmark included in libMemcached v1.0.18 [3]. Memaslap is used to stress MemcachedGPU with varying key-value sizes at different request rates. We evaluate the effectiveness of the hash table modifications in MemcachedGPU on more realistic workload traces with Zipfian distributions in Section 5.2. The Memcached ASCII protocol is used with a minimum key size of 16 bytes (packet size of 96 bytes). Larger value sizes impact both the CPU response packet fill rate and network response rate. However, we find that *GNoM* becomes network bound, not CPU bound, as value sizes increase. Thus, the smallest value size of 2 bytes is chosen to stress per packet overheads.

The single client system is unable to send, receive and process packets at 10 Gbps with the Memaslap microbenchmark. We created a custom client Memcached stressmark using *PF_RING* zero-copy [41] for sending and receiving network requests at 10 Gbps, and replay traces of *SET* and *GET* requests generated from Memaslap. We populate the MemcachedGPU server with 1.3 million key-value pairs through TCP *SET* requests and then send *GET* requests over UDP. However, processing all response packets still limits our send rate to ∼6 Gbps. To overcome this, we used a technique from [36] to forcefully drop response packets at the client using hardware packet filters at the NIC to sample a subset of packets. Using this technique, the server still performs all of the required per-packet operations. However, if we create a static set of requests to repeatedly send to the server (e.g., 512 requests), we are able to send and receive at the full rate at the client. We use this to measure packet drop rates at the client more accurately.

Power is measured using the Watts Up? Pro ES plug load meter [54] and measures the total system wall power for all configurations. An issue with PCIe BAR memory allocation between the BIOS and NVIDIA driver on the server system restricted the Tesla K20c and GTX 750Ti from being installed in isolation. We measured the idle power of the Tesla K20c (18W) and GTX 750Ti (6.5W) using the wall power meter and the *nvidia-smi* tool and subtracted the inactive GPU idle power from the total system power when running experiments on the other GPU. The GTX Titan did not have this issue and could be installed in isolation.

5.2 Hash Table Evaluation

We first evaluate the impact of our modifications to the hash table on the hit-rate. We created an offline hash table simulator that measures the hit-rate for a trace of key-value *GET* and *SET* requests. As in the baseline Memached, a *GET* request miss triggers a *SET* request for that item.

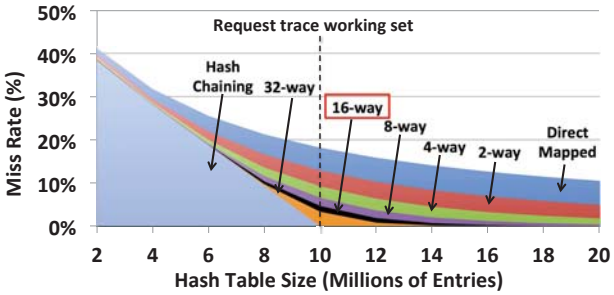


Figure 2: Miss-rate versus hash table associativity and size compared to hash chaining for a request trace with a working set of 10 million requests following the Zipf distribution.

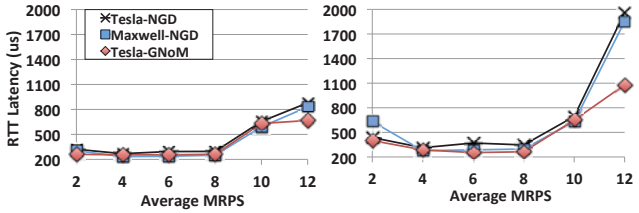
We evaluate the hash chaining (HC) and set-associative (SA) hash tables, as described in Section 4.2.1. For both hash tables, we fix the maximum number of key-value elements stored. HC evictions only occur when the total number of elements is exceeded. However, evictions may occur for the SA hash table even with fewer elements than the maximum due to set conflicts.

We used a modified version of the Yahoo! Cloud Serving Benchmark (YCSB) [13] provided by [19] to generate three Memcached traces with different item access distributions. Real workloads tend to follow a Zipfian (Zipf) distribution [13], where some items are accessed very frequently, while most others are accessed infrequently. Thus, it is important that the SA hash table miss-rate closely matches the HC hash table for the Zipf workload (Figure 2). We also evaluated the Latest distribution, where the items stored most recently are accesses more frequently than others, and Uniform random, where items are chosen at uniform random.

For each workload distribution, we generated a runtime trace of 10 million key-value pairs with 95% *GET* requests and 5% *SET* requests. The hash tables were first warmed up using a *load* trace of all *SET* requests. Figure 2 plots the miss-rate for an increasing hash table size and associativity compared to hash chaining for the Zipf distribution. For example, with a maximum hash table size of 8 million entries, SA has a 21.2% miss-rate with 1-way (direct mapped) and 10.4% miss-rate with 16-ways. HC achieves a 0% miss-rate when the hash table size is larger than the request trace (dotted line at 10 million entries) since it is able to allocate a new entry for all new key-value pairs. At smaller hash table sizes, none of the configurations are able to effectively capture the locality in the request trace resulting in comparable miss-rates. As the hash table size increases, increasing the associativity decreases the miss-rate. At 16-ways for all sizes, SA achieves a minimum of 95.4% of the HC hit-rate for the Zipf distribution. The other distributions follow similar trends with different absolute miss-rates. However, increasing the associativity also increase the worst-case number of entries to search on an access to the hash table. From exper-

Table 3: *GET* request throughput and drop rate at 10 GbE.

Key Size	16 B	32 B	64 B	128 B
Tesla drop rate server	0.002%	0.004%	0.003%	0.006%
Tesla drop rate client	0.428%	0.033%	0.043%	0.053%
Tesla MRPS/Gbps	12.92/9.92	11.14/9.98	8.66/9.98	6.01/10
Maxwell-NGD drop rate server	0.47%	0.13%	0.05%	0.02%
Maxwell-NGD MRPS/Gbps	12.86/9.87	11.06/9.91	8.68 10	6.01/10



(a) Mean RTT (b) 95-percentile RTT

Figure 3: Mean and 95-percentile RTT vs. throughput for Tesla with *GNoM* and NGD, and Maxwell with NGD.

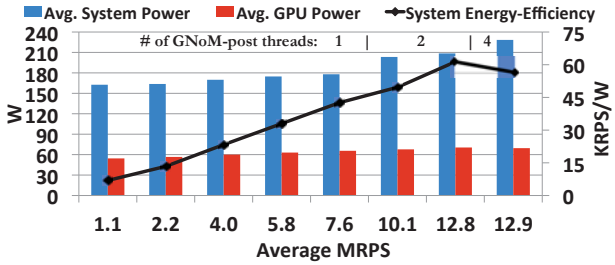


Figure 4: Power and energy-efficiency vs. throughput for MemcachedGPU and *GNoM* on the Tesla K20c.

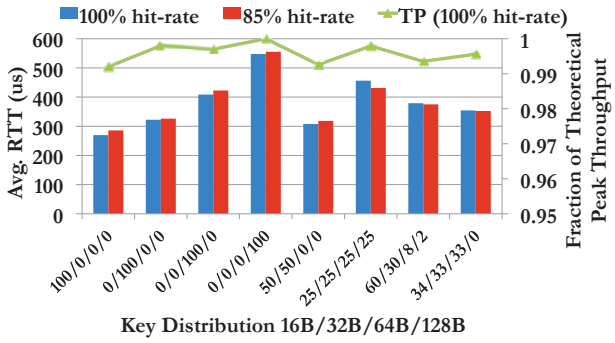


Figure 5: Impact of key length and hit-rate variation on RTT and throughput for MemcachedGPU and *GNoM* on the Tesla K20c.

imentation, we found that an associativity of 16-ways provides a good balance of storage efficiency and performance.

5.3 MemcachedGPU Evaluation

Next, we evaluate the full end-to-end MemcachedGPU on the high-performance NVIDIA Tesla K20c (Tesla) using *GNoM* with GPUDirect, and on the low-power NVIDIA GTX 750Ti (Maxwell) using the non-GPUDirect (NGD) framework (Section 3.2.1). Throughput is measured in millions of requests per second (MRPS) and energy-efficiency is measured in thousands of RPS per Watt (KRPS/W). For latency experiments, the 8 byte Memcached header is modified to contain the client send timestamp and measures the request's complete round trip time (RTT).

Throughput: Memcached tolerates dropped packets by treating them as misses in the caching layer. However, excessive packet dropping mitigates the benefits of using Memcached. We measure the packet drop rate at the server and client for packet traces of 500 million requests with equal length keys at peak throughputs, averaging over three runs. As shown in Table 3, MemcachedGPU is able to processes packets near 10 GbE line-rate for any Memcached request size with server packet drop rates $< 0.006\%$ (*GNoM*) and $< 0.5\%$ (NGD). The client drop rates are measured using the static trace described in Section 5.1 with no packet loss at the server. Increasing to the full 13.02 MRPS at 16B keys increases the server drop rate due to the latency to recycle the limited number of GRXBs to the NIC for new RX packets.

RTT Latency: For many scale-out workloads, such as Memcached, the longest latency tail request dictates the total latency of the task. While the GPU is a throughput-oriented accelerator, we find that it can provide reasonably low latencies under heavy throughput. Figure 3 measures the mean and 95-percentile ($p95$) client-visible RTT versus request throughput for 512 requests per batch on the Tesla using *GNoM* and NGD, and the Maxwell using NGD. As expected, *GNoM* has lower latencies than NGD (55-94% at $p95$ on the Tesla) by reducing the number of memory copies on packet RX with GPUDirect. As the throughput approaches the 10 GbE line-rate the latency also increases, with the $p95$ latencies approaching $1.1ms$ with *GNoM* and $1.8ms$ with NGD. We also evaluated a smaller batch size of 256 requests on *GNoM* and found that it provided mean latencies between 83-92% of 512 requests per batch when less than 10 MRPS, while limiting peak throughput and slightly increasing the mean latency by $\sim$2% at 12MRPS. At lower throughputs (< 4 MRPS), we can see the effects of the batching delay on the $p95$ RTT (Figure 3b). For example, at 2 MRPS with a batch size of 512 requests, the average batching delay per request is already $128\mu s$, compared to $32\mu s$ at 8 MRPS. While not shown here, a simple timeout mechanism can be used to reduce the impact of batching at low request rates by launching partially full batches of requests.

Energy-Efficiency: Figure 4 plots the average power consumption (full system power and GPU power) and energy-efficiency of MemcachedGPU and *GNoM* on the Tesla K20c at increasing request rates. The Tesla K20c power increases by less than 35% when increasing the throughput by $\sim$13X, leading to the steady increase in energy-efficiency. The jumps in system power at 10.1 and 12.9 MRPS are caused by adding *GNoM-post* threads to recycle the GRXBs fast enough to maintain low packet drop rates at the higher throughputs. However, increasing the *GNoM-post* threads from two to four decreases energy-efficiency as the system power is increased by 9%. At peak throughputs, the low-power GTX 750Ti using NGD consumes 73% of the total system power consumed by the Tesla K20c using *GNoM* (151.4 W at 84.8 KRPS/W).

Table 4: Concurrent GETs and SETs on the Tesla K20c.

GET MRPS (% peak)	7 (54)	8.8 (68)	9.7 (74)	10.6 (82)	11.7 (90)
SET KRPS (% peak)	21.1 (66)	18.3 (57)	18 (56)	16.7 (52)	15.7 (49)
SET:GET	0.3%	0.21%	0.19%	0.16%	0.13%
Server Drop	0%	0.26%	3.1%	7.5%	8.8%

The Tesla K20c consumes roughly one third of the total system power. Note that the peak GPU power of 71W is less than 32% of the K20c's TDP, suggesting low utilization of the total GPU resources. The system has an idle power of 84W without any GPUs. Thus, *GNoM-host* consumes roughly 15%, 25%, and 33% of the total system power when using one, two, or four *GNoM-post* threads respectively. Much of this is an artifact of GPUs being offload-accelerators, which rely on the CPU to communicate with the outside world. This leaves large opportunities to further improve the energy-efficiency of *GNoM* through additional hardware I/O and system software support for the GPU.

Branch Divergence: Next, we evaluate the impact of branch divergence on performance in MemcachedGPU, which stems from each GPU thread handling a separate *GET* request. For example, differences in key lengths, potential hits on different indexes in the hash table set, or misses in the hash table can all cause threads to perform a different number of iterations or execute different blocks of code at a given time. Each of these scenarios reduce SIMD efficiency and consequently performance. Figure 5 plots the average peak throughput as a fraction of the theoretical peak throughput for a given key length distribution. We find that the throughput performs within 99% of the theoretical peak, regardless of the key distribution. That is, even when introducing branch divergence, MemcachedGPU becomes network bound before compute bound.

Figure 5 also plots the average RTT for *GET* requests at 4 MRPS under different distributions of key lengths and at 100% and 85% hit-rates. Note that this experiment still consists of 100% *GET* requests. The results match the intuition that because there is no sorting of key lengths to batches, the latency should fall somewhere between the largest and smallest key lengths in the distribution. If there are large variations in key lengths and tight limits on RTT, the system may benefit from a pre-sorting phase by key length. This could help reduce RTT for smaller requests, however, the maximum throughput is still limited by the network. Additionally, there is little variation between average RTT with different hit-rates. While reducing the hit-rate forces more threads to traverse the entire hash table set, the traversal requires a similar amount of work compared to performing the key comparison on a potential hit.

GETs and SETs: While the main focus of MemcachedGPU is on accelerating *GET* requests, we also evaluate the throughput of the current naive *SET* request handler and its impact on concurrent *GET* requests. *SET* requests are sent over TCP for the same packet trace as the *GETs* to stress conflicting locks and update evictions.

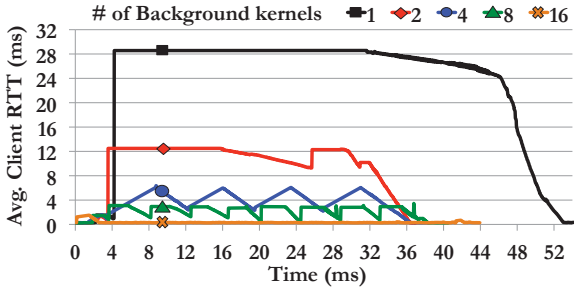


Figure 6: Client RTT (avg. 256 request window) during BGT execution for increasing # of fine-grained kernel launches.

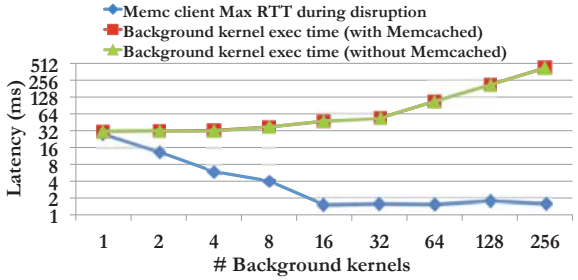


Figure 7: Impact on BGT execution time with increasing # of kernel launches and max client RTT during BGT execution.

The maximum *SET* request throughput is currently limited to 32KRPS in MemcachedGPU, ∼32% of the baseline. This is a result of the naive *SET* handler described in Section 4.1.2, which serializes *SET* requests. However, this is not a fundamental limitation of MemcachedGPU as, similar to *GET* requests, *SET* requests could also be batched together on the CPU prior to updating the GPU hash table. Unlike *GET* requests, however, each *SET* requests would need to be serialized or processed per GPU warp instead of per thread to avoid potential deadlocks on the exclusive locks (Section 4.2.2). Improving *SET* support is left to future work. Table 4 presents the *GET* and *SET* request throughputs, resulting SET:GET ratio, and server packet drop rate of MemcachedGPU on the Tesla K20c. As the *GET* request rate increases, the *SET* rate drops due to contention for GPU resources. The low maximum *SET* request throughput limits the SET:GET ratio to <0.5% for higher *GET* request rates. The average server packet drop rate approaches 10% when increasing *GET* throughput to 90% of the peak, which limits the effectiveness of MemcachedGPU. *GET* requests at ∼9 MRPS maintains comparable drop rates to the peak throughput, while also handling 0.26% *SET* requests.

5.4 Workload Consolidation on GPUs

Workload consolidation, running multiple workloads concurrently on the same hardware, improves data center utilization and efficiency [8]. While specialized hardware accelerators, such as ASICs or FPGAs, can provide high efficiency for single applications, they may reduce the flexibility gained by general-purpose accelerators, such as GPUs. For example, long reconfiguration times of reprogrammable hardware, milliseconds to seconds [47, 49], may mitigate the

benefits gained by the accelerator when switching between applications. In this section, we evaluate the potential for workload consolidation on GPUs, which may provide advantages over other hardware accelerators in the data center.

However, current GPUs do not support preemptive [53] or spatial [1] multitasking for GPU computing although they do support preemptive multitasking for graphics [42]. When multiple CUDA applications run concurrently, their individual CUDA kernel launches contend for access to the GPU and, depending on resource constraints, are granted access in a first-come, first-serve basis by the NVIDIA driver. Large CUDA kernels with many CTAs may consume all of the GPU resources, blocking another CUDA kernel from running until completed. However, we can potentially exploit this property through a naive approach to enable finer grained multitasking by splitting a single CUDA kernel into multiple kernels with fewer CTAs.

We study a hypothetical low-priority background task (BGT) that performs a simple vector multiplication in global memory requiring a total of 256 CTAs with 1024 threads each to complete. The low-priority BGT is divided into many smaller short running kernel launches, which can be interleaved with MemcachedGPU processing. This creates a two-level, software/hardware CTA scheduler. For example, if we reduce the background task to 16 CTAs per kernel, we require 16 separate kernel launches to complete the task.

We run MemcachedGPU using *GNoM* on the Tesla K20c at a lower *GET* request throughput of 4 MRPS using 16 byte keys. After some time, the BGT is launched concurrently on the same GPU, varying the number of CTAs per kernel launch. Figure 6 measures the average client RTT based on a 256 request window during the BGT execution. Without the BGT, MemcachedGPU has an average RTT $< 300\mu$s. With 256 CTAs (1 kernel launch), the BGT consumes the GPU resources causing a large disruption in the RTT. Even after the BGT completes with 256 CTAs (around 32*m*s), MemcachedGPU takes over 20*m*s to return back to the original average RTT. As the number of CTAs per kernel is reduced, the impact of the BGT on MemcachedGPU reduces significantly. For example, at 16 CTAs per kernel, the RTT is disrupted for $\sim$2.4*m*s during the initial BGT kernel launch, with a maximum average RTT of $\sim$1.5*m*s, and then returns back to under 300μs while the BGT is executing.

However, increasing the number of BGT kernel launches also impacts the the BGT execution time. Figure 7 measures the BGT execution time with and without MemcachedGPU, as well as the maximum average RTT seen by MemcachedGPU during the BGT execution. At 4 MRPS, MemcachedGPU has very little impact on the BGT execution time due to its low resource utilization and small kernel launches. As the number of BGT kernels increases, the execution time also increases due to the introduction of the software CTA scheduler and contention with competing kernel launches. However, the impact on MemcachedGPU RTT decreases much faster. At 16 CTAs per kernel, the BGT execution time is increased by $\sim$50% versus 256 CTAs, while the MemcachedGPU RTT is reduced by over 18X. Allowing for an increase in the low-priority BGT completion time, *GNoM* is able to provide reasonable QoS to MemcachedGPU when run with another application.

Table 5: MemcachedGPU Offline - 16B keys, 96B packets.

	Tesla K20c	GTX Titan	GTX 750Ti
Throughput (MRPS)	27.5	27.7	28.3
Avg. latency (us)	353.4	301.1	263.6
Energy-efficiency (KRPS/W)	100	89	127.3

5.5 MemcachedGPU Offline Limit Study

In this section, we evaluate an offline, in-memory framework that reads network request traces directly from CPU memory to evaluate the peak performance and efficiency of *GNoM* and MemcachedGPU independent of the network. The same *GNoM* framework described in Section 3.2 is used to launch the GPU kernels (*GNoM-pre*), perform GPU UDP and *GET* request processing (*GNoM-dev*), and populate dummy response packets upon kernel completion (*GNoM-post*). The same packet trace used in Section 5.3, with the minimum key size of 16 bytes to stress *GNoM*, is used in the offline evaluation.

Table 5 presents the offline throughput, latency, and energy-efficiency of MemcachedGPU for the three GPUs in Table 2. Each GPU achieves over 27 MRPS ($\sim$21.5 Gbps), suggesting that the GPUs are capable of handling over 2X the request throughput measured in the online evaluations. Assuming the PCIe bus is not a bottleneck, achieving this high throughput would require additional NICs and removing the limitation on the amount of pin-able GPU memory to allocate more GRXBs for the increased request rate.

The latency in Table 5 measures the time prior to copying the packets to GPU and after populating the dummy response packets at peak throughputs. An interesting result of this study was that the low-power GTX 750Ti reduced the average batch latency compared to the Tesla K20c by $\sim$25%, while also slightly improving peak throughput. This improvement can be attributed to many of the different architectural optimizations in Maxwell over Kepler [46]. Additionally, MemcachedGPU contains many memory accesses with little processing per packet, which reduces the benefits of the higher computational throughput GPUs.

Finally, the energy-efficiency in Table 5 measures the system wall power at the peak throughputs and includes the TDP of the additional NICs required to support the increased throughput. The GTX 750Ti is able to process over 27% and 43% more *GET* requests per watt than the Tesla K20c and GTX Titan respectively.

5.6 Comparison with Previous Work

This section compares MemcachedGPU against reported results in prior work. Table 6 highlights the main points of comparison for MemcachedGPU. Results not provided in

Table 6: Comparing MemcachedGPU with Prior Work.

Platform	MRPS	Lat. (μs)	KRPS/W	KRPS/$	Year / *nm*
MemcGPU Tesla (online)	12.9-13	m <800, $p95$<1100	62	4.3	'14/28
MemcGPU Tesla (online)	9	$p95 < 500$	45	3	'14/28
MemcGPU GTX 750Ti (NGD online)	12.85	m <830, $p95$<1800	84.8	25.7	'14/28
MemcGPU GTX 750Ti (offline)	28.3	–	127.3	56.6	'14/28
Vanilla Memc - 4 threads	0.93	$p95$<677 - 0.5 MRPS	6.6	2.67	'13/22
Vanilla Memc + *PF_RING*- 2 threads	1.82	$p95$<607 - 1 MRPS	15.89	5.2	'13/22
Flying Memc [16]	1.67	$m < 600$	8.9	2.6	'13/28
MICA - 2x Intel Xeon E5-2680 (online,**4 NICs**) [36]	76.9	$p95 < 80$	–	22	'12/32
MICA - 2x Intel Xeon E5-2680 (offline) [36]	156 (avg. of uni. & skew.)	–	–	44.5	'12/32
MemC3 - 2x Intel Xeon L5640 [36]	4.4	–	–	12.9	'10/32
FPGA [11, 26]	13.02	3.5-4.5	106.7	1.75	'13/40

the published work or not applicable are indicated by "–". The cost-efficiency (KRPS/$) only considers the current purchase cost of the CPU and the corresponding accelerator, if applicable. All other costs are assumed to be the same between systems. The last column in Table 6 presents the year the processor was released and the process technology (*nm*).

Table 6 also presents our results for the vanilla CPU Memcached and vanilla CPU Memcached using *PF_RING* to bypass the Linux network stack. No other optimizations were applied to the baseline Memcached. These results highlight that while bypassing the Linux network stack can increase performance and energy-efficiency, additional optimizations to the core Memcached implementation are required to continue improving efficiency and scalability.

Aside from MICA [36], MemcachedGPU improves or matches the throughput compared to all other systems. However, an expected result of batch processing on a throughput-oriented accelerator is an increase in request latency. The CPU and FPGA process requests serially, requiring low latency per request to achieve high throughput. The GPU instead processes many requests in parallel to increase throughput. As such, applications with very low latency requirements may not be a good fit for the GPU. However, even near 10 GbE line-rate MemcachedGPU achieves a 95-percentile RTT under 1.1*m*s and 1.8*m*s on the Tesla K20c (*GNoM*) and GTX 750Ti (NGD) respectively.

MemcachedGPU is able to closely match the throughput of an optimized FPGA implementation [11, 26] at all key and value sizes, while achieving 79% of the energy-efficiency on the GTX 750Ti. Additionally, the high cost of the Xilinix Virtex 6 SX475T FPGA (e.g., $7100+ on digikey.com) may enable MemcachedGPU to improve cost-efficiency by up to 14.7X on the GTX 750Ti ($150). While an equivalent offline study to Section 5.5 is not available for the FPGA, the work suggests that the memory bandwidth is tuned to match the 10 GbE line-rate, potentially limiting additional scaling on the current architecture. This provides promise for the low-power GTX 750Ti GPU in the offline analysis, which may be able to further increase throughput and energy-efficiency up to 2.2X and 1.2X respectively. Furthermore, the GPU can provide other benefits over the FPGA, such as ease of programming and a higher potential for workload consolidation (Section 5.4).

Flying Memcache [16] uses the GPU to perform the Memcached key hash computation, while all other network and Memcached processing remains on the CPU. *GNoM* and MemcachedGPU work to remove additional serial CPU processing bottlenecks in the *GET* request path, enabling 10 GbE line-rate processing at all key/value sizes. Flying Memcache provides peak results for a minimum value size of 250B. On the Tesla K20c with 250B values, MemcachedGPU improves throughput and energy-efficiency by 3X and 2.6X respectively, with the throughput scaling up to 7.8X when using 2B values.

The recent state-of-the-art CPU Memcached implementation, MICA [36], achieves the highest throughput of all systems on a dual 8-core Intel Xeon system with four dual-port 10 GbE NICs. Similar to MemcachedGPU, MICA makes heavy modifications to Memcached and bypasses the Linux network stack to improve performance, some of which were adopted in MemcachedGPU (Section 4). Additional modifications, such as the log based value storage, could also be implemented in MemcachedGPU. MICA's results include *GETs* and *SETs* (95:5 ratio) whereas the MemcachedGPU results consider 100% *GET* requests, however, MICA also modified *SETs* to run over UDP, which may limit the effectiveness in practice. Additionally, MICA requires modifications to the Memcached client to achieve peak throughputs, reducing to ∼44% peak throughput without this optimization. In the online NGD framework, the GTX 750Ti may improve cost-efficiency over MICA by up to 17%. MICA presents an offline limit study of their data structures without any network transfers or network processing, reaching high throughputs over 150 MRPS. In contrast, all of the UDP packet data movement and processing is still included in the offline MemcachedGPU study (Section 5.5); however, UDP packets are read from CPU memory instead of over the network. In the offline analysis, the GTX 750Ti may improve cost-efficiency over MICA up to 27%. We were not able to compare the energy-efficiency of MemcachedGPU with MICA as no power results were presented.

6. Related Work

Concurrent with our work, Zhang et al. [59] also propose using GPUs to accelerate in-memory key-value stores. They use two 8-core CPUs, two dual-port 10 GbE NICs (max. 40 Gbps), and two NVIDIA GTX 780 GPUs to achieve throughputs over 120 MRPS. In contrast to MemcachedGPU, these results use a smaller minimum key size (8B vs. 16B), use a compact key-value protocol independent from the Memcached ASCII protocol, and batch multiple

requests and responses into single network requests through multi-GETs to reduce per-packet network overheads. However, for *GET* requests, the GPU is only used to perform a parallel lookup for the corresponding CPU key-value pointer in a GPU Cuckoo hash table. All other processing, including UDP network processing, request parsing, key hashing, and key comparisons, are done on the CPU. In contrast, the goal of our work was to achieve 10 GbE line-rate performance for all key-value sizes while performing all of the UDP network processing and *GET* request processing on the GPU. We have not yet evaluated the potential for additional scaling of *GNoM* and MemcachedGPU using a more powerful CPU with multiple NICs and GPUs.

Recent work by Kim et al. [33] present GPUnet, a networking layer and socket level API for GPU programs. Similar to *GNoM*, GPUnet aims to improve the support for applications requiring networking I/O on GPU accelerators. However, GPUnet is designed for Infiniband hardware and is directed towards more traditional GPU applications, such as image processing and MapReduce. The current cost of Infiniband hardware is a large contributor to the restricted usage outside of HPC. *GNoM* instead targets commodity Ethernet hardware typically used in data centers and evaluates a non-traditional GPU application, Memcached.

Other works have evaluated packet acceleration on GPUs [21, 32, 55]. Similar to *GNoM*, these works exploit packet-level parallelism on the massively parallel GPU architecture, requiring a host framework to efficiently manage tasks and data movement. Packet Shader [21] implements a software router framework on GPUs to accelerate the lower-level network layer, whereas GNoM and MemcachedGPU focus on transport and application levels. Other recent works [32, 55] evaluate GPUs for higher-level network layer processing, including stateful protocols such as TCP. However, their focus is more towards individual packet processing applications, such as packet filtering and network intrusion detection.

Our prior work [23] initially explored the potential benefits of exploiting request-level parallelism for a subset of Memcached *GET* request processing on GPUs. Dimitris et al. [16] also address the networking bottlenecks in a version of Memcached where the GPU is used for performing the key hash while all other processing remains on the CPU. In contrast, this work performs all Memcached *GET* request and network processing on the GPU, addressing many of the challenges associated with a full system implementation.

Prior work has evaluated server workloads on GPUs, such as Memcached [23], HTTP workloads [2], or database queries [7, 57]. These works highlight the benefits of exploiting request level parallelism on the GPU through batching. However, they focus solely on the workload specific processing. In contrast, this work considers a full system implementation including end-to-end network measurements.

Many prior works have looked at improving the throughput and scalability of Memcached or other general key-value store applications through software or hardware modifications [4, 10, 18, 19, 28, 29, 36, 40, 56], many of which are orthogonal to our work. As described in Section 4, we have adopted some of the Memcached optimizations presented in [19, 36], and many others could be implemented to further improve performance and storage efficiency. While the focus of this paper is on Memcached, one of the long term goals of our work is to provide a general framework for accelerating high-throughput network services on GPUs.

Other works have considered using FPGAs to accelerate Memcached [11, 12, 37]. While the FPGA architecture enables high energy-efficiency and high performance, the flexibility of the general-purpose GPU architecture (e.g., ease of programming, multitasking), may outweigh some of the efficiency gains in the data center.

Recent works have evaluated different operating system abstractions on GPUs, such as file systems [52], resource management and scheduling [30, 50, 53], exceptions and speculative execution [39], and virtualization [17, 20, 51]. These works are complementary to our work with a common goal of improving system-software support on GPUs.

7. Conclusion

We have described *GNoM*, a GPU-accelerated networking framework, which enables high-throughput, network-centric applications to exploit massively parallel GPUs to execute both network packet processing and application code. This framework allows a single GPU-equipped data center node to service network request at $\sim$10 GbE line-rates, while maintaining acceptable latency even while processing background, lower priority batch jobs. Using *GNoM*, we described an implementation of Memcached, MemcachedGPU. MemcachedGPU is able to achieve $\sim$10 GbE line-rate processing at all request sizes, using only 16.1μJ and 11.8μJ of energy per request, while maintaining a client visible $p95$ RTT latency under 1.1ms and 1.8ms on a high-performance NVIDIA Tesla GPU and low-power NVIDIA Maxwell GPU respectively. We also performed an offline limit study and highlight that MemcachedGPU may be able to scale up to 2X the throughput and 1.5X the energy-efficiency on the low-power NVIDIA Maxwell GPU. We believe that future GPU-enabled systems which are more tightly integrated with the network interface and less reliant on the CPU for I/O will enable higher performance and lower energy per request. Overall, we demonstrated the potential to exploit the efficient parallelism of GPUs for network-oriented data center services.

Acknowledgments

The authors would like to thank the reviewers and our shepherd, Joseph Gonzalez, for their insightful feedback. This research was funded by the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[3] B. Aker. libMemcached. http://libmemcached.org/libMemcached.html.

[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)*, 2009.

[5] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

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

[8] L. A. Barroso and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2nd Edition. *Synthesis Lectures on Computer Architecture*, 2013.

[9] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14. ACM, 2014.

[10] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core Key-value Store. In *Proceedings of the 2011 International Green Computing Conference and Workshops (IGCC)*, 2011.

[11] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bar, and Z. Istvan. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[12] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, 2010. ACM.

[14] A. Corporation. Implementing fpga design with the opencl standard. https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf, 11 2013.

[15] J. Dean. Large scale deep learning. Keynote GPU Technical Conference 2015, 03 2015.

[16] D. Deyannis, L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis. Flying memcache: Lessons learned from different acceleration strategies. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 2014.

[17] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Operating Systems Review*, July 2009.

[18] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th Usenix Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

[20] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2009.

[21] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. *SIGCOMM Computer Communications Review*, October 2010.

[22] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *Distributed Computing*. Springer, 2008.

[23] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *Proceeding of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[24] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, July/August 2010.

[25] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W.-M. Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Proceedings of the 2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010.

[26] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A flexible hash table design for 10gbps key-value stores on fpgas. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013.

[27] B. Jenkins. Function for Producing 32bit Hashes for Hash Table Lookup. http://burtleburtle.net/bob/c/lookup3.c, 2006.

[28] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In

Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2012.

[29] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasiur Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, 2011.

[30] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*, 2012.

[31] Khronos OpenCL Working Group. *The OpenCL Specification*, 1.1 edition, 2011.

[32] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15. ACM, 2015.

[33] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. Gpunet: Networking abstractions for gpu programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Oct. 2014.

[34] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26, 2007.

[35] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th Usenix Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[37] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. *SIGARCH Computer Architecture News*, June 2013.

[38] Memcached. A Distributed Memory Object Caching System. http://www.memcached.org.

[39] J. Menon, M. De Kruijf, and K. Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. Mcelroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, V. Venkataramani, and F. Inc. Scaling Memcached at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[41] ntop. PF_RING. http://www.ntop.org/products/pf_ring/.

[42] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[43] NVIDIA Corporation. NVIDIA CUDA C Programming Guide v4.2. http://developer.nvidia.com/nvidia-gpu-computing-documentation/, 2012.

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

[45] NVIDIA Corporation. Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2014.

[46] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti: Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.

[47] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July 2009.

[48] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[49] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[50] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[51] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, June 2012.

[52] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. *SIGARCH Computer Architecture News*, March 2013.

[53] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[54] ThinkTank Energy Products. Watts up? Plug Load Meters. https://www.wattsupmeters.com/secure/index.php.

[55] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. Gaspp: a gpu-accelerated stateful packet processing framework. In *USENIX ATC*, 2014.

[56] A. Wiggins and J. Langston. Enhancing the Scalability of Memcached. https://software.intel.com/sites/default/files/m/0/b/6/1/d/45675-memcached_05172012.pdf.

[57] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUs using Kernel Fusion/Fission. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012.

[58] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[59] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11), 2015.