GPUnet: Networking Abstractions for GPU Programs

MARK SILBERSTEIN, Technion—Israel Institute of Technology
SANGMAN KIM, SEONGGU HUH, XINYA ZHANG, and YIGE HU,
University of Texas at Austin
AMIR WATED, Technion—Israel Institute of Technology
EMMETT WITCHEL, University of Texas at Austin

Despite the popularity of GPUs in high-performance and scientific computing, and despite increasingly general-purpose hardware capabilities, the use of GPUs in network servers or distributed systems poses significant challenges.

GPUnet is a native GPU networking layer that provides a socket abstraction and high-level networking APIs for GPU programs. We use GPUnet to streamline the development of high-performance, distributed applications like in-GPU-memory MapReduce and a new class of low-latency, high-throughput GPU-native network services such as a face verification server.

CCS Concepts: • **Networks → Programming interfaces**; • **Software and its engineering → Input/ output;**

Additional Key Words and Phrases: Operating systems design, GPGPUs, network servers, accelerators

ACM Reference Format:
Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking abstractions for GPU programs. ACM Trans. Comput. Syst. 34, 3, Article 9 (September 2016), 31 pages.
DOI: http://dx.doi.org/10.1145/2963098

1. INTRODUCTION

GPUs have become the platform of choice for many types of parallel general-purpose applications, from machine learning to molecular dynamics simulations [NVIDIA 2016]. However, harnessing GPUs' impressive computing capabilities in complex software systems like network servers remains challenging: GPUs lack software abstractions to direct the flow of data within a system, leaving the developer with only low-level control over I/O. Therefore, certain classes of applications that could benefit from the GPU's computational density require unacceptable development costs to realize their full performance potential.

Whereas GPU hardware architecture has matured to support general-purpose parallel workloads, the GPU software stack has hardly evolved beyond bare-metal interfaces (e.g., memory transfer via direct memory access (DMA)). Without core

M. Silberstein was supported by the Israel Science Foundation (grant 1138/14) and the Israeli Ministry of Science. The authors also gratefully acknowledge funding from National Science Foundation grants CNS-1017785 and CCF-1333594.

Authors' addresses: M. Silberstein and A. Wated, Electrical Engineering Department, Technion; emails: mark@ee.technion.ac.il, amirw@tx.technion.ac.il; S. Kim, S. Huh, X. Zhang, Y. Hu, and E. Witchel, Computer Science Department, University of Texas at Austin; emails: sangmank@cs.utexas.edu, seonggu.huh@gmail.com, {zxy, yige, witchel}@cs.utexas.edu.

DOI: http://dx.doi.org/10.1145/2963098

I/O abstractions like sockets available to GPU code, GPU programs that access the network must coordinate low-level, machine-specific details among a CPU, GPU, and a NIC, such as managing buffers in weakly consistent GPU memory or optimizing NIC-to-GPU transfers via P2P DMAs.

This article introduces GPUnet, a native GPU networking layer that provides a socket abstraction and high-level networking APIs to GPU programs. GPUnet enables individual threads in one GPU to communicate with threads in other GPUs or CPUs via standard and familiar socket interfaces, regardless of whether they are in the same or different machines. Native GPU networking cuts the CPU out of GPU-NIC interactions, simplifying code and increasing performance. It also unifies application compute and I/O logic within the GPU program, providing a simpler programming model. GPUnet uses advanced NIC and GPU hardware capabilities and applies sophisticated code optimizations that yield high application performance equal to or exceeding hand-tuned traditional implementations.

GPUnet is designed to foster GPU adoption in two broad classes of high-throughput data center applications: network servers for back-end data processing (e.g., media filtering or face recognition) and scale-out distributed computing systems (e.g., MapReduce). The requirements of high compute density, throughput, and power efficiency in such systems motivates using discrete high-end GPUs as the target platform for GPUnet. Discrete GPUs are mounted on an expansion card with dedicated hardware resources like memory. Although discrete GPUs are broadly used in supercomputing systems, their deployment in data centers has been limited. We blame the added design and implementation complexity of integrating GPUs into complex software systems; consequently, GPUnet's goal is to facilitate such integration.

Three essential characteristics make developing efficient network abstractions for discrete GPUs challenging: massive parallelism, slow access to CPU memory, and low single-thread performance. GPUnet accommodates parallelism at the API level by providing coalesced calls invoked by multiple GPU threads at the same point in data-parallel code. For instance, a GPU program computing a vector sum may receive input arrays from the network by calling `recv()` in thousands of GPU threads. These calls will be coalesced into a single receive request to reduce the processing overhead of the networking stack. GPUnet uses recent hardware support for network transmission directly into/from GPU memory to minimize slow accesses from the GPU to system memory. It provides a reliable stream abstraction with GPU-managed flow control. Finally, GPUnet minimizes control-intensive sequential execution on performance-critical paths by offloading message dispatching to the NIC via remote direct memory access (RDMA) hardware support. The GPUnet prototype supports sockets for network communications over InfiniBand RDMA and supports interprocess communication on a local machine (often called *UNIX-domain sockets*).

We build a face verification server using the GPUnet prototype that matches images and interacts with `memcached` directly from GPU code, processing 53K client requests/second on a single NVIDIA K20Xm GPU, exceeding the throughput of a six-core Intel CPU and a CUDA-based server by 1.5× and 2.3×, respectively, while maintaining 3× lower latency than the CPU and requiring half as much code than other versions. We also implement a distributed in-GPU-memory MapReduce framework, where GPUs fully control all of the I/O: they read and write files (via GPUfs [Silberstein et al. 2014a]), and communicate over InfiniBand with other GPUs. This architecture demonstrates the ability of GPUnet to support complex communication patterns across GPUs, and for word count and K-means workloads, it scales to four GPUs (over a 40Gb InfiniBand QDR network) providing speedups of 2.9 to 3.5× over one GPU.

This article begins with the motivation for building GPUnet (Section 2), a review of the GPU and network hardware architecture (Section 3), and high-level design considerations (Section 4). It then makes the following contributions:

—It presents for the first time a socket abstraction, API, and semantics suitable for use with general-purpose GPU programs (Section 5).
—It presents several novel optimizations for enabling discrete GPUs to control network traffic (Section 6).
—It develops three substantial GPU-native network applications: a matrix product server, in-GPU-memory MapReduce, and a face verification server (Section 7).
—It evaluates GPUnet primitives and entire applications including multiple workloads for each of the three application types (Section 8).

2. MOTIVATION

GPUs are widely used for accelerating parallel tasks in high-performance computing, and their architecture has been evolving to enable efficient execution of complex, general-purpose workloads. However, the use of GPUs in network servers or distributed systems poses significant challenges. The list of 200 popular general-purpose GPU applications recently published by NVIDIA [2016] has no mention of GPU-accelerated network services. Using GPUs in software routers and SSL protocols [Jang et al. 2011; Han et al. 2010; Sun and Ricci 2013], as well as in distributed applications [Coates et al. 2013], results in impressive speedups but requires significant development efforts. Recent work shows that GPUs can boost power efficiency and performance for Web servers [Agrawal et al. 2014], but the GPU prototype lacked an actual network implementation because GPU-native networking support does not yet exist. We believe that enabling GPUs to access network hardware and the networking software stack directly, via familiar network abstractions like sockets, will hasten GPU integration in modern network systems.

GPUs currently require application developers to build complicated CPU-side code to manage access to the host's network. If an input to a GPU task is transferred over the network, for example, the CPU-side code handles system-level I/O issues, such as how to overlap data access with GPU execution and how to manage the size of memory transfers. The GPU application programmer has to deal with bare-metal hardware issues like setting up peer-to-peer (P2P) DMA over the PCI Express (PCIe) bus. P2P DMA lets the NIC directly transfer data to and from high-bandwidth graphics double data rate (GDDR) GPU local memory. Direct transfers between the NIC and GPU eliminate redundant PCIe transfers and data copies to system memory, improving data transfer throughput and reducing latency (Section 8.1). Enjoying the benefits of P2P DMA, however, requires intimate knowledge of hardware-specific APIs and characteristics, such as the underlying PCIe topology.

These issues dramatically complicate the design and implementation of GPU-accelerated networking applications, turning their development into a low-level system programming task. Modern CPU operating systems provide high-level I/O abstractions like sockets, which eliminate or hide this type of programming complexity from ordinary application developers. GPUnet is intended to do the same for GPU programmers.

Consider an internal data center network service for on-demand face-in-a-crowd photo labeling. The algorithm detects faces in the input image, creates face descriptors, fetches the name label for each descriptor from a remote database, and returns the location and the name of each recognized face in the image. This task is a perfect candidate for GPU acceleration because some face recognition algorithms are an order of magnitude faster on GPUs than on a single CPU core [Gupta 2013], and by connecting multiple GPUs, server compute density can be increased even further. Designing such a GPU-based service presents several system-level challenges.

No GPU network control. A GPU cannot initiate network I/O from within a GPU kernel. Using P2P DMA, the NIC can place network packets directly in local GPU memory, but only CPU applications control the NIC and perform send and receive. In the traditional GPU-as-coprocessor programming model, a CPU cannot retrieve partial results from GPU memory while a kernel producing them is still running. Therefore, a programmer needs to wait until all GPU threads terminate to request a CPU to invoke network I/O calls. This awkward model effectively forces I/O to occur only on GPU kernel invocation boundaries. In our face recognition example, a CPU program would query the database soon after detecting even a single face to pipeline continued facial processing with database queries. Current GPU programming models make it difficult to achieve this kind of pipelining because GPU kernels must complete before they perform I/O. Thus, all database queries will be delayed until after the GPU face detection kernel terminates, leading to increased response time.

Complex multistage pipelining. Unlike in CPUs, where operating systems use threads and device interrupts to overlap data processing and I/O, GPU code traditionally requires all input to be transferred in full to local GPU memory before processing starts. To overlap data transfers and computations, optimized GPU designs use pipelining: they split inputs and outputs into smaller chunks and asynchronously invoke the kernel on one chunk while simultaneously transferring the next input chunk to the GPU and the prior output chunk from the GPU. Although effective for GPU-CPU interaction, pipelines can quickly grow into a complex multistage dataflow involving GPU-CPU data transfers, GPU invocations, and processing of network events. In addition to the associated implementation complexity, achieving high performance requires tedious tuning of buffer sizes that depend on a particular generation of hardware.

Complex network buffer management. If P2P DMA functionality is available, CPU code must set up the GPU-NIC DMA channel by preallocating dedicated GPU memory buffers and registering them with the NIC. Unfortunately, these GPU buffers are hard to manage because the network transfers are controlled by a CPU. For example, if the image data exceeds the allocated buffer size, the CPU must allocate and register another GPU buffer (which is slow and may exhaust NIC or GPU hardware resources), or the buffer must be freed by copying the old contents to another GPU memory area. GPU code must be modified to cope with input stored in multiple buffers. While on a CPU, the networking API hides system buffer management details and lets the application determine the buffer size according to its internal logic rather than GPU and NIC hardware constraints.

GPUnet aims to address these challenges. It exposes a single networking abstraction across all processors in the system and provides a standard, familiar API, thereby simplifying GPU development and facilitating the integration of GPU programs into complex software systems.

3. HARDWARE ARCHITECTURE OVERVIEW

We now provide an overview of the GPU software/hardware model, RDMA networking, and P2P DMA concepts. We use NVIDIA CUDA terminology because we implement GPUnet on NVIDIA GPUs, but most other GPUs that support the cross-platform OpenCL standard [Khronos Group 2016] share the same concepts.

3.1. GPU Software/Hardware Model

GPUs are parallel processors that expose programmers to hierarchically structured hardware parallelism (for full details, see Kirk and Wen-mei [2010]). They comprise several big cores, streaming multiprocessors (SMs), each having multiple hardware

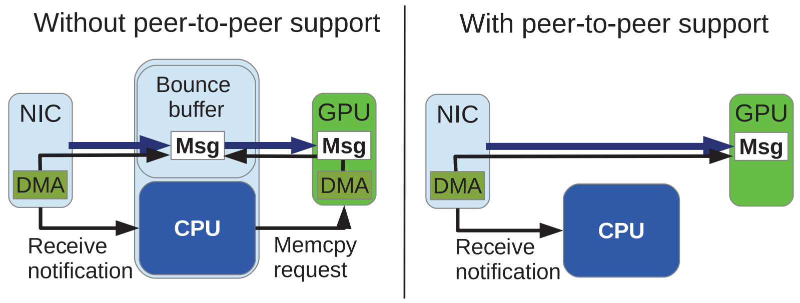

Fig. 1. Receiving network messages into a GPU. Without P2P DMA, the CPU must use a GPU DMA engine to transfer data from the CPU bounce buffer.

contexts and several single instruction, multiple data (SIMD) units. All SMs access global GPU memory and share an address space.

The programming model associates a GPU thread with a single element of a SIMD unit. Threads are grouped into threadblocks, and all threads in a threadblock are executed on the same SM. The threads within a threadblock may communicate and share state via on-die shared memory and synchronize efficiently. Synchronization across threadblocks is possible, but it is much slower and limited to atomic operations. Therefore, most GPU workloads comprise multiple loosely coupled tasks, each running in a single threadblock. Each task in a threadblock is parallelized for tightly coupled parallel execution by the threads of the threadblock. Once a threadblock has been dispatched to an SM, the scheduled threadblock cannot be preempted and occupies that SM until all of the threadblock's threads terminate.[1]

The primary focus of this work is on discrete GPUs, which are peripheral devices connected to the host system via a standard PCIe bus. Discrete GPUs feature their own physical memory on the device, with a separate address space that cannot be referenced directly by CPU programs. Moving data in and out of GPU memory efficiently requires DMA.[2] The CPU prepares the data in GPU memory, invokes a GPU kernel, and retrieves the results after the kernel terminates.

Interaction with I/O devices. P2P DMA refers to the ability of peripheral devices to exchange data on a bus without sending data to a CPU or system memory. Modern discrete GPUs support P2P DMA between GPUs themselves, and between GPUs and other peripheral devices on a PCIe bus (e.g., NICs). For example, the Mellanox Connect-IB network card (Host Channel Adapter (HCA)) is capable of transferring data directly to/from the GPU memory of NVIDIA K20 GPUs (Figure 1). P2P DMA improves the throughput and latency of GPU interaction with other peripherals because it eliminates an extra copy to/from bounce buffers in CPU memory and reduces the load on system memory [Potluri et al. 2013a, 2013b].

RDMA and InfiniBand. RDMA allows remote peers to read from and write directly into application buffers over the network. Multiple RDMA-capable transports exist, such as Internet Wide Area RDMA Protocol (iWARP), InfiniBand, and RDMA over Converged Ethernet (RoCE). As network data transfer rates grow, RDMA-capable technologies have been increasingly adopted for in-data center networks, enabling

[1]Recent NVIDIA GPUs enable a threadblock to invoke another GPU kernel, which under the hood might result in preempting the running threadblock to free hardware resources for the new kernel. However, there is no API that allows preemption directly.
[2]NVIDIA CUDA 6.0 provides CPU-GPU software shared memory for automatic data management, but the data transfer costs remain.

high-throughput and low-latency networking, surpassing legacy Ethernet performance and cost efficiency [Tuneja Group Technology Analysts]. For example, the state-of-the-art Fourteen Data Rate (FDR) InfiniBand provides 56Gbps throughput and submicrosecond latency, with the 40Gbps quad data rate (QDR) technology widely deployed since 2009. InfiniBand is broadly used in supercomputing systems and enterprise data centers, and analysts anticipate significant growth in the coming years.

An InfiniBand NIC is referred to as an HCA and like other RDMA networking hardware, it performs full network packet processing in hardware, enables zero-copy network transmission to/from application buffers, and bypasses the OS kernel for network API calls.

The HCA efficiently dispatches thousands [InfiniBand Trade Association 2007] of network buffers, registered by multiple applications. In combination with P2P DMA, the HCA may access application buffers in GPU memory. RDMA functionality is exposed via a low-level *VERB* interface that is not easy to use. Instead, system software uses VERBs to implement higher-level data transfer abstractions. For example, the rsockets [Hefty 2012] library provides a familiar socket API in user-space for the RDMA transport. Rsockets are a drop-in replacement for sockets (via LD_PRELOAD), providing a simple path for networking over RDMA.

4. DESIGN CONSIDERATIONS

There are many alternative designs for GPU networking. This section discusses important high-level trade-offs.

4.1. Sockets and Alternatives

The GPUnet interface uses sockets. We believe that sockets are appropriate for a GPU-centric communication abstraction because they are standard, general, familiar, and convenient to use.

Sockets are a versatile abstraction and are currently used by a diverse set of applications. Since their inception in 4.2 BSD (1989), sockets have been the API of choice for various types of communication [Stevens 1993; Stevens et al. 2004]. The socket API remains valuable for a variety of communication tasks, as evidenced by the number of address families supported by the socket interface. Linux has added support for more socket address families over time, from 13 in kernel version 2.0.40 (released in June 1996), 35 in version 2.6.35 (in August 2011), to 39 in version 4.0 (in April 2015). Each address family represents a particular way applications use the socket interface. For example, UNIX domain sockets allow local processes to communicate (`AF_UNIX`), Internet sockets support wide-area communication (`AF_INET`, `AF_INET6`), and some sockets provide wireless communication (`AF_BLUETOOTH`, `AF_IRDA`) or communication for security features like key management (`AF_KEY`). Address families added between Linux kernel 2.6.35 and 4.0 show new types of applications embracing the socket interface, such as InfiniBand (`AF_IB`), near-field communication (`AF_NFC`), virtualization stack (`AF_VSOCK`), and the cryptographic API used in `dm-crypt` and OpenSSL (`AF_ALG`).

We also considered several alternatives to sockets. For example, RDMA is usually performed via the VERBs API, but this API is notoriously difficult to use [Trivedi et al. 2013]. Message Passing Interface (MPI) [Ohio State University Network-Based Computing Laboratory 2015] provides a high-level message passing API widely used in high-performance computing systems for building parallel applications, but it is quite uncommon in other networking applications, such as network servers. GPUnet aims to support a broad range of server-side network services, striking the balance between the simplicity and generality by providing the high-level reliable streaming abstraction and the low-level versatile socket API.

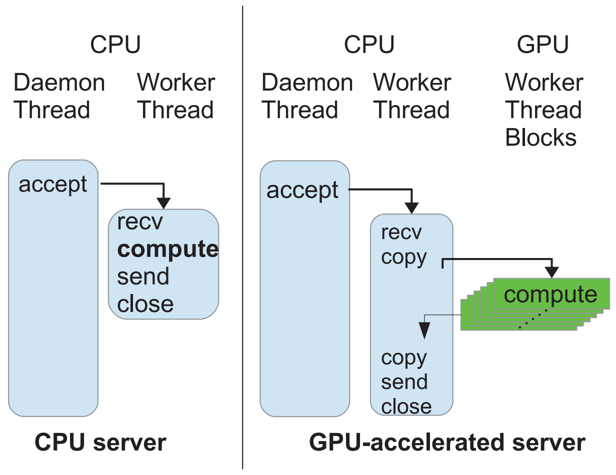

Fig. 2. Architecture of a network server on a CPU, using a GPU as a coprocessor, and with GPUnet (daemon architecture).

Although recent efforts to improve the network performance in modern multi-core computers proposed several changes to the socket design and implementation [Han et al. 2012; Shalev et al. 2010], sockets are currently the most commonly used standard interface for networking and serve as a reasonable first step toward providing networking capabilities in GPU programs.

4.2. Discrete GPUs

We develop GPUnet for discrete GPUs, which can fulfill the high performance requirements of data center server systems. GPUs integrated on hybrid CPU-GPU processors and system-on-chip designs are also gaining market share in low-end computer devices (e.g., Intel graphics, AMD APUs, NVIDIA Tegra, and Qualcomm Snapdragon.) Compared to discrete GPUs, these hybrid GPUs are more tightly integrated with the CPU and therefore embody different trade-offs between power consumption, production costs, and system performance. As a result, some of the GPUnet design choices suitable for discrete GPUs, such as the use of the network adapter for transport layer processing, might require revision to fit hybrid systems that have enhanced low-latency CPU-GPU communication.

We believe, however, that discrete and hybrid GPUs will continue to coexist for years to come. The aggressive, throughput-optimized hardware designs of discrete GPUs rely heavily on a multibillion transistor budget, tight integration with specialized high-throughput memory, and increased thermal design power (TDP). Therefore, discrete GPUs outperform hybrid GPUs by an order of magnitude in compute capacity and memory bandwidth, making them attractive for the data center and therefore a reasonable choice for prototyping GPU networking support.

4.3. Network Server Organization

Figure 2 depicts different organizations for a multithreaded network server. In a CPU server (left), a daemon thread accepts connections and transfers the socket to worker threads. In a traditional GPU-accelerated network server (middle), the worker threads invoke computations on a GPU. GPUs are treated as bulk-synchronous high-performance accelerators, so all of the inputs are read on the CPU first and transferred to the GPU across a PCIe bus. This design requires large batches of work to amortize

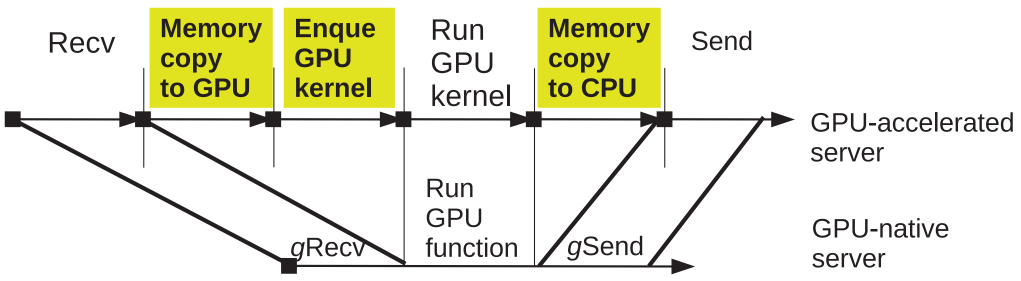

Fig. 3. Logical stages for a task processed on a GPU-accelerated CPU server (top) and GPU-native network server (bottom). Highlighted stages are eliminated by the GPU networking support.

CPU-GPU communications and invocation overheads, which otherwise dominate the execution time. For example, SSLShader [Jang et al. 2011] needs 1,024 independent network flows on a GTX580 GPU to surpass the performance of 128-bit AES-CBC encryption of a single AES-NI enabled CPU. Batching complicates the implementation and leads to increased response latency, because GPU code does not communicate with clients directly.

GPUnet makes it possible for GPU servers to handle multiple independent requests without having to batch them first (far right in Figure 2), much like multitasking in multicore CPUs. We call this organization the *daemon architecture*. It is also possible to have a GPUnet server where each threadblock acts as an independent server, accepting, computing, and responding to requests. We call this the *independent architecture*. We measure both in Section 8.

GPUnet changes the trade-offs that a designer must consider for a networked service because GPUnet removes the need to batch work so heavily, thereby greatly simplifying the programming model. It is our hope that this model will make the computational power of GPUs more easily accessible to networked services, but it will require the development of native GPU programs.[3]

4.4. In-GPU Networking Performance Benefits

A native GPU networking layer can provide significant performance benefits for building low-latency servers on modern GPUs, as it eliminates the overheads associated with the standard programming model of using GPUs as accelerators.

Figure 3 illustrates the flow of a server request on a traditional GPU-accelerated server (top) and compares it to the flow on a server using GPU-native networking support. In-GPU networking eliminates the overheads of CPU-GPU data transfer and kernel invocation, which penalize short requests. For example, computing the matrix product of two 64×64 matrices on a TESLA K20c GPU requires about 14μsec of computation. In comparison, we measure GPU kernel invocation requiring an average of 25μsec, and CPU-GPU-CPU data transfers for this size input average 160μsecs.

In-GPU networking may eliminate the kernel invocation entirely and provides a convenient interface to network buffers in GPU memory. One potential caveat, however, is that I/O activity on a GPU reduces the GPU's computing capacity, because unlike in CPUs, GPU I/O calls do not relinquish the GPU's resources, as we discuss in Section 8. Future hardware support for coarse-grain preemption mechanisms may alleviate this limitation [Zeno and Silberstein 2016].

[3]Unlike traditional GPU-accelerated programs where computations are offloaded to a GPU from the main CPU process that runs the bulk of the program logic, native GPU programs are self-sufficient, contain no CPU code, and run almost entirely on a GPU.

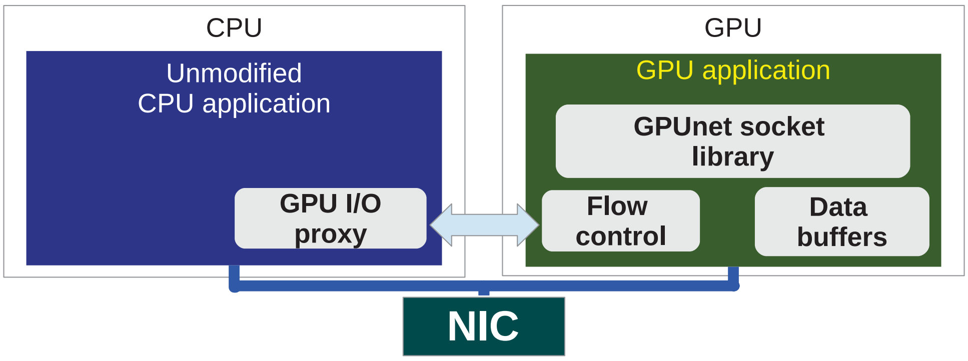

Fig. 4. GPUnet high-level design.

5. GPUNET DESIGN

Figure 4 shows the high-level architecture of GPUnet. GPU programs can access the network via standard socket abstractions provided by the GPUnet library, linked into the application's GPU code. CPU applications may use standard sockets to connect to remote GPU sockets. GPUnet stores network buffers in GPU memory, keeps track of active connections, and manages control flow for the network streams associated with the sockets. The GPUnet library works with the host OS on the CPU via a GPUnet I/O proxy to coordinate GPU access to the NIC and to the system's network port namespace.

Our goals for GPUnet include the following:

(1) *Simplicity*: Enable common network programming practices and provide a standard socket API and an in-order reliable stream abstraction to simplify programming and leverage existing programmer expertise.
(2) *Compatibility with GPU programming*: Support common GPU programming idioms like threadblock-based task parallelism and use on-chip scratchpad memory for application buffers.
(3) *Compatibility with CPU endpoints*: A GPUnet network endpoint has identical capabilities as a CPU network endpoint, ensuring compatibility between networked services on CPUs and GPUs.
(4) *NIC sharing*: Enable all GPUs and CPUs in a host to share the NIC hardware, allowing concurrent use of a NIC by both CPU and GPU programs.
(5) *Namespace sharing*: Share a single network namespace (ports, IP addresses, UNIX domain socket names) among CPUs and GPUs in the same machine to ensure backward compatibility and interoperability of CPU- and GPU-based networking code.

5.1. GPU Networking API

Socket abstraction. GPUnet sockets are similar to CPU sockets. As in a CPU, a GPU thread may open and use multiple sockets concurrently and GPU sockets are shared across all GPU threads. GPUnet supports the main calls in the standard network API, including `connect`, `bind`, `listen`, `accept`, `send`, `recv`, `sendto`, `recvfrom`, `shutdown`, and `close` and their nonblocking versions. In the article and in the actual implementation, we add a "*g*" prefix to emphasize that the code executes on a GPU. These calls work similarly to sockets on the CPUs, although we introduce coalesced multithreaded API calls, as we now explain.

Coalesced API calls. A traditional CPU network API is single threaded—that is, each thread can make independent API calls and receive independent results. GPU

```
increment_by_one_server(int csoc)
{
  //buffer shared by all threadblock threads
   __shared__ float buf[NUM_THREADS];
  size_t len=NUM_THREADS*sizeof(float);
  //collaborative recv into buf
  grecv(csoc, buf, len);
  //data parallel code per thread
  //thread_id provided by GPU hardware
  buf[thread_id]++;
  //collaborative send from buf
  gsend(csoc, buf, len);
}
```

Fig. 5. GPU network server that receives an array, increments every element by one, and sends the result back. The kernel is invoked with `NUM_THREADS` per threadblock.

threads, however, behave differently from CPU threads. They are orders of magnitude slower, and the hardware is designed to run groups of threads (e.g., 32 in an NVIDIA warp or 64 in an AMD wavefront) in lockstep, performing poorly if these threads execute divergent control paths. GPU hardware facilitates collaborative processing inside a threadblock by providing efficient sharing and synchronization primitives for the threads in the same threadblock. GPU programs, therefore, are designed with hierarchical parallelism in mind: they exploit coarse-grain task parallelism across multiple threadblocks and process a single task using all threads in a threadblock jointly rather than in each thread separately. Performing data-parallel API calls in such code is more natural than the traditional per-thread API used in CPU programs. Furthermore, networking primitives tend to be control-flow heavy and often involve large copies between system and user memory buffers (e.g., `recv` and `send`), making per-threadblock calls superior to per-thread granularity.

GPUnet requires applications to invoke its API at the granularity of a single threadblock. All threads in a threadblock must invoke the same GPUnet call together in a coalesced manner: with the same arguments and at the same point in application code (similar to vectorized I/O calls [Vasudevan et al. 2012]). These collaborative calls together comprise one logical GPUnet operation. This idea was inspired by a similar design for the GPU file system API [Silberstein et al. 2013].

We illustrate coalesced calls in Figure 5. It shows a simple GPU server that increments each received character by one and sends the results back. All GPU threads invoke the same code, but each threadblock executes it independently from others. The threads in a threadblock collaboratively invoke the GPUnet functions to receive/send the data to/from a shared buffer but perform computations independently in a data-parallel manner. The GPUnet functions are logically executed in lockstep.

Currently, GPUnet sockets cannot be migrated to processes running on other GPUs or CPUs in the same host. Socket migration might be desirable, however, for cases where program logic or data is distributed across multiple processors, such as in a processing pipeline or in case of data sharding between GPUs and CPUs. We leave the implementation of interprocessor socket migration for future work.

5.2. GPU-NIC Interaction

Building a high-performance GPU network stack requires offloading nontrivial packet processing to NIC hardware.

The majority of existing GPU networking projects (with the notable exception of the GASPP packet processing framework [Vasiliadis et al. 2014]) employ the CPU OS network stack with network buffers in CPU memory and the application explicitly moving data to and from the GPU. Accelerated network applications, such as SSL protocol offloading [Jang et al. 2011], cannot operate on raw packets and first require transport-level processing by a CPU. However, the CPU-GPU memory transfers inherent in the CPU-side processing approach are detrimental to application performance, as we show in our evaluation.

P2P DMA allows network buffers to reside in GPU memory. However, forwarding all network traffic to a GPU would render the NIC unusable for processes running on a CPU and on other GPUs in the system. Further, having a GPU receive and process raw network packets makes it difficult to provide a reliable in-order socket abstraction without porting major parts of the CPU network stack to the GPU. Efficiently processing raw packets on a GPU would require thousands of packets to be batched to hide the overheads of the control-heavy and memory-intensive code inherent to packet processing [Vasiliadis et al. 2014].

To bypass CPU memory, eliminate packet processing, and enable NIC sharing across different processors in the system, we leverage RDMA-capable, high-performance NICs. The NIC performs all low-level packet management tasks, assembles application-level messages, and stores the messages directly in application memory, ready to be delivered to an application memory buffer without additional processing. The NIC can concurrently dispatch messages to multiple buffers and multiple applications while placing source and destination buffers in both CPU and GPU memory. As a result, multiple CPU and GPU applications share the NIC without coordinating their access to the hardware for every data transfer.

GPUnet uses both a CPU and a GPU to interact with the NIC. It stores network buffers for GPU applications in GPU memory and leaves the buffer memory management to the GPU socket layer. The per-connection receive and send queues are also managed by the GPU. On the other hand, the CPU controls the NIC via a standard host driver, keeping the NIC available to all system processors. In particular, GPUnet uses the standard CPU interface to initialize the GPU network buffers and to register GPU memory with the NIC's DMA hardware.

5.3. Socket Layer

The GPU socket layer implements the reliable in-order stream abstraction over low-level network buffers and reliable RDMA message delivery. We adopt the RDMA term *channel* to refer to the RDMA connection. The CPU processes all channel creation related requests (e.g., `bind`), allowing GPU network applications to share the OS network namespace with CPU applications. Once the channel has been established, however, the CPU steps out of the way, allowing the GPU socket to manage the network buffers as it sees fit. Using the CPU for channel creation allows GPU network applications to share the OS network namespace with CPU applications.

Mapping streams to channels. GPUnet maps streams one-to-one onto RDMA channels. A channel is a low-level message-oriented RDMA connection that provides reliable in-order message delivery into application memory buffers. To implement a streaming abstraction on top of it, GPUnet must provide flow control described in Section 6.1.[4] The socket keeps track of the amount of available space in the receiver's receive buffer. If there is no free space left, the send call blocks. The receiver side notifies when the received data is consumed by an application, allowing the blocked send to proceed.

[4]Whereas the InfiniBand transport layer has its own flow control for each message, our mechanism enables end-to-end flow control for each datastream.

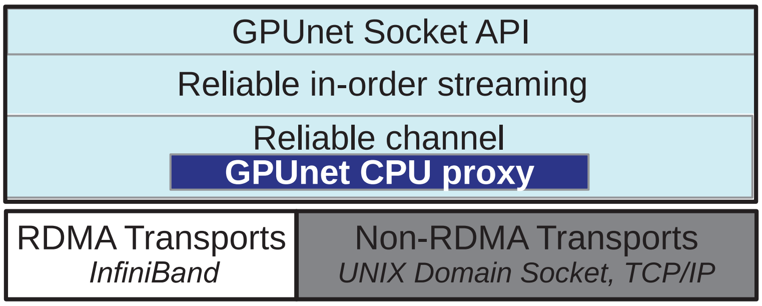

Fig. 6. GPUnet network stack.

By associating each socket with a channel and its private, fixed-sized send and receive buffers, there is no sharing between streams and hence no costly synchronization. Per-stream channels allow GPUnet to offload message dispatch to the highly scalable NIC hardware. The NIC is capable of maintaining a large number of channels associated with one or more memory buffers.[5]

We considered multiplexing several streams over a single channel, similar to SST [Ford 2007], which could improve network buffer utilization and increase PCIe throughput due to the increased granularity of memory transfers. We dismissed this design because handling multiple streams over the same channel would require synchronization of concurrent accesses to the same network buffer, which is slow and complicates the implementation. The buffer sharing may also require an extra memory copy of stream contents into a private per-stream temporary buffer—for example, if the data is not consumed by all the streams at once, thereby potentially reducing system performance. Finally, this design is not compatible with existing CPU socket libraries like rsockets for performing streaming over RDMA.

Naming and address resolution. GPUnet relies on the CPU standard name resolution mechanisms for RDMA transports (CMA) that provide IP-based addressing for RDMA services to initiate the connection. Therefore, unmodified clients may identify GPUnet hosts using domain names and IP addresses as usual.

Wire protocol and congestion control. GPUnet uses reliable RDMA transport services provided by the NIC hardware and thus relies on the underlying transport packet management and congestion control.

6. IMPLEMENTATION

We implement GPUnet for NVIDIA GPUs and use Mellanox InfiniBand HCAs for inter-GPU networking [Silberstein et al. 2014b].

The stream sockets of GPUnet follow a layered design as shown in Figure 6. The lowest layer exposes a reliable channel abstraction to upper layers, and its implementation depends on the underlying transport. We currently support RDMA, UNIX domain sockets, and TCP/IP. There are components that run on both the CPU and GPU. The middle socket layer implements a reliable, in-order, connection-based stream abstraction on top of each channel. It manages flow control for the network buffers associated with each connection. Finally, the top layer implements the blocking and nonblocking versions of standard socket API for the GPU.

[5]Millions for Mellanox Connect-IB, according to the Mellanox Solution Brief, which can be found at http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf.

6.1. Socket Layer

GPUnet's socket interface is compatible with and builds on the open-source rsockets [Hefty 2012] library for a socket-compatible interface over RDMA for CPUs. Rsockets is a drop-in replacement for sockets (via LD_PRELOAD) that provides a simple way to use RDMA over InfiniBand. GPUnet extends the library to use network buffers in GPU memory and integrates GPU-based flow control mechanisms.

GPUnet maintains a private socket table in GPU memory. Each active socket is associated with a single reliable transport channel and holds the flow control metadata for its receive and send buffers. The primary task of the socket layer is to implement the reliable stream abstraction, which requires flow control management as we describe next.

Flow control. Flow control allows the sender to block if the receiver's network buffer is full. Therefore, an implementation requires the receiver to update the sender on buffer consumption. Performing this notification efficiently requires careful coordination of the GPU, CPU, and HCA.

Certain limitations of the existing software and hardware force us to deviate from the original design in which the HCA is controlled entirely from the GPU and instead fall back to using the CPU to assist with send/receive operations. Specifically, the HCA driver does not support placing its work and completion queues in GPU memory. These queues are frequently accessed by the application to enqueue each send/receive request and retrieve completion notifications, and currently can be stored only in CPU memory. GPUs may map CPU memory into their address space, but the latency of accessing that memory from the GPU is too high, and when accessed frequently as in the case of the HCA queues, it might significantly degrade the network throughput and increase latency. Another problem is that the NVIDIA driver does not allow accessing the HCA's "door-bell" hardware registers to trigger a send operation. The door-bell registers are accessed via memory mapped I/O, and GPUs cannot map that memory. These limitations are expected to be resolved in future generations of software and hardware.[6] However, currently, a CPU is necessary to assist every GPU send and receive operation.

Using a CPU to handle completion notifications introduces an interesting challenge for the flow control implementation. The flow control counters must be shared between a CPU and a GPU, as they are updated by a CPU as a part of the completion notification handler, and by a GPU for every `gsend`/`grecv` call. To guarantee consistent concurrent updates, these writes have to be performed atomically, but the updates are performed via a PCIe bus that does not support atomic operations. The solution is to treat the updates as two independent instances of producer-consumer coordination: between a GPU and an HCA (which produces the received data in the GPU network buffer), and between a GPU and a remote host (which consumes the sent data from the GPU network buffer). In both cases, a CPU serves as a mediator for updating the counters in GPU-accessible memory on behalf of the HCA or remote host. Assuming only one consumer and producer, each instance of a producer-consumer coordination can be implemented using a ring buffer data structure shared between a CPU and a GPU.

Figure 7 shows the ring buffer processing a receive call. The GPU receives the data into the local buffer via direct RDMA memory copy from the remote host (1). The CPU gets notified by the HCA that the data was received (2) and updates the ring buffer as a producer on behalf of the remote host (3). Later, the GPU calls `grecv()` (4), recognizes the received data through the ring buffer, reads the data, and updates the

[6]Specifically, memory mapped I/O from GPU kernels is expected to be possible with the future release of the GPUdirect async technology [Rossetti et al. 2016].

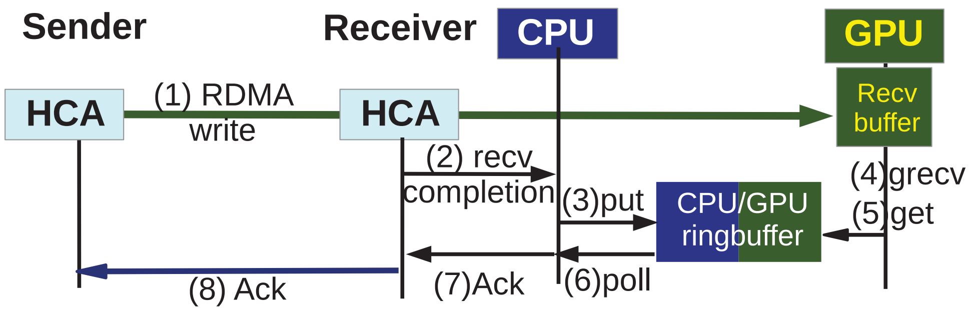

Fig. 7. Ring buffer updates for GPU flow control mechanism in `grecv()` call.

ring buffer that the data has been consumed (5). This update triggers the CPU (6) to send a notification (7) to the remote host (8).

This design decouples the GPU API calls and the CPU I/O transfer operations, allowing the CPU to handle GPU I/O request asynchronously. As a result, the GPU I/O call returns faster, without waiting for the GPU I/O request to propagate through the high-latency PCIe bus, and data transfers and GPU computations are overlapped. This feature is essential to achieve high performance for bulk transfers.

6.2. Channel Layer

The channel layer mediates the GPU's access to the underlying network transport and runs on both the CPU and GPU. On the GPU side, the channel layer manages the network buffers in GPU memory, whereas the CPU-side logic ensures that the buffers are delivered to and from the transport mechanism.

Memory management. GPUnet allocates a large contiguous region of GPU memory, which it uses for network buffers. To enable RDMA hardware transport, CPU code registers GPU memory to the InfiniBand HCA with the help of CUDA's GPUDirectRDMA mechanism. The maximum total amount of HCA-registered memory is limited to 234MB in NVIDIA TESLA K20c GPUs due to the base address register (BAR) size constraints of the current hardware. We allocate the memory statically during GPUnet initialization because the memory registration is expensive (details are provided in Section 6.4) and because the GPU memory allocation may be blocked indefinitely when the GPU kernel is running. GPUnet uses registered memory as a pool for allocating receive and send buffers for each channel.

Bounce buffers and support for non-RDMA transports. If P2P DMA functionality is not available, the underlying transport mechanism has no direct access to GPU network buffers. Therefore, network data must be explicitly staged to and from bounce buffers in CPU memory.

Using bounce buffers incurs higher latency because of the extra memory copy to and from CPU memory. Bounce buffers also increase PCIe occupancy because the HCA and CPU are connected via PCIe. To better hide PCIe latency, GPUnet socket send/receive buffers are configured to be larger than the buffers used with direct GPU-NIC memory transfers.

Bounce buffers are useful to enable GPUnet on hardware that lacks RDMA or on chipsets that experience low bandwidth of P2P DMA. For example, we encounter 15× bandwidth degradation when storing send buffers in GPU memory; using bounce buffers helped to achieve close-to-maximum bandwidth in this case. Similarly, P2P DMA is only possible for certain PCIe topologies; thus, for our dual-socket configuration, only one of the three PCIe attached GPUs can perform P2P DMA with the

InfiniBand HCA. Until the software and hardware support stabilizes, bounce buffers are an interim solution that hides the implementation complexity of CPU-GPU-NIC coordination mechanisms.

6.3. GPUnet CPU Proxy

GPUnet encapsulates all CPU-related functionality in a GPUnet proxy module. The module is invoked in a new user-level CPU thread when the application using GPUnet is initialized. The thread runs in the application's address space, and therefore it can easily access the state of the GPU application. The proxy is implemented as an event-driven server that exposes a simple remote procedure call (RPC) interface and enables GPU programs to invoke certain I/O-related operations on a CPU. Along with handling RPCs, the GPUnet proxy polls the send ring buffer and updates the receive ring buffer when a data receive is notified.

6.4. Performance Optimizations

Single threadblock I/O. While developing GPUnet applications, we found it convenient to dedicate some threadblocks to performing network operations and using others only for computation, such as the receiving threadblock in MapReduce (Section 7.2) or a daemon threadblock in the matrix product server (Section 7.1). In such a design, the performance-limiting factor for send operations is the latency of two steps performed in the `gsend()` call: memory copy between the system and user buffers in GPU, and the update of the flow control ring buffer metadata. Therefore, an important optimization goal has been to maximize the I/O throughput of a single threadblock.

Unfortunately, a single threadblock is allocated only a small fraction of the total GPU compute and memory bandwidth resources (e.g., up to 7% of the total GPU memory bandwidth according to our measurements). Improving the memory throughput of a single threadblock requires issuing many memory requests per thread to enable memory-level parallelism [Volkov 2010]. We resorted to PTX, NVIDIA GPU's low-level assembly, to implement 128-bit/thread vector accesses to global memory that bypass the L2 and L1 caches. Cache bypassing is required to ensure a consistent buffer state when RDMA operations access GPU memory. This optimization improves memory copy throughput almost $3\times$, from 2.5GB/s to 6.9GB/s for a threadblock with only 256 threads.

Ring buffer updates. Ring buffer updates were slow initially because the ring buffer head pointers are shared between the CPU and GPU, and we placed them in "zero-copy" memory, which physically resides on a CPU. Therefore, reading this memory from the GPU incurs a significant penalty of about 1 to 2 μsec. Updating the ring buffer requires multiple accesses to the head pointers, and the latency accumulates to tens of μseconds.

We improved the performance of ring buffer updates by converting reads from remote memory into remote writes into local memory. For example, the head location of a ring buffer, which is updated by a producer, should reside in the consumer's memory to enable the consumer to read the head quickly. To implement this optimization, however, we must map GPU memory into the CPU's address space. NVIDIA CUDA does not natively provide an API for mapping GPU memory, and we implement a kernel driver that leverages GPUDirect to `mmap` the GPU BAR memory into the CPU user address space.[7] This optimization reduces the latency of ring buffer updates to 2.5μsec.

Additional overhead is caused by memory fences that are necessary to guarantee consistent ordering of writes into shared memory between the CPU and GPU. For example, we use `__threadfence_system()` on the GPU to ensure that the ring buffer

[7]A similar approach is also used in the recent `gdrcopy` library from NVIDIA.

head pointer updates are ordered strictly after the network buffer updates. We found that each `__threadfence_system()` call adds about 2μsec to each `gsend()` call and affects the data transfer performance significantly. Therefore, we minimize the use of memory fences on GPUs. For example, adding a memory fence after every metadata update of the ring buffer results in simpler implementation but can be eliminated to reduce the latency of critical processing paths.

Memory registration cache. An HCA performs virtual-to-physical address translation to enable zero-copy access to the user memory. An application needs to initialize the translation table, a process called *memory registration*.

GPUnet uses GPU memory for network buffers, and therefore GPU memory buffers need to be registered in the HCA. A naive implementation might register network buffers every time a new connection is established and unregister when the socket is closed. However, we found that GPU memory registration is extremely slow—for example, 5msec for a 256KB GPU buffer (which is about 80× slower than registering the same-sized CPU buffer). To eliminate this overhead, GPUnet allocates and registers all network buffers during initialization, following a common idiom called *registration cache* [Liu et al. 2004]. This technique reduces the latency of connection establishment from 23msec (two 256KB buffers at both endpoints) to 30μsec with 256KB buffers.

6.5. Limitations

Many NVIDIA GPUs only expose limited memory for P2P DMA due to the PCIe BAR memory size [NVIDIA 2015]. The K20c GPUs used for our experiments have a modest memory budget (234MB) for the BAR area. Recent high-end NVIDIA GPUs like NVIDIA K80 and M40 support larger memory sizes, up to 16GB. Limited memory for P2P DMA memory restricts the number of connections that a single GPU can support with a given system buffer size.

A large GPU BAR area would allow all GPU memory to be registered to an HCA, eliminating the need to register smaller buffers and suffer the memory registration overhead. However, our current implementation of GPUnet does not support large BAR areas because none of our GPUs support it.

GPUnet does not provide a mechanism for socket migration between a GPU and a CPU, which might be convenient for load balancing.

Perhaps the most significant limitation of the prototype is that it relies on the ability of a GPU to guarantee consistent reads from its memory when it is concurrently accessed by a running kernel and the NIC RDMA hardware. Specifically, dependent writes to GPU memory performed by the NIC via PCIe must be observed in the same order by the GPU kernel. NVIDIA GPUs provide such consistency guarantees only if the reads are performed from a kernel invoked after the writes have been completed but do not guarantee consistent reads from the kernels executing concurrently with the writes.

In practice, however, we do not observe consistency violations in GPUnet. To validate our current implementation, we implement a 32-bit cyclic redundancy check (CRC) for the GPU and instrument our applications to check the data integrity of all network messages with 4KB granularity. We detect no data integrity violations for experiments reported in the article (although this experiment surfaced a small bug in GPUnet itself).

It is our hope, perhaps encouraged by GPUnet itself, that GPU vendors will provide such consistency guarantees in the near future (e.g., in NVIDIA CUDA async [Rossetti et al. 2016]). Furthermore, the necessary CPU-GPU memory consistency will be a part of the future releases of OpenCL 2.0-compliant GPU platforms, thereby supporting our expectation that it will become the standard guarantee of future systems.

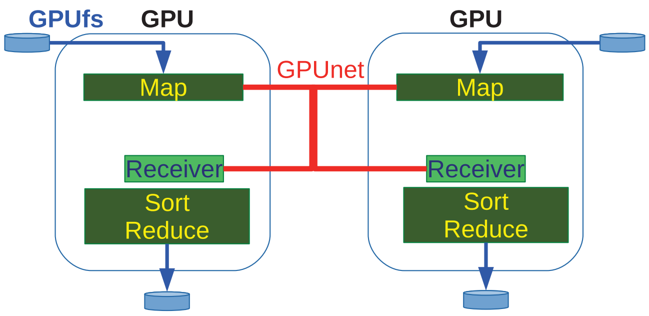

Fig. 8. GimMR design. Mappers push their outputs to the receiver threadblock in the GPU that will run the respective reducer.

7. APPLICATIONS

We now describe the design and implementation of three GPU server applications that use GPUnet for network communications. These are native GPU applications with all of their logic encapsulated in GPU code and without any CPU code.

7.1. Matrix Product Server

The matrix product server is implemented using both the daemon and independent architectures (Section 4.3). In the daemon architecture, the daemon threadblock (one or more) accepts a client connection, reads the input matrices, and enqueues a multiplication kernel. The multiplication kernel gets pointers to the input matrices and the socket for writing the results. The number of threads—a critical parameter defining how many GPU computational resources a kernel should use—is derived from the matrix dimensions as in the standard GPU implementation. When the execution completes, the threadblock that finalizes the computation sends the data back to the client and closes the connection.

In the independent architecture, each threadblock receives the input, runs the computations, and sends the results back.

Implementation details. The daemon server cannot invoke the multiplication kernel using dynamic parallelism (which is the ability to execute a GPU kernel from within an executing kernel, present since NVIDIA Kepler GPUs). Current dynamic parallelism support in NVIDIA GPUs lacks a parent-child concurrency guarantee, and in practice the parent threadblock blocks to ensure the child starts its execution. Our daemon threadblock must remain active to accept new connections and handle incoming data, so we do not use NVIDIA's dynamic parallelism and instead invoke new GPU kernels via the CPU using a custom mechanism. Section 8.2 provides performance measurements.

Parallel-I/O design. We also tried an alternative design where the daemon threadblock invokes the multiply kernel, which then is responsible for receiving the matrix data through `grecv()`. While providing better performance for a single client, it resulted in all compute threadblocks being idle waiting for the I/O completion, wasting GPU computing resources and reducing multiclient throughput substantially.

7.2. MapReduce Design

We design an in-GPU-memory distributed MapReduce framework that keeps intermediate results of map operations in GPU memory while input and output are read from disk using GPUfs [Silberstein et al. 2013]. We call the system *GimMR* (*G*PU *in* *m*emory *M*ap*R*educe). Its design is presented in Figure 8. The number of GPUs in our system is small, so all of them are used to execute both mappers and reducers. Shuffling (i.e., the exchange of intermediate data produced by mappers between different hosts) is done by mappers, and reducers only start once all mappers and data transfer have completed. Our mappers push data, whereas in traditional MapReduce, the reducers pull [Dean and Ghemawat 2004]. Each GPU runs multiple mappers and reducers, each of which is executed by multiple GPU threads.

At the start of the Map phase, a mapper reads its part of the input via GPUfs. The input is split across all threadblocks, so they can execute in parallel. A GPU may run tens of mappers, each invoked in one threadblock and executed by hundreds of its threads. Mappers generate intermediate <key,value> pairs that they assign to buckets using consistent hashing or a predefined key range. Buckets contain pointers to data chunks. A mapper accumulates intermediate keys and data into local chunks. When a chunk size exceeds a threshold, the mapper sends the chunk to the GPU, which will run the reducer for the keys in that bucket, thereby overlapping mapper execution with the shuffle phase, similar to ThemisMR [Rasmussen et al. 2012].

In addition to many mapper threadblocks, each GPU runs one or more receiver threadblocks, which receive buckets from remote GPUs. Each receiver threadblock is assigned a fixed number of connections from a remote GPU. The receivers receive data by making nonblocking calls to `grecv()` on the mappers' sockets in round-robin order (implementing `poll()` on the GPU is left as future work). A mapper threadblock connects to its receiver threadblock when it starts executing. For example, consider a GimMR system with total of five GPUs, each running 12 mappers and 12 receiver threadblocks. Then each GPU will have a total of 48 incoming connections, one per mapper from every other GPU. Each of its 12 receiver threadblocks will handle four incoming connections. Local mappers update local buckets without sending them through the network.

GPU mappers are coordinated by a CPU-side centralized mapper master, accessed over the network. The master assigns jobs, balancing load across the mappers. The master tells each mapper the offset and size of the data to read from its input file.

Similar to Map, each Reduce function is also invoked in a single threadblock. Each reducer identifies the set of buckets it must process, (optionally) performs parallel sort of all key-value pairs in each bucket separately, and finally invokes the user-provided Reduce function. As a result, the GPU exploits the standard coarse-grain data parallelism of independent input keys but also enables the finer-grain parallelism of a function processing different values from the same key (e.g., by parallel sorting or reduction). Enabling each reducer to sort the key/values independently of other reducers is important to avoid a GPU-wide synchronization phase at the end of sorting.

GimMR takes advantage of the dynamic communication capabilities of GPUnet for ease and efficiency of implementation. Without GPUnet, enabling overlapped communications and computations would require significant development effort involving fine-tuned pipelining among CPU sends, CPU-GPU data transfers, and GPU kernel invocations.

7.2.1. GimMR Workloads. We implement word count and K-means. In word count, the mapper parses free-form input text and generates <word, 1> pairs, which are reduced by summing up their values. CUDA does not provide text processing functions, so we implement our own parser. We presample the input text and determine the range of keys being reduced by each reducer.

The mappers in K-means calculate the distance of each point to the cluster centroids and then recluster the point to its nearest centroid. Intermediate data is pairs of <centroid number, point>. The reducer sums the coordinates of all points in a centroid. K-means is an iterative algorithm, and our framework supports iterative MapReduce. A CPU process receives the new centroids produced by the reducers and sends them to all of the GPUs for the next round. We preprocess the input file to piecewise transpose the input points, thereby enabling coalesced memory accesses for mapper threads.

7.3. Face Verification

A client sends a photo of a face, along with a text label identifying the face, to a verification service. The server responds positively if the label matches the photo (i.e., the server has the same face in its database with the proffered label) and negatively otherwise. The server uses a well-known local binary patterns (LBP) algorithm for face verification [Ahonen et al. 2006]. LBP represents images by a histogram of their visual features. The server stores all LBP histograms in a `memcached` database. In our testbed, we have three machines: one for clients, one for the verification server, and one for the `memcached` database.

We believe that our organization is a reasonable choice, as opposed to alternatives such as having the client perform the LBP and send a histogram to the server. Face verification algorithms are constantly evolving, and placing them on the server makes upgrading the algorithm easy for the service provider. In addition, sending actual pictures to the server provides a useful human-checkable log of activity.

Client. The client uses multiple threads, each running on its own CPU and maintaining multiple persistent nonblocking connections with the server. Clients use rsockets for network communications with the server. For each connection, the client performs the following steps and repeats them forever:

(1) Read a (random) 136×136 grayscale image from a (cached) file.
(2) Choose a (random) face label.
(3) Send verification request to server.
(4) Receive response from server: 0 (mismatch) or 1 (match).

Server. We implement three versions of the server: a CPU version, a CUDA version, and a GPUnet version. Each server performs the following steps repeatedly (in different ways):

(1) Receive request from client.
(2) Fetch LBP histogram for client-provided name from the remote memcached database.
(3) Calculate LBP histogram of the image in the request.
(4) Calculate Euclidean distance between the histograms.
(5) Report a match if the distance is below a threshold.
(6) Send integer response.

The CPU server consists of multiple independent threads, one per CPU core. Each thread manages multiple, persistent, nonblocking connections with the client.

The CUDA server is the same as the CPU server, but the face verification algorithm executes on the GPU by launching a kernel (see the middle picture in Figure 2).

The GPUnet server is a native GPU-only application using GPUnet for network operations. It uses the independent architecture (Section 4.3) and consists of multiple threadblocks running forever, with each acting as an independent server. Each threadblock manages persistent connections with the client and `memcached` server. This design is appropriate, as the processing time per image is low and there is enough parallelism per request.

Implementation details. We use a standard benchmarking face recognition dataset,[8] resized to 136×136 and reformatted as raw grayscale images. We implement a GPU

[8]http://www.itl.nist.gov/iad/humanid/feret/feret_master.htm.

Table I. Hardware and Software Configuration

Node	Chipset Intel	CPU Intel	GPU NVIDIA	DMA	Software
A B	Z87	E3-1220V3 Haswell	K20c	N	RHEL 6.5, gcc 4.4.7, GPU driver 331.38
C	C602	E5-2620 Sandy Bridge	C2075	Y	RHEL 6.3, gcc 4.4.6, GPU driver 319.37
D	5520	2× L5630 Westmere	2× C2075	Y	RHEL 6.3, gcc 4.4.6, GPU driver 319.37

Note: The DMA column indicates the presence of a DMA performance asymmetry (Section 6.2).

`memcached` client library; `memcached` uses the InfiniBand RDMA transport provided by the rsockets library. We modified a single line of `memcached` to work with rsockets by disabling the use of `accept4`, which is not supported by rsockets.

8. EVALUATION

Hardware. We run our experiments on a cluster with four nodes (Table I) connected by a QDR 40Gbps InfiniBand interconnect, using Mellanox HCA cards with MT4099 and MT26428 chipsets.

All machines use CUDA 5.5. We disable ECC on GPUs, hyperthreading, SpeedStep, and Turbo mode on all machines for reproducible performance. Nodes A and B feature a newer chipset with a PLX 8747 PCIe switch that enables full-bandwidth P2P DMA between the HCA and the GPU. Nodes C and D provide full bandwidth for DMA writes from the HCA to GPU (`grecv()`) but perform poorly with only 10% of the bandwidth for DMA reads from GPU (`gsend()`) due to chipset limitation. We are not the first to observe such asymmetry [Potluri et al. 2013b].

GPUnet delegates connection establishment and teardown to a CPU. Our benchmarks exclude connection establishment from the performance measurement to measure the steady-state behavior of persistent connections. Using persistent connections is a common optimization technique for data center applications [Benson et al. 2010].

8.1. Microbenchmarks

We run microbenchmarks with two complementary goals: to understand the performance consequences of GPUnet design decisions and to separate the essential bottlenecks from the ephemeral issues due to current hardware. We run them between nodes A and B with 256 threads per threadblock. All results are the average of 10 iterations, with the standard deviation within 1.1% of the mean.

Single-stream performance. Single-stream performance is important to applications with a dedicated networking threadblock like GimMR, as discussed in Section 7.2.

We implement the CPU version of the benchmark using the unmodified rsockets library. Figure 9 shows the round-trip time (RTT) for messages with different sizes. The latency of GPU transfers using RDMA is significantly higher than the baseline CPU-to-CPU latency. Yet bounce buffers (marked BB in the legend) significantly worsen the latency almost twofold.

Figure 10 shows the single-stream bandwidth with different socket buffer sizes. The maximum performance is reached when full I/O pipelining is achieved with sufficiently large buffers. The GPU reaches about 98% of the peak performance of CPU-based rsockets. Since the bounce buffer case results in almost twice the latency of RDMA, the socket buffers need to be twice as large to reach the peak throughput.

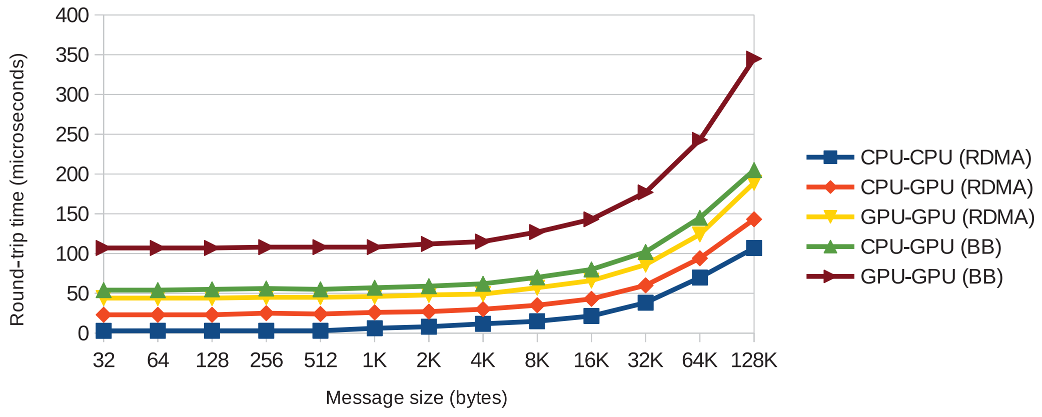

Fig. 9. Single-stream RTT with different message sizes. The CPU uses rsockets.

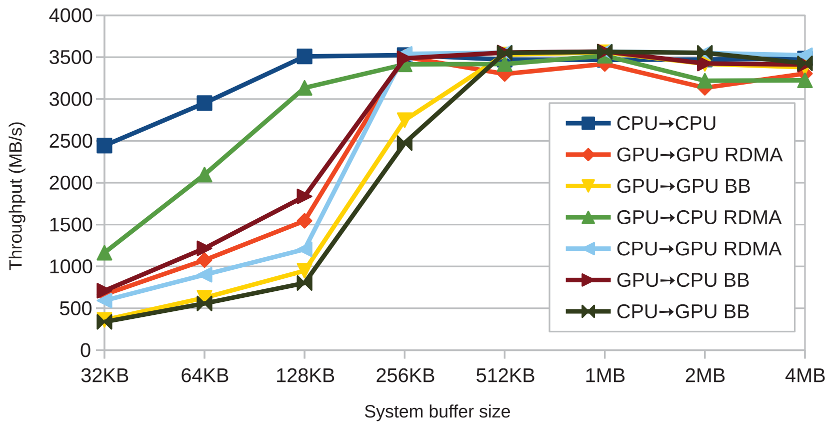

Fig. 10. Single threadblock throughput with different socket buffer sizes.

The graph shows an unexpected discrepancy between the send and receive throughput for CPU-GPU transfers with buffers smaller than 256KB. For example, with 64KB buffers, the throughput from the GPU to the CPU is twice as high as the throughput in the opposite direction. The penalty is due to higher management overheads on the GPU. Specifically, for the case with a GPU sender and a CPU receiver, the GPU sender produces data slower than the CPU receiver consumes it, so no flow control logic is actively involved. However, in the opposite direction, a GPU receiver does not keep up with a CPU sender. Therefore, the flow control prevents further sends, which in turn lowers the throughput substantially. The throughput asymmetry disappears with larger buffers.

We present the latency of individual steps in `gsend()` call sending 64KB (Table II). We measure T_1, T_2, T_3 on the GPU by instrumenting the GPU code using `clock64()`, the GPU intrinsic that reads the hardware cycle counter. T_5 is effectively the latency of the `send()` call performed from the CPU but transferring data between memories of two GPUs. For this data size, the overhead of GPU-related processing is about 50%. The user-to-system buffer copy, T_2, is the primary bottleneck. Accessing CPU-GPU shared data structures (T_1, T_3) and the latency of the update propagation through the PCIe bus (T_4) account for 20% of the total latency, but these are constant factors.

Table II. Latency Breakdown for a GPU `gsend()` Request with a 64KB Message with P2P RDMA

Steps	Latency (μsec)
T_1 GPU ring buffer	1.4
T_2 GPU copies buffer	15.7
T_3 GPU requests to CPU	3.8
T_4 CPU reads GPU request	2.5
T_5 CPU RDMA write time to completion	22.2
Total one-way latency	45.6

We believe that T_2 and T_4 will improve in future hardware generations. Specifically, T_4 can be reduced by enabling a GPU to access the HCA doorbell registers directly, without CPU mediation. We believe that T_2 can be optimized by exposing the already existing GPU DMA engine for performing internal GPU DMAs, similar to the Intel I/OAT DMA engine. Alternatively, a zero-copy API may help to eliminate T_2 in software.

Multistream bandwidth. We measure the aggregate bandwidth of sending over multiple sockets from one GPU. We run 26 threadblocks (2 threadblocks per GPU SM core), each having multiple nonblocking sockets. Each send is 32KB. We test up to 416 active connections—the maximum number of sockets that GPUnet may concurrently maintain given 256KB send buffers, which provide the highest single-stream performance. As we explained in Section 6, the maximum number of sockets is constrained by the total amount of RDMA-registered memory available for network buffers, which in our hardware is limited to 220MB.

We run the experiment between two GPUs. Starting from 2 connections, GPUnet achieves throughput of 3.4GB/s and gradually falls to 3.2GB/s at 416 connections, primarily due to the increased load on the CPU-side proxy having to handle more requests. Using bounce buffers shows slightly better throughput: 3.5GB/s with two connections and 3.3GB/s with 208 connections.

8.2. Matrix Product Server

We implement three versions of the matrix product server to examine the performance of different GPU server organizations.

The *CUDA* server runs the I/O logic on the CPU and offloads matrix product computations to the GPU using standard CUDA. It executes a single CPU thread and invokes one GPU kernel per request. We use `matrixMul`, the matrix product kernel distributed with the NVIDIA SDK.

The *daemon* server uses GPUnet and follows the daemon architecture (Section 4.3), listening on a network socket and receiving the input, and launches a GPU kernel with threadblocks computing the matrix product. GPU resources are partitioned between daemon threadblocks and computing threadblocks. The number of daemon threadblocks is an important server configuration parameter as we discuss in the following. Both the CUDA server and the daemon server invoke the matrix product kernel via the CPU; however, the latter receives/sends data directly to/from GPU memory.

The *independent* server also employs GPUnet, but the GPU is not statically partitioned between daemon and compute threadblocks. Instead, all threadblocks handle I/O and perform computations, and no additional GPU kernels are launched.

The CUDA, daemon, and independent server versions are 894, 391, and 220 LOC for their core functionality.

Resource allocation in the daemon server. The performance of the daemon server is particularly sensitive to the way GPU resources are partitioned between I/O and compute tasks performed by the server. The GPU nonpreemptive scheduling model implies

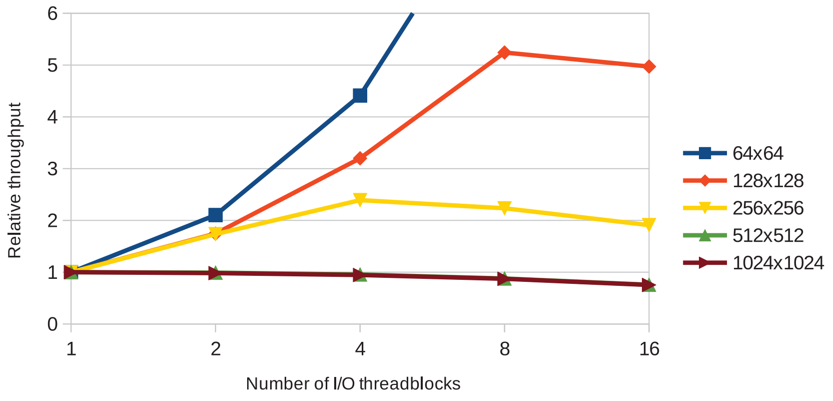

Fig. 11. Relative throughput of the matrix multiplication server with different numbers of I/O threadblocks normalized by the server throughput using one I/O threadblock.

Table III. Optimal Configurations for Matrix Multiplication Workloads

Workload	Optimal Daemon Threadblocks	Threads per Daemon Threadblock
64×64	16	64–256 threads
128×128	8	256–512 threads
256×256	4	128–512 threads
512×512	1	128–1024 threads
1024×1024	1	Not sensitive

that GPU resources allocated to I/O tasks cannot execute computations even while I/O tasks are idle waiting for the input data. Therefore, if the server is configured to run too many daemon threadblocks, the compute kernels will get fewer GPU resources and computations will execute slowly. On the other hand, too few daemon threadblocks may fail to feed the execution units with data fast enough, thereby decreasing the server throughput. In our current implementation, the number of daemon threadblocks is configured at server invocation time and does not change during execution.

The optimal number of I/O threadblocks depends on the compute-to-I/O ratio of the workload. Figure 11 shows how different numbers of I/O threadblocks affect the throughput of the server. The experiment runs with 64 active clients and uses 256 threads in a threadblock. Each value is relative to the single threadblock case. Workloads with smaller matrices have a lower compute-to-I/O ratio, and our measurement shows that `grecv()` dominates the processing time for each matrix computation task. For such I/O-bound jobs, increasing the number of I/O threadblocks significantly enhances the performance, by a factor of 14 with 64×64 matrices and 16 I/O threadblocks, compared to the computation with a single I/O threadblock. However, workloads with larger matrices are compute bound, and increasing the number of I/O threadblocks negatively affects server performance. For medium-size matrices and many I/O threadblocks (e.g., 16 threadblocks for 128 × 128 matrices), the asynchronous kernel invocation, instead of computation, becomes the dominant overhead that reduces the throughput.

To find the best configuration that maximizes the server throughput, we search across several combinations of the number of I/O threadblocks and the number of threads in each I/O threadblock and then record the throughput of each configuration

Table IV. Cost of Misconfiguration: Throughput in a Given Configuration Relative to the Maximum Throughput Using the Best Configuration for That Workload

	Workload				
Configuration	64×64	128×128	256×256	512×512	1024×1024
Optimal for 64×64	100%	95%	80%	76%	76%
Optimal for 128×128	68%	95%	91%	87%	85%
Optimal for 256×256	36%	60%	100%	96%	94%
Optimal for 512×512	8.5%	19%	43%	100%	95%
Optimal for 1024×1024	8.6%	19%	40%	94%	100%

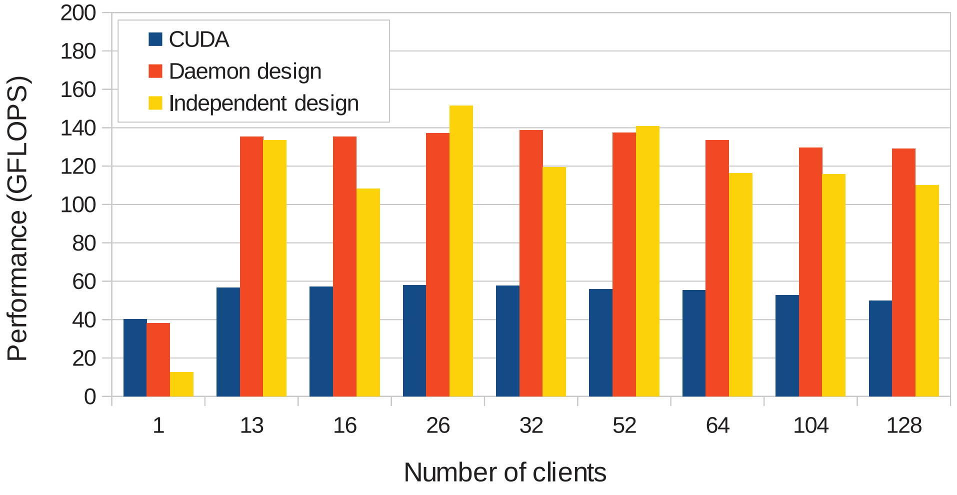

Fig. 12. Throughput comparison for different matrix product servers.

on each workload. Table III shows the optimal configurations found by the search.[9] Table IV shows the relative throughput of each workload under different configurations, normalized by the throughput of the best configuration. We observe significant, up to 10-fold, degradation in server performance when compared to its performance under the best configuration. Finding the best server configuration and dynamically adjusting it to suit the workload is left for future work.

Performance comparison of server designs. We compare the throughput of different server designs while changing the number of concurrent clients. We use the 256×256 matrices for input and configure the daemon server to have the number of daemon threadblocks that maximizes its throughput for this workload. The results are shown in Figure 12.

With multiple clients, both GPUnet-based implementations consistently outperform the traditional CUDA server across all workloads and are competitive with each other.

As expected, the performance of the independent design is sensitive to the number of clients. Our implementation assigns one connection per threadblock, so the number of clients equals the number of server threadblocks. Configurations where the number of clients is divisible by the number of GPU SMs (13 in our case) have the best performance. Other cases suffer from load imbalance. The performance of the independent design is particularly low for one client because the server runs with a single threadblock using a single SM, leading to severe underutilization of GPU resources.

[9]A result within 1% of the peak performance is considered optimal.

Table V. Throughput of GPUnet-Based Matrix Product Servers Under Different Workload Types

Server design	Light workload	Medium workload	Heavy workload
Daemon (GFLOPS)	11	137	201
Independent (GFLOPS)	37 (3.4×)	151 (1.1×)	207 (1.01×)

Table VI. Single-Node GimMR Versus Other MapReduce Systems

Workload	8-Core Phoenix++	1-Node Hadoop	1-GPU GimMR
K-means	12.2s	71.0s	5.6s
Word count	6.23s	211.0s	29.6s

The performance of the independent design is 8× to 20× higher than a single-threaded CPU-only server that uses the highly optimized BLAS library (not shown in the figure).

Table V shows the throughput of the GPUnet servers serving different workload types. We fixed the number of active connections to 26 to allow the independent server to reach its full performance potential.

The independent server achieves higher throughput for all workload types, but its advantages are most profound for light tasks (with low compute-to-I/O ratios). The independent server does not incur the overhead of GPU kernel invocations, which dominate the execution time for shorter tasks in the daemon server. This performance advantage makes the independent design particularly suitable for our face verification server, which also runs tasks with low compute-to-I/O ratio as we describe in Section 8.4.

8.3. MapReduce

We evaluate the standard word count and K-means tasks on our GimMR MapReduce. Table VI compares the performance of the single-GPU GimMR with the single-node Hadoop and Phoenix++ [Talbot et al. 2011] on an eight-core CPU. We use a RAM disk when evaluating K-means on Hadoop. For both word count and K-means on Hadoop, we run 8 map jobs and 16 reduce jobs per node.

Word count. The word count serves as a feasibility proof for distributed GPU-only MapReduce, but the workload characteristics make it inefficient on GPUs.

The benchmark counts words in a 600MB corpus of English language Wikipedia in XML format. A single GPU GimMR outperforms the single-node eight-core Hadoop by a factor of 7.1×, but it is 4.7× slower than Phoenix++ [Talbot et al. 2011] running on eight CPU cores. GimMR word count spends a lot of time sorting strings, which is expensive on GPUs because comparing variable length strings creates divergent, irregular computations. In the future, we will adopt the optimization done by ThemisMR [Rasmussen et al. 2012], which uses the hash of the strings as the intermediate keys to sort quickly.

Scalability. When invoked on the same input on four network-connected GPUs, GimMR performance increases by 2.9×. The scalability is affected by three factors: (1) the amount of computation is too low to fully hide the intermediate data transfer overheads, (2) reducers experience imbalance due to the input data skew, and (3) only two machines enable GPU-NIC RDMA (the other two use bounce buffers).

K-means. We chose K-means to evaluate GimMR under a computationally intensive workload. We compute 500 clusters on a randomly generated 500MB input with 64K vectors, each with hundreds of floating-point elements.

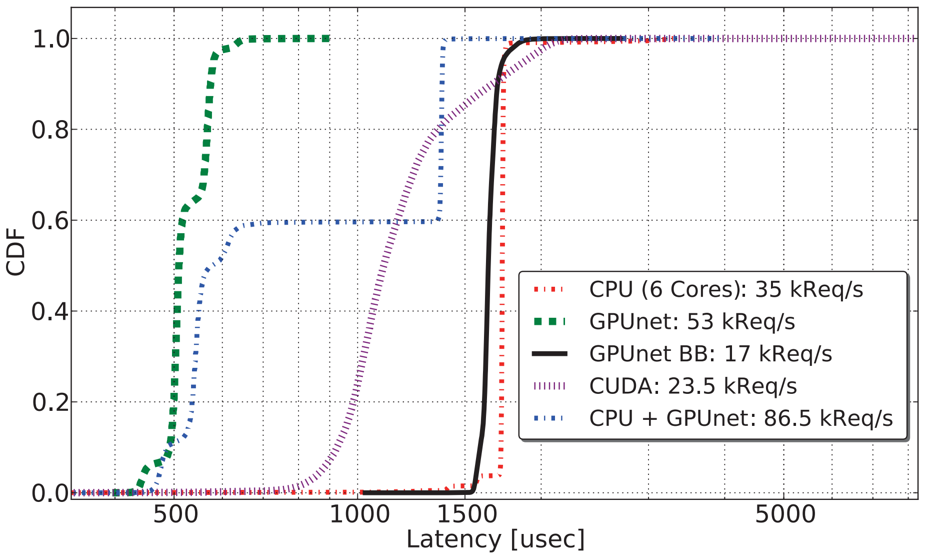

Fig. 13. Face verification latency CDF for different servers.

Table VI compares the performance of GimMR with single-node Hadoop and Phoenix++ using 200 dimension vectors. GimMR on a single GPU outperforms Phoenix++ on eight CPU cores by up to 2.2× and Hadoop by 12.7×.

Scalability. When invoked on the same input on four network-connected GPUs, GimMR performance increases by 2.9×. With 100 dimension vectors, the 4-GPU GimMR achieves up to 3.5× speedup over a single GPU.

8.4. Face Verification

We evaluate the face verification server on a different cluster with three nodes, each with a Mellanox Connect-IB HCA, 2× Intel E5-2620 six-core CPU, and connected via a Mellanox Switch-X bridge. The server executes on NVIDIA K20Xm GPUs. The application's client, server, and `memcached` server run on their own dedicated machines. We verified that both the CPU and GPU algorithm implementations produce the same results and also manually inspected the output using the standard FERET dataset and hand-modified images. All reported results have variance below 0.1% of their mean.

Lower latency, higher throughput. Figure 13 shows the CDF of the request latency for different server implementations and some of their combinations. The legend for each server specifies the effective server throughput observed during the latency measurements. GPUnet and CUDA are invoked with 28 threadblocks, 1024 threads per threadblock, which we found to provide the best trade-off between latency and throughput. Other configurations result in higher throughput but sacrifice latency or slightly lower latency but much lower throughput.

The GPUnet server has the lowest average response time of 524±41 μsec per request while handling 53 KRequests/sec, which is about 3× faster per request, and 50% more requests than the CPU server running on a single six-core CPU. The native CUDA version and GPUnet with bounce buffers suffer from 2× and 3× higher response time, and 2.3× and 3× lower throughput, respectively. They both perform extra memory copies, and the CUDA server is further penalized for invoking a kernel per request. Dynamic kernel invocation accounts for the greater variability in the response time of

Table VII. Face Verification Throughput for Different Servers

Server Type	CPU	2× CPU	CUDA	GPUnet BB	GPUnet	2×GPUnet	2×GPUnet + CPU
Thpt (Req/s)	35K	69K	23K	17K	67K	136K	188K

the CUDA server. The combination of CPU and GPUnet achieves the highest throughput and improves the server response time for all requests, not only for those served on a GPU.

Maximum throughput and multi-GPU scalability. The throughput-optimized configuration for the GPUnet server differs from its latency-optimized version, with 4× more threadblocks, each with 4× fewer threads (112 threadblocks, each with 256 threads). Although the total number of threads remains the same, this configuration serves 4× more concurrent requests. With 4× fewer threads processing each request, the processing time grows only by about 3×. Therefore, this configuration achieves about 30% higher throughput, as shown in Table VII, which is within 3% of the performance of two 2×6-core CPUs.

Adding another GPU to the system almost doubles the server throughput. Achieving linear scalability, however, requires adding a second InfiniBand card. The PCIe topology on the server allows only one of the two GPUs to use P2P DMA with the same HCA, and the second GPU has to fall back to using bounce buffers, which has inferior performance in this case. To work around the problem, we added a second HCA to enable P2P DMA for the second GPU.

Finally, invoking both the CPU and GPUnet servers together results in the highest throughput. Because each GPU in GPUnet requires one CPU core to run, the CPU server gets two fewer cores than the stand-alone CPU version, and the final throughput is lower than the sum of the individual throughputs. The total server throughput is about 172% higher than the throughput of a 6x2-core CPU-only server.

The GPUnet-based server I/O rate with a single GPU reaches nearly 1.1GB/s. I/O activity accounts for about 40% of the server runtime. GPUnet enables high performance with a relatively modest development complexity compared to other servers. The CUDA server has 596 LOC, CPU server - 506 LOC, and GPUnet server- only 245 lines of code.

9. RELATED WORK

GPUnet is the first system to provide native networking abstractions for GPUs. This work emerges from a broader trend to integrate GPUs more cleanly with operating system services, as exemplified by recent work on a file system layer for GPUs (GPUfs) [Silberstein et al. 2013] and virtual memory management (RSVM [Ji et al. 2013]).

OS services for GPU applications. GPU applications operate outside of the resource management scope of the operating system, often to the detriment of system performance. PTask [Rossbach et al. 2011] proposes a dataflow programming model for GPUs that enables the OS to provide fairness and performance isolation. TimeGraph [Kato et al. 2011] allows a device driver to schedule GPU processors to support real-time workloads.

OSes for heterogeneous architecture. Barrelfish [Baumann et al. 2009] proposes multikernels for heterogeneous systems based on memory decoupled message passing. A multikernel approach suggests the possibility of providing OS services to GPUs, and similar approaches to GPUnet can be used as a low-level substrate to aid communication between processors. K2 [Lin et al. 2014] shows the effectiveness of tailoring a mature OS to the details of a heterogeneous architecture. GPUnet demonstrates how to bring system services to a heterogeneous system.

GPUs for network acceleration. There have been several projects targeting acceleration of network applications on GPUs. For example, PacketShader [Han et al. 2010] and Snap [Sun and Ricci 2013] use GPUs to accelerate packet routing at wire speed, whereas SSLShader [Jang et al. 2011] offloads SSL computations. Numerous high-performance computing applications (e.g., deep neural network learning [Coates et al. 2013]) use GPUs to achieve high per-node performance in distributed applications. These works use GPUs as coprocessors and do not provide networking support for GPUs. GASPP [Vasiliadis et al. 2014] accelerates stateful packet processing on GPUs, but it is not suitable for building client/server applications.

P2P DMA. P2P DMA is an emerging technology, and published results comport with the performance problems that GPUnet has on all but the very latest hardware. Potluri et al. [2013a, 2013b] use P2P DMA for NVIDIA GPUs and Intel MICs in an MPI library and report much less bandwidth with P2P DMA than communication through CPU. They suggest an optimization that uses CPU as a relay, similar to our bounce buffers. Kato et al. [2013] and APEnet+ [Ammendola et al. 2012] also propose low-latency networking systems with GPUDirect RDMA but report hardware limitations to their achieved bandwidth. Trivedi et al. [2013] point out the limitation of RDMA with its complicated interaction with various hardware components and the effect of architectural limits on RDMA.

Network stack on accelerators. Intel Xeon Phi is a coprocessor akin to a GPU but features x86 compatible cores and runs embedded Linux. Xeon Phi enables direct access to the HCA from the coprocessor and runs a complete network stack [Woodruf 2013]. GPUnet provides a similar functionality for GPUs and naturally shares some design concepts, such as the CPU-side proxy service. However, GPUs and Xeon Phi have fundamental differences. For example, GPUs have a fine-grain data-parallel programming model and lack hardware support for an operating system. These differences warrant different approaches to key design components such as the coalesced API and CPU-GPU coordination.

Scalability on heterogeneous architecture. Dandelion [Rossbach et al. 2013] is a language and system support for data-parallel applications on heterogeneous architectures. It provides a familiar language interface to programmers, insulating them from the heterogeneity.

GPMR [Stuart and Owens 2011] is a distributed MapReduce system for GPUs that uses MPI over InfiniBand for networking. However, it uses both CPUs and GPUs depending on the characteristics of the steps of the MapReduce.

Network server design. Scalable network server design has been heavily researched as processor and networking architecture advance [Welsh et al. 2001; Von Behren et al. 2003; Krohn et al. 2007; Han et al. 2012; Shalev et al. 2010; Beckmann et al. 2014], but most of this work is specific to CPUs.

Rhythm [Agrawal et al. 2014] is one of the few GPU-based server architectures that use GPUs to run PHP Web services. It promises throughput and energy efficiency that can exceed CPU-based servers, but its current prototype lacks the in-GPU networking that GPUnet provides.

Low-latency networking. More networked applications are demanding low-latency networking. RAMCloud [Ousterhout et al. 2010] notes the high latency of conventional

Ethernet as a major source of latency for a RAM-based server and discusses RDMA as an alternative that is difficult to use directly.

10. FUTURE WORK

Recent developments in the GPU hardware and software stack may help to boost GPUnet performance and broaden its capabilities, motivating us to revisit certain design and implementation decisions in the future. First, new NVIDIA GPUs, such as the Tesla K80, introduce support for large BAR sizes (up to 16GB), thus alleviating the memory pressure that prevents us from scaling the number of active connections on older GPUs. Further, recent work [Daoud et al. 2016] shows the feasibility of controlling a network device from the GPU without CPU involvement. This control mechanism can eliminate the CPU control bottleneck in GPUnet. Finally, the new CUDA async technology [Rossetti et al. 2016] may potentially resolve the GPUnet data consistency problem, paving the way toward GPUnet deployment in production systems.

These system enhancements continue the trend toward more flexible, versatile, and efficient GPU-accelerated systems where GPUs have more control over their I/O operations, underscoring the importance of the GPU-native high-level OS abstractions presented in this article for future GPUs.

REFERENCES

Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. 2014. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. 2006. Face description with local binary patterns: Application to face recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 12, 2037–2041.

R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini. 2012. APEnet+: A 3D Torus network optimized for GPU-based HPC systems. *Journal of Physics: Conference Series* 396, 1–11.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 29–44.

Nathan Z. Beckmann, Charles Gruenwald III, Christopher R. Johnson, Harshad Kasture, Filippo Sironi, Anant Agarwal, M. Frans Kaashoek, and Nickolai Zeldovich. 2014. *PIKA: A Network Service for Multikernel Operating Systems*. Technical Report MIT-CSAIL-TR-2014-002. Massachusetts Institute of Technology, Cambridge, MA. http://hdl.handle.net/1721.1/84608.

Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, New York, NY, 267–280.

Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. 1337–1345.

Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'16)*. Article No. 6.

J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*.

Bryan Ford. 2007. Structured streams: A new transport abstraction. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, New York, NY, 361–372. DOI:http://dx.doi.org/10.1145/1282380.1282421

Shalini Gupta. 2013. Efficient Object Detection on GPUs Using MB-LBP Features and Random Forests. Retrieved August 21, 2016, from http://on-demand.gputechconf.com/gtc/2013/presentations/S3297-Efficient-Object-Detection-GPU-MB-LBP-Forest.pdf.

Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 40, 4, 195–206. DOI:http://dx.doi.org/10.1145/1851275.1851207

Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*.

Sean Hefty. 2012. Rsockets. Available at https://www.openfabrics.org/index.php/resources/document-downloads/public-documents/doc_download/495-rsockets.html.

InfiniBand Trade Association. 2007. *InfiniBand Architecture Specification, Volume 1—General Specification, Release 1.2.1*. InfiniBand Trade Association.

Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*. http://portal.acm.org/citation.cfm?id=1972457.1972459

Feng Ji, Heshan Lin, and Xiaosong Ma. 2013. RSVM: A region-based software virtual memory for GPU. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, Los Alamitos, CA, 269–278.

Shinpei Kato, Jason Aumiller, and Scott Brandt. 2013. Zero-copy I/O processing for low-latency GPU computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS'13)*. ACM, New York, NY, 170–178. DOI: http://dx.doi.org/10.1145/2502524.2502548

Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*. http://portal.acm.org/citation.cfm?id=2002181.2002183

Khronos Group. 2016. OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems. Retrieved August 21, 2016, from http://www.khronos.org/opencl.

David B. Kirk and W. Hwu Wen-mei. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference*. http://dl.acm.org/citation.cfm?id=1364385.1364392

Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY.

Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. 2004. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming* 32, 3, 167–198.

NVIDIA. 2015. Developing a Linux Kernel Module Using GPUDirect RDMA. Retrieved August 21, 2016, from http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

NVIDIA. 2016. GPU Applications. Retrieved August 21, 2016, from http://www.nvidia.com/object/ gpu-applications.html.

Ohio State University Network-Based Computing Laboratory. 2015. MVAPICH2: High Performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse. ohio-state.edu. (2015).

John Ousterhout et al. 2010. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM Operating Systems Review* 43, 4, 92–105.

Sreeram Potluri, Devendar Bureddy, Khaled Hamidouche, Akshay Venkatesh, Krishna Kandalla, Hari Subramoni, and Dhabaleswar K. Panda. 2013a. MVAPICH-PRISM: A proxy-based communication framework using InfiniBand and SCIF for Intel MIC clusters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'13)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2503210.2503288

Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. 2013b. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *Proceedings of the 2013 42nd International Conference on Parallel Processing (ICPP'13)*. IEEE, Los Alamitos, CA, 80–89.

Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. 2012. Themis: An I/O efficient MapReduce. In *Proceedings of the ACM Symposium on Cloud Computing*.

Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*. 233–248.

Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 49–68. DOI: http://dx.doi.org/10.1145/2517349.2522715

Davide Rossetti, Sreeram Potluri, and David Fontaine. 2016. State of GPUdirect Technologies. Retrieved August 21, 2016, from http://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.

Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. 2010. IsoStack: Highly efficient network processing on dedicated cores. In *Proceedings of the USENIX Annual Technical Conference*. http://dl.acm.org/citation.cfm?id=1855840.1855845

Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating file systems with GPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 13.

Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014a. GPUfs: Integrating file systems with GPUs. *ACM Transactions on Computer Systems* 32, 1, Article No. 1.

Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2014b. GPUnet: Networking Abstractions for GPU Programs. Retrieved August 21, 2016, from https://sites.google.com/site/silbersteinmark/GPUnet.

W. Richard Stevens. 1993. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley.

W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. 2004. *UNIX Network Programming*. Vol. 1. Addison-Wesley Professional.

Jeff A. Stuart and John D. Owens. 2011. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE, Los Alamitos, CA, 1068–1079.

Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE, Los Alamitos, CA, 25–36. http://dl.acm.org/citation.cfm?id=2537857.2537861

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. 2011. Phoenix++: Modular MapReduce for shared-memory systems. In *Proceedings of the 2nd International Workshop on MapReduce and Its Applications*. ACM, New York, NY, 9–16.

Taneja Group Technology Analysts. 2012. InfiniBand Data Center March. Retrieved August 21, 2016, from https://cw.infinibandta.org/document/dl/7269.

Animesh Trivedi, Bernard Metzler, Patrick Stuedi, and Thomas R. Gross. 2013. On limitations of network acceleration. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'13)*. ACM, New York, NY, 121–126. DOI: http://dx.doi.org/10.1145/2535372.2535412

Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-accelerated stateful packet processing framework. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 321–332.

Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. 2012. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2391229.2391237

Vasily Volkov. 2010. Better Performance at Lower Occupancy. Retrieved August 21, 2016, from http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

Rob Von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable threads for Internet services. *ACM Operating Systems Review* 37, 268–281.

Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. *ACM Operating Systems Review* 35, 230–243.

Bob Woodruf. 2013. OFS Software for the Intel Xeon Phi. *In Proceedings of the OpenFabrics Alliance International Developer Workshop*.

Lior Zeno and Mark Silberstein. 2016. The case for I/O preemption on discrete GPUs. In *Proceedings of the International Workshop on GPU Computing Systems (GPGPU'16)*. 63–71.

Received January 2016; accepted June 2016