GPUnet: Networking Abstractions for GPU Programs

Sangman Kim Seonggu Huh Yige Hu
Xinya Zhang Emmett Witchel
The University of Texas at Austin

Amir Wated Mark Silberstein[1]
Technion – Israel Institute of Technology

Abstract

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.

1. Introduction

GPUs have become the platform of choice for many types of parallel general-purpose applications from machine learning to molecular dynamics simulations [3]. However, harnessing impressive GPU 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 GPU's computational density require unacceptable development costs to realize their full performance potential.

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

This paper 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 programing 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 like MapReduce. While 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 inter-process 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 6-core Intel CPU and a CUDA-based server by $1.5\times$ and $2.3\times$ respectively, while maintaining $3\times$ 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 [35]), and communicate over Infiniband with other GPUs. This architec-

[1] Corresponding author: mark@ee.technion.ac.il

ture 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 providing speedups of 2.9–3.5× over one GPU.

This paper begins with the motivation for building GPUnet (§2), a review of the GPU and network hardware architecture (§3), and high-level design considerations(§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 (§5).
- It presents several novel optimizations for enabling discrete GPUs to control network traffic (§6).
- It develops three substantial GPU-native network applications: a matrix product server, in-GPU-memory MapReduce, and a face verification server (§7).
- It evaluates GPUnet primitives and entire applications including multiple workloads for each of the three application types (§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 [3] has no mention of GPU-accelerated network services. Using GPUs in software routers and SSL protocols [16, 19, 37], as well as in distributed applications [12] resulted in significant speedups but required heroic development efforts. Recent work shows that GPUs can boost power efficiency and performance for web servers [5], 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 optimize 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 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 (§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 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 [4] 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 in order 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, in order 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 the database queries will be delayed until after the GPU face detection kernel terminates, leading to increased response time.

Complex multi-stage 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. While effective for GPU-CPU interaction, the pipeline grows into a complex multi-stage data flow in-

volving 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 which 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 pre-allocating dedicated GPU memory buffers and registering them with the NIC. Unfortunately, these GPU buffers are hard to manage since 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 system processors and allows using it via a standard, familiar API, thereby simplifying GPU development and facilitating integration of GPUs into complex software systems.

3. Hardware architecture overview

We provide an overview of the GPU software/hardware model, RDMA networking and peer-to-peer (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 [15] 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 [23]). They comprise several big cores, *Streaming Multiprocessors* (*SMs*), each having multiple hardware contexts and several Single Instruction, Multiple Data (SIMD) units. All the 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 the 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, and each parallelized for tightly-coupled parallel execution by the threadblock threads. Once a threadblock has been dispatched to an SM, it cannot be preempted and occupies that SM until all of the threadblock's threads terminate.

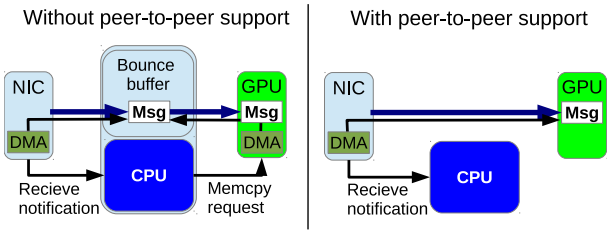

Figure 1: Receiving network messages into a GPU. Without P2P DMA, the CPU must use a GPU DMA engine to transfer data from the CPU bounce buffer.

The primary focus of this work is on discrete GPUs, which are peripheral devices connected to the host system via a standard PCI Express (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 the data in and out of GPU memory efficiently requires DMA.[2] 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 (HCA) is capable of transferring data directly to/from the GPU memory of NVIDIA K20 GPUs (see 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 load on system memory [27, 28].

RDMA and Infiniband. *Remote Direct Memory Access* (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 high throughput and low latency networking, surpassing legacy Ethernet performance and cost efficiency [8]. For example, the state-of-the-art fourteen data rate (FDR) Infiniband provides 56Gbps throughput and sub-microsecond 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 called a Host Channel Adapter (HCA) and like other RDMA networking hardware, it

[2] NVIDIA CUDA 6.0 provides CPU-GPU software shared memory for automatic data management, but the data transfer costs remain.

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 [18] of network buffers, registered by multiple applications. In combination with P2P DMA, the HCA may access buffers in GPU memory. The low-level *VERB* interface to RDMA is not easy to use. Instead, system software uses VERBs to implement higher-level data transfer abstractions. For example, the *rsockets* [32] library provides a familiar socket API in user-space for RDMA transport. Rsockets are a drop-in replacement for sockets (via LD_PRELOAD), providing a simple way to perform streaming over RDMA.

4. Design considerations

There are many alternative designs for GPU networking; this section discuses important high-level tradeoffs.

4.1 Sockets and alternatives

The GPUnet interface uses sockets because we believe they offer the best blend of properties, being generic, familiar, convenient to use, and versatile (e.g., inter-process communication over UNIX domain sockets). Alternatives like remote direct memory access (RDMA) via a VERBs API are too difficult to program [39]. Existing message passing frameworks (e.g., MPI) [2] allow zero-copy transfers into GPU memory, but they keep all network I/O control on the CPU, and suffer from the conceptual limitations of the GPU-as-slave model that we address in this work.

4.2 Discrete GPUs

We develop GPUnet for discrete GPUs, even though hybrid CPU-GPU processors and system-on-chip options like AMD Kaveri and Qualcomm Snapdragon are gaining market share. We believe discrete and hybrid GPUs will continue to co-exist for years to come. They embody different tradeoffs between power consumption, production costs and system performance, and thus serve different application domains. The aggressive, throughput-optimized hardware designs of discrete GPUs rely heavily on a multi-billion 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 multi-threaded 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 CPU-GPU communications and invocation overheads, which otherwise dominate the execution time. For example, SSLShader [19] 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.

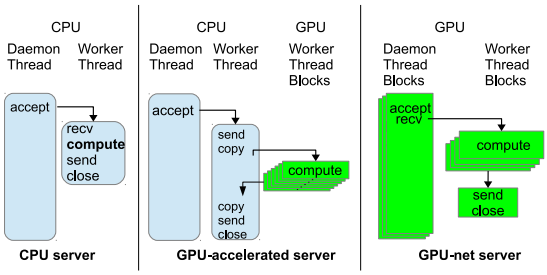

Figure 2: The architecture of a network server on a CPU, using a GPU as a co-processor, and with GPUnet (daemon architecture).

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 multi-core CPUs. We call this 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 §8.

This organization changes the tradeoffs a designer must consider for a networked service because it removes the need to batch work so heavily, thereby greatly simplifying the programming model. We hope 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*.

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, because it eliminates the overheads associated with 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 64x64 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.

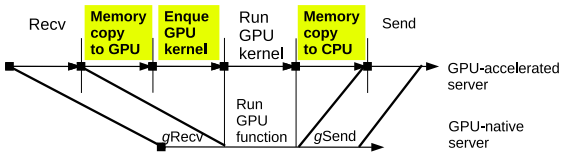

Figure 3: The logical stages for a task processed on a GPU-accelerated CPU server (top) and GPU-native network server(bottom). Highlighted stages are eliminated by the GPU networking support.

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 GPU I/O calls do not relinquish the GPU's resources, as discussed in Section 8.

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 their associated network streams. 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 using 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. GPU sockets are shared across all GPU threads, but cannot be migrated to processes running on other GPUs or CPUs in the same host.

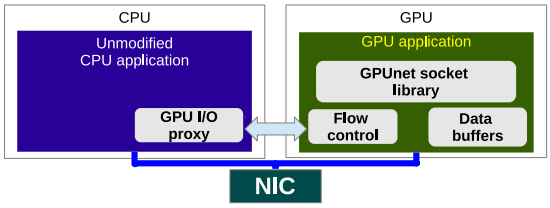

Figure 4: GPUnet high-level design.

GPUnet supports the main calls in the standard network API, including `connect`, `bind`, `listen`, `accept`, `send`, `recv`, `shutdown`, and `close` and their non-blocking versions. In the paper and in the actual implementation we add a "g" prefix to emphasize that the code executes on a GPU. These calls work mostly as expected, though we introduce coalesced multithreaded API calls as we now explain.

Coalesced API calls. A traditional CPU network API is single-threaded, i.e., each thread can make independent API calls and receive independent results. GPU threads, however, behave differently from CPU threads. They are orders of magnitude slower, and the hardware is optimized to run groups of threads (e.g. 32 in an NVIDIA warp or 64 in an AMD wavefront) in lock-step, 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 this hierarchical parallelism in mind: they exploit coarse-grain task parallelism across multiple threadblocks, and process a single task using all the 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 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, at the same point in application code (similar to vectorized I/O calls [42]). These collaborative calls together comprise one logical GPUnet operation. This idea was inspired by a similar design for the GPU file system API [34].

We illustrate coalesced calls in Figure 5. It shows a simple GPU server which 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.

```
increment_by_one_server(int csoc)
{
  //buffer shared by all TB threads
   __shared__ char buf[THREADS_IN_TB];
  //collaborative recv into buf
  len=THREADS_IN_TB;
  grecv(csoc, buf, len);
  //data parallel code per thread
  buf[thread_id]++;
  //collaborative send from buf
  gsend(csoc, buf, len);
}
```

Figure 5: A GPU network client using GPUnet (TB – threadblock).

5.2 GPU-NIC interaction

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

The majority of existing GPU networking projects (with the notable exception of the GASPP packet processing framework [40]) employ the CPU OS network stack with network buffers in CPU memory, and explicit application data movement to and from the GPU. Specifically, accelerated network applications, like SSL protocol offloading [19], cannot operate on raw packets and first require transport-level processing by a CPU. However CPU-GPU memory transfers associated with CPU-side network processing are detrimental to performance as we show in the 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, since a GPU would receive raw network packets, achieving the goal of providing a reliable in-order socket abstraction would require porting major parts of the CPU network stack to the GPU – a daunting task, which to be efficient requires thousands of packets to be batched in order to hide the overheads of the control-heavy and memory intensive processing involved [40].

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 them directly in application memory, ready to be delivered to an application 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 can 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 register the GPU memory with the NIC's DMA hardware.

5.3 Socket layer

The GPU socket layer implements a reliable in-order stream abstraction over low-level network buffers and reliable RDMA message delivery. We adopt an 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 name space 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.

Mapping streams to channels. GPUnet maps streams one-to-one onto RDMA channels. A channel is a low-level RDMA connection that does not have flow control,[3] so GPUnet must provide flow control using a ring buffer described in Section 6.1. 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 allows 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.[4]

We considered multiplexing several streams over a single channel, similar to SST [14], 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.

Naming and address resolution. GPUnet relies on the CPU standard name resolution mechanisms for RDMA transports (CMA) which provide IP-based addressing for RDMA services to initiate the connection.

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

6. Implementation

We implement GPUnet for NVIDIA GPUs and use Mellanox Infiniband Host Channel Adaptors (HCA) for inter-GPU networking [1].

[3] While the Infiniband transport layer does have its own flow control, it is message-oriented and we do not use it for streaming.

[4] Millions for Mellanox Connect-IB, according to Mellanox Solution Brief `http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf`

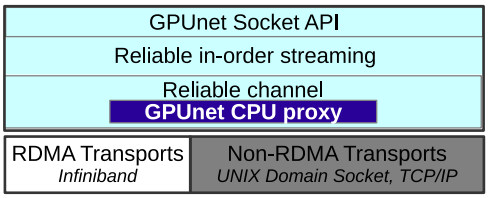

Figure 6: GPUnet network stack.

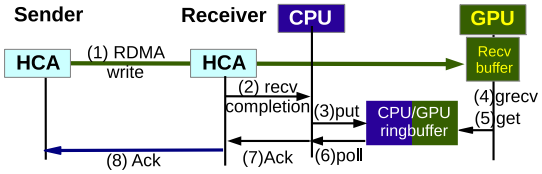

Figure 7: Ring buffer updates for GPU flow control mechanism in `grecv()` call.

GPUnet follows a layered design 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. 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 stream. Finally, the top layer implements the blocking and non-blocking versions of standard socket API for the GPU.

6.1 Socket layer

GPUnet's socket interface is compatible with and builds upon the open-source *rsockets* [32] library for socket-compatible data streams over RDMA for CPUs. Rsockets is a drop-in replacement for sockets (via LD_PRELOAD) which provides a simple way to use RDMA over Infiniband. GPUnet extends the library to use network buffers in GPU memory and integrates it with the GPU flow control mechanisms.

GPUnet maintains a private socket table in GPU. Each active socket is associated with a single reliable 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. The flow control mechanism allows the sender to block if the receiver's network buffer is full. Therefore, an implementation requires the receiver to update the sender upon buffer consumption.

Unfortunately, our original design to handle flow control entirely on the GPU is not yet practical on current hardware. NVIDIA GPUs cannot yet control an HCA directly, without additional help from a CPU. They cannot access the HCA's "door-bell" registers in order to trigger a send operation, because accessing the door-bell registers is done through memory mapped I/O, and GPUs cannot currently map that memory. Further, the HCA driver does not yet allow placement of completion queue structures in GPU memory. The HCA uses completion queues to deliver completion notifications, e.g., when new data arrives. Therefore, a CPU is necessary to assist every GPU send and receive operation.

Using a CPU for handling completion notifications introduces an interesting challenge for the flow control implementation. The flow control counters must be shared between a CPU and a GPU, since 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 which 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), reads the data and updates the 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 CPU and GPU. On the GPU side it manages the network buffers in GPU memory, while the CPU side logic ensures that the buffers are delivered to and from the transport mechanism underneath, as we describe shortly.

Memory management. GPUnet allocates a large contiguous region of GPU memory which it uses for network buffers. To enable RDMA hardware transport, the CPU code registers the GPU memory into the Infiniband HCA with the help of CUDA's GPUDirectRDMA mechanism. The maximum total amount of HCA registered memory

is limited to 220MB 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, and also because we were unable to register it while the GPU kernel is running. GPUnet uses this RDMA-registered memory as a memory pool for allocating a receive and send buffer 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 has higher latency and requires larger system buffer than native RDMA, as we measure in Section 8.1. However, this functionality serves to bridge current hardware constraints, which often make the use of RDMA impossible or inefficient. P2P DMA for GPUs and other peripherals has been made available only since early 2013, and its hardware and software support is still immature. For example, on some modern server chipsets we encountered $15\times$ bandwidth degradation when storing send buffers in GPU memory, and as a result had to use bounce buffers. Similarly, P2P DMA is only possible in a certain PCIe topology, so 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 Performance optimizations.

Single threadblock I/O. While developing GPUnet applications we found that it is convenient to dedicate some threadblocks to performing network operations, while using others only for computation, like the receiving threadblock in MapReduce (§7.1), or a daemon threadblock in the matrix product server (§7). In such a design, the performance-limiting factor for send operations is the *latency* of two steps performed in the `send()` call: memory copy between the system and user buffers in GPU, and the update of the flow control ring buffer metadata.

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 in order to enable memory-level parallelism [41]. We resorted to PTX, NVIDIA GPU low-level assembly, in order to implement 128-bit/thread vector accesses to global memory which also bypass the L2 and L1 caches. This 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 metadata is shared between the CPU and GPU, and we placed it in "zero-copy" memory, which physically resides on a CPU. Therefore, reading this memory from the GPU incurs a significant penalty of about 1-2μsec. Updating the ring buffer requires multiple reads, and the latency accumulates to tens of μsec.

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 in order 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, which is not supported by CUDA. We implement our own mapping using NVIDIA's GPUDirect from a Linux kernel module. This optimization reduces the latency of ring buffer updates to 2.5μsec.

6.4 Limitations

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

More significantly, the prototype relies on the ability of a GPU to provide the means to guarantee consistent reads to its memory when it is concurrently accessed by a running kernel and the NIC RDMA hardware. NVIDIA GPUs do not currently provide such consistency guarantees. In practice, however, we do not observe consistency violations in GPUnet. Specifically, to validate our current implementation, we implement a GPU CRC32C library and instrument the applications to check the data integrity of all network messages with 4KB granularity. We detect no data integrity violations for experiments reported in the paper (though this experiment surfaced a small bug in GPUnet itself).

We hope, perhaps encouraged by GPUnet itself, that GPU vendors will provide such consistency guarantees in the near future. In fact, the necessary CPU-GPU memory consistency will be supported in the future releases of OpenCL 2.0-compliant GPU platforms, thereby supporting our expectation that it will become the standard guarantee in future systems.

7. Applications

Matrix product server. The matrix product server is implemented entirely on the GPU, using both the daemon and independent architectures (§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 param-

eter 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 which 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 CPU by a custom mechanism. See Section 8.2 for performance measurements.

7.1 MapReduce design

We design an in-GPU-memory distributed MapReduce framework that keeps intermediate results of map operation in GPU memory, while input and output are read from disk using GPUfs [34]. We call the system *GimMR* for *G*PU *i*n *m*emory *M*ap *R*educe. GimMR is a native GPU application without CPU code. 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 has completed. Our mappers push data, while in traditional MapReduce, the reducers pull [13]. Each GPU runs multiple mappers and reducers, each of which are 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 with hundreds of threads. Mappers generate intermediate <key,value> pairs that they assign to *bucket*s 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 [29].

Each Map function is invoked in one threadblock and is executed by all the threadblock threads. On each GPU, there are many mapper threadblocks and consumer threadblocks, with the consumer threadblocks receiving buckets from remote GPUs. Each consumer threadblock is assigned a fixed number of connections from a remote GPU. The receivers get data by making non-blocking calls to `grecv()` on the mappers' sockets in round-robin order (using `poll()` on the GPU is left as future work).

The network connections are set up at the beginning of the Map phase, between each pair of consumer threadblock and remote threadblock. For example, a GPU node in a GimMR system with five GPUs, each with 12 mapper and 12 consumer threadblocks, will have a total of 48 incoming connections, one per mapper from every other GPU. And each of its 12 consumers will handle 4 incoming connections. Local mappers update local buckets without sending them through the network.

GPU mappers coordinate with 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 it should read from its input file.

Similar to the Map, each Reduce function is also invoked in one threadblock. Each reducer identifies the set of buckets it must process, (optionally) performs parallel sort of all the 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-grained parallelism of a function processing 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 in 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.

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 pre-sample 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 re-cluster 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 results of the reducers, and calculates the new centroids for the next round.We preprocess the input file to piecewise transpose the input

points, thereby coalescing memory accesses for threads in a threadblock.

7.2 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 [6]. 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 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 possible. Also, 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 non-blocking 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) 136x136 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, non-blocking 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 Figure 2, middle picture).

The GPUnet server is a native GPU-only application using GPUnet for network operations. It uses the independent architecture (§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 since the processing time per image is low and there is enough parallelism per request.

Implementation details. We use a standard benchmarking face recognition dataset[5], resized to 136x136 and reformatted as raw grayscale images. We implement a GPU `memcached` client library. `memcached` uses 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.

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

Table 1: Hardware and software configuration. The DMA column indicates the presence of a DMA performance asymmetry (§6.2).

8. Evaluation

Hardware. We run our experiments on a cluster with four nodes (Table 1) connected by a QDR 40Gbps Infiniband interconnect, using Mellanox HCA cards with MT4099 and MT26428 chipsets.

All machines use CUDA 5.5. ECC on GPUs, hyperthreading, SpeedStep, and Turbo mode of all the machines are disabled for reproducible performance. Nodes A and B feature a newer chipset with a PLX 8747 PCIe switch which enables full bandwidth P2P DMA between the HCA and the GPU. Nodes C and D provide full bandwidth for DMA writes from HCA to GPU (`grecv()`), but perform poorly with only 10% of the bandwidth for DMA reads from GPU (`gsend()`). We are not the first to observe such asymmetry [28].

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 [11].

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.

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

	C-C	C-G RDMA	C-G BB	G-G RDMA	G-G BB
RTT 64 byte(μsec)	2.86	26.9	60.3	50.0	117
Bandwidth (GB/s)	3.44	3.44	3.48	3.38	3.46

Table 2: Single stream latency (round trip time) and bandwidth for GPUnet, CPU uses rsockets. C–CPU, G-GPU, BB–bounce buffer.

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

Table 3: Latency breakdown for a GPU `gsend()` request with a 64KB message with peer-to-peer RDMA.

Single stream performance. We run a simple single-threadblock GPU echo server and client using a single GPUnet socket. We implement the CPU version of the benchmark using the unmodified rsockets library. Table 2 shows the round trip time (RTT) for 64 byte messages and bandwidth for 64KB messages and 256KB (512KB for bounce buffer) system buffers. The GPU reaches about 98% of the peak performance of CPU-based rsockets. Bounce buffers (entries marked BB in the table) increase latency two-fold versus RDMA transfers, but its throughput is close to RDMA thanks to twice larger system buffers for better latency hiding.

The latency of GPU transfers is significantly higher than the baseline CPU-to-CPU latency. To understand the reasons, Table 3 provides the breakdown for the latency of individual steps of `gsend()` sending 64KB.

We measured 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.

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 eliminate T_2 in software.

Duplex performance. The CPU rsocket library achieves 6.65 GB/s of the aggregate duplex bandwidth for two concurrent data streams in opposite directions – twice the bandwidth of a single stream. With GPUnet, we found that `gsend` and `grecv` interfere when invoked concurrently on two sockets, but the reasons for this interference is still unclear. Specifically, when using a CPU end-point, the throughput of `grecv` and `gsend` is 3.31 GB/s and 2.63 GB/s respectively. As a result, in a GPU-GPU experiment with two opposite streams, the one-directional bandwidth is constrained by the `gsend` performance on both sides, hence the aggregate bandwidth is 5.26 GB/s.

Multistream bandwidth. We measured the aggregate bandwidth of sending over multiple sockets from one GPU. We run 26 threadblocks (2 threadblocks per GPU SM core) each having multiple non-blocking 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 § 6, the maximum number of sockets is constrained by the total amount of RDMA-registered memory available for network buffers, which is currently limited to 220MB.

We run the experiment between two GPUs. Starting from 2 connections, GPUnet achieves a 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 (`matrixMul`), the matrix product kernel distributed with the NVIDIA SDK.

The *daemon* server uses GPUnet and follows the daemon architecture (§4.3). The GPU resources are partitioned between daemon threadblocks and computing threadblocks. The number of daemon threadblocks is an important server configuration parameter as we discuss below. 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 the 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 non-preemptive scheduling model implies that GPU resources allocated to I/O tasks cannot execute computations even

Configuration	Workload Light	Medium	Heavy
Light	92%	81%	74%
Medium	44%	99%	88%
Heavy	12%	44%	100%

Table 4: The cost of misconfiguration: the throughput in a given configuration relative to the maximum throughput using the best configuration for that workload.

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 [6]. In our current implementation the number of daemon threadblocks is configured at server invocation time and does not change during execution.

The best server configuration depends on the workload. Intuitively, the more computation that is performed per byte of I/O, the fewer GPU resources should be allocated for I/O threadblocks and, consequently, more resources allocated for computation. The optimal server configuration depends on the compute-to-I/O ratio of its tasks.

Balancing the allocation of threadblocks between computation and I/O is a high-stakes game. Table 4 shows how we separate three matrix multiplication workloads by their compute-to-I/O ratio: light (64x64 and 128x128), medium (256x256) and heavy (512x512 and 1024x1024).

We exhaustively search the configuration space for each workload (with varying number of clients) to find the configuration of compute and I/O threadblocks that maximizes throughput. Then we run all workloads on all configurations and measure the penalty for using the best configuration for each class of workload. Splitting workloads into three classes allows us to find configurations that perform very well for all instances of that class (the diagonal is all above 90% of optimal). However, dedicating too many or too few threadblocks to I/O can be terrible for performance, with the worst misconfiguration reducing throughput to 12% of optimal. Future work includes a generic method of finding the best server configuration and dynamically adjusting it to suit the workload.

Performance comparison of different 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

[6] In practice, the number of threads per a daemon threadblock also affects the server performance, but we omit these technical details for simplicity.

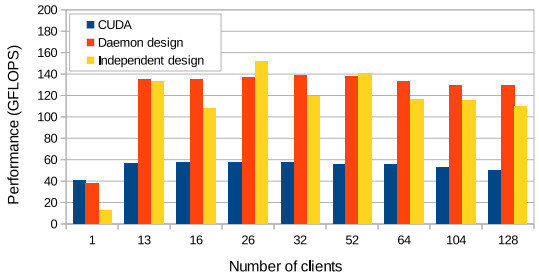

Figure 8: Throughput comparison for different matrix product servers.

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 5: The throughput of GPUnet-based matrix product servers under different workload types.

which maximizes its throughput for this workload. The results are shown in Figure 8.

Both GPUnet-based implementations consistently outperform the traditional CUDA server across all the 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 are 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.

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

Table 5 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 of the 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 below (§ 8.4).

8.3 Map reduce

We evaluate the standard word count and K-means tasks on our GimMR MapReduce. Table 6 compares the performance of the single-GPU GimMR with the single-node Hadoop and Phoenix++ [38] on a 8-core CPU. We

Workload	8-core Phoenix++	1-Node Hadoop	1-GPU GimMR
K-means	12.2 sec	71.0 sec	5.6 sec
Wordcount	6.23 sec	211.0 sec	29.6 sec

Table 6: Single-node GimMR vs. other MapReduce systems.

use RAM disk and IP over IB when evaluating K-Means on Hadoop. For both wordcount and kmeans 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 8-core Hadoop by a factor of 7.1×, but is 4.7× slower than Phoenix++ [38] running on 8 CPU cores. GimMR word count spends a lot of time sorting strings, which is expensive on GPUs because comparing variable length strings create divergent, irregular computations. In the future we will adopt the optimization done by ThemisMR [29] which uses the hash of the strings as the intermediate keys, in order 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, (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.

Table 6 compares the performance of GimMR with single-node Hadoop and Phoenix++ using 200 dimension vectors. GimMR on a single GPU outperforms Phoenix++ on 8 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 Mellanox Connect-IB HCA, 2× Intel E5-2620 6-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 the reported results have variance below 0.1% of their mean.

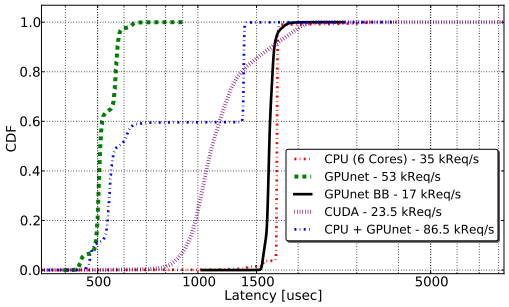

Figure 9: Face verification latency CDF for different servers.

Lower latency, higher throughput. Figure 9 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 tradeoff 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 6-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 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). While 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 7, 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

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

Table 7: Face verification throughput for different servers.

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 standalone 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 - 506, and GPUnet– 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) [34] and virtual memory management (RSVM [20]).

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 [30] proposes a data flow programming model for GPUs that enables the OS to provide fairness and performance isolation. TimeGraph [22] allows a device driver to schedule GPU processors to support real-time workloads.

OSes for heterogeneous architecture. Barrelfish [9] proposes multikernels for heterogeneous systems based on memory decoupled message passing. K2 [25] shows the effectiveness of tailoring a mature OS to the details of a heterogeneous architecture. GPUnet demonstrates how to bring system services into a heterogeneous system.

GPUs for network acceleration. There have been several projects targeting acceleration of network applications on GPUs. For example, PacketShader [16] and Snap [37] use GPUs to accelerate packet routing at wire speed, while SSLShader [19] offloads SSL computations. Numerous high-performance computing applications (e.g., Deep Neural Network learning [12]) use GPUs to achieve high per-node performance in distributed applications. These works use GPUs as co-processors, and do not provide networking support for GPUs. GASPP [40] accelerates stateful packet processing on GPUs, but it is not suitable for building client/server applications.

Peer-to-peer DMA. P2P DMA is an emerging technology, and published results comport with the performance problems GPUnet has on all but the very latest hardware. Potluri et. al. [27, 28] 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. Kato et. al [21] and APEnet+ [7] also propose low-latency networking systems with GPUDirect RDMA, but report hardware limitations to their achieved bandwidth. Trivedi et al. [39] 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 co-processor akin to a GPU, but featuring x86 compatible cores and running embedded Linux. Xeon Phi enables direct access to the HCA from the co-processor and runs a complete network stack [45]. GPUnet provides a similar functionality for GPUs, and naturally shares some design concepts, like the CPU-side proxy service. However, GPUs and Xeon Phi have fundamental differences, e.g. fine-grain data parallel programming model, and the lack of hardware support for operating system, which warrant different approaches to key design components such as the coalesced API and the CPU-GPU coordination.

Scalability on heterogeneous architecture. Dandelion [31] 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 [36] is a distributed MapReduce system for GPUs, which 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 [10, 17, 24, 33, 43, 44], but most of this work is specific to CPUs.

Rhythm [5] 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 [26] 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. Acknowledgments

Mark Silberstein was supported by the Israel Science Foundation (grant No. 1138/14) and the Israeli Ministry of Science. We also gratefully acknowledge funding from NSF grants CNS-1017785 and CCF-1333594.

References

[1] *GPUnet project web page*. https://sites.google.com/site/silbersteinmark/GPUnet.

[2] MVAPICH2: High performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse.ohio-state.edu.

[3] *Popular GPU-accelerated applications*. http://www.nvidia.com/object/gpu-applications.html.

[4] Efficient Object Detection on GPUs using MB-LBP features and Random Forests. GPU Technology Conference, 2013. http://on-demand.gputechconf.com/gtc/2013/presentations/S3297-Efficient-Object-Detection-GPU-\MB-LBP-Forest.pdf.

[5] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[6] T. Ahonen, A. Hadid, and M. Pietikainen. Face description with local binary patterns: Application to face recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(12):2037–2041, 2006.

[7] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini. APEnet+: a 3D Torus network optimized for GPU-based HPC Systems. In *Journal of Physics: Conference Series*, volume 396. IOP Publishing, 2012.

[8] T. G. T. analysts. InfiniBand data center march, 2012. https://cw.infinibandta.org/document/dl/7269.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[10] N. Z. Beckmann, C. Gruenwald III, C. R. Johnson, H. Kasture, F. Sironi, A. Agarwal, M. F. Kaashoek, and N. Zeldovich. PIKA: A network service for multikernel operating systems. Technical Report MIT-CSAIL-TR-2014-002, MIT, January 2014.

[11] T. Benson, A. Akella, and D. A. Maltz. 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*, pages 267–280. ACM, 2010.

[12] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345, 2013.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[14] B. Ford. Structured streams: A new transport abstraction. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 361–372, New York, NY, USA, 2007. ACM.

[15] K. Group. *OpenCL - the open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/opencl.

[16] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40:195–206, August 2010.

[17] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[18] InfiniBand Trade Association. *InfiniBand Architecture Specification, Volume 1 - General Specification, Release 1.2.1*, 2007.

[19] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Berkeley, CA, USA, 2011. USENIX Association.

[20] F. Ji, H. Lin, and X. Ma. RSVM: a region-based software virtual memory for GPU. In *Proceedings of 22nd International Confreence on Parallel Architectures and Compilation Techniques (PACT)*, pages 269–278. IEEE, 2013.

[21] S. Kato, J. Aumiller, and S. Brandt. 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, pages 170–178, New York, NY, USA, 2013. ACM.

[22] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011. USENIX Association.

[23] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[24] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007. USENIX Association.

[25] F. X. Lin, Z. Wang, and L. Zhong. 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)*. ACM, 2014.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM Operating Systems Review*, 43(4):92–105, 2010.

[27] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. Panda. 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)*, New York, NY, USA, 2013. ACM.

[28] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89. IEEE, 2013.

[29] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An I/O Efficient MapReduce. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[30] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 233–248, 2011.

[31] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 49–68, New York, NY, USA, 2013. ACM.

[32] Sean Hefty. Rsockets. OpenFabrics International Workshop, 2012. `https://www.openfabrics.org/index.php/resources/document-downloads/public-documents/doc_download/495-rsockets.html`.

[33] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack: Highly efficient network processing on dedicated cores. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010. USENIX Association.

[34] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013.

[35] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[36] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[37] W. Sun and R. Ricci. Fast and Flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 25–36, Piscataway, NJ, USA, 2013. IEEE Press.

[38] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[39] A. Trivedi, B. Metzler, P. Stuedi, and T. R. Gross. On limitations of network acceleration. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 121–126, New York, NY, USA, 2013. ACM.

[40] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. Gaspp: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 321–332, Philadelphia, PA, June 2014. USENIX Association.

[41] Vasily Volkov. Better performance at lower occupancy. GPU Technology Conference, 2010. `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.

[42] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2012. ACM.

[43] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *ACM Operating Systems Review*, volume 37, pages 268–281. ACM, 2003.

[44] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[45] B. Woodruf. OFS software for the Intel Xeon Phi. OpenFabrics Alliance International Developer Workshop, 2013.