Extended Task Queuing: Active Messages for Heterogeneous Systems

Michael LeBeane*†, Brandon Potter†, Abhisek Pan†, Alexandru Dutu†, Vinay Agarwala†, Wonchan Lee‡, Deepak Majeti§, Bibek Ghimire‖, Eric Van Tassell†, Samuel Wasmundt¶, Brad Benton†, Mauricio Breternitz†, Michael L. Chu†, Mithuna Thottethodi**, Lizy K. John*, Steven K. Reinhardt†

*University of Texas at Austin
{mlebeane, ljohn}@utexas.com

‡Stanford University
wonchan@cs.standford.com

¶University of California, San Diego
wasmundt@eng.ucsd.edu

‖Louisiana State University
gbibek@gmail.com

§HPE Vertica
deepak.majeti@hpe.com

**Purdue University
mithuna@purdue.edu

†Advanced Micro Devices, Inc.
{Michael.Lebeane, Brandon.Potter, Abhisek.Pan, Alexandru.Dutu, Vinay.Agarwala, Eric.Vantassell, Brad.Benton, Mauricio.Breternitz, Mike.Chu, Steve.Reinhardt}@amd.com

***Abstract*—Accelerators have emerged as an important component of modern cloud, datacenter, and HPC computing environments. However, launching tasks on remote accelerators across a network remains unwieldy, forcing programmers to send data in large chunks to amortize the transfer and launch overhead. By combining advances in intra-node accelerator unification with one-sided Remote Direct Memory Access (RDMA) communication primitives, it is possible to efficiently implement lightweight tasking across distributed-memory systems.**

This paper introduces Extended Task Queuing (XTQ), an RDMA-based active messaging mechanism for accelerators in distributed-memory systems. XTQ's direct NIC-to-accelerator communication decreases inter-node GPU task launch latency by 10-15% for small-to-medium sized messages and ameliorates CPU message servicing overheads. These benefits are shown in the context of MPI accumulate, reduce, and allreduce operations with up to 64 nodes. Finally, we illustrate how XTQ can improve the performance of popular deep learning workloads implemented in the Computational Network Toolkit (CNTK).

***Index Terms*—Accelerator architectures, Computer architecture, Computer networks, Distributed computing, Network interfaces.**

I. Introduction

Accelerators have emerged as ubiquitous components of many datacenter, cloud computing, and HPC ecosystems [1–3]. At the time of this writing, over 100 of the top 500 supercomputers use accelerators [4], typically GPUs or Intel Xeon Phi co-processors [5]. Similarly, Amazon offers GPU-enabled nodes as part of their virtualized Elastic Compute Cloud service [6]. However, despite widespread adoption, accelerators are traditionally deployed as peripheral components that must marshal data to and from main memory at the directive of a kernel driver.

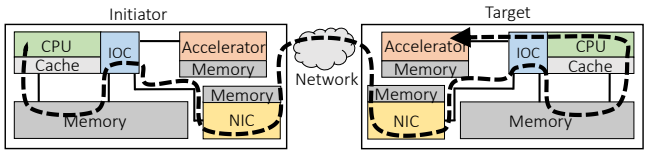
Many of the major heterogeneous platform providers are striving to incorporate accelerators more tightly into a node's compute ecosystem [7–10]. Most of the major hardware vendors offer a number of standardized features that enable accelerators to participate in computation as peers to the host CPU. These features typically include user-mode task invocation, shared virtual memory with a well-defined memory consistency model, shared-memory-based synchronization, and accelerator context switching. We refer to frameworks that implement the above features at a node level as *tightly coupled* compute ecosystems.

Eliminating data copies and privileged device-driver operations from the critical path has helped tightly coupled architectures drive down kernel launch latencies from approximately 30μs for a high-performance discrete GPU to less than 7μs for a device such as AMD's A10-7850K Accelerated Processing Unit (APU) [11]. Low-overhead kernel launches and data-copy elimination have the potential to change how applications use accelerators. Tasks that previously were too small to amortize dispatch and data movement costs will begin to benefit from accelerator offload. We expect these trends to continue, as kernel launch latency is of paramount importance to chip designers [8, 12] and researchers [13, 14].

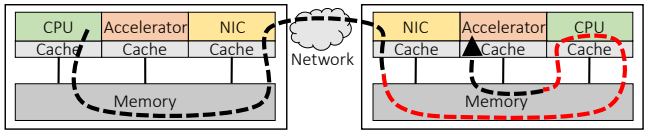
While node-level, tightly coupled frameworks remove data copies and heavyweight driver invocations for intra-node accelerators, Network Interface Controllers (NICs) with Remote Direct Memory Access (RDMA) offer these same benefits for inter-node data transfers. RDMA-capable NICs and fabrics [15–19] move data from one node to another without involving the target-node CPU by enabling the target-node NIC to perform DMA directly to and from application memory.

This paper introduces NIC primitives which combine RDMA with tightly coupled, user-level task queuing. These primitives enable applications to efficiently enqueue tasks on any compute device in a distributed-memory system, without involving the target-node CPU or the operating system on ei-

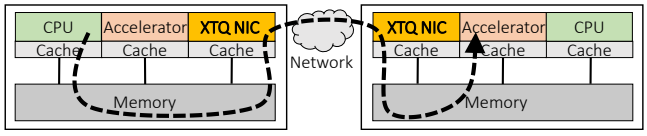
SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16$31.00



(a) Conventional heterogeneous systems



(b) Tightly coupled heterogeneous systems



(c) XTQ-enabled, tightly coupled heterogeneous systems

Fig. 1. Remote task enqueue control path on different heterogeneous, distributed-memory systems.

ther the initiator or the target node. We call this mechanism Extended Task Queuing (XTQ), since it extends the lightweight, user-mode task queuing in modern shared-memory platforms across distributed memory systems. XTQ offers a novel, highly efficient active messaging [20] platform for accelerators that improves upon the state of the art.

Figure 1 shows the control path of a point-to-point remote task invocation implemented on three types of systems. In these examples, a CPU on the initiator node schedules work on a remote accelerator. In a conventional heterogeneous node, the communication flow is similar to Figure 1a. The initiator CPU uses a high-performance NIC sitting on an I/O bus to transfer the task and associated data to the target via RDMA. While emerging technologies such as NVIDIA's GPUDirect RDMA [21] allow for NICs to transfer data directly to a GPU's onboard memory, launching a kernel still requires the intervention of the target CPU's runtime and kernel driver.

Figure 1b shows the same operation implemented on a contemporary, tightly coupled SoC. The CPU and the accelerator share the same memory, obviating the need to transfer data from the target's main memory to the accelerator's local device memory. However, the target-side CPU must still service the request from the NIC and explicitly schedule work on its local accelerator.

XTQ provides a mechanism enabling the direct CPU-to-accelerator communication presented in Figure 1c. Our scheme enables tightly integrated accelerators to efficiently and directly communicate with each other through a customized hardware NIC. The target-side NIC participates in a tightly coupled queuing model and can directly schedule work to the accelerator, completely eliminating the CPU communication path in Figure 1b.

Directly interfacing an intra-node tasking framework with inter-node RDMA through XTQ offers a number of benefits, including:

- **A unified active messaging framework for all compute devices in the system.** By leveraging the user-mode task invocation of tightly coupled systems, it is possible to design an active messaging framework that uses the same interface to spawn remote tasks on any device (CPU, GPU, FPGA, Processor-in-Memory (PIM), etc.) in the system. Unified active messaging offers exciting new acceleration possibilities for future applications and runtime libraries.
- **A reduction in remote accelerator task launch latency.** RDMA provides the means for low-latency, CPU-less data transfer without redundant data copies. Tightly coupled architectures provide the means for lightweight, direct accelerator-to-accelerator communication within a shared-memory node. By marrying the two, a target-side NIC can schedule work directly on a local accelerator without critical-path software on the CPU. Bypassing the CPU decreases accelerator launch latency and opens up the possibility of fine-grained remote tasking models for accelerators.
- **Removal of message processing and task launch overheads on the target CPU.** Message progress threads can impose a significant overhead in distributed systems [22]. This problem is exacerbated when the progress thread not only has to handle messages, but also construct command packets and schedule work on accelerators. Direct NIC-to-accelerator task invocation frees the CPU to either perform more useful computation or to enter a low power state.

This paper provides an overview of an XTQ-enabled, tightly coupled system architecture. Our exploration of XTQ is organized into the following three topics:

- **The NIC hardware design.** We illustrate that cross-node heterogeneous integration can be achieved with the addition of a small amount of hardware to an RDMA-capable NIC.
- **Programming via lightweight extensions to an RDMA-capable host API.** We implement the XTQ remote tasking primitive as a simple extension to a generic RDMA network programming interface. This API extension leverages one-sided communication semantics to allow programmers to schedule active messages on any computing device in the cluster. We illustrate that it is easy to express XTQ task invocations using only a few API calls.
- **Performance on a number of important primitive operations and machine learning applications.** We illustrate that XTQ can improve GPU-bound active message performance by 10-15% over non-XTQ enhanced messages, while eliminating message and task enqueue overheads on the target-side CPU. We present latency decompositions for important steps in the XTQ task flow and show performance improvements for microbenchmarks. We show that XTQ can enhance important MPI primitives such as accumulate, reduce, and allreduce operations, and that XTQ benefits scale as the number of nodes increases for a fixed problem size. Finally, we illustrate the real-world impact of XTQ on distributed deep learning workloads implemented using the Computational Network Toolkit (CNTK) [23].

II. Background

In this section, we discuss the concepts, trends, and recent advances on which XTQ is built.

A. Intra-node Accelerator Integration

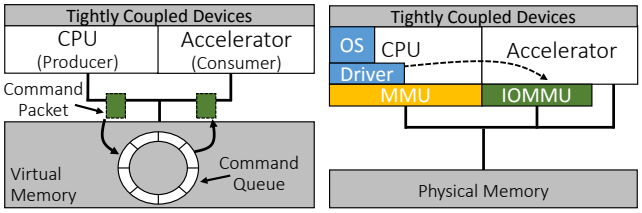
Modern system designs are increasingly providing tighter coupling between CPUs, GPUs, and other accelerators. There are a number of recent frameworks offering varying features and levels of support [7–10]. XTQ-like active messaging can be implemented on many tightly coupled, node-level architectures. However, it is helpful to explain this work in the context of an existing industry standard. For the remainder of the paper, we will use the Heterogeneous System Architecture (HSA) [8] as our prototypical example of a tightly coupled architecture. HSA is an open industry standard from the HSA Foundation, a consortium formed by AMD, ARM®, Imagination Technologies, MediaTek, Texas Instruments, Samsung Electronics, Qualcomm, and others with the objective of helping system designers integrate different kinds of computing elements (e.g., CPUs and GPUs) to enable efficient data sharing and work dispatch. Some important features of the HSA specification are illustrated in Figure 2 and are described in the following paragraphs.

User-Level Command Queuing: In HSA, applications allocate accelerator task queues in user memory. Devices fetch and execute tasks directly out of these queues, thereby eliminating OS kernel transitions and device-driver overheads on common paths. Figure 2a illustrates HSA user-level command queuing. User-mode queues are arranged as circular buffers, with the read and write pointers implemented as monotonically increasing indices. The queue entry format is defined by HSA's Architected Queuing Language (AQL). AQL packets contain all the information needed to launch and synchronize a GPU kernel or CPU function.

Shared Virtual Memory: HSA requires that devices access memory using the same virtual addresses seen by the application program. This feature is necessary to allow users to pass pointers directly to devices through the HSA task queues without device driver intervention or validation. Devices must also be capable of initiating page faults to avoid the overhead of pinning pages in memory. Shared translations can be provided to devices by an IOMMU that references the same page tables as the host CPUs [24,25], as shown in Figure 2b.

Shared-Memory Synchronization: HSA uses memory-based signal objects for synchronization. Processes indicate to a device that work has been placed in its command queue using a doorbell signal associated with the queue. The doorbell signal can map to a memory-mapped device range (e.g., for a firmware- or hardware-dispatched device such as a GPU), or to a shared-memory location (on which a software-dispatched device such as a CPU can poll). Devices or threads can also wait on tasks to finish using completion signals.

While HSA improves programmability and task invocation in a shared-memory environment, distributed-memory setups cannot naturally leverage these features. Through XTQ, distributed-memory applications are able to directly interface with the intra-node HSA tasking model.



(a) HSA tasking model. (b) Shared virtual memory.

Fig. 2. Intra-node accelerator integration in HSA.

B. Inter-node RDMA frameworks

Modern high-performance clusters employ RDMA adaptors for lightweight and efficient inter-node data movement. RDMA protocols expose connections directly to user-level applications, enabling data movement without involving the operating system or target-side processor. High-speed RDMA fabrics are increasingly available on commodity systems. Technologies such as InfiniBand™ [15], iWARP [19], and RDMA over Converged Ethernet (RoCE) [18] enable these systems to scale to thousands of nodes. Current technologies can reliably provide read latencies as low as 1.19μs [26], with emerging network technologies promising less than 100ns network latency per hop [16,27]. RDMA technology provides the communication infrastructure for XTQ to implement heterogeneous tasking across nodes. XTQ is built on a generic remote *Put* operation and can be easily integrated into any RDMA transport layer exposing one-sided communication semantics.

While XTQ can be added to any RDMA transport, we use Portals 4 [17] as the framework of reference when explaining XTQ in the context of an existing network transport layer. Portals 4 is a connectionless low-level network API designed to support the Message Passing Interface (MPI) [28] and various partitioned global address space (PGAS) languages [29, 30]. It is agnostic to the underlying fabric and exposes both flow-control and RDMA data transfer to higher-level applications and libraries.

III. XTQ Architecture

This section illustrates the main components of the XTQ hardware architecture. XTQ introduces one basic primitive to an RDMA-capable NIC: direct, user-mode task invocation on a remote accelerator. The lightweight hardware that implements this operation is described in the following paragraphs. For the purposes of our exploration of XTQ, we will refer specifically to a system with tightly coupled CPU and GPU. However, the same scheme is generalizable to other accelerators in a tightly coupled system architecture.

A. XTQ Message Format

Figure 3 illustrates the main components of a typical XTQ message along with the AQL-like [8] command packet formats for CPU and GPU tasks. For GPU tasks, the command packet

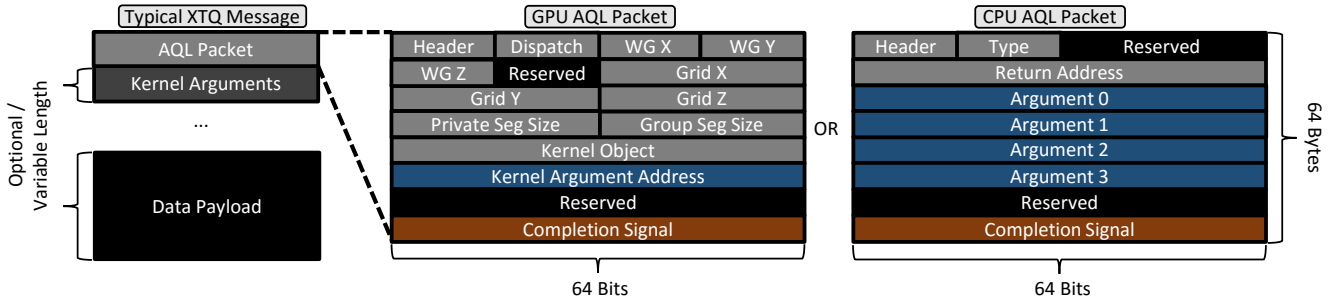


Fig. 3. XTQ message format.

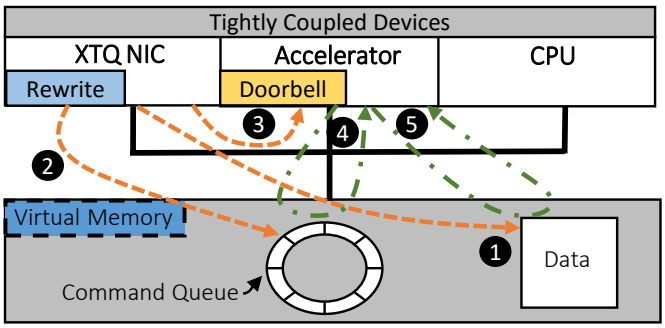


Fig. 4. Target side steps in *XtqPut* operation.

contains all the information needed to launch a kernel, such as a pointer to the kernel code object, a pointer to the kernel arguments, and workitem/workgroup sizes. Variable-sized kernel arguments are appended to the message at the end of the command packet. For CPU tasks, the command packet contains fields such as a function pointer and embedded scalar arguments.

After the command packet and kernel arguments is a variable-size payload. This payload is generally a task input buffer provided by the initiator, but there are no specific requirements for its usage. Our API and NIC consume two separate pointers for the command packet/kernel argument combination and the payload. The NIC performs a gather operation on the two buffers before transmitting to the target. Gathering the payload and command packet separately avoids an unnecessary memory copy in the application code.

B. XTQ Put Overview

This section illustrates the steps involved for a CPU to schedule a unit of work on a remote GPU. The first step in a remote task invocation is for the initiator's CPU to enqueue one or more remote *XtqPut* operations on the NIC's command queue. The NIC's software interface is provided pointers to two distinct memory buffers: one for the command packet and kernel/function arguments and one for the optional data payload. The host library notifies the NIC of the local memory buffers through a shared command queue and doorbell mechanism. The NIC then performs a local gather operation and transfers data over the network.

Figure 4 illustrates the steps involved in queuing a task on the target GPU from the NIC. First, the target NIC receives the XTQ message from the network. The payload portion of the message is streamed directly into the receive buffer in main memory ❶. The NIC also extracts the command packet from the message and performs the rewriting services described in Section III-C. Before enqueuing the packet, the NIC first accesses the command queue descriptor. If the queue is full, then the NIC triggers the flow control mechanism discussed later. Otherwise, the rewritten command packet is enqueued to the target GPU's command queue ❷. After the payload write and command enqueue is completed, the NIC writes the command queue index to the GPU's memory-mapped doorbell register ❸. In our configuration, the GPU contains a Command Processor (CP), which is responsible for reading packets from the command queue when the doorbell is updated with the newest write index. The CP proceeds to dequeue the packet from its command queue ❹, decodes the launch parameters, and schedules the work on the GPU's compute threads. The GPU threads then perform global load and store operations to access the kernel arguments and payload data ❺. Optionally, the GPU can notify the local CPU of kernel completion using a shared-memory signal (not shown).

It is important to note that all of the node-local operations take place in a unified virtual memory environment. The pointer addresses used for the command queue and data buffer are virtual addresses. In our scheme it is assumed that both the NIC and GPU will have access to an IOMMU as described in Section II-A.

Security and process isolation are critical concerns for distributed, multi-process workloads. For the XTQ extensions, the security concerns are handled by the underlying transport layer. As an example, the Portals 4 network programming API provides clear semantics to guarantee isolation of its own per-process data structures. An XTQ extension to the Portals 4 framework would inherit these same security mechanisms to protect its task queues and other auxiliary data structures.

A similar argument exists for flow control. XTQ utilizes shared-memory queues as the interface between the NIC and the GPU, which can become full. XTQ can utilize any existing hardware transport-level flow-control mechanism. In our continuing example with Portals 4, the philosophy is to provide building blocks for a higher software layer to implement arbitrarily sophisticated policies. Portals 4 can identify when target-side resources are full and generate events to notify the initiator or target that a message was not successfully

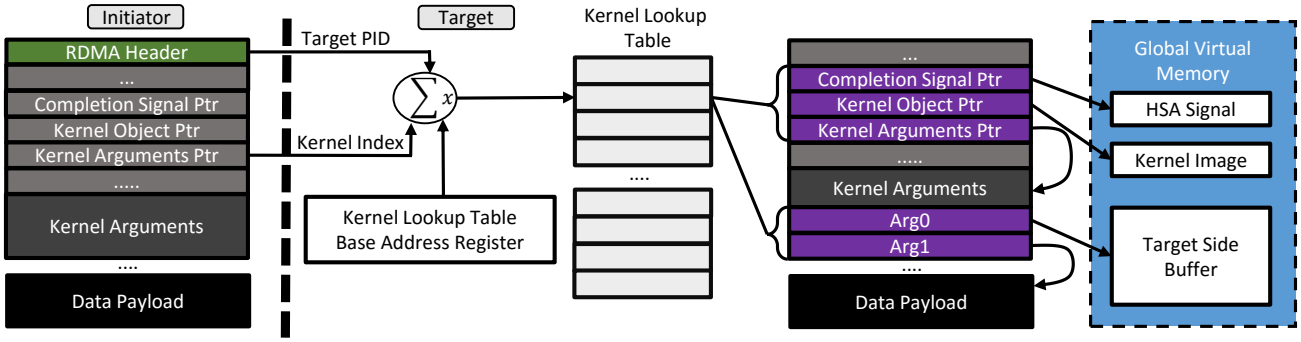


Fig. 5. Target-side XTQ rewrite semantics.

delivered. The same monitoring and messaging facilities are used for the XTQ extension to Portals 4.

C. XTQ Rewrite Semantics

One issue with remote task invocation is that the initiator is unaware of the addresses of important resources which are dynamically allocated by the target. Some examples are target-resident kernel input/output buffers, completion signals, and the GPU's command queue. One simple way to solve this issue is to broadcast the virtual addresses of all dynamically allocated data needed for task execution. However, it is costly for initiators to keep track of resource descriptors for thousands of possible target nodes; this scenario is particularly germane in scenarios when resources are frequently allocated and deallocated during program execution.

XTQ solves this problem by leveraging coordinated indices to refer to all target-resident data. The initiator populates the command packet with these indices, and the target performs a translation from an index to the correct target-local virtual address. Therefore, the initiator does not need to store any target address information, and the target only needs to keep index translations for its own resident data. Since one of the design goals is to avoid invoking the CPU on tasks that are not specifically destined for it, the XTQ framework incorporates the logic to substitute virtual addresses into the target-side NIC. We will describe the semantics of this "XTQ rewrite" operation in detail for both CPU and GPU bound tasks. We use the term "rewrite" as opposed to the arguably more accurate term "translation" to avoid confusion with virtual to physical address translations which occur during device IOMMU accesses.

1) Lookup Tables: The NIC manages a number of per-process lookup tables to hold the index-to-virtual address rewrites needed for the NIC to enqueue tasks on a compute device. There are three different types of lookup tables: the Kernel Lookup Table, Function Lookup Table, and Queue Lookup Table. Every XTQ packet will perform one lookup in either the Kernel or Function Lookup Table depending on the type of the packet. Additionally, all messages will trigger a lookup in the Queue Lookup Table to extract the base pointer of the target command queue. The entries in the lookup tables are populated by the host CPU using the XTQ API described in Section IV.

One lookup table entry is needed for each function, kernel, and queue that wishes to participate in XTQ's direct NIC-to-compute device tasking. For the applications and microbenchmarks that we studied, 64 kernel and function registrations were sufficient, producing Kernel and Function Lookup Tables that are around 4KB per process. For a small number of nodes, these data structures can be resident on dedicated tables on the NIC. For larger numbers of nodes, these tables would need to reside in DRAM. In this case, the NIC can implement an on-chip cache to ensure low-latency access to frequently used table entries.

2) Rewrite Procedure: Figure 5 shows how the NIC selectively replaces certain fields in a GPU AQL packet. The initiator places a lookup table index in the field reserved for the kernel object pointer. The target NIC uses this index to offset into the Kernel Lookup Table to replace the kernel pointer with the correct value in the target's virtual address space. The actual kernel arguments are aligned directly after the command packet in the receive buffer. XTQ replaces the kernel argument pointer in the command packet with the address at which the kernel arguments will be written to memory. Finally, the first two kernel arguments are replaced with a pointer to a pre-registered target side buffer and the initiator-provided data payload, respectively. A pointer table can be registered instead of a target-side buffer if more registrations are needed. Kernel Lookup Table entries also contain room for the registration of a target-resident, shared-memory completion signal. When the GPU finishes execution of a task, this completion signal is decremented by the CP to let other agents efficiently wait for the task to complete. XTQ replaces the command packet completion signal entry with the completion signal registered in the Kernel Lookup Table if it is valid. The rest of the fields are passed through as they are received and are assumed to be properly populated by the initiator.

A similar rewrite procedure is performed for CPU tasks. For CPU tasks, the NIC references the Function Lookup Table instead of the Kernel Lookup Table. The primary difference is that the first four function arguments are embedded directly in the AQL packet.

Finally, the NIC must identify in which of many possible user-mode queues to place the AQL packet. The queue address is extracted with one final table lookup, using an index

embedded in the AQL reserved bits to access the Queue Lookup Table.

IV. XTQ API

The XTQ tasking framework defines an API that the host CPU can use to schedule remote tasks on a target compute device. This API is implemented as an extension to a generic RDMA network programming interface that supports a remote *Put* operation. The XTQ-specific extensions can be broken down into the remote task launch function (*XtqPut*) and a number of registration functions used to populate the XTQ lookup tables.

A. XtqPut Function

XTQ contains one remote-tasking operation: *XtqPut*. The *XtqPut* operation performs the same one-sided, RDMA data movement operation as a basic Put operation, with additional semantics for launching tasks on the target. The exact mechanism for launching tasks is described in detail in Section III.

B. XTQ Lookup Table Registration

The XTQ lookup tables are populated by the host CPU using a registration API. The API provides three lookup table registration functions:

- *XtqRegisterFunction:* Associates a lookup table index with a function pointer and an optional target resident buffer.
- *XtqRegisterKernel:* Associates a lookup table index with a kernel pointer, an optional target resident buffer, and an optional completion signal.
- *XtqRegisterQueue:* Associates a lookup table index with a command queue descriptor.

These functions are meant to be invoked using globally known, coordinated indices. Such indices are common in SPMD programming techniques and are already available in distributed, multiprocess programming frameworks such as MPI.

C. Example Program

To ground our discussion of the API, Figure 6 illustrates a small program written in the SPMD style utilizing the primary components of XTQ. In this example, the initiator CPU enqueues a task on the target node's GPU. The target-side CPU simply waits on a shared-memory signal until the target-side GPU has completed the task.

Both CPUs begin by initializing the RDMA communication layer and NIC ❶. On the initiator side, the CPU allocates a payload and creates a command packet encapsulating the task to execute on the target ❷. In this example, the coordinated index 42 is used to associate this command with queue, kernel, and signal registrations at the target. This task is then supplied to the NIC using the *XtqPut* operation ❸. An *XtqPut* triggers the NIC to send the input data and command packet to the target.

Meanwhile, the target CPU posts the receive buffer using the RDMA communication layer ❹. Next, it initializes the local accelerator runtime and creates a kernel, completion

```
int main(int argc, char *argv[]) {
  // ❶ Initialize RDMA comm layer
  int rank = RdmaInit();
  int index = 42;
  if (rank == INITIATOR) {
    // ❷ Construct XTQ payload and command
    void *payload = malloc(BUFFER_SIZE);
    void *cmd = ConstructCmd(CMD_SIZE, 42);
    // Post initialization sync with target
    ExecutionBarrier();
    // ❸ Launch on remote GPU using XTQ
    XtqPut(TARGET, cmd, CMD_SIZE,
           payload, BUFFER_SIZE);
  } else {
    // ❹ Post receive buffer
    void *recv_buf = malloc(BUFFER_SIZE);
    RdmaPostBuffer(recv_buf);
    // ❺ Initialize HSA CPU Runtime
    signal_t signal;
    kernel_t kernel;
    queue_t queue;
    TaskingInit(&signal, &kernel, &queue);
    // ❻ Register Kernel/Queues
    XtqRegisterKernel(signal, kernel, 42);
    XtqRegisterQueue(queue, 42);
    // Post initialization sync with initiator
    ExecutionBarrier();
    // ❼ Wait for GPU to complete task
    SignalWait(signal);
  }
}
```

Fig. 6. Pseudocode for *XtqPut* operation.

signal, and user-mode command queue ❺. The target then registers the kernel, signal, and queue with the XTQ NIC using the *XtqRegisterKernel* and *XtqRegisterQueue* functions at index 42 ❻. These functions populate the lookup tables used by the target-side NIC. When the initiator's RDMA operation arrives at the target, the target NIC recognizes it as an XTQ-enabled RDMA and uses the Kernel Lookup Table and Queue Lookup Table to replace components of the packet and select a command queue. After the RDMA operation is complete, the NIC enqueues the packet to the GPU, which begins execution of the kernel. Meanwhile, the target CPU waits for the operation to complete using a shared-memory signal ❼.

After initialization, an execution barrier is entered between steps ❷ and ❸ on the initiator and steps ❻ and ❼ on the target. This barrier ensures that the initiator does not send a command to the target before the target's lookup tables have been populated.

In addition to illustrating the XTQ API, this example program drives home two related contributions of the XTQ framework. First, the NIC delivers the task descriptor directly to the GPU without host CPU involvement, minimizing the task launch latency for the GPU. Second, the host CPU does not have to use any cycles servicing the network request and scheduling the task on the GPU. In this simple example, the target CPU waits for the GPU to complete its operation, but in more complex workloads the CPU could be free to perform meaningful computations.

While this example may seem primitive, it is actually possi-

TABLE I
XTQ SIMULATION CONFIGURATION.

CPU and Memory Configuration	
Type	4 Wide OoO
Clock	3Ghz, 8 cores
I, D-Cache	64K, 2-way, 2 cycles
L2-Cache	2MB, 8-way, 4 cycles
L3-Cache	16MB, 16-way, 20 cycles
DRAM	DDR3, 8 Channels, 800MHz
GPU Configuration	
Clock	1 GHz, 24 Compute Units
Wavefronts	40 (each 64 lanes)/ oldest job first
D-Cache	16kB, 64B line, 16-way, 4 cycles
I-Cache	32kB, 64B line, 8-way, 4 cycles
L2-Cache	768kB, 64B line, 16-way, 24 cycles
Network Configuration	
Switch Latency	100ns
Link Bandwidth	100Gbps
Topology	Star

ble to support many features with a small amount of software support from higher-level runtime libraries. For example, the target can send the data to another node for further computation by issuing another *XtqPut* operation after the shared-memory signal resolves. Complex, cross-node dependencies and task graphs can be implemented using shared-memory signals and XTQ.

V. EVALUATION

XTQ provides new opportunities to re-evaluate design decisions in existing applications and to redesign communication in emerging applications. In this section, we evaluate the XTQ tasking model on microbenchmarks designed to expose latencies, primitives that are intrinsic to many MPI programs, and the CNTK deep learning framework. We illustrate that XTQ offers significant improvements over standard remote GPU dispatch.

A. Experimental Setup

To evaluate XTQ, we employ a simulation framework based on the open-source gem5 simulator [31] including the AMD public GPU compute model [32]. We have configured our hardware to resemble a potential future server-class APU system, with each APU containing a NIC, GPU, and CPU cores. None of the devices share a cache, and all memory accesses are coherent and occur in a unified virtual address space through IOMMU technology. This system does not suffer from the off-chip bandwidth issues [33] that plague current integrated CPU/GPU solutions, since we expect future high-performance chips to contain a larger number of memory channels [34] or integrated die-stacked memory. Our infrastructure simulates multi-node configurations with a simple switch and wire delay model incorporating bandwidth and latencies configured similarly to high-performance fabrics such as Intel Omni-Path [16]. The NIC model implements the Portals 4 [17] network programming specification with custom XTQ extensions implemented on top of *Put* operations. The GPU tasking interface is implemented using HSA task descriptors and queues. Table I shows the specific configuration for the major components of our infrastructure.

Our simulation environment models forward-looking GPU launch latencies. Once the CP has been notified of an available command packet using its doorbell, it performs a read of the command packet and then immediately schedules the kernel when a compute unit becomes available. We believe that this limit represents the conceptual overhead of launching a kernel, and that real-world launch overheads will trend towards this limit as tightly coupled frameworks continue to integrate GPUs more closely with CPUs.

In our experiments, we compare three different remote tasking interfaces that we will refer to as CPU, HSA, and XTQ. These configurations are defined as follows:

- **CPU:** Remote tasking is accomplished by using two-sided send/receive pairs through user-space RDMA. These results use the application thread for message progress and active message execution unless otherwise indicated. The CPU configuration represents a non-GPU-accelerated system, and is included to separate the baseline benefits of GPU acceleration from those of XTQ.
- **HSA:** HSA also uses two-sided send/receive pairs over RDMA, but launches the active message on the GPU after the CPU thread has received the data. The CPU communicates to the GPU through user-mode command queues. The HSA configuration represents user-space tasking on a tightly-coupled architecture without XTQ.
- **XTQ:** XTQ uses one-sided *XtqPut*s through user-space RDMA to remotely enqueue active messages on the target's GPU. The NIC places AQL packets directly into the GPU's command queue for execution.

B. Latency Analysis

To precisely quantify the benefits of XTQ in our model, we have coded a microbenchmark very similar to the API code example presented in Figure 6. Figure 7 shows a time breakdown of XTQ and HSA remote task spawn for a small 64B payload and a 4KB payload. The small 64B payload over XTQ takes approximately 1.2μs to complete a remote task, while the same payload over HSA takes approximately 1.5μs. In XTQ, the data transfer phases from initiator to target take slightly longer than the HSA transfers. XTQ's usage of 64B HSA-like packet format slightly increases the payload size over an optimized Portals 4 implementation. However, the performance penalty incurred from transferring the command structure is dwarfed by the benefits in task launch latency. XTQ saves approximately 300ns over HSA during the task launch phase, since the NIC can directly schedule work on the GPU's command queues, while HSA requires the CPU to process an RDMA event, create the GPU task descriptor, and place it in the GPU's command queue. For the 4KB payload, we see similar savings during task enqueue, although the speedup is proportionally less due to the increase in data transfer and kernel execution time.

C. MPI Integration

The Message Passing Interface (MPI) [28] is the *de facto* communication library for distributed-memory HPC applica-

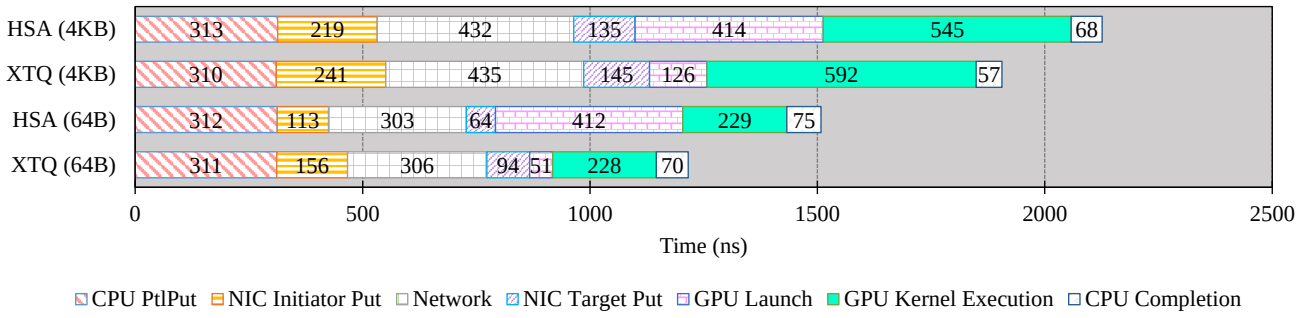
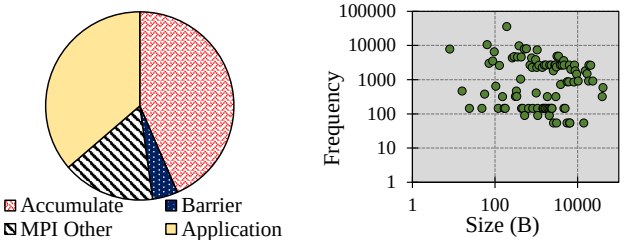


Fig. 7. Time breakdown of remote GPU kernel launch.



(a) Time distribution. (b) Accumulate data size distribution.

Fig. 8. NWChem *tce_ozone* accumulate statistics.

tions. Several MPI functions, such as one-sided accumulate operations and reduction collectives, have a strong computational component that can be parallelized. Using XTQ to offload this computation to the GPU can lead to improved performance and reduced CPU overheads. By freeing the CPU to do independent work, XTQ enables non-blocking variants to achieve substantial computation overlap between the primary application thread and the accelerator.

We extended the one-sided and collective frameworks of the Open MPI [35] implementation of the MPI-3.0 specification to incorporate XTQ-based accumulates and reductions. Our implementation enables library users to reap the benefits of XTQ acceleration without altering any code.

1) MPI One-Sided Accumulates: The *MPI_Accumulate* function is a one-sided operation used to combine initiator-resident data with target-resident data through a specified operation. The target data is replaced with the result without any explicit participation of the target process.

Accumulate operations are preceded by a collective window creation operation during which the target process makes a portion of its memory space available to other members in the window for direct updates. Subsequent accumulate calls to the target are bracketed by synchronization operations which define access epochs. An initiator process can make multiple accumulate calls to a target within an epoch. However, the accumulate operations are considered complete only after the synchronization operation that closes the epoch.

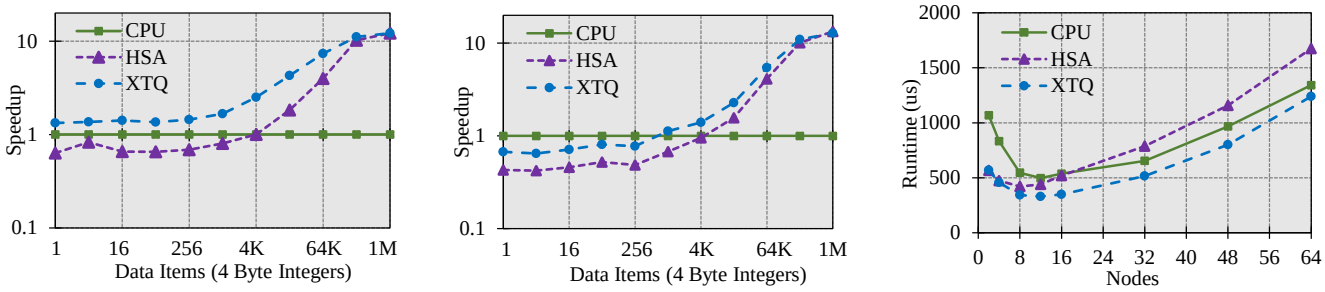
The baseline CPU accumulate implementation for Portals 4 in OpenMPI is not a true one-sided communication model: it is layered over two-sided, send-receive calls. Our HSA-based implementation is similar, except for the fact that the MPI library at the target enqueues the operation on a local GPU instead of performing it directly.

XTQ-based accumulates, however, are completely one-sided, relieving the target-side MPI process from receiving data and executing the operation. On receipt of data from the initiator, the target-side NIC enqueues the appropriate command directly on the GPU's command queue. When execution completes, the GPU updates the target-side accumulate buffer with the result and an internal progress thread sends an acknowledgement to the initiator.

MPI accumulate operations are used extensively in the NWChem [36] computational chemistry package through the ARMCI/MPI3 library [37]. As an example, the *tce_ozone* workload from the NWChem regression tests spends over 58% of its time in the MPI library on a 4 node cluster. Figure 8a shows the percentage of time spent performing various MPI functions on a single rank. The chart shows that the 43% of its total execution time performing accumulate-related operations. Figure 8b shows a histogram of the payload size and number of accumulates that occur during *tce_ozone*. We see a large concentration of small-to-medium sized accumulates, which are ideal for XTQ lightweight messaging acceleration. Unfortunately, we are unable to simulate this application directly because it requires a version of MPI not supported by our simulation infrastructure. However, our expectation from the profiling data is that NWChem will realize measurable benefits from XTQ-based GPU acceleration.

2) MPI Reduce and Allreduce Collectives: *MPI_Reduce* and *MPI_Allreduce* are collective operation that use a binary operation (e.g., SUM, PROD, or MAX) to combine the corresponding elements in the input buffer of each participating process [28]. For reduce, the combined results are stored in a result buffer in the root process's address space. For allreduce, the combined results are stored in the result buffers of all participating processes.

Our implementation of both reductions using XTQ is built on the LibNBC library [38, 39]. LibNBC was designed to support non-blocking collectives on generic architectures. In doing so, it creates a schedule: a directive to execute a set of operations. We modify the schedule to layer reductions on $XtqPut$s instead of two-sided, send-receive operations. An inherent problem with implementing reductions with active messages is that the target-side resources must be allocated before an active message can be received at the target. To ad-



(a) XTQ accumulate acceleration on 2 nodes. (b) XTQ reduce acceleration on 2 nodes. (c) XTQ 4MB allreduce acceleration.

Fig. 9. Acceleration of MPI accumulate, reduce, and allreduce operations over XTQ.

dress this problem, XTQ encodes an explicit synchronization step into the schedule by issuing zero-length send/receive pairs between initiators and targets. This step guarantees that the target resources are available before any $XtqPut$s are issued.

3) MPI Benchmarks: Figure 9 shows performance results for accumulate, reduce, and allreduce implemented on CPU, HSA, and XTQ as defined in Section V-A. For sufficiently large benchmarks, GPUs perform much better than CPUs for these data-parallel operations. For a two-node accumulate operation (Figure 9a), XTQ performs significantly better than HSA. Interestingly, XTQ also performs better than CPU even for very small accumulate sizes. We believe this result occurs because CPU implements one-sided accumulates using two-sided send/recvs, while XTQ leverages one-sided puts directly through Portals 4. XTQ offers a reasonable performance improvement of around 10-15% for a two-node reduction (Figure 9b) over a standard HSA-enabled GPU. For both operations, the benefits of XTQ over HSA decrease as the payload increases over approximately 64KB. All of these algorithms, however, are amenable to software pipelining, which will push the payload size back into a range where XTQ shows significant benefits, even for very large transfers.

Figure 9c illustrates how XTQ performs on allreduce when strong scaling up to 64 nodes for a fixed-size global data set of 4MB. Unlike accumulates and reductions, allreduce requires the result to be transmitted to all nodes participating in the collective operation. We implement allreduce as a reduce-scatter followed by an allgather, which is an efficient implementation for vector allreduce operations [40]. With a fixed-size data set, increasing the number of nodes decreases the computation per node while increasing the total number of messages required to complete the reduction. The inflection point at approximately 12 nodes indicates the point where the overhead of sending more messages outweighs the benefits of less computation.

The figure illustrates that for small node counts, the size of each round's messages are large enough to benefit from GPU acceleration with or without XTQ. However, for larger node counts, non-XTQ-enabled GPUs are unable to amortize the high launch latency over the execution of smaller messages. Only XTQ-enabled GPUs are able to maintain performance improvements over a CPU reduction up to 64 nodes.

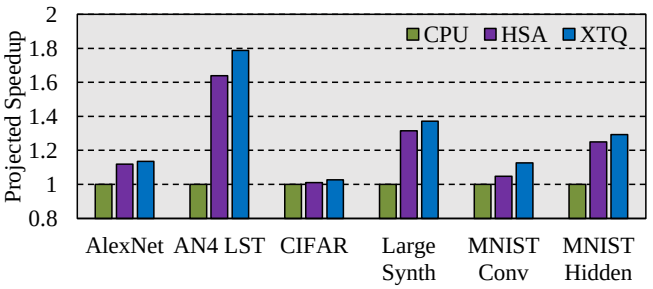


Fig. 10. XTQ performance on CNTK workloads across 8 GPU-enabled nodes.

D. Machine Learning Results

In this section, we evaluate XTQ on a distributed deep-learning framework. Our vehicle of exploration for this case study is the Computational Network Toolkit (CNTK) [23]. CNTK provides a general programming framework that allows researchers to express arbitrarily complex deep-learning computations and deploy them across multiple nodes.

Training deep-learning networks is a very compute- and network-intensive calculation that is difficult to parallelize efficiently. For our experiments, we configure CNTK to perform data-parallel training, where the model is replicated across all nodes. We select six sample applications from the CNTK distribution that represent a broad range of data sizes and communication patterns. Each node computes gradients using Stochastic Gradient Descent (SGD) for a subset of the training data and aggregates the gradients across the cluster using an allreduce communication pattern. This cycle of computation using SGD and global gradient allreduce is repeated until the model has been trained to a user-defined metric. The frequency with which gradients are aggregated between nodes is a tradeoff between convergence time and time to complete a round of training.

For our analysis, we collected profiling data on the Stampede [3] supercomputer, where GPUs were used for local SGD, InfiniBand™ NICs were used for communication, and the CPUs were used for the computation in the allreduce phase. By combining these real-world runs with detailed simulation of allreduce operations obtained from gem5, we are able to project the performance improvement of accelerating the allreduce function with XTQ.

To perform this projection, we profile our Stampede runs

to extract the data size and duration of allreduce calls from CNTK applications. We simulate allreduce calls of the same sizes in gem5 to estimate the performance improvement attainable by HSA and XTQ. We then project the communications performance improvement into the time taken to perform the allreduce in real hardware without XTQ. Since CNTK uses blocking allreduce calls, we do not need to worry about any communication and computation overlap.

Figure 10 shows the results for CNTK across 8 high performance compute nodes. Utilizing GPUs for allreduce compute provides a 23% average improvement in runtime over a baseline CPU version. XTQ-enabled acceleration gives on average an additional 8% improvement over the HSA baseline, with AN4 LST improving by 15%. CIFAR does not significantly benefit from either form of GPU allreduce acceleration, since it is more bound by the local SGD compute phase than the gradient reduction.

VI. Related Work

The seminal Active Messages work [20] embeds computation in network messages by directly invoking a user message handler on the target. Willcock et al. [41] developed AM++, which extends AM with type-safe object-oriented and generic programming, enabling optimizations such as message coalescing. More recently, Besta and Hoefler [42] modify the IOMMU to invoke active-message-like handlers as a side effect of RDMA put and get operations. Our work uses HSA's architected task queues to extend active messages to non-CPU accelerators. The use of explicit user-allocated task queues rather than direct handler invocation also provides a more flexible buffering model than the original AM.

Several machines proposed in the '90s coupled light-weight tasks with explicit message passing, including the J-Machine [43], M-Machine [44], Star-T Voyager [45]; or some combination of messaging passing and shared memory such as MIT Alewife [46], Typhoon [47], and FLASH [48]. Our work applies some of these concepts to emerging commodity hardware with tightly coupled accelerators and RDMA NICs.

XTQ's rewrite mechanism is similar to the Cray T3E's global address translation scheme [49], in that global references are translated to local addresses at the target to decouple local resource management from global resource identifiers. The T3E also supported remote enqueuing to memory-based user-level message queues, similar to how XTQ enqueues to HSA task queues. The key differences are first, that HSA queue entries are architecturally defined task descriptors, so XTQ can provide additional semantics (such as argument rewriting) that enable direct enqueuing of accelerator tasks without target-side CPU intervention; and second, the HSA doorbell mechanism provides a notification mechanism other than the T3E's options of interrupts and polling. Note that XTQ is independent of the global addressing mechanism used for conventional RDMA, which in our example system's case is provided by Portals 4.

XTQ shares some similarities with the scale-out NUMA inspired designs which optimize RDMA for NUMA memory fabrics [26] and NIC layouts for tiled, scale-out architectures [50]. These works are primarily concerned with efficient data movement, while ours is concerned with efficient task invocation for heterogeneous compute devices.

Oden et al. propose Global GPU Address Spaces (GGAS) [51] and evaluate its performance on GPU-accelerated reductions [52]. While XTQ also exploits GPUs for reductions, there are a number of significant differences between GGAS and XTQ. GGAS focuses on data transfer, creating a hardware-supported global address space that spans all GPU device memory and supports direct remote memory accesses from the GPUs. XTQ builds on a more traditional RDMA model with explicit put and get operations, but adds remote task invocation to the data-transfer model.

A number of commercial and open source offerings provide the ability to utilize GPUs in a distributed environment. NVIDIA's GPUDirect [21] enables direct high-performance RDMA between InfiniBand™ NICs and the local memory of discrete GPUs. However, task launch on the discrete GPU still must be initiated by the CPU using the CUDA runtime and GPU kernel driver. Our work focuses on HSA-like systems with unified physical memory and tightly coupled accelerators. As such, we are able to completely bypass the CPU for both data transfer and kernel launch. Another platform for remote GPU task invocation is rCUDA [53], which allows CPUs without locally attached GPUs to take advantage of GPU resources on another node. However, rCUDA is designed primarily to virtualize a cluster's GPU resources for coarse-grain task offload, while XTQ is intended to enable fine-grain interaction of low-latency communication and accelerator-based computation. Finally, GPUNet [54] allows GPUs to source and sink network messages directly through a sockets interface with coordination from a CPU runtime library.

Some NICs provide built-in offload capabilities for collective operations such as reductions, including Quadrics programmable engines [55], Cray's Aries collective engine [56], and Mellanox's CORE-Direct technologies [57]. XTQ provides a more general solution by making a broader range of existing accelerated compute engines easily accessible to the NIC. However, for small computations, NIC-based offload will further reduce overheads by avoiding all inter-device interactions. A NIC with built-in offload functionality could complement other accelerators in an XTQ/HSA environment.

VII. Conclusion

Emerging node-level architectures tightly couple accelerators into host platforms. These frameworks significantly reduce task launch latency via user-level task queues and eliminate data copy overhead via shared virtual memory, paving the path for fine-grained, heterogeneous tasking models in shared-memory environments. Concurrently, RDMA enables highly efficient user-level network data transfers. This paper proposes Extended Task Queuing (XTQ), a mechanism that combines tightly coupled, user-level task queuing with RDMA to provide heterogeneous, lightweight tasking across distributed-memory systems. XTQ is implemented as an extension to a tightly

integrated, RDMA-capable NIC and enables applications to schedule tasks on accelerators and CPUs across nodes. Using XTQ, applications can send messages to remote accelerators, bypassing the operating system on both nodes and not involving the target CPU. Bypassing the target-side CPU reduces task launch latency by 10-15% for small-to-medium sized messages, and frees a CPU thread from message processing to perform more useful computation.

Because XTQ lies at the intersection of several emerging paradigms, such as accelerator-based programming and lightweight distributed tasking, few existing applications directly leverage its benefits. We believe that XTQ opens up new possibilities for applications to leverage accelerators for tightly coupled communication and computation in distributed systems. Towards this end, we demonstrate that XTQ enables the use of GPUs to accelerate MPI reduction, allreduce, and accumulate operations across a variety of payload sizes and on clusters up to 64 nodes. Finally, we illustrate how XTQ can provide up to 15% performance improvement for emerging deep learning workloads in the Computational Network Toolkit. We are excited to see how existing applications will adapt to the unique benefits of XTQ, and how new applications can be designed to leverage these features.

ACKNOWLEDGMENTS

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. ARM is the registered trademark of ARM Limited in the EU and other countries. HyperTransport is a licensed trademark of HyperTransport Technology Consortium. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Results were obtained in part using resources from the Texas Advanced Computing Center. This research was supported by the US Department of Energy under the DesignForward and DesignForward 2 programs, and by National Science Foundation grant CCF-1337393. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the DoE or NSF.

REFERENCES

[1] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Int. Symp. on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Int. Symp. on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[3] TACC, "Stampede supercomputer user guide," https://portal.tacc.utexas.edu/user-guides/stampede, 2015.

[4] TOP500.org, "Highlights - November 2015," http://www.top500.org/lists/2015/11/highlights, 2015.

[5] Intel, "Intel Xeon Phi product family," www.intel.com/XeonPhi, 2015.

[6] Amazon, "Amazon EC2 cloud computing," https://aws.amazon.com/ec2/, 2015.

[7] Nvidia, "CUDA toolkit 6.0," https://developer.nvidia.com/cuda-toolkit-60, 2014.

[8] HSA Foundation, "HSA platform system architecture specification 1.0," http://www.hsafoundation.com/standards/, 2015.

[9] Intel, "The compute architecture of Intel processor graphics gen9," https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf, 2015.

[10] B. Wile, "Coherent accelerator processor interface (CAPI) for POWER8 systems," http://www-304.ibm.com/webapp/set2/sas/f/capi/CAPI_POWER8.pdf, IBM, Tech. Rep., 2014.

[11] AMD, "AMD A-Series desktop APUs," http://www.amd.com/en-us/products/processors/desktop/a-series-apu, 2015.

[12] Nvidia. (2012) Hyperq sample. http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[13] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2013, pp. 354–365.

[14] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Int. Symp. on Computer Architecture (ISCA)*, 2015, pp. 528–540.

[15] InfiniBand Trade Association. (2000) InfiniBand architecture specification: Release 1.0.2. http://www.infinibandta.org/content/pages.php?pg=technology_download.

[16] Intel. (2015) Omni-Path fabric 100 series. https://fabricbuilders.intel.com/.

[17] Sandia National Laboratories, "The Portals 4.0.2 network programming interface," http://www.cs.sandia.gov/Portals/portals402.pdf, 2014.

[18] InfiniBand Trade Association, "RDMA over converged ethernet v2," https://cw.infinibandta.org/document/dl/7781, 2014.

[19] Intel, "Internet wide area RDMA protocol (iWARP)," http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf, 2010.

[20] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Int. Symp. on Computer Architecture (ISCA)*, 1992, pp. 256–266.

[21] Mellanox, "Mellanox GPUDirect RDMA user manual," http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.2.pdf, 2015.

[22] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *Int. Conf. on Cluster Computing (Cluster)*, 2008, pp. 213–222.

[23] A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, T. R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, H. Parthasarathi, B. Peng, M. Radmilac, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, H. Wang, Y. Wang, K. Yao, D. Yu, Y. Zhang, and G. Zweig, "An introduction to computational networks and the Computational Network Toolkit," Microsoft, Technical Report, 2014.

[24] AMD, "AMD I/O virtualization technology (IOMMU) specification," http://support.amd.com/TechDocs/48882_IOMMU.pdf, 2015.

[25] M. Ben-Yehuda, J. Mason, L. Van Doorn, and E. Wahlig, "Utilizing IOMMUs for virtualization in Linux and Xen," in *In proc. of the Linux Symp.*, 2006.

[26] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 3–18.

[27] B. Towles, J. Grossman, B. Greskamp, and D. Shaw, "Unifying on-chip and inter-node switching within the Anton 2 network," in *Int. Symp. on Computer Architecture (ISCA)*, 2014, pp. 1–12.

[28] MPI Forum, "MPI: A message-passing interface standard. ver. 3," www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[29] D. Bonachea, "GASNet specification, v1.1," http://gasnet.lbl.gov/CSD-02-1207.pdf, 2002.

[30] D. Bonachea, P. H. Hargrove, M. Welcome, and K. Yelick, "Porting GASNet to Portals: Partitioned global address space (PGAS) language support for the Cray XT," in *Cray User Group (CUG)*, 2009.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, pp. 1–7, 2011.

[32] AMD. (2015) The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5. http://gem5.org/GPU_Models.

[33] M. Daga, A. Aji, and W.-C. Feng, "On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing," in *Symp. on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011, pp. 141–149.

[34] M. Schulte, M. Ignatowski, G. Loh, B. Beckmann, W. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *Micro, IEEE*, vol. 35, no. 4, pp. 26–36, 2015.

[35] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004, pp. 97–104.

[36] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[37] J. Hammond, "ARMCI-MPI - MPICH wiki," https://wiki.mpich.org/armci-mpi/index.php/Main_Page, 2015.

[38] T. Hoefler and A. Lumsdaine, "Design, implementation, and usage of LibNBC," Open Systems Laboratory, Technical Report, 2006.

[39] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2007, pp. 52:1–52:10.

[40] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[41] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 401–410.

[42] M. Besta and T. Hoefler, "Active Access: A mechanism for high-performance distributed data-centric computations," in *Int. Conf. on Supercomputing (ICS)*, 2015, pp. 155–164.

[43] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine multicomputer," in *Int. Symp. on Microarchitecture (MICRO)*, 1995, pp. 146–156.

[44] M. Noakes, D. Wallach, and W. Dally, "The J-Machine multicomputer: An architectural evaluation," in *Int. Symp. on Computer Architecture (ISCA)*, May 1993, pp. 224–235.

[45] B. S. Ang, D. Chiou, D. L. Rosenband, M. Ehrlich, L. Rudolph, and Arvind, "StarT-Voyager: A flexible platform for exploring scalable SMP issues," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 1998, pp. 1–13.

[46] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife machine: architecture and performance," in *Int. Symp. on Computer Architecture (ISCA)*, 1995, pp. 2–13.

[47] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: user-level shared memory," in *Int. Symp. on Computer Architecture (ISCA)*, 1994, pp. 325–336.

[48] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of message passing and shared memory in the Stanford FLASH multiprocessor," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994, pp. 38–50.

[49] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996, pp. 26–36.

[50] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network interfaces for in-memory rack-scale computing," in *Int. Symp. on Computer Architecture (ISCA)*, 2015, pp. 567–579.

[51] L. Oden and H. Froning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *Int. Conf. on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[52] L. Oden, B. Klenk, and H. Froning, "Energy-efficient collective reduce and allreduce operations on distributed GPUs," in *Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, 2014, pp. 483–492.

[53] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Int. Conf. on High Performance Computing and Simulation (HPCS)*, 2010, pp. 224–231.

[54] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking abstractions for GPU programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 201–216.

[55] D. Roweth and A. Pittman, "Optimised global reduction on QsNetII," in *Symp. on High Performance Interconnects*, 2005, pp. 23–28.

[56] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," http://www.cray.com/sites/default/files/resources/CrayXC30Networking.pdf, 2015.

[57] R. Graham, S. Poole, P. Shamis, G. Bloch, G. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations," in *Int. Conf. on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 53–62.