NETWORK-BASED COMPUTING LABORATORY

Multi-Threaded UPC Runtime for GPU to GPU communication over InfiniBand

Miao Luo, Hao Wang,

& D. K. Panda

Network-Based Computing Laboratory

Department of Computer Science and Engineering

The Ohio State University, USA

OHIO STATE

Outline

- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

Outline

- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

Introduction

- Unified Parallel C (UPC)
  - One of the most popular PGAS programming languages
  - High-productivity and better applicability on hierarchical architectures
  - Irregular parallelism
- Graphics Processing Units (GPUs) Clusters:
  - High peak performance
  - Cost-efficiency
  - OpenCL / CUDA
  - High performance interconnects (i.e. InfiniBand)
- UPC (PGAS) + CUDA (GPU)?

Outline

- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

Motivation

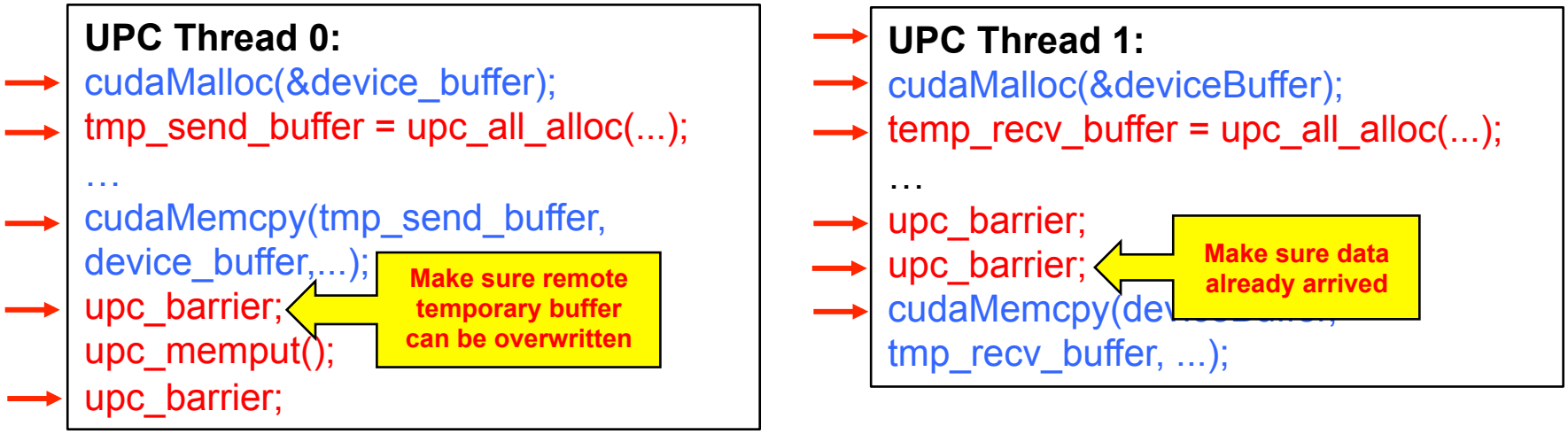
UPC Thread 0:

```
cudaMalloc(&device_buffer);
tmp_send_buffer = upc_all_alloc(...);
…
cudaMemcpy(tmp_send_buffer,
device_buffer,...);
upc_barrier;
upc_memput();
upc_barrier;
```

UPC Thread 1:

```
cudaMalloc(&deviceBuffer);
temp_recv_buffer = upc_all_alloc(...);
…
upc_barrier;
upc_barrier;
cudaMemcpy(deviceBuffer,
tmp_recv_buffer, ...);
```

Motivation



- Complicated CUDA functions & temporary host buffer
- Explicit synchronizations
- Involvement of remote UPC thread: poor latency when remote is busy …

Motivation

UPC Thread 0:

```
cudaMalloc(&device_buffer);
tmp_send_buffer = upc_all_alloc(...);
…
cudaMemcpy(tmp_send_buffer,
device_buffer,...);
upc_barrier;
upc_memput();
upc_barrier;
```

UPC Thread 1:

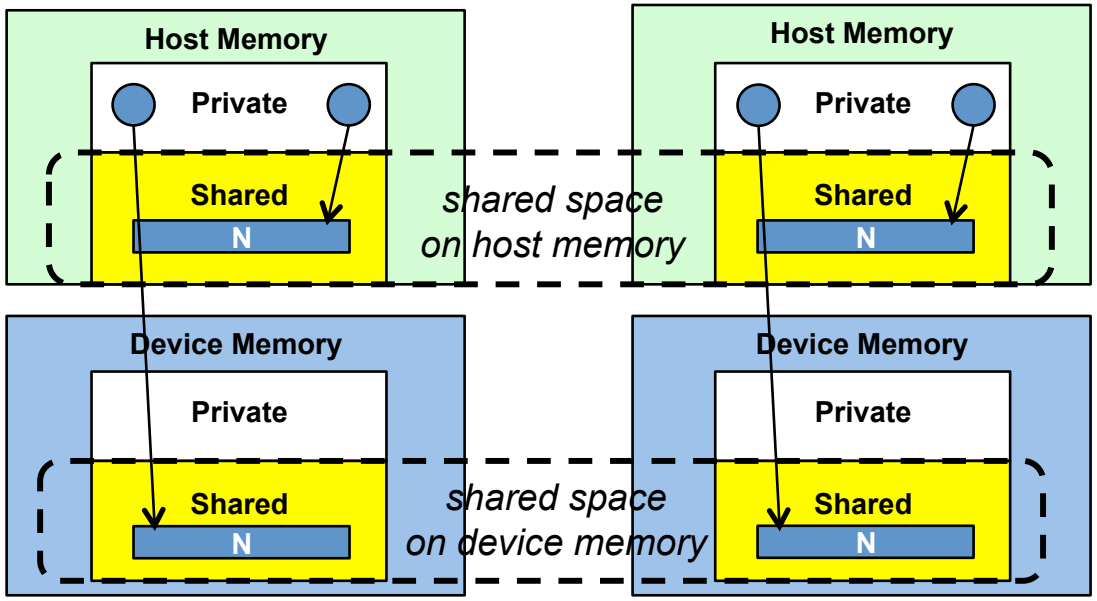
```
cudaMalloc(&deviceBuffer);
temp_recv_buffer = upc_all_alloc(...);
…
upc_barrier;
upc_barrier;
cudaMemcpy(deviceBuffer,
tmp_recv_buffer, ...);
```

- Can both device and host memory be part of shared space and be accessed by the same UPC thread at the same time through UPC standard APIs?
- How to provide efficient GPU to GPU communication based on RDMA features?
- How to ensure low-latency non-uniform data access while the destination is busy?

Outline

- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

GPU Global Address Space with Host and Device Memory




```
..
upc_on_device();
/* allocated on device shared segment  */
upc_all_alloc(THREADS, N*sizeof(int));
…

upc_off_device();
/* allocated on host shared segment */
upc_all_alloc(THREADS, N*sizeof(int));
…
```

- Extended APIs:
  - upc_on_device/upc_off_device
- Return true device memory through Unified Virtual Addressing (UVA)

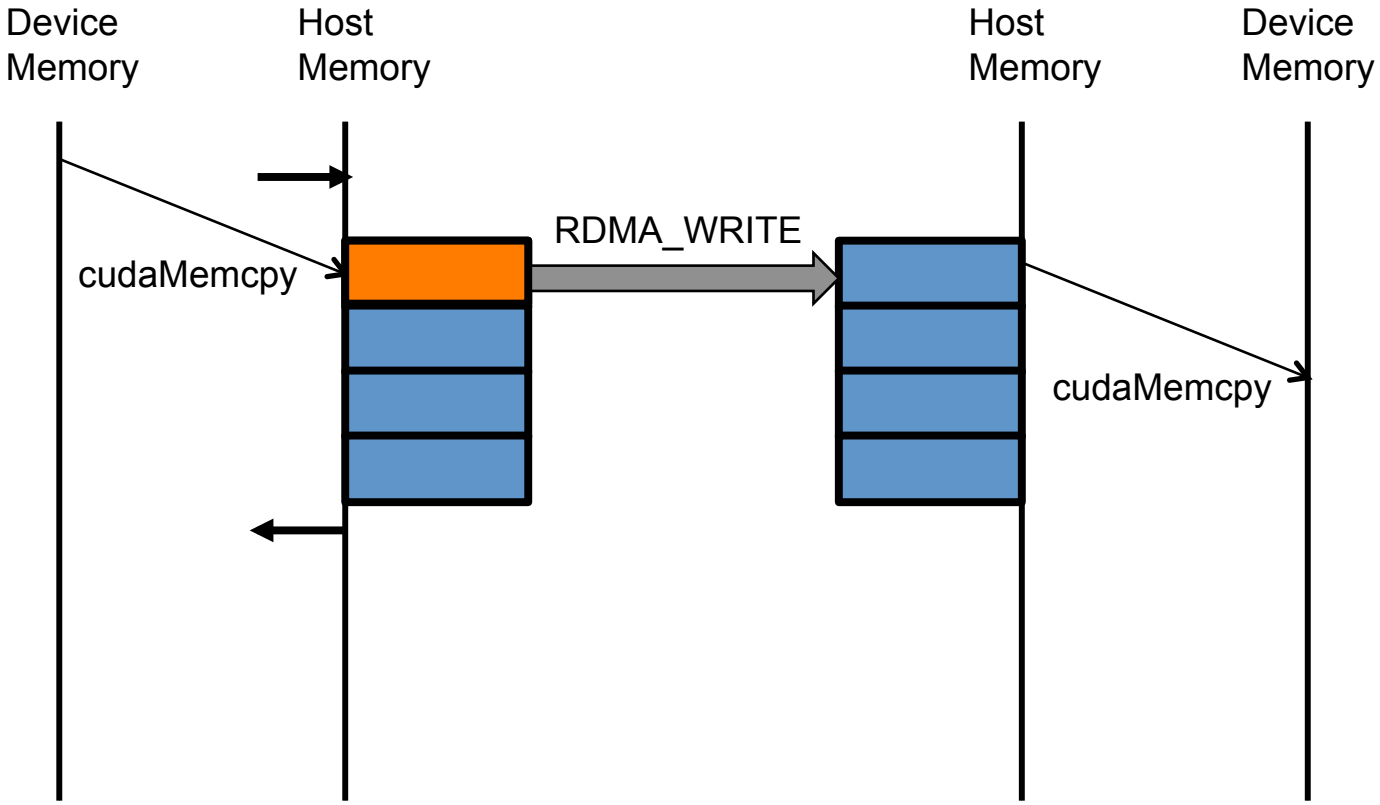
Design for Remote Memory Operation

- After device memory becomes part of the global shared space:
  - Accessible through standard UPC APIs
  - Data movement and communication over network both hidden inside runtime

- Goal: same or better performance compared to existing UPC/CUDA device to device memory access operations

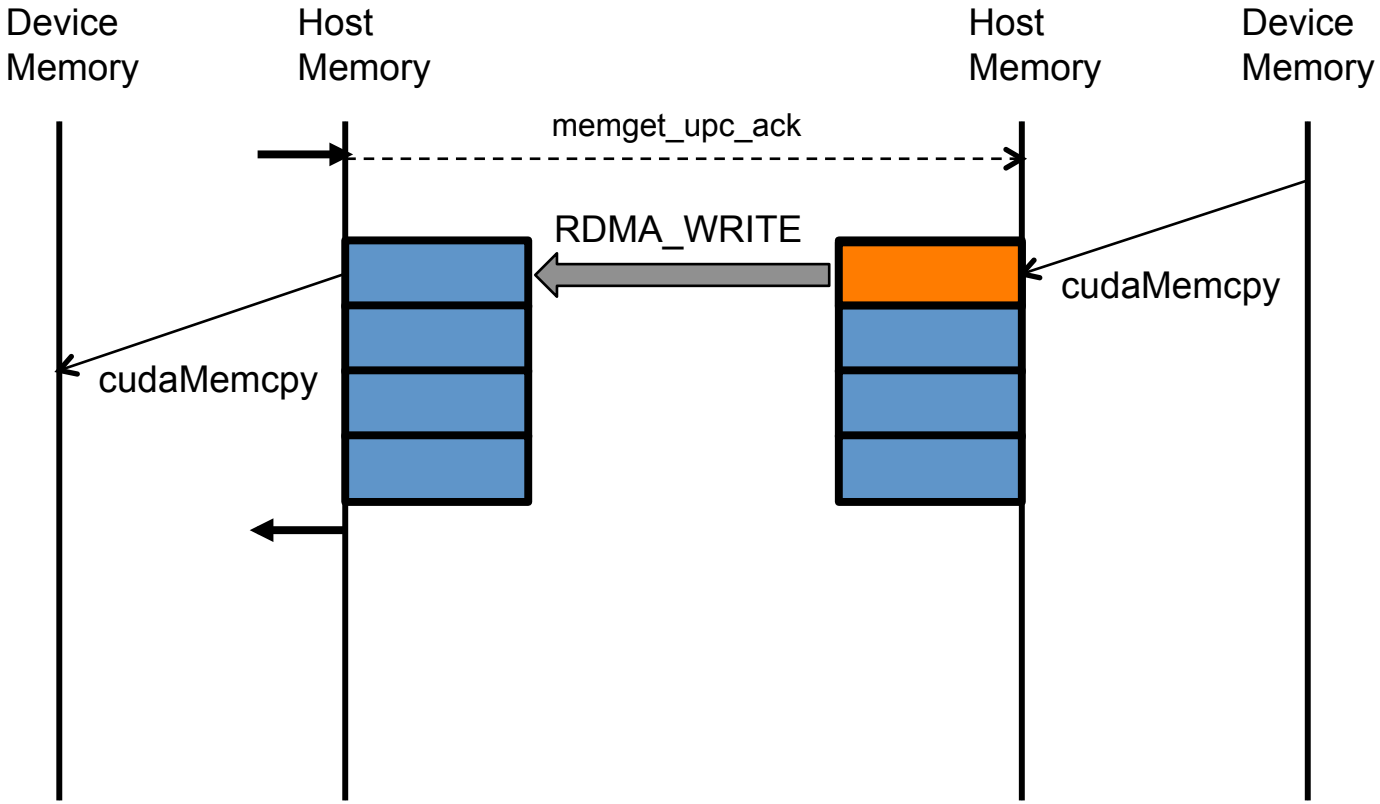
Design for Remote Memory Operation

upc_memput for small and medium message through RDMA Fastpath design



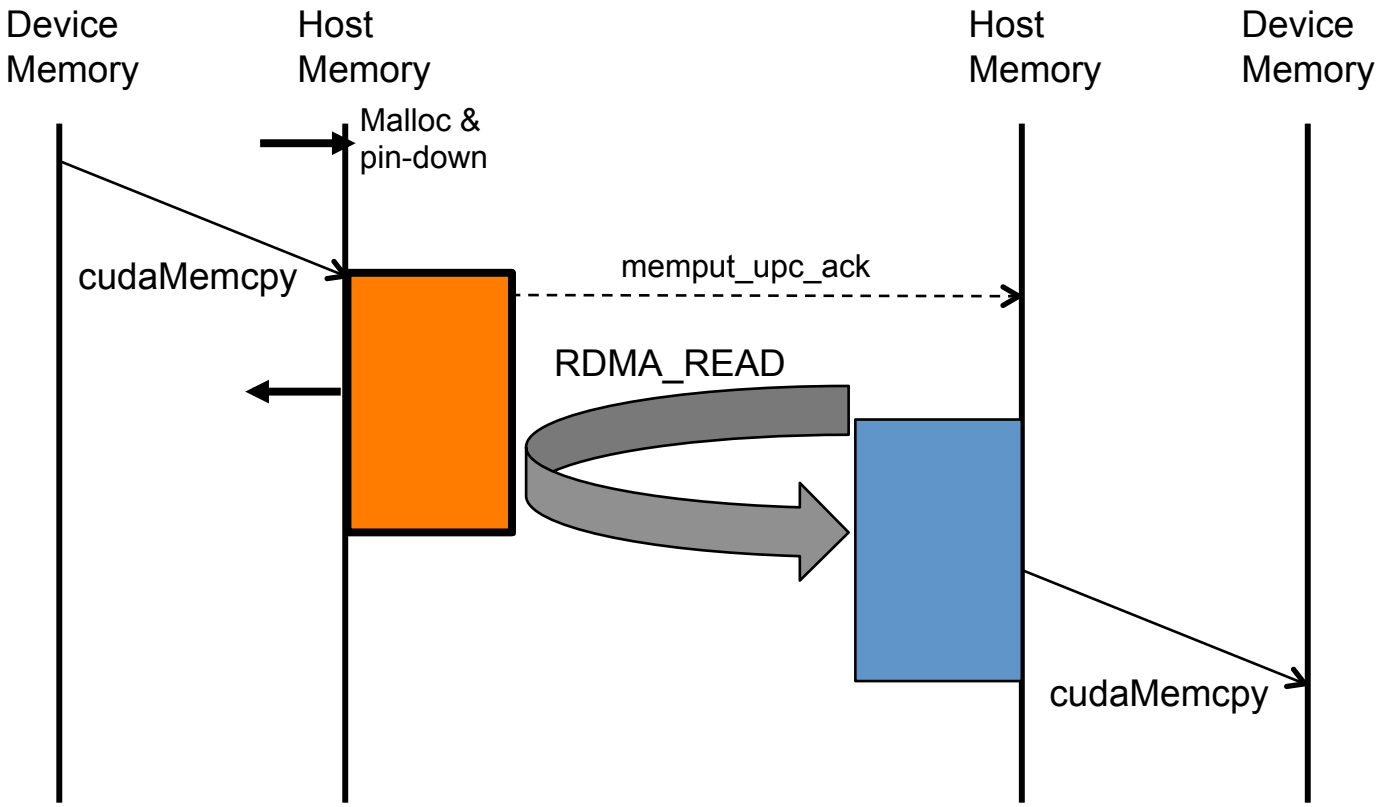
Design for Remote Memory Operation

upc_memget for small and medium message through RDMA Fastpath design



Design for Remote Memory Operation

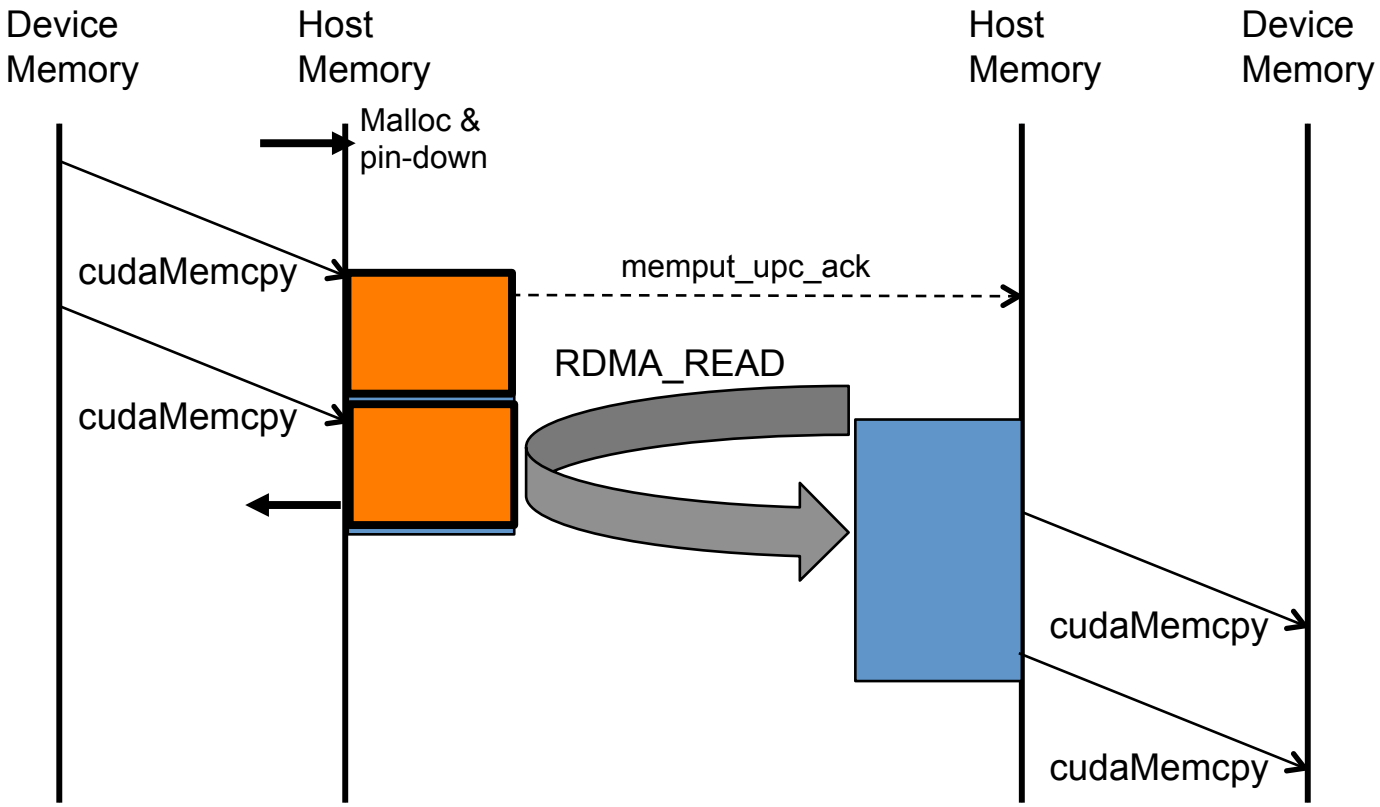
upc_memput for large message



Registration Cache
Free-delayed buffer

Design for Remote Memory Operation

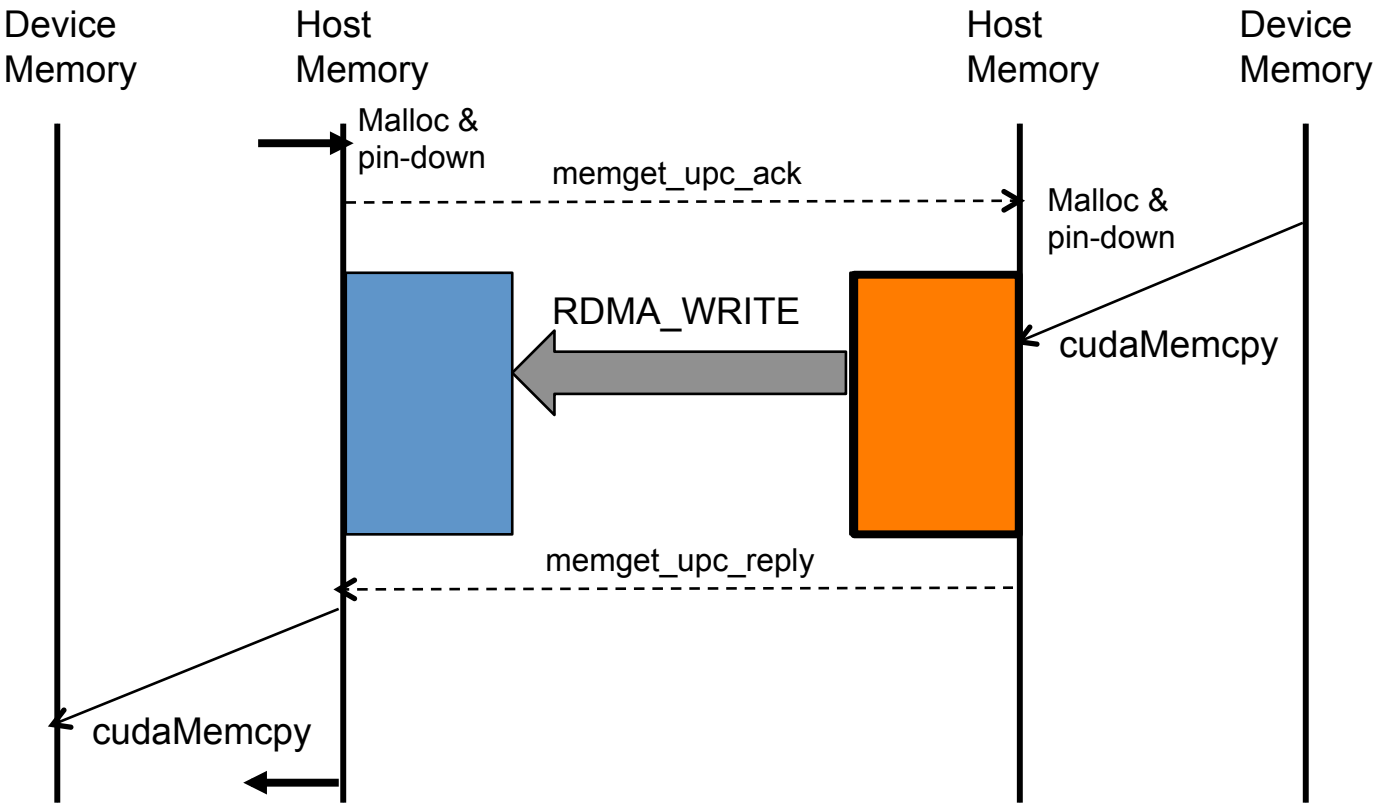
upc_memput for large message



Overlapping of data movement and network communication

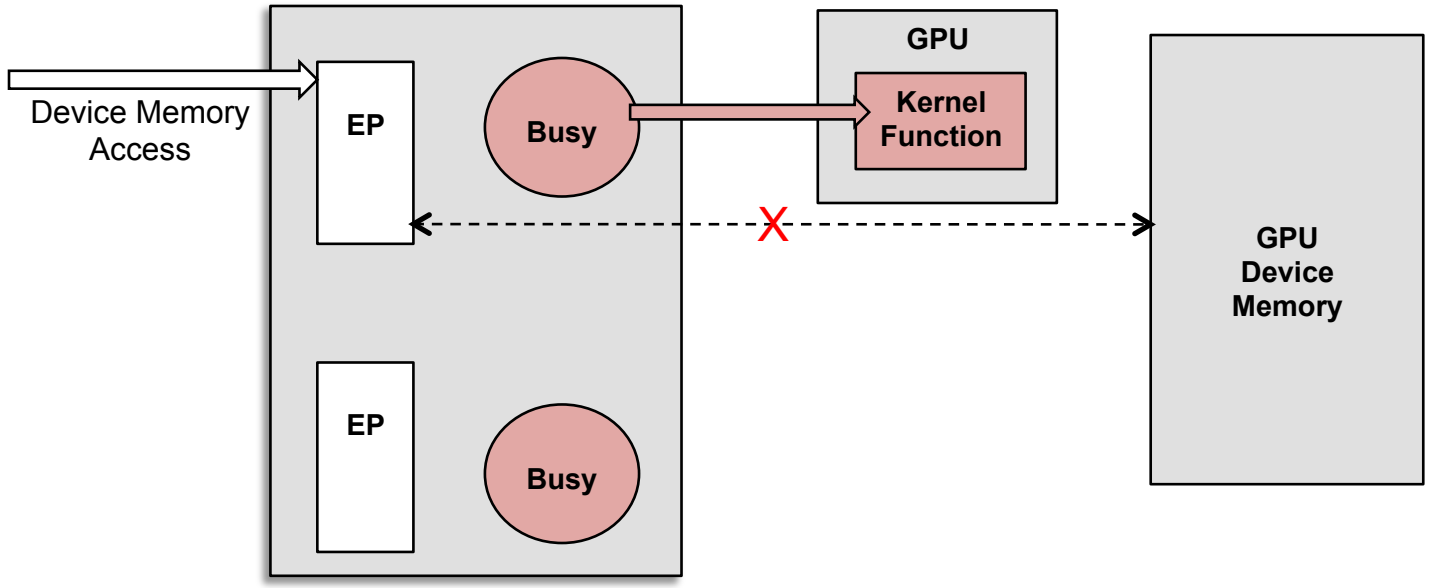
Design for Remote Memory Operation

upc_memget for large message



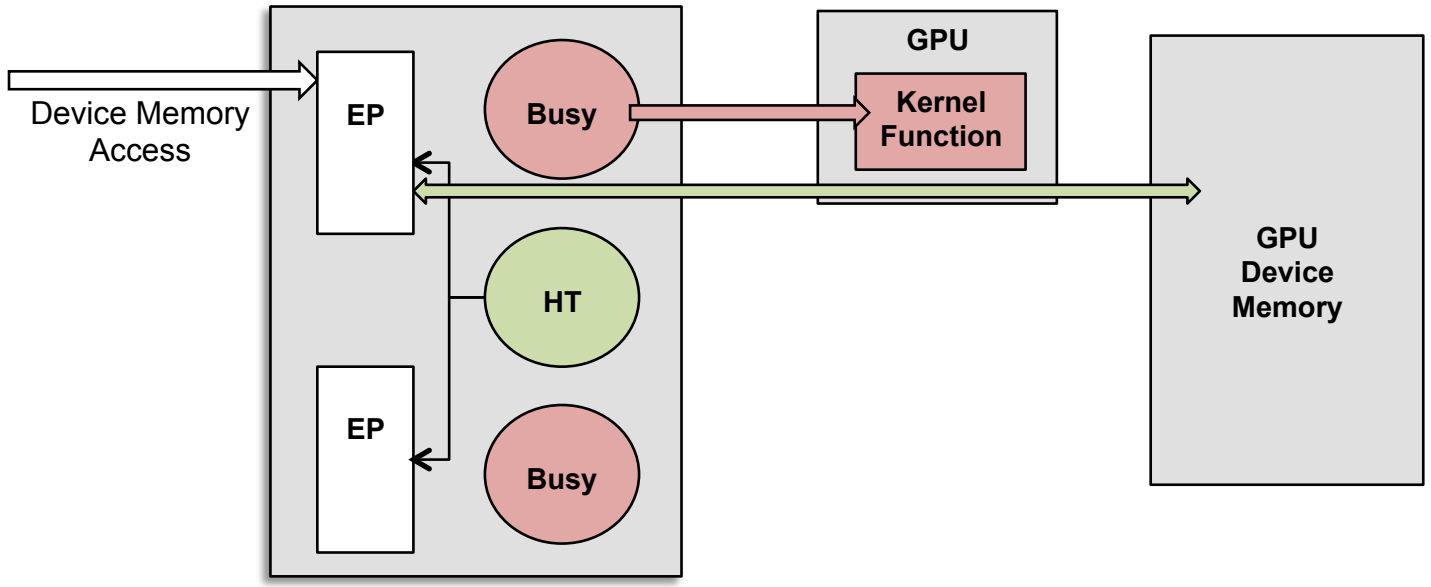
Helper Thread for Improved Asynchronous Access

- Remote UPC threads are busy?
- Helper threads managed by user?



Helper Thread for Improved Asynchronous Access

- True runtime helper thread
  - Poll endpoints of busy UPC threads
  - Helper thread complete memory access
  - Multi-GPUs are supported by multi-endpoints



United Communication Runtime

- Designed and implemented with multi-threaded Unified Communication Runtime (UCR):
  - Support both MPI and PGAS programming models on InfiniBand clusters
  - Based on MVAPICH2 project
- MVAPICH2-X 1.9a release:
  - http://mvapich.cse.ohio-state.edu
  - OpenSHMEM support in current release
  - UPC support in next release

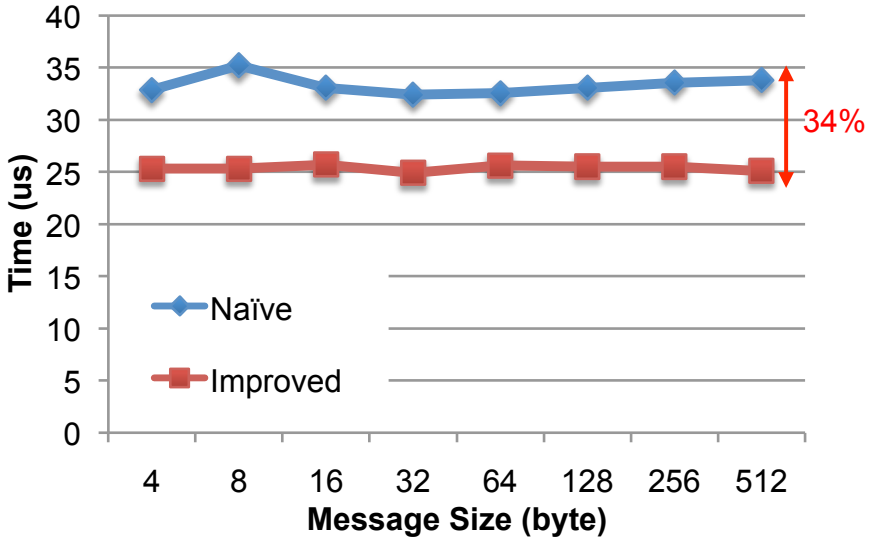
Outline

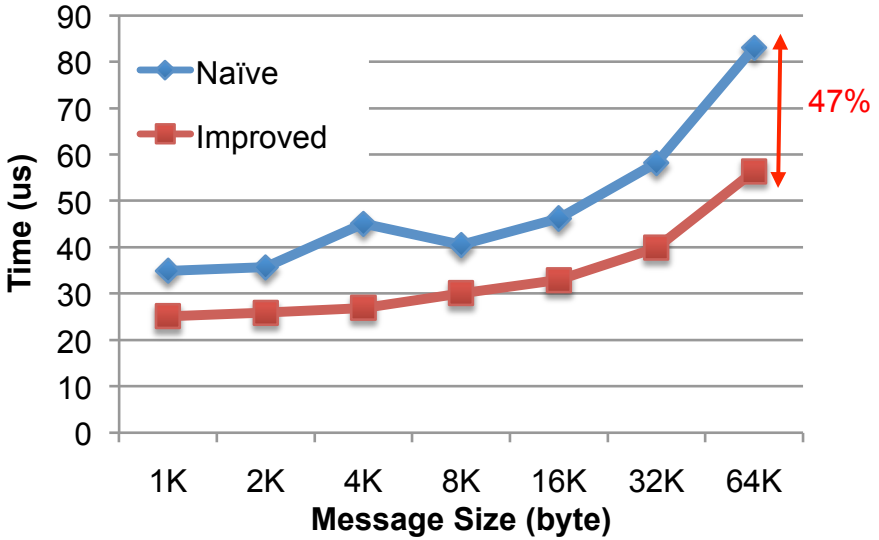
- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

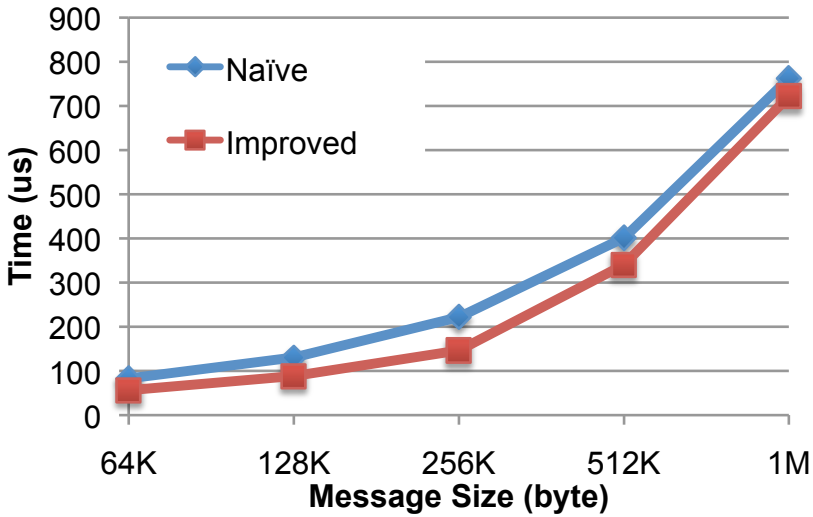
Experimental Platform

- The experiments are carried out on following platform:
  - Four nodes, Each contains two sockets
  - Intel Xeon Quad-core Westmere CPUs operating at 2.53GHz and 12GB of host memory
  - Each node has one Tesla C2050 GPU with 3GB DRAM
  - MT26428 QDR ConnectX HCAs (36Gbps)
  - Red Hat Linux 5.4, OFED 1.5.1, and CUDA Toolkit 4.0
- Comparison to user level UPC/CUDA implementations
  - Naïve: explicity cudaMemcpy and cudaMalloc; temporary host buffers
  - Improved: multi-threaded UCR + our proposed designs

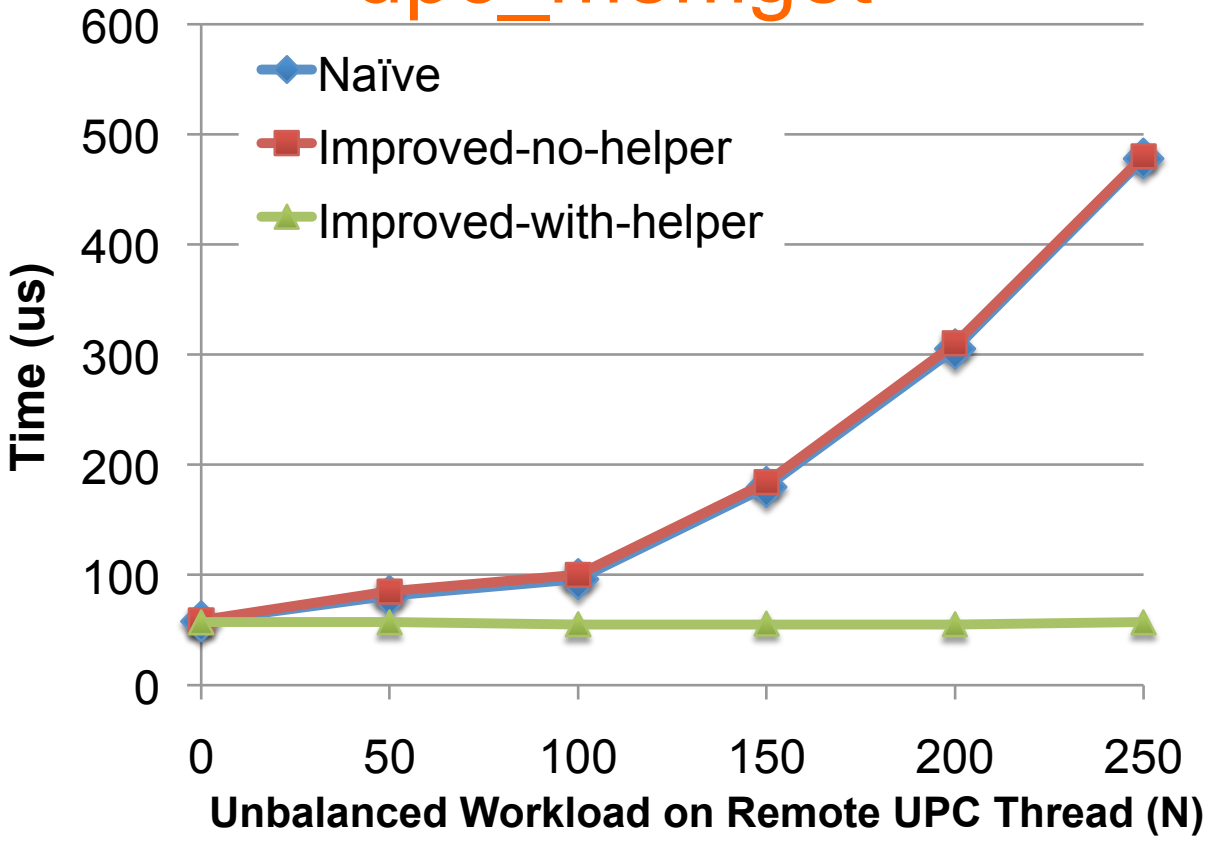
Micro-benchmark Evaluation
upc_memput latency





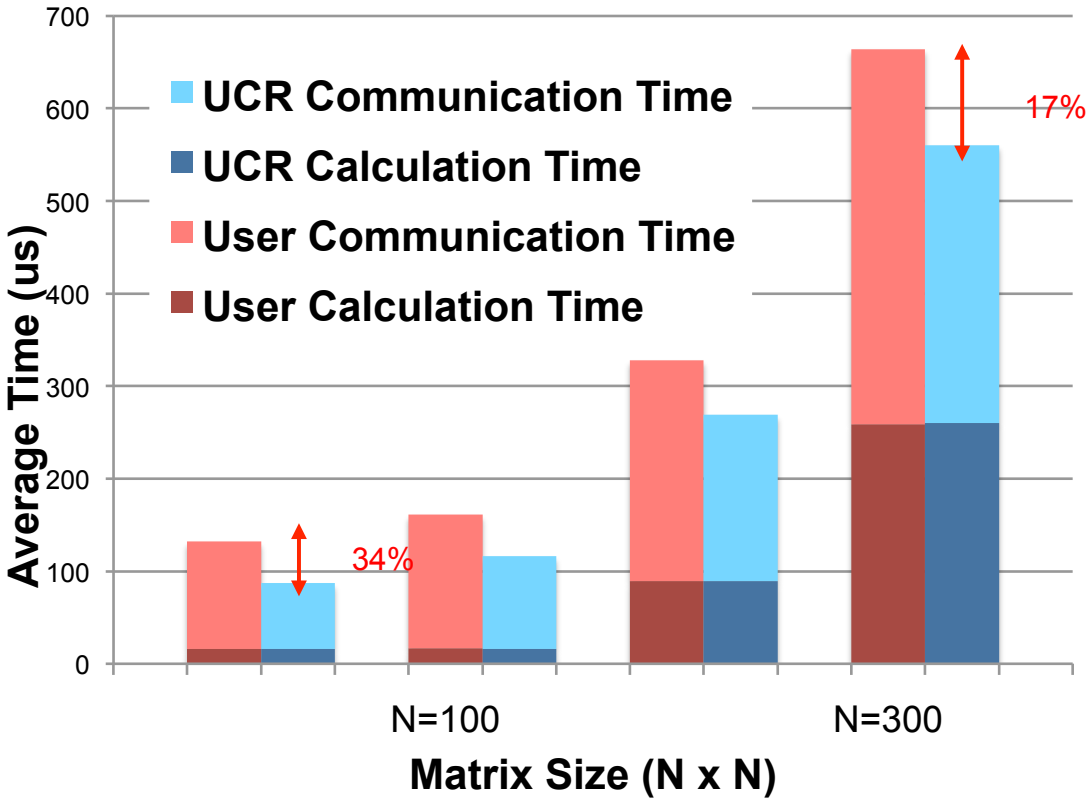


Micro-benchmark Evaluation upc_memget



- The local UPC thread calls upc memget operation to read a piece of 8K byte data on the remote device memory
- Remote UPC is busy with CUDA kernel function doing matrix multiplication
- Kernel function is not calculating on the required data

Sample Application Evaluation



- Matrix Multiplication with 4 GPU nodes
- Communication between root node and other nodes happens before/after computation in every iteration

Outline

- Introduction
- Motivation
- Proposed Designs
- Performance Evaluations
- Conclusion & Future Work

Conclusion

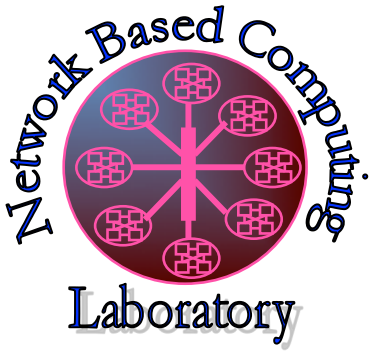
- Identify problems in current UPC/CUDA applications
- A new multi-threaded UPC runtime is proposed:
  - GPU global address space
  - Design for remote memory access
  - Runtime helper thread for improved asynchronous access
- Evaluation through micro-benchmarks and sample benchmark:
  - 47% for upc_memput
  - Helper thread micro-benchmark evaluation
  - 17% ~ 34% improvement for a parallel matrix-multiplication sample benchmark

Future Works

- Adapting UPC/CUDA for irregular applications
- Further study on the helper thread and work-stealing based on multi-threaded UPC runtime at real application level

Thank You!

{luom, wangh, panda}@cse.ohio-state.edu



Network-Based Computing Laboratory

http://nowlab.cse.ohio-state.edu/

MVAPICH Web Page

http://mvapich.cse.ohio-state.edu/

Matrix Multiplication

- C[N][N] = A[N][N] ∗ B[N][N]
- B is divided into 4 (the number of GPUs) matrix B_i[N][N/4] and B_i is associated with UPC thread with thread ID i.
- Kernel function: C_i[N][N/4] = A[N][N] ∗ B_i[N][N/4]
- C_i will be sent to UPC thread 0