



University of Tennessee, Knoxville
**TRACE: Tennessee Research and Creative
Exchange**

[Doctoral Dissertations](#)

[Graduate School](#)


5-2017

Programming Models' Support for Heterogeneous Architecture

Wei Wu

University of Tennessee, Knoxville, wwu12@vols.utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_graddiss

 Part of the [Numerical Analysis and Scientific Computing Commons](#), [Programming Languages and Compilers Commons](#), [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Wu, Wei, "Programming Models' Support for Heterogeneous Architecture. " PhD diss., University of Tennessee, 2017.
https://trace.tennessee.edu/utk_graddiss/4510

This Dissertation is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a dissertation written by Wei Wu entitled "Programming Models' Support for Heterogeneous Architecture." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

James Plank, Michael Berry, Yingkui Li

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Programming Models' Support for Heterogeneous Architecture

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Wei Wu

May 2017

© by Wei Wu, 2017
All Rights Reserved.

To my wife and parents for their wholehearted love

Acknowledgments

For the past six years spent at the University of Tennessee and Innovative Computing Laboratory (ICL), not a single day I did not appreciate how the school, my advisors and my fellow colleagues treated me. They have guided, inspired, and embraced me in so many ways that as a foreigner, I felt like home. For that, I am forever grateful and I will miss them deeply.

The University of Tennessee has a beautiful campus. It has a welcoming, passionate, and diverse environment that gave me not only a phenomenal academic life but also a productive after-school life. People on campus are so nice and easy. I greatly enjoyed my time here on campus and in Knoxville.

Dr. Jack Dongarra, a highly respected and reputable scientist, remain down to earth and approachable. He cares about every student and every staff member. It was a great privilege to join the ICL, to work with and have him as my advisor. His wisdom, his knowledge and his experiences will always be my guide and my example in my future work and life. I couldn't help but feel lucky to be a member of ICL everyday.

My project leader, Dr. George Bosilca, was an outstanding teacher. He didn't only care about the work I do at ICL, but he also cared about things such as what the grades I got on the classes I was taking. When it comes to my internships, he reviewed each position carefully and gave recommendations based on which position would further help me with my academic career. I feel so lucky to be his student. Dr.

Bosilca taught me everything I know. He made me confident and competent to face all kinds of challenges forthcoming in my career.

I am grateful to my committee, Dr. James Plank, Dr. Michael Berry and Dr. Yingkui Li for agreeing to serve on my dissertation committee. I greatly appreciate their time and invaluable guidance on this dissertation.

I would also like to thank my wife. She is loving and caring. She supported me all the way and never complained about the long journey to get my PHD. She encourages me when I have difficulties and she was always by my side. She is my rock and I love her deeply.

To my parents, I would like to thank them for letting me leave home and come to the United States almost 10 years ago in the pursuit of my dreams. They always support and respect my decisions.

Last but not the least, I would like to thank my colleagues. Xi Luo, Chongxiao Cao, Anthony Danalis, Mathieu Faverge and Thananon Patinyasakdikul (Arm), to name a few. I enjoyed working with them very much. I wish them all the best.

Stay Hungry, Stay Foolish. - Steve Jobs

Abstract

Accelerator-enhanced computing platforms have drawn a lot of attention due to their massive peak computational capacity. Heterogeneous systems equipped with accelerators such as GPUs have become the most prominent components of High Performance Computing (HPC) systems. Even at the node level the significant heterogeneity of CPU and GPU, i.e. hardware and memory space differences, leads to challenges for fully exploiting such complex architectures. Extending outside the node scope, only escalate such challenges.

Conventional programming models such as data-flow and message passing have been widely adopted in HPC communities. When moving towards heterogeneous systems, the lack of GPU integration causes such programming models to struggle in handling the heterogeneity of different computing units, leading to sub-optimal performance and drastic decrease in developer productivity. To bridge the gap between underlying heterogeneous architectures and current programming paradigms, we propose to extend such programming paradigms with architecture awareness optimization.

Two programming models are used to demonstrate the impact of heterogeneous architecture awareness. The PaRSEC task-based runtime, an adopter of the data-flow model, provides opportunities for overlapping communications with computations and minimizing data movements, as well as dynamically adapting the work granularity to the capability of the hardware.

To fulfill the demand of an efficient and portable Message Passing Interface (MPI) implementation to communicate GPU data, a GPU-aware design is presented based on the Open MPI infrastructure supporting efficient point-to-point and collective communications of GPU-residential data, for both contiguous and non-contiguous memory layouts, by leveraging GPU network topology and hardware capabilities such as GPUDirect. The tight integration of GPU support in a widely used programming environment, free the developers from manually move data into/out of host memory before/after relying on MPI routines for communications, allowing them to focus instead on algorithmic optimizations.

Experimental results have confirmed that supported by such a tight and transparent integration, conventional programming models can once again take advantage of the state-of-the-art hardware and exhibit performance at the levels expected by the underlying hardware capabilities.

Table of Contents

1	Introduction	1
1.1	Motivations and Contributions	1
1.1.1	Data-flow Programming Model	4
1.1.2	Message Passing Programming Model	6
1.2	Dissertation Outline	10
2	Background and Literature Review of Related Works	11
2.1	Data-flow Programming Model	11
2.1.1	DAG-Based Representation	11
2.1.2	Parallel Runtime Scheduling and Execution Controller	12
2.1.3	Tiled Dense Linear Algebra	12
2.1.4	Literature Review	15
2.2	GPU-aware MPI	17
2.2.1	MPI Derived Datatype	17
2.2.2	MPI Point-to-point Communications	19
2.2.3	MPI Collective Communications	21
2.2.4	Literature Review	22
3	PaRSEC's Support for Heterogeneous System	27
3.1	Issues of PaRSEC in Heterogeneous System	27
3.1.1	Data Granularity of CPU/GPU Tasks	28
3.1.2	Different Memory Spaces	30

3.2	Hierarchical DAG	31
3.2.1	Methodology	32
3.2.2	Case Study: Cholesky Factorization	33
3.2.3	Hybrid Data Layout	36
3.2.4	Hierarchical DAG Task Scheduler in PaRSEC	37
3.3	Employing Multiple CUDA Streams	38
3.4	Data Coherence between CPU and GPUs	39
3.5	Out of Core Execution	41
3.6	CPU/GPU Load Balance	42
3.7	Performance Evaluation	43
3.7.1	Overhead from Runtime Task Subdivision	44
3.7.2	Number of CUDA Stream Tuning	45
3.7.3	Tile Size Tuning	47
3.7.4	Shared Memory	49
3.7.5	Distributed Memory	51
3.8	Summary	54
4	GPU-aware Point-to-point Communication	56
4.1	Issues of Point-to-point Communication of non-contiguous GPU Data in Open MPI	56
4.2	Design of GPU Datatype Engine	60
4.2.1	Vector Type	60
4.2.2	Less Regular Memory Patterns	61
4.3	Integration of GPU Datatype Engine into Open MPI	64
4.3.1	RDMA Protocol	65
4.3.2	Copy In/Out Protocol	68
4.3.3	Analysis of Pipelining of Pack/unpack with Data Movement	70
4.4	Performance Evaluation	71
4.4.1	Performance Evaluation for Datatype Engine	72

4.4.2	Full Evaluation: GPU-GPU Communication with MPI	77
4.4.3	GPU Resources of Packing/Unpacking Kernels	83
4.4.4	Pipeline and Resource Contention Effects	85
4.5	Summary	87
5	GPU-aware Collective Communication	88
5.1	Issues of Traditional Collective Communication Algorithms in Hetero- geneous System	88
5.1.1	Multiple Networks	89
5.1.2	Process Mapping	90
5.2	Design of Topology-aware Collective Communication	92
5.2.1	Topology-aware Tree	92
5.2.2	Algorithm Selection of Sub-trees	94
5.2.3	Collective Communication with Topology-aware Tree	96
5.2.4	Minimize Communications Over PCI-Express	97
5.2.5	PCI-Express Switch Level Process Reorder	100
5.2.6	Offload Reduction Operation on GPU	100
5.3	Performance Evaluation	101
5.3.1	Performance Scalability	102
5.3.2	Strong Scalability	102
5.3.3	Process Mapping	105
5.4	Summary	108
6	Conclusions and Future Directions	109
6.1	Conclusions	109
6.2	Future Directions	111
	Bibliography	113
	Vita	123

List of Tables

3.1	Status transition of data copies of tile A and C after each task.(Only tiles A and C are presented as they are shared accessed by tasks)	40
5.1	Latency of different broadcast algorithms with Hockney model	95

List of Figures

2.1	The Framework of PaRSEC.	13
2.2	Cholesky factorization on matrix of 4×4 tiles.	13
2.3	DAG representation of Cholesky factorization on matrix of 4×4 tiles	14
3.1	Performance of compute kernels on CPU and GPU depending on problem granularity	29
3.2	DAG representation of Cholesky factorization on matrix of 4×4 tiles	34
3.3	DAG of “hierarchical DAG” Cholesky factorization, whose size is 4×4 large tiles and then each CPU task is split into 3×3 small tiles. . . .	35
3.4	Different data layout: tile and LAPACK. For sub-tiles in the fine grain DAG (red), the data layout is the same as the LAPACK layout with interleaved data, while tile layout (blue) is used for large tiles and permits a much more efficient data transfer to/from the accelerators.	37
3.5	Step 0 and step 1 of cholesky factorization of 4×4 tiles. RW refers to Read and Write of data; R refers to Read data	40
3.6	GPU memory management strategy in PaRSEC.	42
3.7	Overhead incurred from the hierarchical DAG subdivision management (DPOTRF, CPU only). The h-PaRSEC version uses an big tiling of $B=900$, all tasks are subdivided into small tiles of size $b=180$ (same as standard PaRSEC)	45

3.8	Performance difference between hierarchical DAG and the standard version on DPOTRF with a varying number of CUDA streams (Bunsen using 1 K40 GPU).	46
3.9	Performance for different tile size parameters (DPOTRF, using 1 GPU on Bunsen).	48
3.10	Performance of h-PaRSEC DPOTRF with regular PaRSEC and MAGMA.	50
3.11	Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (KFS, 3 M2090 GPUs and 16 cores per node)	52
3.12	Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (Titan, 1 K20 GPU and 8 cores per node)	53
3.13	Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (Dancer, 1 C2050 GPU and 8 cores per node)	54
4.1	Four possible solutions for sending/receiving non-contiguous data residing in GPU memory.	58
4.2	Access pattern of GPU pack/unpack kernels of <i>vector</i> type. The size of a CUDA block is a multiple of the warp size.	61
4.3	Access pattern of GPU pack/unpack kernels using the DEV methodology. The left <i>struct</i> describes a work unit for a CUDA WARP.	63
4.4	Pipelined RDMA protocol for send/receive of non-contiguous GPU-resident data.	65
4.5	Communication time of using pipeline compared with communication without pipeline.	69
4.6	Triangular matrix (red one) vs Stair triangular matrix (red and green one), width and height of stair <i>nb</i> is multiple of CUDA block size	72

4.7	GPU memory bandwidth of packing kernels for sub-matrix and lower triangular matrix comparing with contiguous data of the same size. “T” represents triangular matrix, “V” represents sub-matrix, “C” represents contiguous matrix	74
4.8	Performance of pack and unpack sub-matrix and lower triangular matrix varies by matrix size.	75
4.9	<i>vector</i> pack/unpack performance vs <i>cudaMemcpy2D</i> . “kernel” represents our pack/unpack kernels. “mcp2d” represents <i>cudaMemcpy2D</i> . “d2d” represents non-contiguous data packed into a GPU buffer. “d2d2h” represents “d2d” followed by a device-host data movement. “d2h” means non-contiguous GPU data moved directly into CPU buffer.	76
4.10	PCI-Express bandwidth of vector and indexed data type comparing with contiguous data.	78
4.11	Ping-pong benchmark with matrices on shared memory machine. “V” refers to sub-matrix, “T” refers to triangular matrix.	79
4.12	Ping-pong benchmark with matrices on distributed memory machine. “V” refers to sub-matrix, “T” refers to triangular matrix.	80
4.13	Ping-pong benchmark with vector and contiguous data type.	82
4.14	Ping-pong benchmark for matrix transpose in both shared and distributed memory environment.	83
4.15	GPU memory and PCI-Express bandwidth of pack/unpack sub-matrix “V” data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.	84
4.16	GPU memory and PCI-Express bandwidth of pack/unpack triangular matrix “T” data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.	85
4.17	Ping-pong benchmark with partial GPU resources available. In the legend, the number after the matrix size is the ratio of GPU resources occupied.	86

5.1	Topology-aware tree of broadcast algorithm on multi-GPU cluster. (4 GPUs per socket and 2 sockets per node)	93
5.2	Data flow of non-root node leader MPI process	98
5.3	Process mapping using hardware topology of PCI-Express switch . . .	99
5.4	Performance of broadcast and reduce with GPU data varies by message size using 8 nodes 32 GPUs	103
5.5	Strong scalability of broadcast and reduce with GPU data varies by number of nodes, message size is 32 MB	104
5.6	Bandwidth of broadcast with GPU data varies by different process mapping with message size 32MB	106
5.7	Bandwidth of broadcast with GPU data on 1 nodes varies by different process mapping on the level of PCI-Express switch with message size 32MB	107

Chapter 1

Introduction

1.1 Motivations and Contributions

Throughput-oriented architectures, such as GPUs, are becoming ubiquitous assistants for computationally intensive tasks in scientific applications. Compared with traditional CPU, GPU has much higher peak performance and memory bandwidth. For example, the peak double precision floating point performance and memory bandwidth of the Nvidia Kepler K40 is approximately 1.43 Tflop/s and 288 GB/s, dwarfing the performance of any existing CPU family. As a consequence, an increasing number of production systems feature GPUs. In Top500 [Top500 \(2016\)](#) list, 20% of the top 500 machines and 40% of the top 10 machines are equipped with GPUs. Such trend is expected to persist in the future towards ex-scale machine: the coming machine, Oak Ridge National Laboratory's "Summit" and Lawrence Livermore National Laboratory's "Sierra", will both use GPUs as accelerators.

The hardware of CPU and GPU are significant different: GPU features thousands of light-weight cores while CPU features much less heavy-weight cores, and the cost of thread context switch of GPU is lower than CPU, but the latency of issuing instructions of GPU is relevantly higher than CPU, so the way of efficiently programming in GPU and CPU are different. Typically, GPU programs launch thousands of

threads and switch contexts frequently to hide the latency of instructions. While CPU programs run on less threads and need minimal thread context switch. Therefore, GPU programming model extracts parallelism by operating on large granularity of data to achieve optimal occupancy of GPU. In contrast, CPU programming model is much more flexible with less restriction, hence less data granularity is able to feed a modern CPU. To fully exploit the resources of CPU and GPU, it is expected to select the proper execution unit for programs based on their degrees of available parallelism.

Machines equipped with accelerators, such as GPUs, are called heterogeneous systems, in which GPUs are connect to host machine as peripheral devices via PCI-Express. More recent advances, CPUs are connecting these GPUs with their own dedicated network, NVLink, allowing for a notable increase in the data movement capabilities, especially between accelerators. However, for a long time, GPUs have a separate memory space than the host. Explicit memory copy directives are necessary to move data between host and GPU, before being available to computations or communication on CPU/GPU. This memory separation has been fused with the introduction of the Unified Memory Architecture (UMA), allowing the host memory to be directly accessed from GPUs, and inversely, GPU memory to be directly accessed from CPUs. However, the connection between CPU and GPUs is bandwidth oriented not latency oriented, data parallel programs which involves frequently memory access of small independent data, is not able to fully utilize bandwidth. Therefore, it is always better to explicitly move data into GPU memory prior execution on GPU for such programs. Limited by CPU-GPU link bandwidth even with NVLink, such data transfers are expensive, hence, in order utilize both CPUs and GPUs efficiently, developers have to carefully overlap data movements between host and GPU with computations, as well as to minimize data movement and reuse data in GPU memory if available, in order to fully exploit the performance of both CPU and GPU.

Satisfying the increasing demand for computation from the scientific computing community, led to the trends of super large scale clusters. A typical large scale heterogeneous cluster usually is consisted of thousands of computer nodes. Computer

node, the building block of super computers, usually contains multiple CPU sockets connected by high speed inter-socket connection (e.g. Intel QPI or AMD Hyper-transport), and multiple GPUs. Scaling up, several computer nodes are coupled together through high performance network and form a computer blade, which are organized in racks and then finally large scale, super-computers. All these advances at the hardware level, cause a drastic increase in the hierarchization of different components, with wild differences between the different levels of the hierarchy. Communication cost between GPUs are dramatically different depending on location of GPUs: intra-socket communication is able to use CUDA Inter-process communication (CUDA IPC) [NVIDIA \(2016b\)](#) to achieve RDMA between two GPUs; inter-socket communication has to fall back to stage through CPU memory; inter-node communication can either use GPUDirect RDMA [NVIDIA \(2015\)](#) or go through intermediate CPU memory. Hence, maintaining good network performance requires efficiently utilization of all different networks and taking care of GPU locality as well as network topology.

Programming models such as data-flow and message-passing have been proved efficient for conventional distributed homogeneous systems. When moving towards distributed heterogeneous systems, the straight forward approach to port such programming models is to explicitly move data from GPU to host memory prior engaging CPU-based conventional programming model and move data back to GPU memory afterwards. However, such directly porting can not efficiently utilize resources of both CPU and GPU due to the lacking of architecture awareness optimization. Therefore, modifications of traditional programming models must take account of the characteristic of heterogeneous systems (significant hardware differences between CPU and GPU, different memory space of CPU and GPU, and complicated network topology of entire system). In the dissertation, we focus on two widely used programming models: data-flow programming model and message-passing model.

1.1.1 Data-flow Programming Model

Data-flow programming model has seen a revival, with the emergence of numerous task-based programming frameworks, where an algorithm is divided into computations entities (tasks) connected by data dependencies, and forms a Direct Acyclic Graph (DAG) (nodes and edges represent tasks and data dependencies of tasks respectively). This programming paradigm has been successfully used in different projects to depart from tightly coupled or fork-join programming paradigms, and express the parallelism in a form that allows for more execution flexibility and portability across many types of hardware resources. One of the early adopters of this programming paradigm is the ParSEC [Bosilca et al. \(2012\)](#) framework, which encompasses a toolbox to help express algorithms in the data-flow programming paradigm, and a task runtime component whose role is to efficiently schedule the resultant DAG, on large scale distributed heterogeneous systems.

Tiled linear algebra algorithm is one of the beneficiaries of data-flow programming model [Agullo et al. \(2009\)](#). With tiled linear algebra, a matrix is divided into square tiles and each task operates on tiles. In heterogeneous systems, execution union of tasks can be either a CPU core or GPU. Lack of GPU knowledge, when deploying tasks, there are two common issues in traditional task-based runtime that could slow down performance. First, the size of tiles is one of the critical tuning parameters that impacts the efficiency of kernels, the degree of parallelism and the communication volume. As discussed before, due to different architectures of CPU and GPU, optimal data granularities (represented as tile size in the context of linear algebra) of GPU and CPU tasks are different: usually GPUs require large data set while CPUs benefit from smaller ones. Traditional tiled linear algebra algorithms [Bosilca et al. \(2011\)](#), which require all tasks to have a unique tile size to reach reasonable performance, but fail to provide the runtime with the means to achieve an adapted load-distribution on heterogeneous systems. Second, data dependencies indicate data transfers between tasks if they are executed in different execution units. As CPU and GPU have

different memory space, most linear algebra GPU kernels are not able to be beneficial from UMA because of their memory access pattern, therefore, it requires developers to explicitly move data into/out of GPU memory prior/after GPU tasks. Limited by the PCI-Express bandwidth, such data movement is expensive, hence serialization of the data movement and GPU kernels is not able to deliver optimal performance. Therefore, it is desired to overlap communication with computation. Application developers without rich experience in GPU programming are unlikely to efficiently handle such overlapping, which calls for task runtime to automatically infer data transfers between CPU and GPU, and provide better overlapping of communications with computations to fully exploit the resources of both CPU and GPU.

In this dissertation, we integrate the GPU knowledge including architecture and memory space into task runtime and achieve the following contributions:

- We propose a method called “hierarchical DAG” to adapt the granularity of tasks with a multi-level approach, where tiles of different sizes coexist in the runtime. In the hierarchical DAG approach, tasks operated on large granularity data (large tile size) are organized in an outer DAG level, which are executed on GPUs. When executed on CPU, each large granularity task can be dynamically subdivided into a finer granularity inner DAG, operating on smaller tiles, so that the larger number of finer granularity tasks increases the available parallelism to levels adequate for multi-core processors.
- We design a data coherence protocol to track the data copies in CPU and GPU memory. With the help of coherence protocol, data is cached in GPU memory to reduce data movement. Later, data movement and GPU kernel execution are overlapped with each other by offloading them to different CUDA streams.
- We develop a multi-level GPU memory management, which reuses GPU memory based on Least Recent Used (LRU) strategy when running out of memory, and therefore, it supports out of core execution (problem size larger than GPU memory size).

- We showcase a popular linear algebra algorithms - Cholesky factorization to motivate the need for “hierarchical DAG” design to adjust task granularity and integration the knowledge of separated memory space of CPU and GPU into task runtime to overlap communication with computation and minimize data movement.

1.1.2 Message Passing Programming Model

In data-flow model, data flows from one task to another. There are several ways to implement the underlying data movement. One of the popular approach is to use message passing. Message passing model is another traditional programming paradigm used in distributed system. Processes communicate with each other by messages without resorting to shared variables. Message Passing Interface (MPI) is a standard, which defines a set of communication pattern with message passing. Since the MPI standard [MPI Forum \(1995\)](#) does not define interactions with GPU-based data, it is expected that application developers have to explicitly initiate data movements between host and device memory prior to use MPI to move data across node boundaries. Such approach imposes a significant complexity on programmers, renders explicit management of hierarchies which defeats performance portability, In heterogeneous system, it is expected MPI implementations to provide GPU-aware capability by unifying MPI routines for both CPU and GPU data, and freeing programmers from explicit CPU-GPU data movements. However, the current MPI implementations obviously can not satisfy the requirement of high efficiency and portability in communication of GPU-resident data. In this dissertation, we adapt Open MPI, one of the state-of-the-art MPI implementations, to heterogeneous system to provide efficient point-to-point and collective communication of GPU data.

Point-to-point Communication

Since point-to-point is the basic building block routines of MPI, the performance of point-to-point communication is critical. As many scientific applications operate on multi-dimensional data, manipulating parts of these data becomes complicated because the underlying memory layout is not-contiguous. The MPI standard proposes a rich set of interfaces to define regular and irregular memory patterns, the so called Derived Datatypes (DDT). The DDTs provide a general and flexible solution to describe any collections of contiguous and non-contiguous data with a compact format. Once constructed and committed, an MPI datatype can be used as an argument for any MPI communication routines. Thus, the scientific application developers do not have to manually pack and unpack data in order to optimize non-contiguous data transfers, but instead they can safely rely on the MPI runtime to make such operations trivial and portable. To improve point-to-point communication between GPUs, the GPUDirect technique are proposed to enable RDMA-like data movement between GPUs without staging through host memory. Recent state-of-the-art implementations of MPI, such as MVAPICH and Open MPI already utilize GPUDirect to provide the capability of direct GPU data movement between processes. Unfortunately, these optimizations were designed with a focus on contiguous data, leaving the most difficult operations, the packing and unpacking of non-contiguous memory patterns, in the charge of developers. There are effective packing/unpacking implementations for datatypes in host memory [Ross et al. \(2003\)](#). However, exposing the same level of support for a non-contiguous MPI datatype based on GPU memory remains an open challenge.

Since MPI collective operations are based on point-to-point communication, DDT support is usually integrated in point-to-point level, and then collective communications are automatically able to support non-contiguous data layout. In this dissertation, we achieve the following contributions on non-contiguous point-to-point communication of GPU data:

- We present the design of a datatype engine for non-contiguous GPU-resident data, which is able to take advantage of the embarrassingly parallel nature of the pack and unpack operations and efficiently map them onto GPU threads.
- We incorporate the GPU datatype engine into the Open MPI infrastructure, and takes advantage of the latest NVIDIA hardware capabilities, such as GPUDirect, not only to minimize the overheads but also to decrease the overall energy consumption. For contexts where GPUDirect is not available, we provide a copy-in/copy-out protocol using host memory as an intermediary buffer.
- We present a light-weight pipeline protocol to allow pack and unpack operations to work simultaneously.
- We demonstrate the performance improvement of point-to-point communication of non-contiguous GPU data by comparing with state-of-the-art MVAPICH library via variety of benchmarks.

Collective Communication

Collective communications are another set of communication patterns, which messages are exchanged within a group of processors. Since collective communications are widely used in scientific and deep learning application, it is crucial for MPI libraries to sustain the parallel applications by providing the most optimal collective routines. According to underlying link properties between GPUs, a collective operation in heterogeneous systems includes inter-node, inter-socket and intra-socket communications, whose bandwidth and latency are different. Therefore, a smart collective algorithm should be able to utilize the knowledge of GPU network topology to rearrange the processes involved in the collective pattern, in order to shift the burden from low performance networks and minimize communications on these slow channels. However, traditional collective algorithms do not worry about hierarchical networks, resulting in sub-optimal performance when mapping of MPI processes

does not strictly follow the hardware hierarchy of processors and topology of GPUs participated in collective communications. Indeed, recent advances in MPI collective communications have already demonstrated that such performance issues can be solved by integration of network topology information into collective operations [Graham et al. \(2011\)](#) [Kandalla et al. \(2010\)](#). However, insufficient cooperation of communications of different topology levels (i.e. intra-socket, inter-socket and inter-node levels) leads to sub-optimal overlapping and pipelining of different levels' communications, and to algorithms that are not adaptable to the fluctuating network conditions. This calls for a collaborative approach between multiple levels of collective algorithms, dedicated to holistically managing all levels of the network hierarchies.

In this dissertation, we propose a new GPU-aware collective framework by taking account of GPU network topology and achieve the following contributions:

- We present a topology-aware collective framework in Open MPI, which orchestrates collaborations between multiple levels of network, toward a common goal. Instead of creating isolated communicators for different levels, we incorporate all processes into process groups based on their closeness, build communication tree based on network topology. In such way, we eliminate all topology levels boundaries and allow for fine grain pipelining between the different communications. Also since network of different topology levels are independent, we allow for more concurrent communications, eventually providing more opportunities for offloaded communications to overlap.
- We minimize communications over PCI-Express by caching data in CPU memory, so that inter-node and inter-socket communications are directly use the cached data instead of pulling data from GPU memory. One directional of PCI-Express is used for intra-socket GPU RDMA communication, and the other is used for update cached data from CPU memory back to GPU memory, therefore, PCI-Express is fully utilized but no congestion.

- As a side-effect of using GPUs, we also have the opportunity minimize the cost of the reduction operations by offloading them on the accelerators.
- We showcase two popular collective operations - broadcast and reduce - to prove the advantage of our topology-aware collective framework over the state-of-the-art MVAPICH.

1.2 Dissertation Outline

The rest of this dissertation is organized as follows: Chapter ?? introduces task-based runtime with an application of dense linear algebra and GPU-aware MPI, as well as a survey of the literature of these two aspects. Chapter 3 discusses the approaches developed for PaRSEC to optimal utilization of resources in heterogeneous systems for computational tasks, including “hierarchical DAG” for optimal occupancy of both CPU and GPU, data coherence protocol for minimize data movement and multi-level GPU memory management for out of core execution. Chapter 4 presents the implementation of non-contiguous point-to-point communication of GPU data with benchmarks to demonstrate the performance improvement over other MPI implementations. Next, Chapter 5 describes the design of GPU-aware collective communication by integrating the knowledge of GPU network topology and locality with MPI. Two collective operations, broadcast and reduce, are use as example to prove the higher performance obtained compared with other MPI implementations under any process placement. Finally, Chapter 6 concludes the dissertation and outlines the future work.

Chapter 2

Background and Literature Review of Related Works

2.1 Data-flow Programming Model

2.1.1 DAG-Based Representation

Different from traditional control-flow programming model, data-flow programming paradigm emphasizes the movement of data and models programs as a series of connections. Explicitly defined inputs and outputs data connect different tasks. A task runs as soon as all of its inputs become valid. Thus, data-flow programming paradigm are inherently parallel and can work well in large, decentralized systems. With data-flow programming model, applications are divided into a set of different type of tasks, and described as a DAG $D = (V, E)$. Tasks, also called kernels, are a set of sequential computations, which is fundamental of an application. In a DAG representation, a vertex $v \in V$ represent a task and a edge represents data dependencies between a task v_1 and its predecessor task v_2 . If an edge (v_1, v_2) exists in E , then the output data of task v_1 should be transferred to the execution location of task v_2 as its input data, and task v_2 can not start until its all input data is ready.

2.1.2 Parallel Runtime Scheduling and Execution Controller

In order to deploy tasks in DAGs efficiently to a proper execution unit including CPU and GPUs, it is desired to provide a task-based runtime system. The Parallel Runtime Scheduling and Execution Controller (PaRSEC) [Bosilca et al. \(2012\)](#), developed by Innovative Computing Laboratory, is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. PaRSEC is an adoption of data-flow program paradigm, which takes this DAG-based representation and assigns tasks to the computing resources, and uses a dynamic, fully-distributed scheduler based on cache awareness, data-locality and task priority.

Figure [2.1](#) presents the detailed framework of PaRSEC, which is consisted of 3 levels. The first level is hardware level, which interacts with different hardware architecture, including multi-core CPUs, memory hierarchies and accelerators. The middle level is the functionalities of the parallel runtime in PaRSEC, including distributed scheduling, data distribution and movement, task management and creating specialized kernels. The third level is the extension for domain specific applications, including a concise format of representing tasks called Parameterized Task Graph (PTG) [Cosnard et al. \(1999\)](#), and a dynamic representation of tasks called Dynamic Task Discovery (DTD) [Haidar et al. \(2011\)](#).

2.1.3 Tiled Dense Linear Algebra

In the area of dense linear algebra, DAGs have been demonstrated to be an extremely effective way to describe tiled linear algebra algorithms [Agullo et al. \(2009\)](#). In tiled linear algebra algorithms, an $N \times N$ matrix is split into $NT \times NT$ tiles, each of size B ($\lceil N/B \rceil = NT$). A “tile” can be considered a sub-matrix of the original matrix. Therefore, instead of computing element by element, each computation task/kernel executes on tiles.

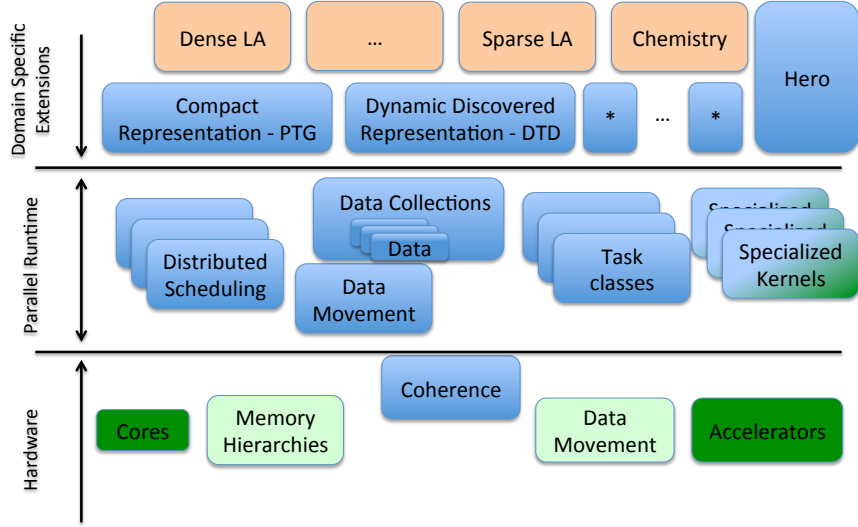


Figure 2.1: The Framework of PaRSEC.

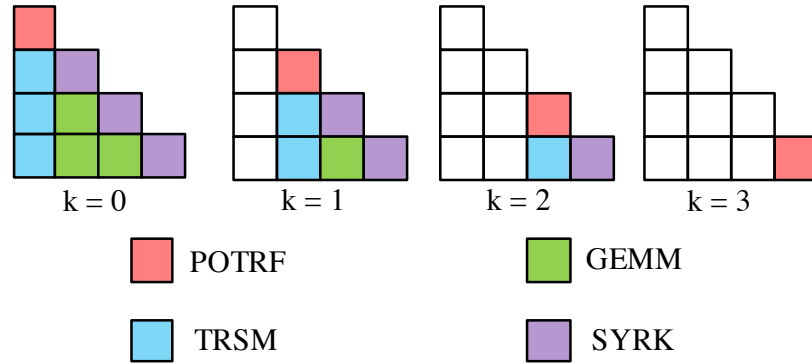


Figure 2.2: Cholesky factorization on matrix of 4×4 tiles.

Cholesky factorizations is a classic linear algebra algorithms that are widely used for solving linear systems $Ax = b$, and as basic blocks in computing eigenvalues and singular values. It is composed of four kernels (POTRF, TRSM, SYRK and GEMM) [Ltaief et al. \(2011\)](#) that are successively applied on the trailing sub-matrix at each step, as illustrated in Figure 2.2 for matrices of 4×4 tiles.

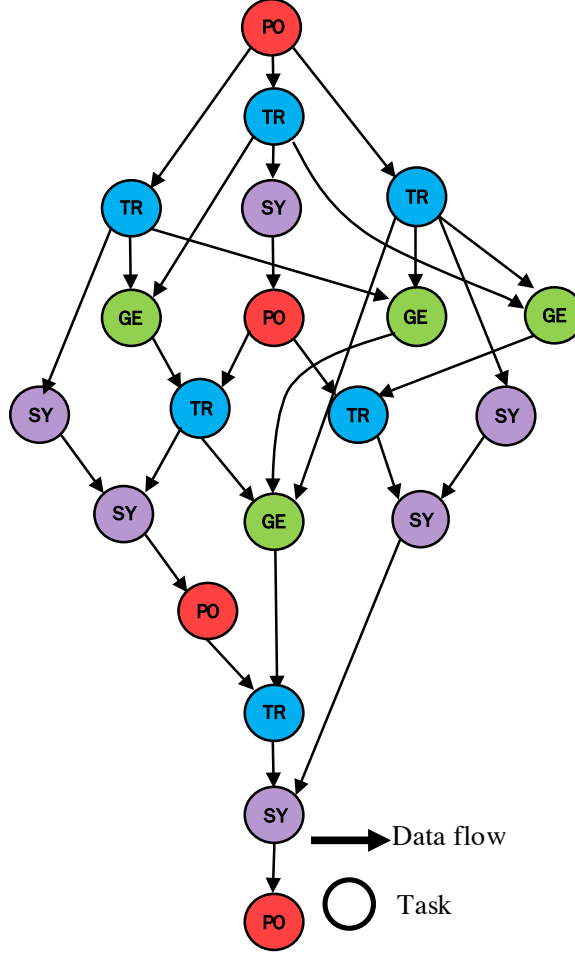


Figure 2.3: DAG representation of Cholesky factorization on matrix of 4×4 tiles

- POTRF performs the untiled Cholesky factorization of a diagonal tile of the input matrix and overrides it with the final elements of the output matrix.
- SYRK is a symmetric rank-k update, which updates to a diagonal tile of the input matrix.
- TRSM is a triangular system solve, which applies transformation computed by POTRF to an off-diagonal tile below the diagonal tile operated by the last POTRF of the same column.
- GEMM is a matrix-matrix multiplication, used to update tiles in trailing matrix.

Figure 2.3 is the DAG representation of the Cholesky factorization shown on Figure 2.2. Usually, the execution location of a kernel is dynamic, which can be either a CPU core or an accelerator. With the help of task-based runtime such as PaRSEC, application developers only focus on translation of application into DAG representation; PaRSEC take care of data distribution and task scheduling: it automatically deploy tasks to a proper execution unit based on load balance and data locality. In this dissertation, Cholesky factorization is used to demonstrate the performance of GPU-aware design of PaRSEC.

2.1.4 Literature Review

Dense Linear Algebra on Heterogeneous System

Dense linear algebra is one of the computing fields most likely to benefit early from any increase in the computational power of the hardware such as GPUs. Thus, it is not unexpected that every evolution at the hardware level is rapidly reflected in dense linear algebra libraries. MAGMA Agullo et al. (2009) Cao et al. (2013) is a linear algebra library designed for GPUs. It harnesses the power of both the GPU and the CPU by invoking CUDA, OpenCL, or multi-threaded BLAS kernels. Fogue et al. ported the existing LAPACK library to GPU-accelerated clusters Fogue et al. (2010). However, both libraries are not driven by runtimes, their static scheduler distributes tasks equally among GPUs, resulting in potential load imbalance and poor portability. Beside, the current version of the MAGMA library doesn't support distributed memory systems.

Runtime Driven Dense Linear Algebra on Heterogeneous System

When towards heterogeneous system, more and more dense linear algebra libraries trend to transit to DAG-based representation and rely on runtime because of their better portability and load balance management. Quintana-Orti et al. extended the SuperMatrix runtime to shared-memory machines with GPUs Quintana-Ortí

et al. (2009). LibFLAME is another library for dense matrix computations in heterogeneous system Zee et al. (2009). However, in these solutions, only a particular type of computational kernel can execute on the CPU (the less compute-intensive diagonal blocks), which produces a load imbalance between CPUs and GPUs. As discussed in Chapter 1, because of hardware differences of CPU and GPUs, optimal data granularity (described as tile size in dense linear algebra) of CPU and GPU tasks are dramatically different. All these prior works mandate the use of an identical tile size, thereby preventing the adaptation of the task granularity to the considered execution resource.

There are a few prior works trying to resolve the tile size mismatch between CPUs and GPUs. Song et al. presented a heterogeneous tile algorithm Song et al. (2012) which divides square tiles into a skinny tall rectangle tile for the CPU and places the remainder on the GPU. It uses a non-uniform 1D partitioning, and data is statically distributed between GPUs, hence, it is likely to cause imbalance in the Cholesky factorization. Kim et al. adapted the libFLAME library to support different block sizes on different devices in a shared memory environment Kim et al. (2012). However, its write-through GPU data caching policy may incur too many unnecessary data movements between the host and the GPU. Lima et al. presented similar work for Intel Xeon Phi Lima et al. (2013). However, the decision to recursively split a task is made statically at submission time, without runtime insight. Furthermore, in their Cholesky factorization, only the POTRF kernel is recursively split.

Our approach uses a 2D block cyclic data distribution for each host, and data is dynamically assigned to GPUs to maintain good load balance. We maximize the throughput by allowing all operations with the GPU to be asynchronous, overlapping data movements and task submission to the GPU, and allowing threads to migrate between GPU management and CPU execution. Thanks to the parameterized DAG of our solution, the decision is taken dynamically at runtime and is not limited to a single kernel, an important distinction as several kernels can compose the critical path of an application. Moreover, our approach supports multiple node deployments

with automatic network transfers and a distributed scheduler suited for large scale systems.

2.2 GPU-aware MPI

The message passing model has emerged as an expressive, efficient, and well-understood paradigm for parallel programming. The process of creating a standard to enable portability of using message passing for application began at Message Passing Interface (MPI) Forum [MPI Forum \(1995\)](#). MPI is a message passing library standard, together with protocol and semantic specifications for how its features must behave in any implementation. Point-to-point and collective communications are two important and frequently used communication patterns in MPI [Gropp et al. \(1996\)](#). MPI is now already widely used for solving significant scientific and engineering problems on parallel computers. There are several state-of-the-art MPI implementations such as Open MPI [Gabriel et al. \(2004\)](#), MPICH2 [Gropp \(2002\)](#), MVAPICH2 [Huang et al. \(2007\)](#) and Intel MPI. When towards heterogeneous systems, MPI implementations such as Open MPI and MVAPICH2 already provide some levels of support for data residing in GPU memory. With GPU-aware MPI, users can use MPI communication routine to transmit GPU data without hand-made moving data from GPU memory to host memory and vice versa. In this dissertation, we integrate GPU knowledge into Open MPI on both point-to-point and collective communications.

2.2.1 MPI Derived Datatype

MPI provides a powerful and general way to describe arbitrary collections of data in memory. MPI Standard [MPI Forum \(1995\)](#) predefines its primitive data types such as MPI_INT, MPI_CHAR, MPI_DOUBLE and so on, respecting to data type of int, char and double in C or Fortran language. Based on primitive data types, MPI

also provides facilities for users to define their data structures, which is called MPI derived datatypes (DDT). MPI DDTs provide a flexible and general mechanism for working with arbitrary layouts (contiguous or non-contiguous) of data in memory. MPI defines data layouts of varying complexity:

- Contiguous: a number of repetitions of the same datatype without gaps in-between
- Vector: defines a non-contiguous data layout that consists of equally spaced blocks of the same datatype.
- Indexed: specifies a noncontiguous data layout where neither the size of each block nor the displacements between successive blocks are equal.
- Struct: consists of location-blocklength-datatype tuples, allowing for the most flexible type of non-contiguous datatype construction.

Once constructed and committed, an MPI DDTs can be used as an argument for any point-to-point, collective, I/O, and one-sided functions. MPI DDTs allow users to treat non-contiguous data in a convenient manner as though it was contiguous in memory. Because current network is bandwidth-oriented instead of latency-oriented, large messages delivers better bytes per second transfer rates (network bandwidth). Without MPI DDTs, users must manually copy any data to be sent to a contiguous buffer, pass that to the send routine, and then unpack the data when it is received. With MPI DDTs, users can safely rely on MPI runtime to make such pack/unpack operations trivial and portable. Internally, the MPI datatype engine automatically packs and unpacks data based on the type of operation to be realized, in an efficient way while hiding the low-level details from users.

MPI DDTs provide a solution to avoid intermediate packing and unpacking of communication data that might otherwise be necessary when working with non-contiguous data manually, therefore, it is widely adopted by scientific applications. In the 2D stencil application of the Scalable Heterogeneous Computing benchmark

(SHOC) [Danalis et al. \(2010\)](#), two of the four boundaries are contiguous, and the other two are non-contiguous, which can be defined by a *vector* type. In the LAMMPS application from the molecular dynamics domain [Schneider et al. \(2012\)](#), each process keeps an array of indices of local particles that need to be communicated; such an access pattern can be captured by an *indexed* type. Hence, MPI datatypes help application developers alleviate the burden of manually packing and unpacking non-contiguous data. Recent MPI implementations have exhibited significant performance improvement for the handling of non-contiguous datatypes when handling CPU-based data [Ross et al. \(2003\)](#); [Schneider et al. \(2012\)](#). Therefore, it is urgent to extend the MPI DDTs support to GPU data for efficient programming in heterogeneous systems.

2.2.2 MPI Point-to-point Communications

MPI point-to-point communications typically involve message exchanges between two MPI processes. One process is performing a send operation and the other process is performing a matching receive operation. As early as 1994, point-to-point communications have been included into the first MPI standard (MPI-1.1) [MPI Forum \(1995\)](#). MPI point-to-point communication is the basic communication routines, other communication patterns such as collective communication are build on top of point-to-point operations. Therefore, it is very important to deliver high performance point-to-point operations. MPI point-to-point operations can be categorized into two types based on the number of send and receive operations:

- Single send/receive: these types of routines issues only one send/receive operations at once, such as *MPI_Send* and *MPI_Recv*.
- Combined send/receive: these types of routines combine in one call the sending of a message to one destination and the receiving of another message, from another process, such as *MPI_Sendrecv*.

Single send/receive routines support blocking and non-blocking mode. In blocking model, routines only return after it is safe to modify the users' data buffer for reuse. In

non-blocking model, routines return immediately without any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message; actual communications are progressed only after the call to routine *MPI_Wait/MPI_Waitall*. Combined send/receive routines does not have non-blocking model since it force the sequence of receive after send, while non-blocking send/receive do not guarantee such sequence. No matter blocking or non-blocking model, the progressing of communications are the same; the only difference is the moment of progress (immediate or delayed). In this dissertation, we work on the layer of progressing point-to-point data transfer, hence, our work support both blocking and non-blocking point-to-point communications. The data type that is sent/received in MPI point-to-point routines can be either primitive or derived datatypes. The following subsections describe the API definition of single and combined send/receive operations in the MPI standard [MPI Forum \(1995\)](#)

Single Send/Receive

- Send

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
```

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request)
```

MPI_Send blocks until the message is sent to the destination. *MPI_Isend* is non-blocking; the sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

- Receive

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Recv is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started). MPI_Irecv is non-blocking; the receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Combined Send/Receive

- Send/Receive

int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

MPI_Sendrecv executes a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

2.2.3 MPI Collective Communications

MPI collective communications are abstracted from a wide variation of distributed parallel algorithms, which are another set of widely used communication patterns in MPI applications. MPI collective communications involve message exchanges among all processes in the scope of a communicator. Similar to point-to-point communications, collective communications are also included in the first MPI Standard (MPI-1.1) [MPI Forum \(1995\)](#). As well as point-to-point communication, collective communications are frequently used in all kinds of MPI applications. For example, broadcast and reduce are often used in All-Pairs-Shortest-Path algorithm [Plaa et al. \(1999\)](#) and deep learning applications [Yu et al. \(2014\)](#). In this dissertation, we focus

on two types of widely used collective operations: broadcast and reduce, and the performance of these operations are greatly threatened by increasing amounts of data and hardware complexity especially when GPU is engaged. The performance of applications are usually sensitive to quality of implementations of these operations. The following subsections describe the API definition of single and combined send/receive operations in the MPI standard [MPI Forum \(1995\)](#).

Broadcast

```
int MPI_Bcast ( void buffer , int count , MPI_Datatype datatype , int root ,  
MPI_Comm comm)
```

MPI_Bcast sends a message from the root process to all processes within the communicator. It is called by all processes of the communicator with the same arguments for comm and root. Once returned, the contents of roots communication buffer has been copied to all processes.

Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype  
datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI_Reduce performs a reduction operation “op” across all processes within the communicator. Data is gathered from send buffer (sendbuf) and final result is in receive buffer(recvbuf). This routine also support in-place mode, which converts the receive buffer into a send-and-receive buffer.

2.2.4 Literature Review

GPU-aware MPI Point-to-point Communication

Heterogeneous systems equipped with both CPUs and GPUs are currently the most popular platform in high performance computing. Writing efficient applications for such heterogeneous systems is a challenging task as application developers need

to explicitly manage two types of data movements: intra-process communications (device to host) and inter-process communications. Recent versions of well-known MPI libraries such as MVAPICH2 [Wang et al. \(2011b\)](#) and Open MPI already provide some levels of GPU support for point-to-point communications. With these GPU-Aware MPI libraries, application developers can use MPI constructs to transparently move data, even if the data resides in GPU memory. Similar efforts have been made to integrate GPU-awareness into other programming models. Aij et. al. propose the MPI-ACC [Aji et al. \(2012\)](#), which seamlessly integrates OpenACC with the MPI library, enabling OpenACC applications to perform end-to-end data movement. Lawlor presents the cudaMPI [Lawlor \(2009\)](#) library for communication between GPUs, which provides specialized data movement calls that translate to *cudaMemcpy* followed by the corresponding MPI call. Even though the paper discusses non-contiguous data support, the current implementation only includes support for vector types. For the PGAS programming model, Potluri et. al [Potluri et al. \(2013\)](#) extend OpenSHMEM to GPU clusters providing a unified memory space. However, as OpenSHMEM has no support for non-contiguous types, this implementation does not provide sufficient support to communicate non-contiguous GPU data. All these works focus on providing GPU-awareness for parallel programming models, and have been demonstrated to deliver good performance for contiguous data, but none of them provide full and efficient support for non-contiguous data residing in GPU memory.

More recent works have focused on providing non-contiguous MPI datatype functionality for GPU data. Wang et. al. have improved the MVAPICH MPI implementation to provide the ability to transparently communicate non-contiguous GPU memory that can be represented as a single vector, and therefore translated into CUDA’s two-dimensional memory copy (*cudaMemcpy2D*) [Wang et al. \(2011a\)](#). A subsequent paper by the same authors tries to extend this functionality to many data-types by proposing a vectorization algorithm to convert any type of datatype into a set of vector datatypes [Wang et al. \(2014\)](#). Unfortunately, indexed datatypes such as triangular matrices, are difficult to convert into a compact vector type. Using

Wang’s approach, each contiguous block in such an indexed datatype is considered as a single vector type and packed/unpacked separately from other vectors by its own call to *cudaMemcpy2D*, increasing the number of synchronizations and consequently decreasing the performance. Moreover, no pipelining or overlap between the different stages of the datatype conversion is provided, even further limiting the performance.

Jenkins et. al. integrated a GPU datatype extension into the MPICH library [Jenkins et al. \(2014\)](#). His work focuses on the packing and unpacking of GPU kernels, but without providing overlaps between data packing/unpacking and other communication steps. Both Wang and Jenkins’s work require transitioning the packed GPU data through host memory, increasing the load on the memory bus and imposing a significant sequential overhead on the communications. All of these approaches are drastically different from our proposed design, as in our work we favor pipelining between GPU data packing/unpacking and data movements, and also take advantage, when possible, of GPUDirect to bypass the host memory and therefore decrease latency and improve bandwidth.

GPU-aware MPI Collective Communication

In heterogeneous system, according to underlying link properties between processes, when data is residing in GPU memory, communications between any two processes could use different networks depending on the location of GPUs (discussed in Section 1.1). Therefore, message exchanges between processes involve different networks (intra-socket, inter-socket and inter-node). To minimize the data movement over the heavy channels, collective communication should be able to take care of GPU localities, which is represented by GPU network topology.

For data in host memory, several previous works have been done to use topology-aware idea for collective operations to take advantage of communication cost differences at every level in network. MagPie [Kielmann et al. \(1999\)](#) creates hierarchical algorithms for clustered wide-area systems to avoid slow links. MPICH2 [Zhu et al. \(2009\)](#) implements several collective operations by exploits knowledge of

the topology. But these works only consider two network layers. Karonis et. al. [Karonis et al. \(2000\)](#) extends the previous work and presents a multi-level topology-aware tree to support more network layers. Later, MVAPICH2 [Kandalla et al. \(2010\)](#) [Subramoni et al. \(2012a\)](#) introduce Neighbor-Joining techniques to detect network topology on switch level, and adds one more levels in the network hierarchy collective operations. However, all these approaches focus on exploring more and more network topology levels. While they provide interesting performance compared with a single-level approaches, but their inter and intra levels communications do not cooperate tightly, leading to non-communication overlap between different topology levels.

Other researchers try to take the benefit of shared memory and propose hierarchical collective operations. Tipparaju et. al. [Tipparaju et al. \(2003\)](#) uses shared memory as intermediate buffer to reduce number of memcpyies. Cheetah [Graham et al. \(2011\)](#) is a hierarchical collective communication framework. In this framework, a Directed Acyclic Graph is constructed based on characteristics of communication topology. It can take advantage of shared memory for intra-node communications and point-to-point (p2p) or InfiniBand CORE-Direct for inter-node communications. Parsons et. al. [Parsons and Pai \(2014\)](#) decouples the choice of inter-node and intra-node communication algorithms. Similarly, all these work do not have communiation overlap between levels. HierKNEM [Ma et al. \(2012\)](#) enables tight collaboration between the collective algorithms pertaining to different layers of the hierarchy. It combines KNEM(an Linux kernel for memcpyy in shared memory), pipelining and hierarchical idea to allow overlap of inter-node and intra-node communication. But it only have two topology level and in each level the tree is fixed. Our algorithm can support multiple topology levels and each level can select different algorithms base on different characteristic of each group like number of processes and message size. Also HierKNEM is bind to shared memory, but our framework is much more flexible which supports different hardware like GPU.

State-of-the-art MPI libraries such as Open MPI and MVAPICH2 [Singh et al. \(2011\)](#) have provided CUDA-aware collective communications. But they never

integrate GPU knowledge into their MPI, which still move data from GPU memory to host memory for reduction operations, without taking the parallelism feature of GPU to handle large parallel reduction operations. Later, Chu et. al. [Chu et al. \(2016\)](#) and Oden et. al. [Oden et al. \(2014\)](#) have proposed CUDA-aware reduce operations by leveraging CUDA kernels to handle reduction operations. Similarly NVidia introduces NCCL [NVIDIA \(2016\)](#), which is collective communication library targeted to shared memory multi-GPU platform. Overall, none of them take care of network hierarchical topology of GPU clusters. Awan et. al. [Awan et al. \(2016\)](#) have integrated NCCL into MVAPICH2 to provide hierarchical broadcast operations by using NCCL to handle intra-node communications. However, similar to MVAPICH2 in CPU clusters discussed before, there is no communication overlap between different topology levels. Moreover, they never consider the intra-node GPU locality (PCI-Express level). Our design is the first MPI implementation who is integrated with inter- and intra-node GPU network topology and allows communications overlap of inter and intra levels.

Chapter 3

PaRSEC's Support for Heterogeneous System

The portion of this chapter is drawn from the following publication of mine:

- W Wu, A Bouteiller, G Bosilca, M Faverge, J Dongarra, “Hierarchical dag scheduling for hybrid distributed systems”, Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

3.1 Issues of PaRSEC in Heterogeneous System

Data-flow programming paradigm describes an application as a DAG where nodes represent tasks and edges represent data dependencies between tasks. Deploying a task promptly to a proper execution location is critical to performance of application, therefore, it is preferred to apply a task-based runtime to schedule DAGs to achieve fully exploiting of the computing resources in heterogeneous systems. In the context of linear algebra, DAGs have been demonstrated to be an extremely effective way to describe tiled linear algebra algorithms. PaRSEC, an adoption of data-flow program paradigm, takes this DAG-based representation and schedule tasks efficiently in homogeneous systems. However, when porting them to heterogeneous system to use

both CPU and GPUs efficiently, there are several issues caused by the hardware differences and separated memory space between GPU and CPU. This chapter describes how we solve these issue and achieve high occupancy of both CPU and GPUs.

3.1.1 Data Granularity of CPU/GPU Tasks

Tiled linear algebra is a representative class of algorithms that can be expressed efficiently with a data-flow: the parallelism between operations is represented with a DAG that symbolizes the flow of data between several tasks called kernels, which are described as nodes in a DAG. As discussed in Chapter 2.1.3, in tiled linear algebra algorithms, each kernel works on tiles instead of element of matrix. The tile size is a key tuning parameter that affects the efficiency of kernels tremendously. In most linear algebra algorithms, the tile size has been assumed to be constant for all kernels.

In most heterogeneous systems, a computing node features several CPU cores and one or more GPUs. Kernels are executed on CPU cores or GPUs depending on their performance profile and the occupancy on the target execution unit. Compared with CPU cores, a GPU has many more lightweight computing units; hence GPU tasks(kernels) usually require more data parallelism than CPU tasks to achieve high occupancy of GPU as they need to dispatch computation on many individual cores. The optimal data granularity of GPU tasks is larger than CPU. In tiled linear algebra algorithm, such data granularity is described as tile size. Therefore, GPU kernels reach their optimal efficiency when using larger tile sizes; on the other hand, CPU cores often reach good efficiency when using moderate or small tile sizes. Figure 3.1 shows the performance of the SGEMM (real single precision general matrix-matrix multiplication) kernel on different environments varies by tile size.

When running on a 8 cores Intel Nehalem Xeon E5520 CPU, the best CPU implementation of SGEMM (Intel MKL) reaches its peak performance starting from problem sizes larger than 200; while in the best GPU implementation of SGEMM

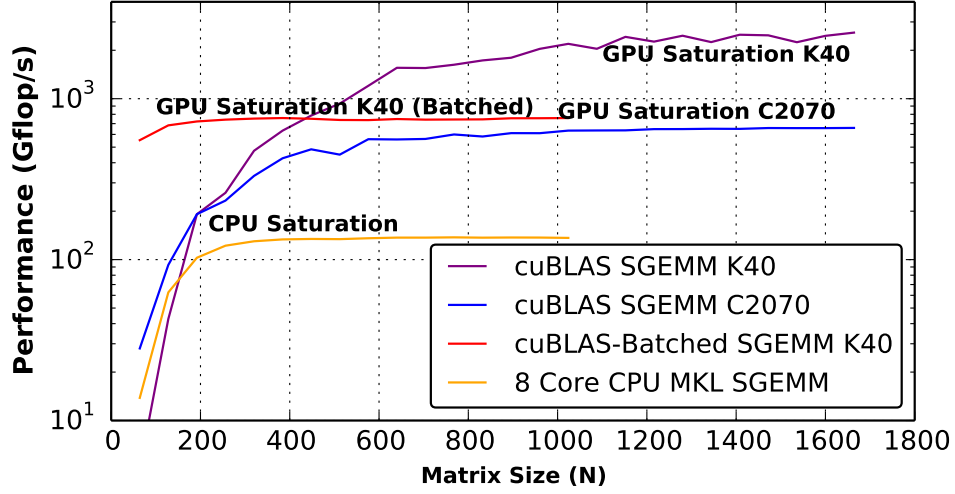


Figure 3.1: Performance of compute kernels on CPU and GPU depending on problem granularity

(cuBLAS), the optimal problem size is larger than 1000 on a Fermi C2070, and larger than 1500 on a Kepler K40. When problem size (matrix size) is fixed, the total number of tasks is directly depended on tile size. Therefore, in a heterogeneous system, selecting a optimal tile size becomes a dilemma:

- If small tile size (optimal for CPU) is used, GPU kernels can not achieve fully utilization of the GPU computing resources since the small problem size cannot efficiently span over all GPU execution units.
- If large tile size (optimal for GPU) is used, given a certain matrix size N , the amount of exploitable parallelism of a DAG is limited by the number of tiles, directly depending on the tile size (N/B). Therefore, when problem size is fixed, increasing the tile size proportionally decreases the parallelism. Furthermore, certain kernels (especially memory bound kernels) are less efficient than their functionally equivalent decomposition into smaller but more compute bound kernels. Executing these large kernels is thereby adding synchronous choke points that delay the execution of other dependent kernels, further decreasing the occupancy of all compute resources.

Traditional solutions used a trade-off approach [Bosilca et al. \(2011\)](#) by choosing an intermediate tile size, larger than the CPU optimal, but smaller than the GPU optimal. Clearly, this trade-off solution is not able to maximize the usage of both the CPU and GPU computing resource. Another solution is to use small tile size for all tasks, and batch execution of GPU tasks by using some batched libraries, i.e. Nvidia introduces batched cuBLAS [NVIDIA \(2016a\)](#) which is able to batched launch a group of GPU kernels of small sizes, to have better GPU occupancy than regular cuBLAS library. However, as shown in [Figure 3.1](#), batched cuBLAS only performs well for very small tasks, otherwise, no matter how many tasks are batched, it is not able to get the best GPU utilization as regular cuBLAS. To address the issue of tile size disagreement of tasks in CPU and GPU, we propose a new solution called “hierarchical DAG”, in which the tile size decomposition varies depending on the target unit executing the task, a decision taken dynamically based on the available parallelism. We describe the details in [Section 3.2](#).

3.1.2 Different Memory Spaces

Since GPU and CPU have different memory spaces, in order to execute a GPU task, it requires developers to move data into GPU memory prior launching GPU kernels, and later move data back to CPU memory after kernel is finished. A easy way to port conventional task-based runtime to heterogeneous systems is to serialize data transfer with GPU kernels. However, limited by the network bandwidth between host and GPU, such data movement is expensive and would alleviate the advantage of high performance GPU kernels. Therefore, the knowledge of different memory space should be integrated into task runtime to provide the capability of overlapping GPU kernels with data transfers, which is discussed in [Section 3.3](#).

Some tasks such as matrix-matrix multiplications have high demand of data (it requires pulling 3 input matrix from host memory, and pushing back a output matrix), hence it is unlikely to perfectly overlap GPU kernels execution with data transfers,

resulting sub-optimal performance. Further optimizations should be taken account of to mitigate the traffic over PCI-Express. When deploying tasks in heterogeneous system, tasks can be either run on CPU or GPUs. With the DAG representation, a edge between two tasks represents data flows from one task to the other. However such data-flow does not require physical data movements if tasks are running on the same devices. Therefore, it is not necessary to move data in/out of GPU memory for each tasks; Instead, data can be cached in GPU memory for reuse by other tasks to minimize the traffic over PCI-Express. we propose a data coherence protocol to track the data in both CPU and GPUs memory to reduce unnecessary data movement, which is discussed in Section 3.4.

In the current design of GPU, GPU memory is usually built on graphic card. Limited by the size of graphic card, it is not possible to integrate many memory chip on graphic card, hence, the size of GPU memory is not larger than 12 GB, and is much smaller than CPU memory whose size can be easily extended by adding more pieces of memory or replacing existing memory with a larger piece. Therefore, it is likely that entire data of an application can not be fit into GPU memory. When towards exascale machines, the scale of application is larger and larger, hence, task-based runtime should be able to support out of core execution, which allows larger application size than GPU memory. To address this issue, we develop a memory management policy to flush least used GPU data back to CPU memory and reuse the memory for data of further tasks, which is discussed in Section 3.5.

3.2 Hierarchical DAG

To solve the data granularity disagreement of CPU and GPU tasks in heterogeneous systems, we propose “hierarchical DAG” approach to allow tasks running on different execution devices operate on data of different granularities. The hierarchical method described below can be generalized to any number of hierarchies, but for the sake of the explanation we will consider a two levels hierarchy, GPU and CPU. In this

section, we use the case of tiled dense linear algebra to demonstrate it is a efficient tasking model for heterogeneous systems. Meanwhile, we describe how to modify the PaRSEC runtime to support “hierarchical DAG”.

3.2.1 Methodology

Assume the optimal tile size for a GPU is B , and the one for a CPU is a smaller tile size b . B and b can be obtained by running a task operating on single tile in both CPU and GPU and tuning the tile sizes. With “hierarchical DAG” method, the input matrix is divided into $NT \times NT$ tiles of size $B \times B$, and the linear algebra algorithm is represented by a DAG whose data granularity of tasks is B . At the top level, all tasks in the original DAG operate on large tiles, and the corresponding tasks are pushed into queues for scheduling on CPUs or GPUs. When retrieving these tasks from the scheduling queues, a decision algorithm (described in Algorithm 1) is executed. If a task is going to be scheduled for GPU execution, then it is executed directly by calling the GPU kernel functions (as a cuBLAS function). If a task does not map well on a GPU, or GPUs are overloaded with other pending tasks, then the task is scheduled on a CPU core. In such case, the CPU task is called only if the data granularity is bellow b . Otherwise, instead of calling the CPU kernel functions directly on the large tile, the CPU task is split into a finer granularity DAG operating on the smaller tiles whose size is b .

Algorithm 1 Generic TASK_X(A) code in the “hierarchical DAG” approach (b :small tile size).

```

if OnGPU  $||((nbrows(A) < b) || (nbcols(A) < b))$  then
    GPUComputeTaskX( A ) // by calling kernel function
    ReleaseDeps( Task_X, A )
else
    o = CreateDAG( Task_X, A,
                  ReleaseDeps( Task_X, A ) )
    Submit(o)
end if

```

When a large grain task is scheduled onto a CPU core, the “hierarchical DAG” capable runtime decomposes the CPU workload into a finer grain parallelism that is more adequate for this type of execution unit. The creation of fine grain DAGs happen online; no preprocessing or static decomposition is required. The runtime engine creates a local data descriptor, a different view of the input sub-matrix representing the large tile divided into smaller tiles. A new DAG is created to represent the fine grain decomposition of the task’s algorithm applied on these smaller tiles. Tasks operating on large tiles that are scheduled for execution on CPU cores are divided into finer grain tasks operating on $nt \times nt$ tiles of size $b \times b$ ($B = nt \times b$). These fine grain tasks are pushed into the scheduling queues and can be executed on any available CPU core. Upon the completion of the final task in the finer grain DAG, the parent coarse grain task is completed through a callback system added as extra-information to the fine grain DAG: the metadata representing the fine grain DAG is released and the dependent coarse grain tasks are pushed into the scheduling queues. Multiple coarse grain tasks can be decomposed simultaneously and the resultant fine grain tasks scheduled concurrently on the available CPU cores. Overall, the “hierarchical DAG” method is based on a dynamic division of a data-flow into smaller flows, allowing for an increase in the available parallelism (as this has the potential to generate more local tasks), and for a decrease in the task execution time.

3.2.2 Case Study: Cholesky Factorization

As discussed in Chapter 2.1.3, Cholesky factorization is a widely used dense linear algebra routine, which is consisted of 4 types of kernels: POTRF, TRSM, SYRK and GEMM. Figure 2.3 is the DAG representation of Cholesky factorization on matrix 4×4 tiles. In heterogeneous systems, the developer determines which kernels are offloaded onto GPUs. In practice, the implementation of these kernels rely on BLAS libraries (MKL on Intel CPUs, cuBLAS for Nvidia GPUs). We made the choice of offloading onto GPU only the most computationally intensive kernels, respectively

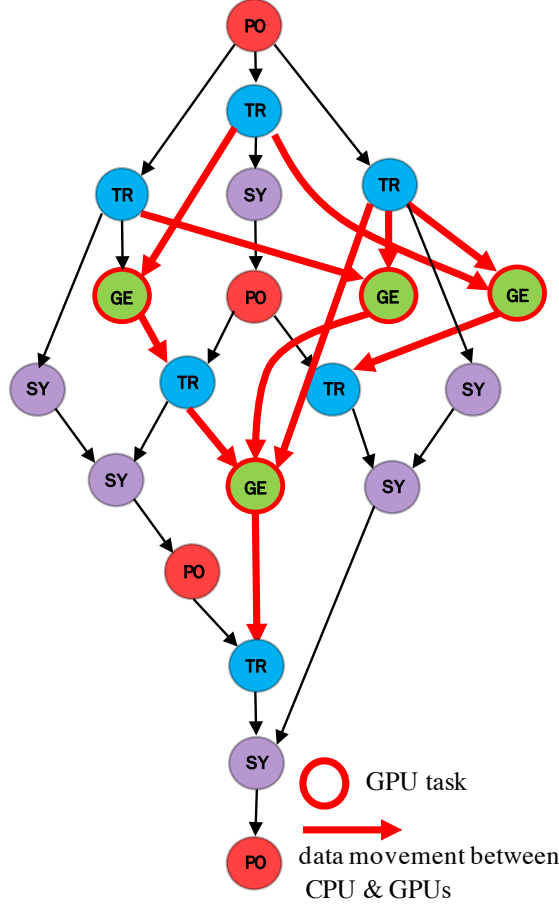


Figure 3.2: DAG representation of Cholesky factorization on matrix of 4×4 tiles

GEMM for the Cholesky factorization. GEMM kernel represents the bulk of the computation time and experience a great speedup when executed on GPU, while the outlook for other kernels is not as favorable. Therefore, in Cholesky factorization, we make our decision to only execute GEMM tasks on GPU, while other tasks stay on CPU. Figure 3.2 presents the DAG of Cholesky factorization in Figure 2.3 by high-lighting GPU tasks. However, GEMM tasks also can be run on CPU cores when GPU is overloaded, according to the load balance strategy discussed in Section 3.6. We now discuss how the Cholesky factorization algorithm can be adapted to take advantage of the adaptive task granularity.

As discussed in Section 3.1.1, tile size is a very important factor to achieve the best performance. Using a large tile size decreases the total number of tasks, increases

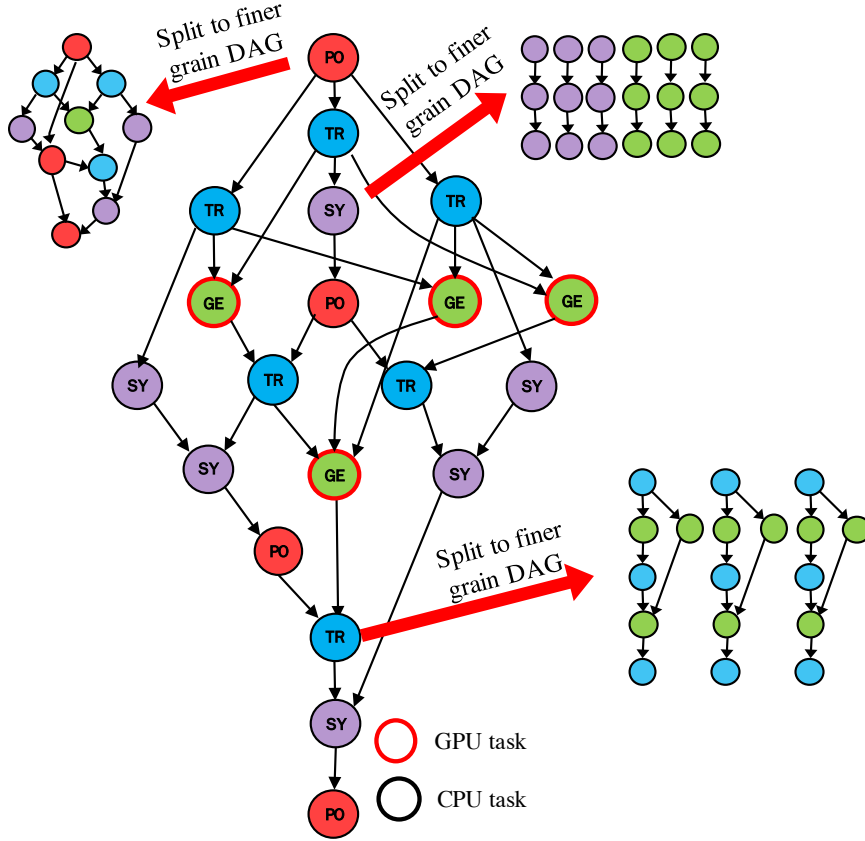


Figure 3.3: DAG of “hierarchical DAG” Cholesky factorization, whose size is 4×4 large tiles and then each CPU task is split into 3×3 small tiles.

the execution time of each CPU task, and therefore delays the release of dependent tasks. The two effects combine to reduce the efficiency of the CPU and generate idle time due to task starvation. When the “hierarchical DAG” approach is applied, the original DAG is dynamically transformed into a new DAG (Figure 3.3) featuring an adapted granularity for both CPU and GPU units. Ideally, all types of tasks should have a fine grain decomposition. In the Cholesky factorization, all four kernels are available in the DPLASMA library as tiled algorithms. When handling POTRF, TRSM and SYRK tasks, a large tile is divided into an $nt \times nt$ tiled matrix whose tile size is $b \times b$, the regular Cholesky factorization kernels can be directly replaced by tiled algorithms version represented by fine grain DAGs, as shown on Figure 3.3 where tiles in coarse DAG are divided into matrices of 3×3 small tiles. PaRSEC

runtime selects a proper execution unit (CPU core or GPUs) for GEMM tasks based on load balance. If a GEMM task stays on a CPU core, similar to other tasks running on CPU, it is split and replaced by a fine grain DAG. If a GEMM task goes to GPU, it is not split, and operates on large tiles. In such way, “hierarchical DAG” achieves tasks running on different devices operate on tiles of optimal sizes, resulting better utilization of both CPU and GPUs than traditional tiled algorithms with unique tile size.

3.2.3 Hybrid Data Layout

In a regular tiled algorithm, data of each tile is stored in contiguous memory (the so called tile layout). When the “hierarchical DAG” approach is applied, tiles used by CPU kernels are treated as a full matrix and a finer grain algorithm is applied on smaller sub-tiles. However, in these sub-tiles, the data layout is not contiguous anymore. Instead, sub-tiles are in the LAPACK data layout, where iterating from one column to the next jumps over a stride. Figure 3.4 shows the resultant hybrid data layout in the “hierarchical DAG” algorithm. We have adapted our tile algorithms to work indifferently on either tile or LAPACK layout, so that our algorithms can be applied directly onto fine or coarse grain tasks. To support “hierarchical DAG” in PaRSEC runtime, We have modified PaRSEC to enable it view data in different layouts for tasks of different hierarchical levels of DAGs. It should be noted that this versatility may come at a performance price since employing the LAPACK layout on small tiles may decrease data locality, but we expect (and demonstrate in the performance section) a profitable trade-off. Another approach would be to perform in-place translation, but this carries a cost of its own, and in the light of the satisfying performance results obtained when operating on LAPACK format directly, we did not pursue such a speculative gain.

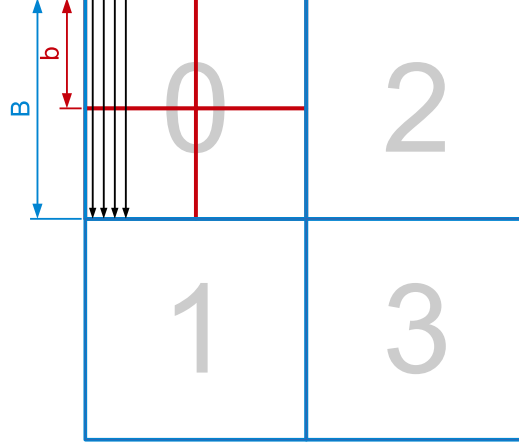


Figure 3.4: Different data layout: tile and LAPACK. For sub-tiles in the fine grain DAG (red), the data layout is the same as the LAPACK layout with interleaved data, while tile layout (blue) is used for large tiles and permits a much more efficient data transfer to/from the accelerators.

3.2.4 Hierarchical DAG Task Scheduler in PaRSEC

In a classical PaRSEC program, creating an instance of a DAG object, which represents the data-flow dependencies of an algorithm, is a collective operation across the entire distributed memory domain. The creation operation generates the local handle that contains the metadata used to track the state of the progress in the data-flow algorithm, but also allocates a unique identifier used to tag the internal messages exchanged between nodes to perform the distributed scheduling and data transfer. In contrast, the fine grain DAG object instances spawned from coarse grain tasks span only the local domain, and do not need to be created collectively across the entire distributed domain. This is an important property, because unlike the creation of the coarse grain DAG object, which happens only once during the initiation, the creation of a fine-grain DAG object happens multiple times asynchronously during the computation. The scheduling between the distributed domains operates on the coarse grain DAG, even without the knowledge of these sub-graphs; thus it can remain unchanged. A new, thread-safe and non-collective DAG object creation operation has been added. It allocates the instance identifier in a local range that never collides

with the global identifiers used for collectively allocated DAG objects. Aside from this initial difference, the local DAG object instances are similar and can be managed concurrently by the same scheduler (with the exception that these tasks must all be scheduled on a shared memory local domain).

3.3 Employing Multiple CUDA Streams

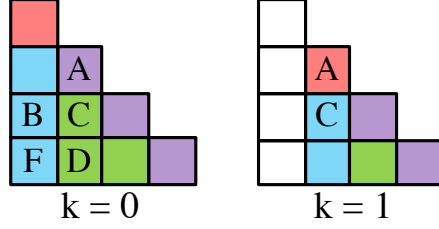
Data transfer between host and GPU memory are expensive, hence, in PaRSEC, we offload data movements and GPU kernels to different CUDA streams to overlap communications with computations. We define the execution of each GPU task as three stages: moving data from the host memory into the GPU memory, kernel execution, and moving data back to the host memory. In order to overlap data movement and kernel execution, each operation type runs in a separate CUDA stream. Since PCI-Express is bidirectional, we reserve one CUDA stream to handle data movement from host to GPU memory and another CUDA stream to handle the opposite direction. A single stream per direction is sufficient to saturate the PCI-Express bandwidth and adding supplementary streams does not improve data movement speed.

GPU streams are also employed to partially circumvent the issues stemming from the conflicting goals of preserving parallelism with smaller tasks and improving per-task GPU efficiency with larger tasks that can employ all execution units of a GPU. By scheduling multiple GPU kernels simultaneously on multiple CUDA streams, the PaRSEC runtime improves the occupancy of the GPU units when moderately sized tasks are submitted: each task employs only a subset of the GPU processing units, but concurrently submitted tasks can employ the unused units. Performance results in Section 3.7.2 demonstrates that even this optimization is insufficient to achieve maximum compute throughput without employing hierarchical DAG.

3.4 Data Coherence between CPU and GPUs

As discussed before, in order to execute tasks on a GPU, data should be moved from CPU to GPU memory, which is an expensive operation. Therefore, to efficiently deploy tasks in heterogeneous systems, task-based runtime should be able to carefully minimize such data movement. PaRSEC minimizes data movement with a careful selection of the computational unit where a task is to be executed, based on the current workload of the unit but also on the cost of moving the data needed for the task execution into the unit memory. In PaRSEC program, data of a matrix tile could have multiple copies, coexisting in different memory spaces of different devices. We developed a data coherence protocol to track the location of these copies by taking the idea of MOESI [AMD \(2010\)](#). A data copy has 5 status: Modified, Owned, Exclusive, Shared and Invalid. Requirement of data movement is determined by checking the transition of data copy status. When dispatching a task to a GPU, PaRSEC runtime checks if data of the task is already available in the target GPU memory, with the effect of reducing the amount of data transiting between the host and GPUs. Figure [3.5](#) shows an example of using data coherence protocol to infer data transfer between host and GPUs. This figure presents the first two steps of Cholesky factorization of 4×4 tiles. Assume the two GEMM tasks of step 0 run on the same GPU and TRSM task of step 1 runs on CPU. Table [3.1](#) shows the transition of data copy status of tiles on both host and GPU memory. By checking the status, the second GEMM task does not need to move tile *A* since the data is already moved in by the first GEMM task; tile *C* is required to be move back to host memory for task TRSM since CPU does not have the most current version of *C*.

PaRSEC overlaps communication with computation by leveraging asynchronous data movement, where data transfer is handled by underlying DMA engines. Hence, when to change the status of data copies becomes a problem: if modifying the status right after issuing a data movement, other tasks who also need this data would consider data is already available in memory while the data could still in transition



GEMM 1 (GPU): RWC; RA & B
GEMM 2 (GPU): RW D; RA & F
TRSM (CPU): RWC; R A

Figure 3.5: Step 0 and step 1 of cholesky factorization of 4×4 tiles. RW refers to Read and Write of data; R refers to Read data

Table 3.1: Status transition of data copies of tile *A* and *C* after each task.(Only tiles *A* and *C* are presented as they are shared accessed by tasks)

Tasks	host Mem	GPU Mem
Init	A:E, C:E	A:I, C:I
GEMM 1	A:S, C:I	A:S, C:M
GEMM 2	A:S, C:I	A:S, C:M
TRSM	A:S, C:M	A:S, C:I

(i.e. the second GEMM could not get the correct tile *A* since its data movement is issued by the first GEMM task, but may not be done yet); if not modifying the status until the completion of data movement, future tasks who need data that is actually under transfer see the invalid data status and would issue another data movement, hence, it would bring unnecessary data movement and increase the traffic of PCI-Express. To solve this issue, we modify the status of data copy right after issuing of data movement, and use extra flags to track the status of the movement. Therefore, in the example above, the second GEMM task can not step to kernel execution until the first GEMM marks the tile *A* as completion of data movement.

Regarding the qualitative aspect of the transfers, PaRSEC also prioritizes the transfer for tasks closer to the critical path of the algorithm. This guarantees that when the main PaRSEC scheduler follows the critical path of the algorithm as closely as possible, the tasks offloaded to an accelerator adhere to the same imperatives.

Overall, with the help of data coherence protocol, PaRSEC infers automatic data movements between host and GPUs, reducing traffics over PCI-Express.

3.5 Out of Core Execution

Allocation of GPU memory (*cudaMalloc*) involves GPU kernel calls, which is an expensive operation. Hence, it is not efficient to use *cudaMalloc* to allocate memory for data of each task; instead, the most common solution is to pre-allocate a large chunk of GPU memory for applications. Some dense linear algebra libraries such MAGMA allocates a large chunk of memory that entire matrix can be fit in. However, limited by size of GPU memory, this method can not support problem size larger than GPU memory, which is called out of core execution. PaRSEC pre-allocates a chunk of GPU memory and manages it as a memory pool. To reduce memory consumption, PaRSEC recycles memory that has been least recent touched for further tasks. Therefore, the size of memory pool can be much smaller than actual problem size. Figure 3.6 presents the GPU memory management strategy of PaRSEC. Based on access mode, data copies of tasks are categorized as two groups: Read and Write, which are managed with LRU (least recent used). Memory in Read LRU has higher priority to be recycled than Write LRU, since data copies in Read LRU have not been modified, hence, is not necessary to be updated back to host memory. In the other side, data copies in Write LRU have to be moved back to host memory when recycling, which is much more costly than recycling memory in Read LRU. To ensure tasks are not delayed by memory recycle procedure, PaRSEC automatically recycles Write LRU when ratio of Write LRU size to entire memory pool size hit to a pre-defined threshold. To enable reusing of data among tasks, each data copies use reference counter to record number of tasks concurrently accessing the data. Reference counter is increased when a new task needs to access it and is decreased when a task is done with the data. Memory recycle only happens to pieces of memory whose reference counter is 0. When there is no memory that can be recycled, PaRSEC

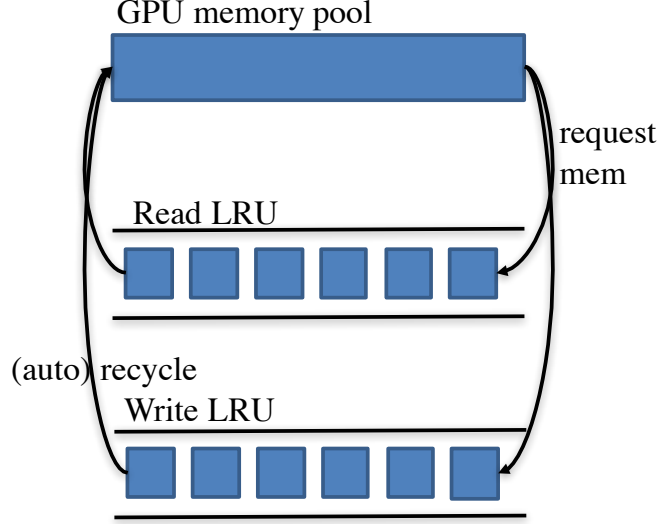


Figure 3.6: GPU memory management strategy in PaRSEC.

suspends tasks and tries to reschedule them in the future. With the GPU memory management strategy, PaRSEC runtime is able to run applications of size much larger than MAGMA, as observed in the Section 3.7.

3.6 CPU/GPU Load Balance

In a complex heterogeneous system, composed by CPUs and GPUs, one additional constraint is to be taken into account. The tasks generated by the algorithm that are distributed on the different computing resources should maintain a balance between the load of the different computing units. Without a load-balance mechanism, the overall computational throughput will decrease as some of the resources will become overloaded while other will starve. In many solutions proposed in the literature, the scheduler is either static with a predefined load distribution, or requires fine knowledge of the duration of each task for each processor type, information we decided to ignore. Instead, our mechanisms are simpler, close to a greedy approach in which we strive to maintain all resources occupied simply based on the current workload of all computing units (CPUs and GPUs).

When scheduling a “hierarchical DAG” program, the runtime load balancing mechanism has two separate levels. The first level separates the workload between CPUs and GPUs at the coarse grain level. Based on the assumption that all tasks of a particular type have a similar duration, and that the driving difference between them is the cost of moving the required data to and from a device, the runtime computes the inverse of the theoretical peak performance of a specified device, and uses it as the weight of a task on this device; a device with a higher computing capacity will have a smaller cost per task. When a new task is considered by the scheduler, its cost is computed for each device, and the task is then assigned to the device which has the lowest current workload. However, to minimize data movement, the selection of the GPU execution device is also determined according to the current data locality: we prioritize the placement of the computation on a GPU that already owns most of the data that will be accessed by the task. The second level of load balancing is realized between fine grain tasks executed on CPU cores, where job stealing according to locality proximity is employed to equilibrate the fine grain tasks workload. Using this simple yet efficient workload management, PaRSEC runtime can distribute tasks on different heterogeneous devices and maintain good load balance, as observed in the Section [3.7](#).

3.7 Performance Evaluation

In this section, we investigate the performance of PaRSEC runtime in heterogeneous system. For a fair comparison, both the hierarchical DAG (shown as “h-PaRSEC in the figures of results) and regular tiled (shown as “PaRSEC” in the figures of results) factorizations are implemented using the PaRSEC framework. We also compare our implementation with the state-of-the-art implementation from MAGMA. All the results presented in this chapter use the real double precision Cholesky factorization (DPOTRF), respectively. Experiments are carried out on four systems:

1. Bunsen is a machine with 3 NVIDIA Kepler K40 GPUs (12GB of memory per GPU) and 2 Intel Xeon E5-2650v2 (16 cores total). We use CUDA 5.0.35 and the Intel compiler 2013.4.183 (includes MKL BLAS).
2. Dancer is an FDR Infiniband small cluster. Each node is equipped with 1 NVIDIA Fermi C2050 GPUs (4GB of memory per GPU) and 2 Intel Xeon E5520 (8 cores total). We use CUDA 5.5 and the Intel compiler 2013.4.183 (includes MKL BLAS).
3. Keeneland Full Scale (KFS) is an FDR Infiniband cluster. Each node is equipped with 3 NVIDIA Fermi M2090 GPUs (6GB of memory per GPU) and 2 Intel Xeon E5-2670 (16 cores total). We use CUDA 5.5 and the Intel compiler 14.0.1 (includes MKL BLAS).
4. Titan is an FDR Infiniband cluster. Each node is equipped with 1 NVIDIA Kepler K20 GPUs (6GB of memory per GPU) and 16 cores AMD Opterons (2 cores share a floating-point unit, so only 8 cores are used). We use CUDA 6.0 and the GNU compiler and BLAS from Cray libsci.

3.7.1 Overhead from Runtime Task Subdivision

In the hierarchical DAG approach, when a task of large data granularity (large tile size B) needs to be executed on the CPU, the task workload is further split into several finer grain tasks (small tile size b), and a temporary, finer grain DAG is created. The initialization of internal PaRSEC objects representing this subdivision happens online, as the execution unfolds, and therefore has a potential to induce management overhead. Figure 3.7 presents the comparison between h-PaRSEC and standard PaRSEC, when running on CPU only. Although an atypical use case scenario for h-PaRSEC, the goal of such an experiment is to emphasize the overhead of DAG subdivision management: in this setup without GPU accelerators, all B sized tasks are subdivided, and all computational kernels eventually execute on tiles of size b ;

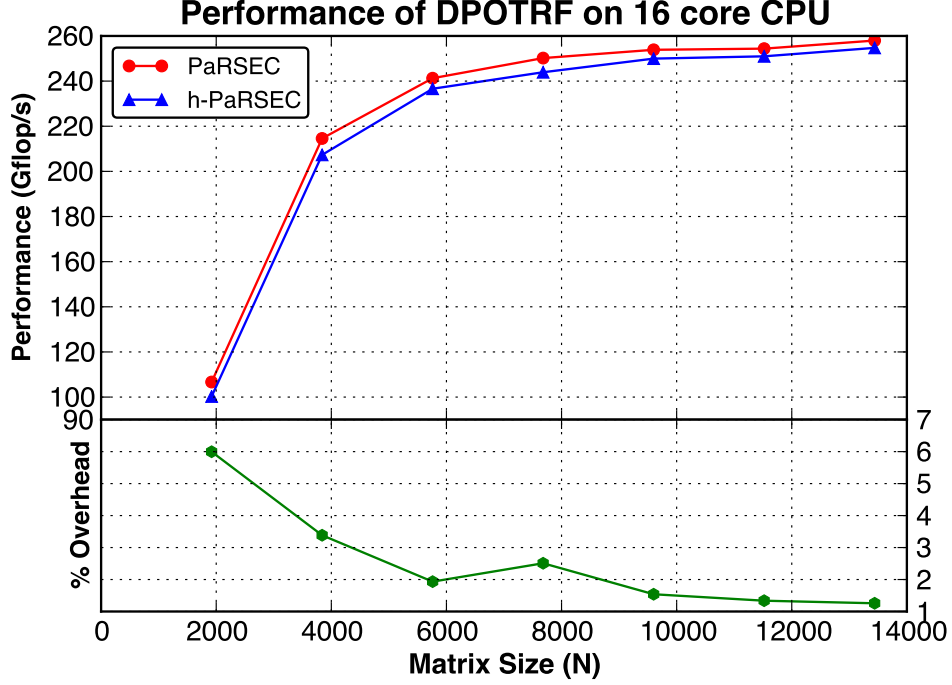


Figure 3.7: Overhead incurred from the hierarchical DAG subdivision management (DPOTRF, CPU only). The h-PaRSEC version uses an big tiling of $B=900$, all tasks are subdivided into small tiles of size $b=180$ (same as standard PaRSEC)

in essence, the only difference with a standard PaRSEC execution comes from the creation and management of DAG subdivisions. As can be observed on the results, both runs outline very similar performance. the h-PaRSEC version is about 4 Gflop/s slower than the standard version. With respect to the overall performance of 260 Gflop/s, this translates into a marginal 1.5% performance overhead, which is easy to overcome when the benefits from using an appropriate task granularity on both CPU and GPU resources is factored in.

3.7.2 Number of CUDA Stream Tuning

CUDA Streams, which represent multiple available execution contexts mapped onto the same physical GPU, can drastically improve the occupancy of GPU units by allowing the device to overlap executions from different streams on all available

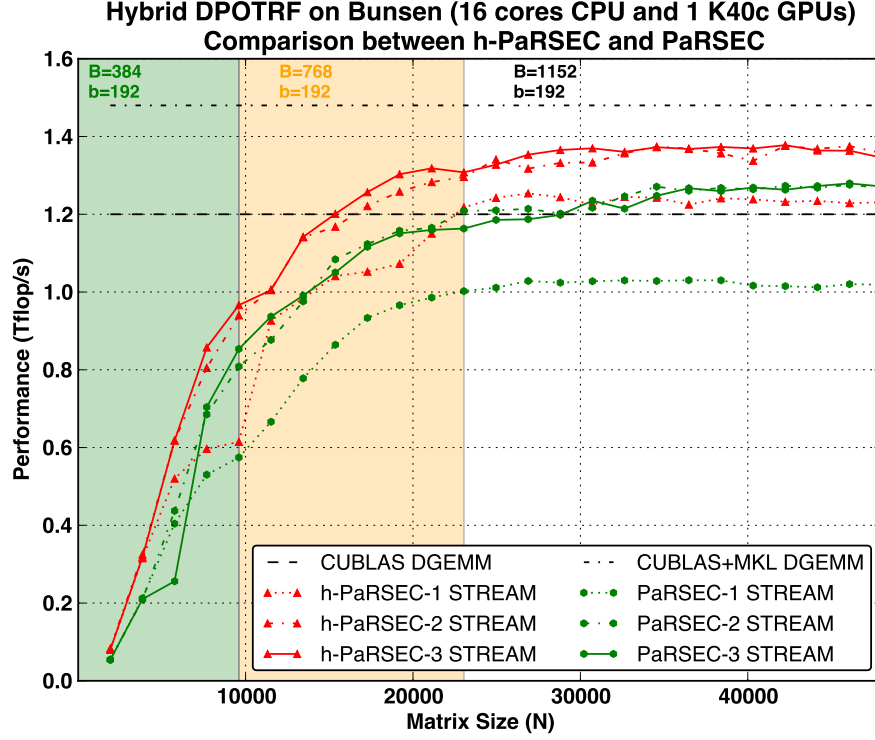


Figure 3.8: Performance difference between hierarchical DAG and the standard version on DPOTRF with a varying number of CUDA streams (Bunsen using 1 K40 GPU).

computational units. The potential for improvement is magnified when executing multiple small grain tasks, as is the case when employing an improperly tuned tile size.

Figure 3.8 presents the performance of the DPOTRF with 1 Kepler K40, when employing a varied number of streams to submit GPU kernels. Employing several CUDA streams improves drastically the throughput of the GPU for both the standard PaRSEC and h-PaRSEC. Using multiple streams has an even greater effect at improving the performance of standard PaRSEC. However, Even with this optimization, the performance of standard PaRSEC can only match that of h-PaRSEC without streams. When CUDA streams are also employed in h-PaRSEC, it outperforms standard PaRSEC for all matrix sizes. Overall, these results outline that multiple streams are not a sufficient optimization to alleviate the need for

employing the hierarchical DAG approach. From the Figure 3.8, it can be observed that the difference between employing 2 or 3 CUDA streams is low. Hence, in later experiments, we always use 2 CUDA streams for kernel execution per GPU.

3.7.3 Tile Size Tuning

Tuning the tile size has traditionally been a difficult issue for linear algebra software [Sawa and Suda \(2010\)](#). In the hierarchical DAG approach, tile sizes of both coarse and fine grain DAGs need to be tuned. Figure 3.9 presents the performance of DPOTRF on the Bunsen machine varies by both the inner (b , executed on CPU) and outer (B , executed on GPU) tile sizes. In the experiment, different matrix sizes ($N=16K, 48K$) are tested to emphasize the impact of the tile size on the amount of available parallelism. Each curve represents a different value for b , for which B varies (on the x-axis). In addition, the performance of standard PaRSEC is also presented (then, the x-axis represents the single tile size used on both GPUs and CPUs). B is set as a multiple of both b and 64 (due to the physical organization of the CUDA warps on Nvidia cards).

On Bunsen, sequential BLAS kernels in Intel MKL executed on the CPU usually obtain their peak performance when b is larger than 180. However, and although GPU kernel performance remains sub-optimal for tile sizes smaller than 1K (see Figure 3.1), the overall performance of standard PaRSEC (gray dash lines) on a heterogeneous platform decreases when increasing the tile size. Two intermingled effects are explaining this phenomenon. First, by increasing the tile size, the number of GEMM tasks in the update of the trailing matrix is reduced, leading to reduced parallelism. Second, the factorization of the panel itself becomes a bottleneck: the associated operations apply to a single column of tiles, yet further progression is conditioned on their completion. With large tiles, panel parallelism is drastically

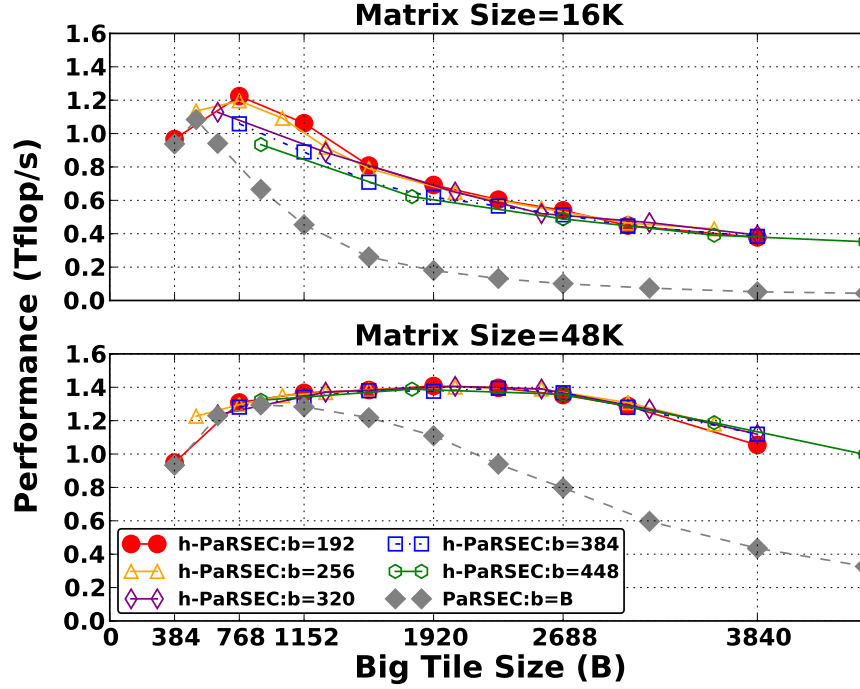


Figure 3.9: Performance for different tile size parameters (DPOTRF, using 1 GPU on Bunsen).

reduced and the more parallel trailing matrix update is delayed, leading to under-utilization of computing resources. As can be seen, this effect persists even for large matrix sizes.

On the contrary, thanks to hierarchical subdivision of tasks into sub-DAGs, h-PaRSEC is much less subject to starvation from lack of parallelism (the panel factorization is divided into many small tasks whose data granularities are adapted to reach peak performance on CPU). Obviously, if the GPU tile size B is set too small (less than 512), the overall performance suffers from poor kernel efficiency. Increasing the value of B delivers the expected performance boost from the compute kernels' efficiency improvement, without suffering as much from lack of parallelism and poor performance on the CPU-executed panel factorization. Another interesting note is, when using the hierarchical DAG approach, finding a value of B that delivers acceptable performance is easier than when tuning for a single tile size. Even for

small matrices that are prone to exacerbate lack of parallelism, the amplitude of performance difference is reduced; while for larger matrices, a very wide band of values deliver more than 90% of the best performing tuning. Developers can select the smallest tile size that maximizes CPU performance as the value for b , and then pick any reasonable multiple (around 1K) to set B . In the remainder of the experiments of DPOTRF, we apply such a tuning, and b is set to 192, while B varies between 384 and 1152 depending on the matrix size.

3.7.4 Shared Memory

Figure 3.10 presents the performance of the DPOTRF on the Bunsen machine with both h-PaRSEC and PaRSEC implementation. In both implementations, the tile size is tuned to perform best for this particular matrix size (the sizes used by h-PaRSEC are illustrated with a background color in the figure, the sizes employed in regular PaRSEC are similarly tuned).

For all matrix sizes, h-PaRSEC always performs better than standard PaRSEC, even for small matrices, when both employ the same tile size for kernels executed on the GPU. In this case, the advantage comes from employing a smaller tile size of 192 for computations executed on CPUs. For larger matrix sizes, h-PaRSEC reaches 1.36Tflops/s for DPOTRF using 1 GPU, which is around 10% faster than standard PaRSEC, demonstrating that when more parallelism is available, higher kernel efficiency gives h-PaRSEC an extra boost.

Since the peak performance of cuBLAS DGEMM on 1 K40 is 1.2 Tflop/s, then based on the performance result from the 1 GPU experiment (1.36 Tflop/s), it can be inferred that CPUs contribute 160 Gflop/s on this platform. Based on these numbers, a perfectly scalable implementation of Cholesky would achieve approximately 2.56 Tflop/s using 2 GPUs and 3.76 Tflop/s using 3 GPUs (the contribution of the CPUs being accounted for only once). In practice, we obtain 2.5 Tflop/s with 2 GPUs and

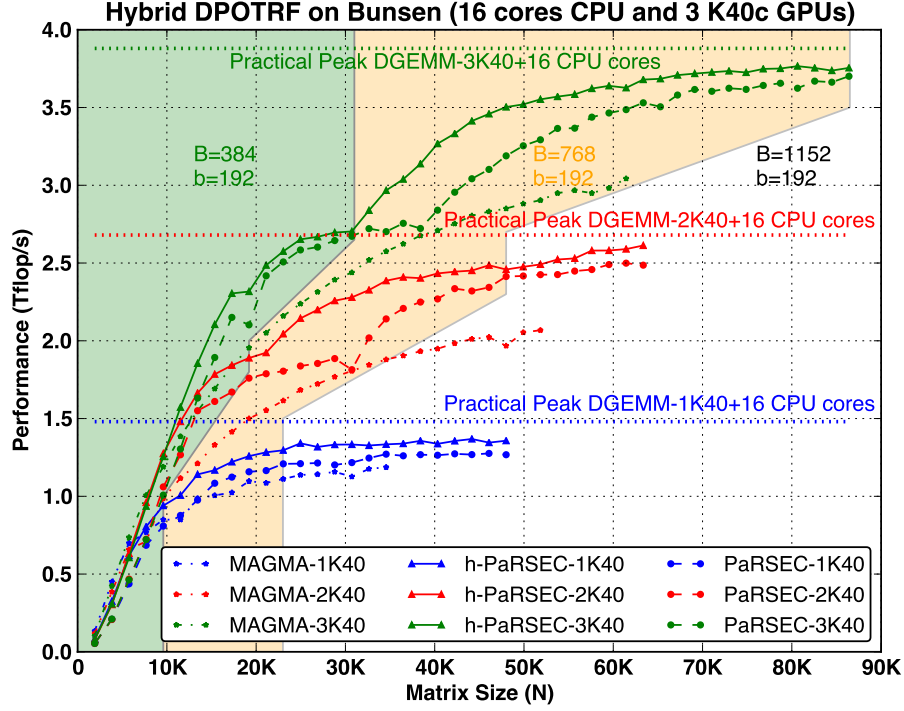


Figure 3.10: Performance of h-PaRSEC DPOTRF with regular PaRSEC and MAGMA.

3.7 Gflop/s with 3 GPUs, which demonstrates the scalability up to 3 GPUs is almost perfect.

Last, Figure 3.10 also presents the performance of the state-of-the-art MAGMA GPU linear algebra package for reference (please note that the MAGMA results do not include the cost of the initial transfer of the dataset to the GPU memory, whereas this cost is implicitly included for h-PaRSEC, when the relevant data are transferred in the background meanwhile computation is progressing). The comparison between MAGMA and h-PaRSEC demonstrates that by retaining a dynamic distribution of tasks, and dynamic load balancing between GPUs, while at the same time improving the efficiency of compute kernels by employing hierarchical DAG subdivision, h-PaRSEC can outperform (as seen for Cholesky) production quality software like MAGMA, whose data distribution and load balancing are static. As discussed in

Section 3.5, MAGMA requires matrix to be fit into GPU memory, while PaRSEC runtime is able to recycle memory for future tasks. Therefore, as seen in Figure 3.10, MAGMA runs out of GPU memory when matrix size is larger than 35K, 52K and 62K, respecting to 1 GPU, 2 GPUs and 3GPUs, and no results are plot.

3.7.5 Distributed Memory

Last, we investigate the performance affection of hierarchical DAG on distributed memory machines. Figure 3.11 3.12 3.13 present the weak scalability performance of h-PaRSEC and standard PaRSEC for the Cholesky factorizations on KFS, Titan and Dancer. In a weak scalability experiment, the problem size is set in accordance to the number of nodes, so that the workload per node keeps constant when increasing the number of nodes. The experiment demonstrates a good weak scalability for both standard PaRSEC and h-PaRSEC. However, as the number of nodes becomes larger, the hierarchical DAG approach shows a better scalability. h-PaRSEC obtains 78% of the ideal scalability on Cholesky factorization (performance at 1 node, multiplied by number of nodes) on KFS, 65% on Titan and 88% on Dancer. When deploying data over $P \times P$ nodes based on 2D block cyclic, for each task, the chance of a particular input data being local is $1/P^2$. When P is very small, many tasks can execute without communications. When p becomes larger and larger, the communication/computation ratio is much lower. Therefore the scalability curve drops at first. However, the effect of varying P for large values of P is negligible.

Now, we investigate the ration of performance to practical peak performance. KFS features 3 GPUs, whose practical GEMM peak performance is around 3 times of its GPU peak. As seen in Figure 3.11, with 64 nodes, h-PaRSEC reaches 59 Tflop/s on DPOTRF, which represents 60.5% of the practical GEMM peak (GEMM performance on 1 node, multiplied by 64). h-PaRSEC performs 10% faster than standard PaRSEC. Although the overall efficiency is not as high as in the shared memory machine, one has to consider that the execution platform is compute over

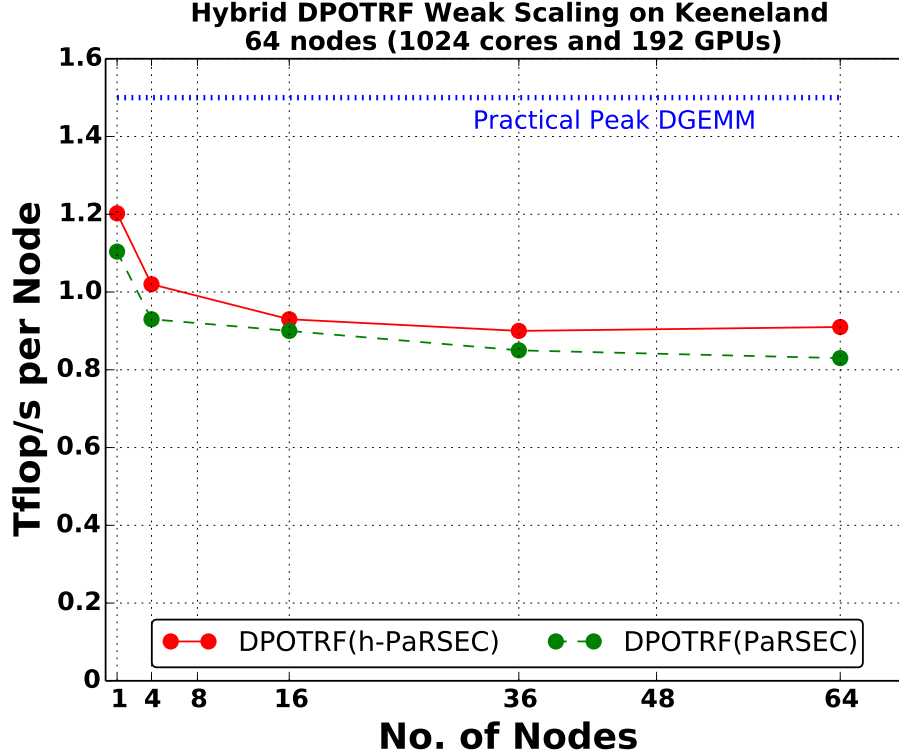


Figure 3.11: Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (KFS, 3 M2090 GPUs and 16 cores per node)

provisioned: Even for compute intensive algorithms such as Cholesky, the Infiniband 40G network is insufficient to feed 3 GPUs. This behavior is customary and can also be observed when comparing the efficiency per core of ScaLAPACK versus LAPACK. Dancer features only 1 GPU, but because its CPU is slow, so the ratio of GPU to CPU performance is also 3. As seen in Figure 3.13, with 8 nodes, h-PaRSEC reaches 2.2 Tflop/s on FPOTRF, which represents 73% of the practical GEMM peak. h-PaRSEC outperforms standard PaRSEC by 15%. The ratio on Dancer is much better than the one on KFS, since there are only one Fermi GPU, and insufficient of network is less significant. However, the number is still less than the one in shared memory. Titan features a fast Kepler K20, but a slow CPU. The ration of CPU to GPU performance is larger than 10. As seen in Figure 3.12, with 256 nodes, h-PaRSEC reaches 128 Tflop/s, which represents 50% of the practical GEMM peak. POTRF and SYRK

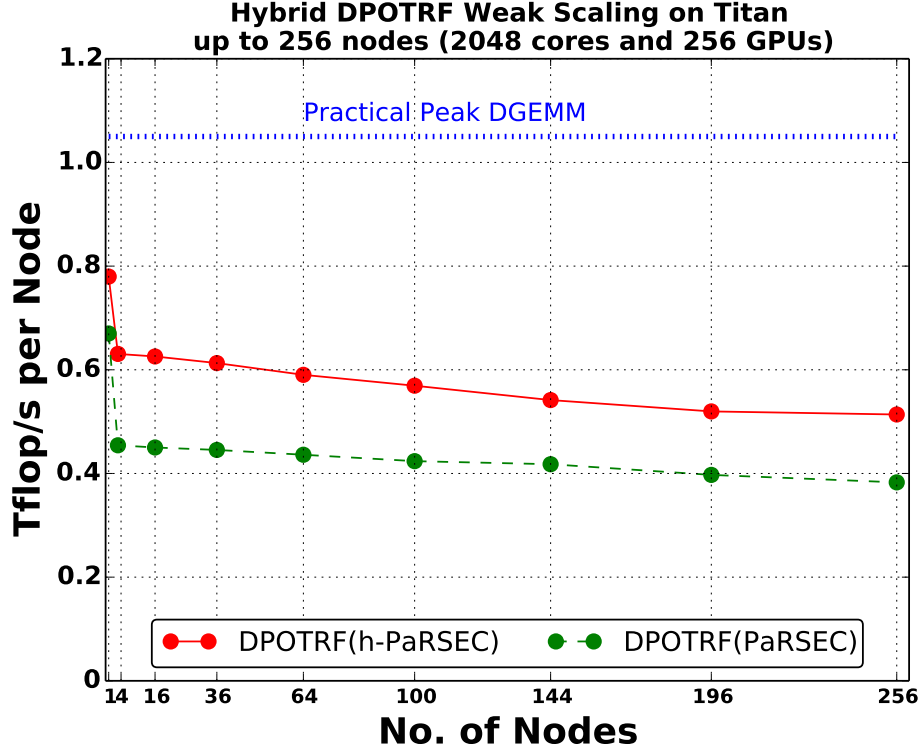


Figure 3.12: Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (Titan, 1 K20 GPU and 8 cores per node)

tasks in the diagonal of matrix are on the critical path of Cholesky factorization algorithm, and are running on CPU. Since the CPU in Titan is much slower than GPU, it is not sufficient to feed GPU tasks, leading to delay execution of GPU GEMM tasks. Therefore, even without the insufficient of network (using 1 node), h-PaRSEC only achieves 75% of practical GEMM peak. This phenomenon can also be observed from the performance of regular PaRSEC. Without hierarchical subdivision of tasks into sub-DAGs, such starvation is more significant in regular PaRSEC, resulting only 39% and 65% of practical GEMM peak performance on 256 nodes and 1 nodes. With the help of hierarchical DAG, h-PaRSEC can split tasks into smaller tasks to provide more parallelism, and somehow promote the execution of critical tasks, leading to 12% faster than regular PaRSEC. Overall the h-PaRSEC strategy better mitigates

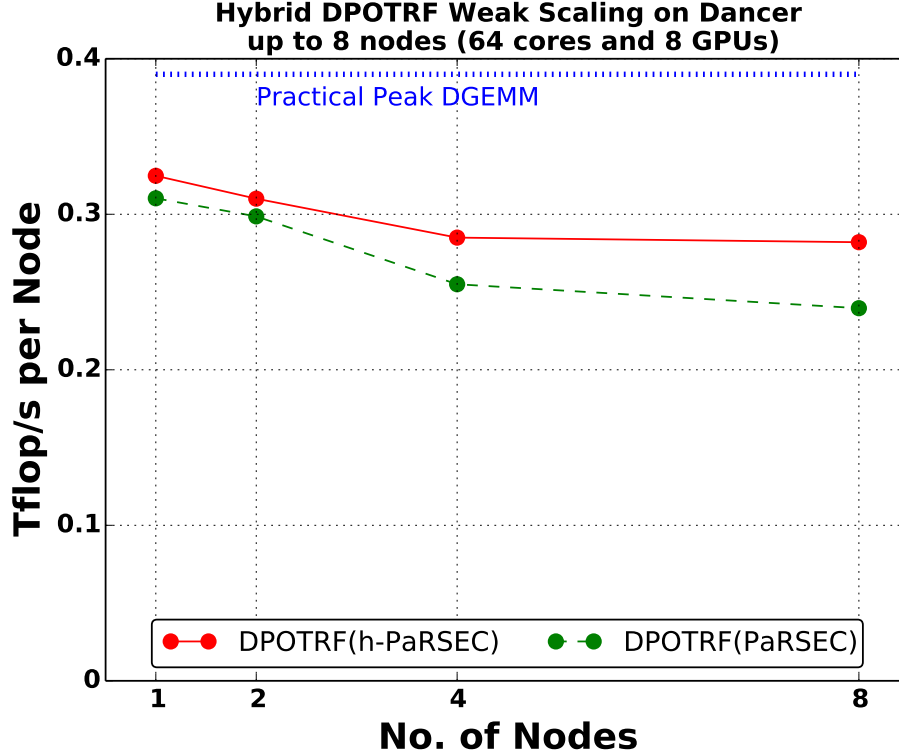


Figure 3.13: Weak Scalability: DPOTRF performance as a function of the number of nodes, with a problem size scaled accordingly (Dancer, 1 C2050 GPU and 8 cores per node)

the heterogeneity within nodes, which translates into a sizable gain on distributed systems.

3.8 Summary

In this chapter, we have extended the PaRSEC runtime to heterogeneous system to maximize the usage of both CPU and GPU resources by the following architecture awareness optimizations. First, we have proposed a “hierarchical DAG” approach, which is able to dynamically adjust the data granularity of tasks, leading to better occupancy on GPU while providing enough parallelism for CPU execution. Second, we have overlapped the data movement between CPU and GPU memory with task executions on GPU by offloading communications and computations on multiple

different CUDA streams. Third, we have presented a software data coherence protocol to track the data copies on both CPU and GPU memory, in order to minimize the data movement by reusing data in GPU memory. Last, we have designed a multi-level GPU memory management strategy to support applications whose required data size is larger than GPU memory size by reusing GPU memory. We have evaluated the impact of all the optimizations described above with an application called Cholesky factorization on both shared and distributed memory machines. Experiment results have demonstrated our optimizations are able to make PaRSEC to utilize both CPU and GPU much more efficiently than previous work.

Chapter 4

GPU-aware Point-to-point Communication

The portion of this chapter is drawn from the following publication of mine:

- W Wu, G Bosilca, R Vandevaraart, S Jeaugey, J Dongarra, “GPU-Aware Non-contiguous Data Movement In Open MPI”, Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, 2016

4.1 Issues of Point-to-point Communication of non-contiguous GPU Data in Open MPI

Message passing paradigm is another widely used program model in high performance area, which is more generic than data-flow program paradigm. MPI point-to-point (p2p) communication emphasizes transit a message between a pair of processes, and is the basic building block of higher level communication routines such as collective operations. Therefore, MPI p2p is critical to overall performance of MPI applications. When towards heterogeneous system, p2p communications involve not only data communication between CPU memory, but also between data of GPU

memory. It is urgent to providing CUDA-aware MPI by integrating the knowledge of GPU into MPI runtime to fully utilize GPU hardware feature and advanced communication techniques such as GPUDirect RDMA. Recently, some state-of-the-art MPI implementations such as Open MPI and MVAPICH already provide some levels of GPU support to enable transparent data movement between processes even if data is in GPU memory, avoiding explicitly data movement between host and GPU memory prior to using MPI routines. However, none of them is able to efficiently transit non-contiguous data. MPI derived datatype gives one the capability to define contiguous and non-contiguous memory layouts, allowing developers to reason at a higher level of abstraction, thinking about data instead of focusing on the memory layout of the data (for the pack/unpack operations). Therefore, extending the same datatype support to GPU data is extremely important for efficient programming in heterogeneous systems.

Current networks are bandwidth-oriented instead of latency-oriented, and fewer large messages provide better network bandwidth. Thus, in the context of non-contiguous data transfers, instead of generating a network operation for each individual contiguous block from the non-contiguous type, it is more efficient to pack the non-contiguous data into a contiguous buffer, and send less – but larger – messages. The same logic can be applied when data resides in GPU memory. In heterogeneous system, GPU hardware features and memory space difference lead to several possible solutions for non-contiguous GPU data communications; the four solutions presented in Figure 4.1 are usually employed:

1. Copy the entire non-contiguous data including the gaps from device memory into host memory. Accordingly, the data in host memory retains the same memory layout as the original, and the traditional CPU datatype engine can handle the pack/unpack operations. This solution provides good performance for memory layouts with little gaps, but cannot be generalized since it wastes a large amount of host memory for the intermediary copies, and has a potential degree of

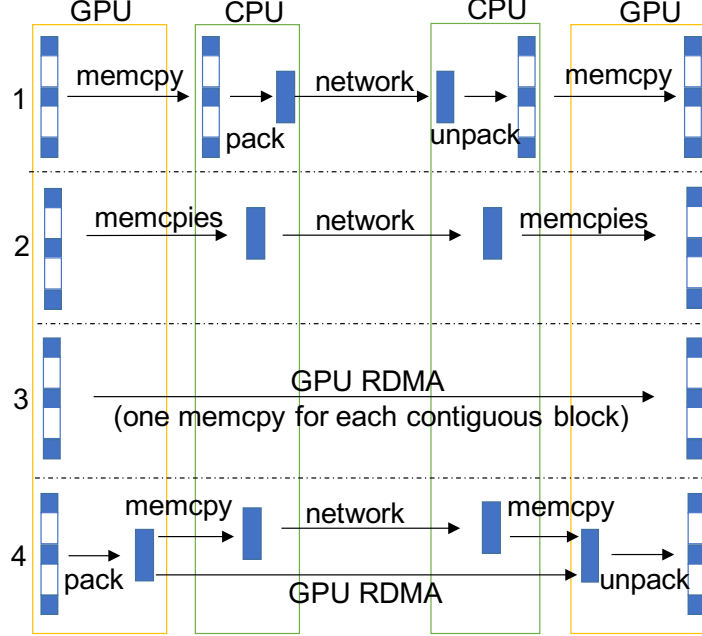


Figure 4.1: Four possible solutions for sending/receiving non-contiguous data residing in GPU memory.

parallelism bounded by the CPU parallelism instead of taking advantage of the computational power of the GPU.

2. The second solution is the one used in Open MPI, which issues one device-to-host memory copy (*cudaMemcpy*) for each piece of contiguous data, packing the data into a single, contiguous buffer. Once packed, the resulting contiguous buffer is sent using a traditional approach. The receiver will also generate the required host-to-device memory copies to scatter the temporary contiguous buffer into the expected locations in device memory. The overhead of launching lots of memory copies degrades performance. Moreover, a memory copy of each small block of contiguous data is not able to utilize the bandwidth of PCI-Express even with the help of multiple CUDA streams. Hence, the performance of this approach is limited.
3. A small improvement upon the second solution, instead of going through host memory, it issues one device-to-device memory copy for each piece of contiguous

data, and directly copies data into the destination device memory. Similar to the previous solution, this alternative suffers from the overhead of launching too many memory copies and the low utilization of PCI-Express. Also, this solution only works when the peers have identical memory layouts and the hardware supports direct device-to-device copy.

4. The last solution is to utilize the GPU to pack and unpack non-contiguous data directly into/from a contiguous GPU buffer. Then the contiguous GPU-based buffer can either be moved between GPUs with hardware support, or – in the worst case – through the host memory.

Among all of the above solutions, we believe the last to be the most promising. From the hardware perspective, GPU has many light-weight cores and significantly larger memory bandwidth than CPU, which might be beneficial for GPU packing/unpacking as these operations can be made embarrassingly parallel (discussed in Section 4.2). Since the kernel is offloaded into the GPU while the CPU is mostly idle (in an MPI call), it also provides the opportunity to pipeline pack/unpack with send/receive (discussed in Section 4.3). From the memory space perspective, packed GPU data is moved to destination process by either going through host memory or not according to different hardware configurations. The 4th approach can be easily adapted to any hardware configuration: if GPUDirect is supported, we can bypass the host memory and use network RDMA capabilities, otherwise the copies to/from host memory can also be integrated in the pipeline, providing end-to-end overlap between pack/unpack and communications. In this chapter, we present the design of non-contiguous GPU data communication based on the 4th approach, taking advantage of CUDA’s many core capability and pipeline techniques to maximally the overlap between pack/unpack operations and communications.

4.2 Design of GPU Datatype Engine

In Open MPI, a datatype is described by a concise stack-based representation. Each stack element records type-specific parameters for a block, such as the number of contiguous elements in the block, the displacement of the first element from the beginning of the corresponding stack frame, and the number of blocks to be packed/unpacked. The most straightforward way to provide datatype support for GPU data would be to port the original (CPU-based) datatype engine into the GPU. However, porting the datatype stack to execute the pack/unpack operation on the GPU generates too many conditional operations, which are not GPU friendly. Thus, in order to minimize the branch operations executed by the GPU, we do not use stack-based representations for GPU datatype, but design two representations (one is for vector like shape, the other is more generic), which are suitable for parallel processing in GPU.

4.2.1 Vector Type

Other than *contiguous* datatype, *vector* is the most regular and certainly the most widely used MPI datatype constructor. A *vector* type is described by blocklength and stride, where blocklength refers to the number of primitive datatypes that a block contains, and stride refers to the gaps between blocks. In our GPU datatype engine, we developed optimized packing/unpacking kernels specialized for a vector-like datatype. The pack/unpack is driven by CPU. The pack kernel takes the address of the source and the destination buffers, blocklength, stride, and block count as arguments, and is launched in a dedicated CUDA stream. The operation is considered complete after a synchronization with the stream. The unpack kernel behaves similarly to the pack kernel.

While accessing global memory, a GPU device coalesces loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Figure 4.2 shows the memory access pattern of GPU packing and unpacking kernels,

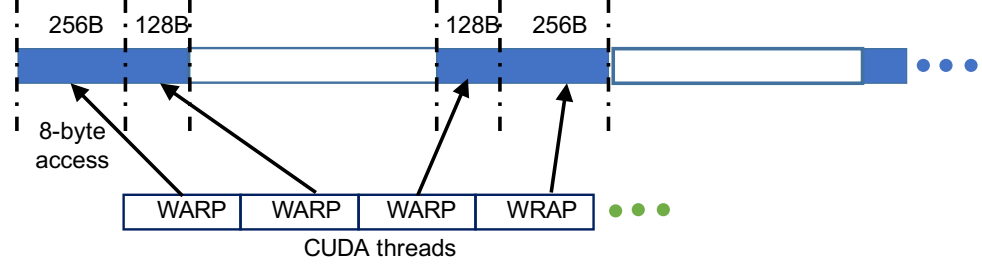


Figure 4.2: Access pattern of GPU pack/unpack kernels of *vector* type. The size of a CUDA block is a multiple of the warp size.

forcing coalesced CUDA threads to access contiguous memory. Since device memory is accessed via 32-, 64-, or 128-byte memory-wide transactions [NVIDIA \(2016b\)](#), in order to minimize memory transactions, each thread theoretically should copy at least 4-bytes of data (128 bytes / 32 threads per warp). In our kernel, we force each thread to copy 8-bytes of data to reduce the number of total loops of each thread. In the case that data is not aligned with 8-bytes, the block is divided into 3 parts: the prologue and epilogue sections follow the original alignment, while the middle one follows the 8-byte alignment.

4.2.2 Less Regular Memory Patterns

Datatypes other than *vector* are more complicated, and cannot be described in a concise format using only blocklength and stride, and instead require a more detailed description including the displacement. However, one can imagine that any type can be described as a collection of vectors, even if some of the vectors have a count of a single element. Thus, it would be possible to fall back on a set of vector-based descriptions, and launch a vector kernel (similar to [4.2.1](#)) for each entry. This design is unable to provide good performance as many kernels need to be launched, overwhelming the CUDA runtime.

Instead, we propose a general solution by re-encoding a representation of any complex datatype into a set of work units with similar sizes as shown in [Figure 4.3](#) by picking a reasonable work unit size. As described above, each entry is identified by a

tuple $\langle \textit{source displacement}, \textit{destination displacement}, \textit{length} \rangle$ named *cuda_dev_dist*. Together with the source and destination buffers, these entries are independent and can be treated in parallel. When entries work on the same length they provide a good occupancy. The incomplete entries can either be delegated into another stream with a lower priority, or treated the same as all the other entries. We choose to treat them equally to the other entries, allowing us to launch a single kernel and therefore minimize launching overhead. In a word, this generic solution can be divided into two stages: first, the host simulates the pack/unpack and generates a list of tuples $\langle \textit{source displacement}, \textit{destination displacement}, \textit{length} \rangle$; the second stage, represented by a kernel executing on a GPU, is using this list to execute – in parallel – as many of these pack/unpack operations as possible. A more detailed procedure for the pack/unpack operations is as follows:

- First, convert the representation of the datatype from stack-based into a collection of Datatype Engine Vectors (DEVs), where each DEV contains the displacement of a block from the contiguous buffer, the displacement of the corresponding block from the non-contiguous data and the corresponding blocklength (the contiguous buffer is the destination for the pack operation, and the source for the unpack).
- The second step is to compute a more balanced work distribution for each CUDA thread. Limited by the number of threads allowed per CUDA block, a contiguous block of data could be too large to use a single CUDA block, resulting in reduced parallelism. To improve parallelism, a DEV is assigned to multiple CUDA blocks. Instead of copying the entire DEV into GPU memory and letting each CUDA block compute its working range, we take advantage of the sequentiality of this operation to execute it on CPU, where each DEV is divided into several *cuda_dev_dist* (called CUDA DEV) of the same size S – plus a residue if needed – and each one is assigned to a CUDA WARP. Similar to the vector approach, each CUDA thread accesses 8-bytes of data each time;

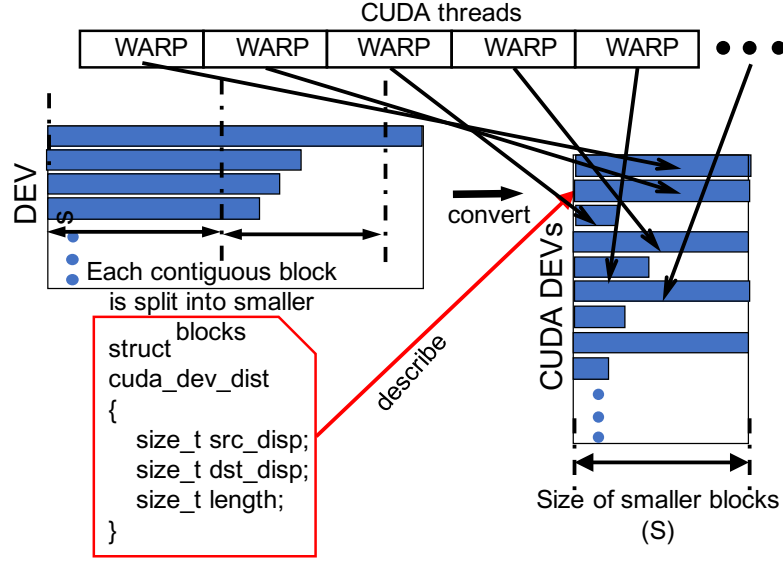


Figure 4.3: Access pattern of GPU pack/unpack kernels using the DEV methodology. The left *struct* describes a work unit for a CUDA WARP.

to fully utilize all threads of a WARP, the size S must be a multiple of 8 times the CUDA WARP size (32). Thus, the lower bound of S is 256 bytes; but since CUDA provides loop unrolling capability, we set the size S to 1KB, 2KB or 4KB to reduce the branch penalties and increase opportunities for instruction level parallelism (ILP).

- Last, once the array of CUDA DEVs is generated, it is copied into device memory and the corresponding GPU kernel is launched. When a CUDA block finishes its work, it would jump N (total number of CUDA blocks) on the CUDA DEVs array to retrieve its next unit of work.

Since any datatype can be converted into DEV, this approach is capable of handling any MPI datatype. However, without a careful orchestration of the different operations, the GPU idles when the CPU is preparing the CUDA DEVs array. To improve the utilization of both GPU and CPU, we pipeline the preparation of the array and the execution of the GPU kernel: instead of traversing the entire datatype, the CPU converts only a part of the datatype, then a GPU kernel is launched to

pack/unpack the converted part into a dedicated CUDA stream. The CPU can then continue converting while the GPU is executing the pack/unpack kernel. As the CUDA DEV is tied to the data representation and is independent of the location of the source and destination buffers, it can be cached, either in the main or GPU memory, thereby minimizing the overheads of future pack/unpack operations.

4.3 Integration of GPU Datatype Engine into Open MPI

This section describes how we integrated the GPU datatype engine with the Open MPI infrastructure. The Open MPI communication framework – outside the MPI API – is divided into three layers, with each one playing a different role. At the top level, the PML (point-to-point management layer) realizes the MPI matching, fragments, and reassembles the message data from point-to-point communications. Different protocols based on the message size (short, eager, and rendezvous) and network properties are available (latency, bandwidth, RMA support), and the PML is designed to pick the best combination in order to maximize network usage. Below the PML, the BML (BTL management layer) manages different network devices, handles multi-link data transfers, and selects the most suitable BTL for a communication based on the current network device where messages go through.

The lowest layer, the BTL (byte transfer layer), is used for the actual point-to-point byte movement. Each BTL provides support for a particular type of network (TCP, shared memory, InfiniBand, Portals, uGNI and so on), and mainly deals with low level network communication protocols where the focus is on optimally moving blobs of bytes. As different network devices have their own optimal communication protocols, the methodology of GPU datatype engine integration is realized at the level of the network device (the BTL). In this paper, we focus on the shared memory and InfiniBand BTL, and propose support for two types of protocols: RDMA and copy

Taking advantage of GPUDirect, a basic GPU RDMA protocol can be implemented as follows: sender packs a non-contiguous GPU datatype into a contiguous GPU buffer, and then exposes this contiguous GPU buffer to the receiver process. If the synchronization is done at the level of an entire datatype packing, the receiver should not access the data until the sender has completed the pack operation. The resulting cost of this operation is therefore the cost of the pack, followed by the cost of the data movement plus the cost of the unpack. However, if a pipeline is installed between the 2 processes, the cost of the operation can be decreased, reaching the invariant (which is the cost of the data transfer) plus the cost of the most expensive operation (pack or unpack) on a single fragment, which might represent a reduction by nearly a factor of 2 if the pipeline size is correctly tuned. This approach also requires a smaller contiguous buffer on the GPU as the segments used for the pipeline can be reused once the receiver completes the unpack and notifies the sender that its operation on a segment is completed.

The Open MPI’s PML layer is already capable of implementing message fragmentation and can send/receive them in a pipelined fashion. However, applying this pipelining feature directly for PML-based RDMA protocols is costly because PML is the top-level layer, and pipelining in this layer requires going through the entire Open MPI infrastructure to establish an RDMA transfer for each fragment. Starting an RDMA transfer requires the sender to send its GPU memory handle to the receiver for mapping to its own GPU memory space, which is a costly operation. With such an approach any benefits obtained from pipelining will be annihilated by the overhead of registering the RDMA fragments. To lower this cost, we implement a light-weight pipelined RDMA protocol directly at the BTL level, which only proposes a single one-time establishment of the RDMA connection (and then caching the registration).

The implementation of our pipelined RDMA protocol uses BTL-level *Active Message* Eicken et al. (1992), which is an asynchronous communication mechanism intended to expose the interconnection network’s flexibility and performance. To reduce the communication overhead, each message header contains the reference of

a callback handler triggered on the receiver side, allowing the sender to specify how the message will be handled on the receiver side upon message arrival.

Taking advantage of *Active Message* communications, the sender and receiver are dissociated, and they synchronize only when needed to ensure smooth progress of the pack/unpack operations. While the sender works on packing a fragment, the receiver is able to unpack the previous fragment, and then notify the sender that the fragment is now ready for reuse. Once the sender receives the notification from the receiver that a fragment can safely be reused, it will pack the next chunk of data (if any) directly inside. Figure 4.4 presents the steps of the pipelined RDMA protocol. Besides the address of a callback handler for invoking the remote pack or unpack functions, the header in our implementation also contains additional information providing a finer grain control of the pack/unpack functions (such as the index of the fragment to be used). In our RDMA protocol, the packing/unpacking is entirely driven by the receiver acting upon a GET protocol, providing an opportunity for a handshake prior to the beginning of the operation. During this handshake, the two participants agree on the type of datatype involved in the operation (contiguous or non-contiguous) and the best strategy to be employed. If the sender datatype is contiguous, the receiver can use the sender buffer directly for its unpack operation, without the need for further synchronizations. Similarly, if the receiver datatype is contiguous the sender is then allowed to pack directly into the receiver buffer, without further synchronizations. Of course, based on the protocol used (PUT or GET), a final synchronization might be needed to inform the peer about the data transfer completion. The more detailed description of the pipelined RDMA protocol is as follows.

- **Sender:** detects if GPU RDMA is supported between the two MPI processes, and requests a temporary GPU-residing buffer from the datatype engine. It then retrieves the memory handle of this temporary GPU buffer, and starts the RDMA connection request providing the memory handle and the shape of the local datatype in a request message. It then waits until a pack request is received

from the receiver. After finishing packing a fragment, an unpack request is sent to the receiver signaling the index of the fragment to be unpacked. In case the GPU buffer is full, or the pipeline depth has been reached, the sender waits until it receives an acknowledgment from the receiver notifying that the unpacking is finished for a particular fragment that can be reused for the next pack. This stage repeats until all the data is packed.

- **Receiver:** upon receiving an RDMA request it maps the memory handle provided by the sender into its own memory, allowing for direct access to the sender's GPU buffer. After the RDMA connection is established, the receiver signals the sender to start packing, and then waits until it receives an unpack request from the sender. After finishing the unpacking of each fragment, the receiver acknowledges the sender, allowing the fragment to be reused. In the case where the sender and the receiver are bound to different GPUs, we provide the option to allow the receiver to allocate a temporary buffer within its device memory and move the packed data from sender's device memory into its own memory before unpacking. In some configurations, going through this intermediary copy delivers better performance than accessing the data directly from remote device memory. When using temporary buffer, receiver is able to acknowledge sender a fragment to be reused right after data is move into the temporary buffer, without waiting for the completion of the unpacking of the fragment. It provides the opportunity to let sender to start packing as early as possible.

4.3.2 Copy In/Out Protocol

In some cases, due to hardware limitations or system level security restrictions, the IPC is disabled and GPU RDMA transfers are not available between different MPI processes. To compensate for the lack of RDMA transfers we provide a copy in/copy out protocol, where all data transfers go through host memory. It is worth noting that

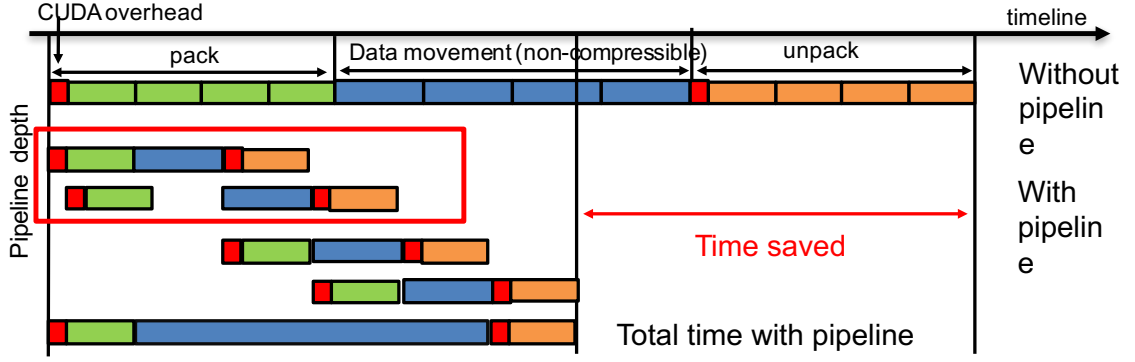


Figure 4.5: Communication time of using pipeline compared with communication without pipeline.

this approach is extremely similar to the case when one process uses device memory while the other only uses host memory.

Open MPI handles non-contiguous datatypes on the CPU by packing them into a temporary CPU buffer prior to communication. When GPU RDMA is not available, we forced Open MPI to always consider all data as being in host memory, and therefore it always provides a CPU buffer even for datatypes residing in device memory. When the datatype engine detects that the corresponding non-contiguous data is actually in device memory, it allocates a temporary GPU buffer (with the same or smaller size than the CPU buffer) for packing. Once this GPU buffer is full, the packed data is copied into the CPU buffer for further processing. This procedure repeats until the entire data is packed. A similar mechanism applies to unpack.

Unlike the RDMA protocol, extra memory copies between device and host memory are required. To alleviate the overhead of such memory transfer, pipelining can also be used by allowing the sender to partially pack the data, fragment after fragment, and allow the receiver to unpack once it receives each packed fragment. Therefore, the pipelining becomes more complex, overlapping packing/unpacking on the GPU, with device-to-host data movement and intra-node communication. Another CUDA capability, zero copy, can be exploited to minimize the memory copy overhead. Instead of using the CPU to explicitly drive memory movement, the CPU buffer is

mapped to GPU memory with the help of CUDA UMA, and then the data movement is implicitly handled by hardware, which is able to overlap it with pack/unpack operations. Overall, as indicated in the experimental Section 4.4, copy in/out protocol is a general solution suitable for most platforms, and delivers good performance – especially once integrated with a pipelined protocol.

4.3.3 Analysis of Pipelining of Pack/unpack with Data Movement

In both RDMA and copy in/out protocols, pipeline technique is applied to overlap pack/unpack kernels on GPU with packed data transfer between processes. Since non-contiguous data communication includes three steps (pack, data movement and unpack), it is necessary to use 3-stage pipeline. Figure 4.5 shows the time line of the 3-stage pipeline compared with communication without pipeline. The time of 2nd stage (data movement) depends on hardware network throughput, thus it is not compressible. Therefore, by overlapping pack/unpack kernels with data movement, the entire time of transition non-contiguous message is time used for packing and unpacking one fragment plus the cost of data movement of entire message, which is much more efficient than communication without pipeline. Pipeline fragment size is an important tuning factors that affect performance. Pipeline fragment size is the size of each fragment to be packed, transferred and unpacked. If fragment size is too big, the cost of pack/unpack one fragment is significant, resulting in increment of overall cost. Fragment size cannot be too small as well. As seen in Figure 4.5, time of pack/unpack kernels is consisted of kernel overhead $T_{koverhead}$ and actual pack/unpack time of one segment (T_{pack}/T_{unpack}). The $T_{koverhead}$ is constant and only related to GPU hardware, and T_{pack}/T_{unpack} varies by fragment size. If fragment size is too small, $T_{koverhead}$ becomes dominated. Moreover, PCI-Express is bandwidth-oriented instead of latency-oriented, and large messages provide better bytes per second transfer rates. Hence, small fragment can not occupy the full bandwidth of

PCI-Express, leading to sub-optimal performance. Overall, for large messages, which is the scope of most GPU applications, the fragment size should be large enough to fulfill the network bandwidth. For modern GPUs, we set the pipeline fragment size to 1MB to 4MB based on networks used (GPUDirect or going through host memory).

As discussed in Section 4.3.1, another benefit of pipeline is to minimize the usage of GPU buffer by reuse it: after the remote process has moved packed data into its own buffer, the packing buffer can be released for reuse by further packing kernels. Limited by GPU memory size, it is important to retain the minimal GPU buffers being used while leave more memory for applications, and still keep the performance beneficial of pipeline. Since pack/unpack operations mainly include intra-GPU data movement, T_{pack}/T_{unpack} reflects the GPU memory bus bandwidth. Time used for data movement over the network ($T_{network}$) depends on the network bandwidth between two processes, which is usually much smaller than GPU memory bus. Therefore, in order to overlap data movement with pack/unpack operations, providing two separate GPU buffer is large enough and take turns to reuse them.

Most GPU applications need to utilize GPU for computations, hence, it is desired to let GPU pack/unpack kernels to occupy GPU cores as few as possible. With pipeline, it provides opportunity to let pack/unpack to use less GPU resources as long as the cost can be hidden by data movement stage. We investigate the minimal resources required in later performance benchmark(Section 4.4.3).

4.4 Performance Evaluation

We evaluate our datatypes packing/unpacking methodology using four types of benchmarks. First, we investigate the performance of the GPU datatype engine. Second, we look at inter-process GPU-to-GPU communication through a non-contiguous data ping-pong test, and compare with MVAPICH2.1-GDR. Third, we figure out the minimal GPU resources required for GPU packing/unpacking kernels to achieve optimal overall performance when communication is engaged. Last, we

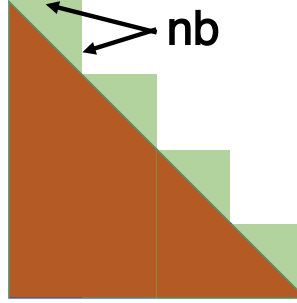


Figure 4.6: Triangular matrix (red one) vs Stair triangular matrix (red and green one), width and height of stair nb is multiple of CUDA block size

analyze the impact on non-contiguous data transfer when access to the GPU resource is limited (the GPU is shared with another GPU intensive application). Experiments are carried out on an NVIDIA PSG cluster: each node is equipped with 6 NVIDIA Kepler K40 GPUs with CUDA 7.5 and 2 deca-core Intel Xeon E5-2690v2 Ivy Bridge CPUs; nodes are connected by FDR IB.

4.4.1 Performance Evaluation for Datatype Engine

In this section, we investigate the performance of our GPU datatype engine by using two commonly used datatypes: *vector* and *indexed*. These datatypes are representative of many dense linear algebra based applications, as they are the basic blocks of the ScaLAPACK data manipulation. More precisely, these types are represented as a sub-matrix and an (upper or lower) triangular matrix.

Considering a sub-matrix with column-major format, each column is contiguous in memory, and the stride between columns is the size of the columns in the original big matrix, which follows the characteristic of a *vector* type (shown as “V” in the following figures). In the lower triangular matrix case, each column is contiguous in memory with a size smaller by one element than the size of the previous column; and the strides between consecutive columns are equal to the previous stride plus 1, which can be described by an *indexed* type (shown as “T” in the following figures). First, we evaluate the performance of our packing/unpacking kernels by measuring GPU

memory bandwidth. Figure 4.7 presents the GPU memory bandwidth achieved from packing these two datatypes into local GPU buffer using our CUDA kernel compared with moving contiguous data of the same size using *cudaMemcpy*. *cudaMemcpy* is already the optimal implementation for moving contiguous GPU data, which can be treated as the practical peak of GPU memory bandwidth. Compared to *cudaMemcpy*, our GPU packing kernel is able to obtain 94% of the practical peak for a *vector* type. The memory instructions in the unpacking kernel are the same as the ones in the packing kernel – but in the opposite direction – and therefore the unpacking kernel delivers the same performance as packing kernels; this is not presented in the figure. For a triangular matrix, each column has a different size, which results in inefficient occupancy of the CUDA kernels; therefore, a GPU packing kernel is only able to achieve 80% of the GPU memory’s peak bandwidth. In order to prove that the bandwidth difference between the sub-matrix and the triangular matrix is indeed from the less efficient GPU occupancy, the triangular matrix is modified to a stair-like triangular matrix (Figure 4.6). Thus, the occupancy issue can be reduced by setting the stair size nb to a multiple of a CUDA block size to ensure no CUDA thread is idle. Sure enough, it is able to deliver almost the same bandwidth as the *vector* type.

After studying the performance of the packing/unpacking kernels, we measure the intra-process performance of packing non-contiguous GPU-resident data to evaluate the GPU datatype engine. Because of the current limitation of GPUDirect, using an intermediate host buffer for sending and receiving over the network is better for large messages than direct communication between remote GPUs in an InfiniBand environment [vandeVaart \(2014\)](#). Thus, studying the case of going through host memory is also necessary. In the following benchmark, one process is launched to pack the non-contiguous GPU data into a local GPU buffer, followed by a data movement to copy the packed GPU data into host memory; and then, the unpacking procedure moves the data from host memory back into the original GPU memory with the non-contiguous layout.

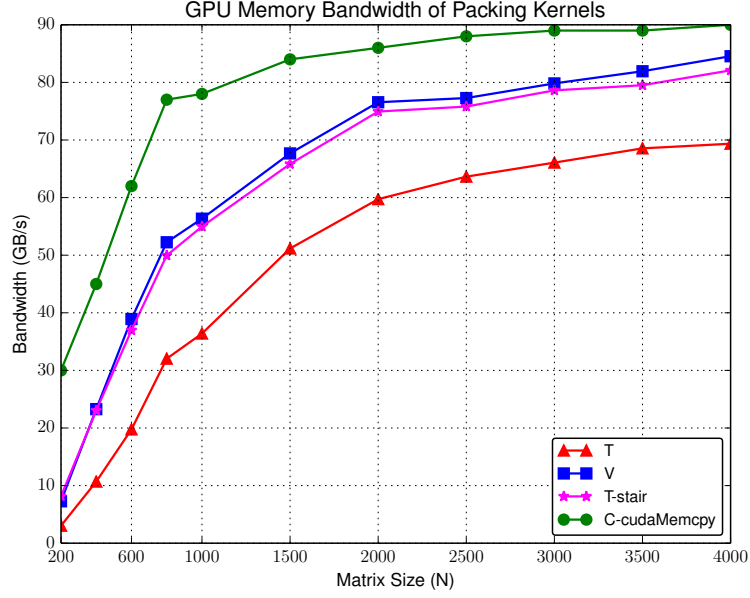


Figure 4.7: GPU memory bandwidth of packing kernels for sub-matrix and lower triangular matrix comparing with contiguous data of the same size. “T” represents triangular matrix, “V” represents sub-matrix, “C” represents contiguous matrix

Accordingly, the time measurement of the benchmarks in this section contains two parts: “d2d” measures the time of packing/unpacking non-contiguous data into/from a contiguous GPU buffer; and “d2d2h” measures the time of packing/unpacking plus the round trip device-host data movements. We also apply zero copy, shown as “0cpy,” to use the CUDA UMA to map the CPU buffer to GPU memory. In this case, the GPU to CPU data movement is taken care of by hardware implicitly. Since zero copy involves implicit data transfer, we are only able to measure its total time without having a separate in-GPU pack/unpack time to show in figures.

Figure 4.8 shows the results of a double precision sub-matrix and lower triangular matrix, with respect to matrix size. From the figure, a number of interesting trends can be observed. First, the pipelining discussed in Section ?? overlaps the preparation of the CUDA DEVs with GPU pack/unpack kernels, almost doubling the performance. If the CUDA DEVs are cached in GPU memory (shown as “cached”), the preparation cost can be omitted; therefore, by caching the CUDA DEVs, the

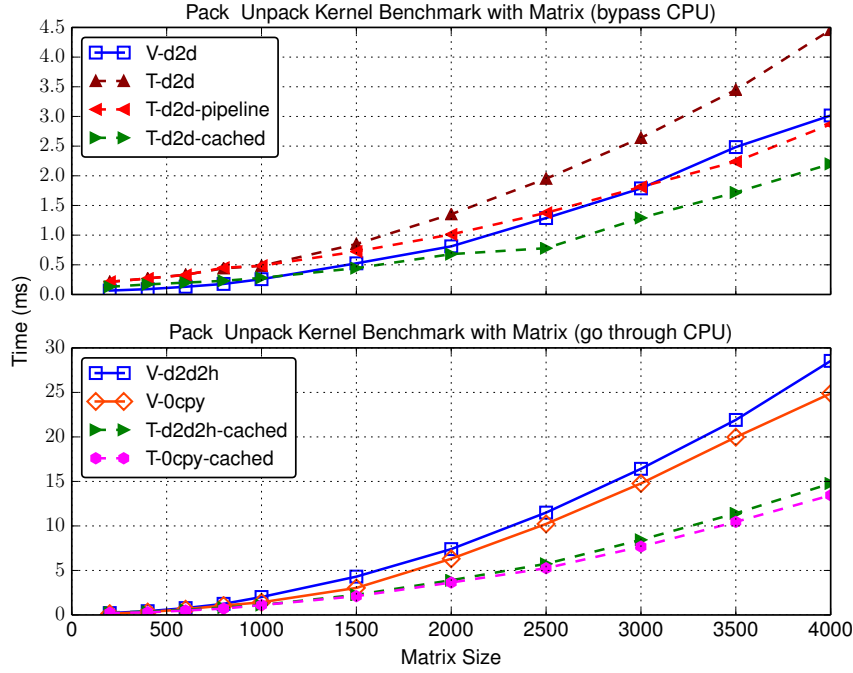


Figure 4.8: Performance of pack and unpack sub-matrix and lower triangular matrix varies by matrix size.

packing/unpacking performance is improved when working on data types of the same format. Second, even though it takes the same time (if CUDA DEVs are not cached) to pack/unpack a sub-matrix and triangular matrix of the same matrix size on a GPU, one must note that the triangular matrix is half the size of a sub-matrix; therefore, compared with a vector approach, the overhead of CUDA DEVs preparation is significant – even with pipelining – which also demonstrates the importance of caching the CUDA DEVs. Since the MPI datatype describes data layout format, not data location, by spending a few MBs of GPU memory to cache the CUDA DEVs, the packing/unpacking performance could be significantly improved when using the same data type repetitively. Third, since zero copy is able to overlap the device-host communication with the GPU kernel, it is slightly faster than explicitly moving data between device and host memory after/before pack/unpack kernels. In all remaining figures, the zero copy is always enabled if going through host memory is required.

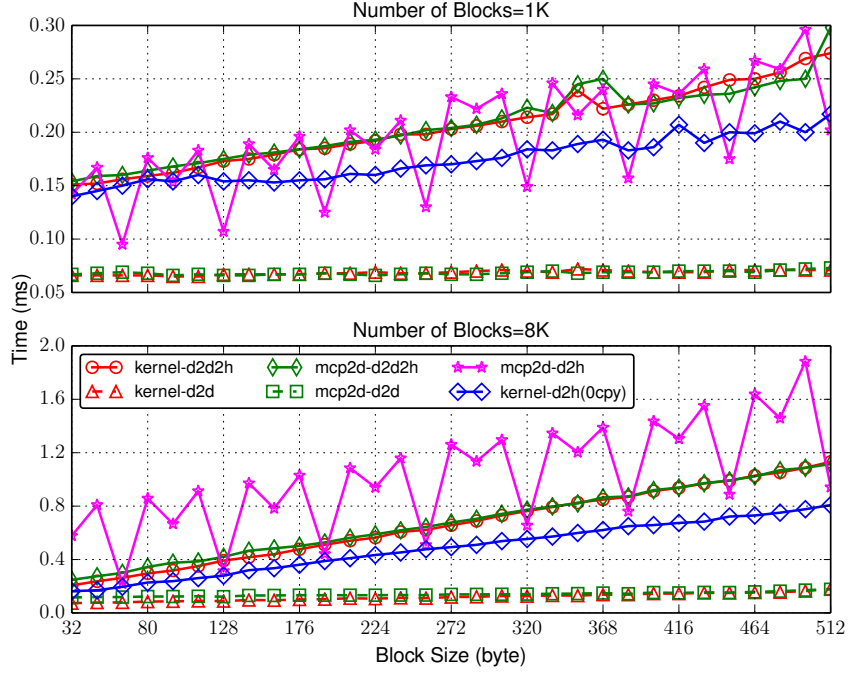


Figure 4.9: *vector* pack/unpack performance vs *cudaMemcpy2D*. “kernel” represents our pack/unpack kernels. “mcp2d” represents *cudaMemcpy2D*. “d2d” represents non-contiguous data packed into a GPU buffer. “d2d2h” represents “d2d” followed by a device-host data movement. “d2h” means non-contiguous GPU data moved directly into CPU buffer.

Alternatively, CUDA provides a two-dimensional memory copy *cudaMemcpy2D* to move vector-like data. Figure 4.9 presents the comparison between our *vector* pack/unpack kernel and *cudaMemcpy2D*, when the numbers of contiguous blocks are fixed at 1000 and 8000, while block size varies covering both small and large problems. Since using our pack kernel to move vector-like non-contiguous GPU data is equivalent to initiating a device to host data movement using *cudaMemcpy2D*, we test it in three ways (device-to-device “mcp-d2d”, device-to-device-to-host “mcp2d-d2d2h”, and device-to-host “mcp2d-d2h”). As seen in the figure, the performance of *cudaMemcpy2D* between device and host memory highly depends on the block size: block sizes that are a multiple of 64 bytes perform better, while others experience significant performance regression – especially when the problem size increases. For

non-contiguous data movement within a GPU, our kernels achieve almost the same performance as *cudaMemcpy2D*. Our DEV pack/unpack kernel is not compared with CUDA since CUDA does not provide any alternative function for irregular non-contiguous GPU data movement.

4.4.2 Full Evaluation: GPU-GPU Communication with MPI

In this section, we evaluate the performance of the GPU datatype engine integration with the Open MPI infrastructure. The performance is assessed using an MPI “ping-pong” benchmark. In a shared memory environment, the RDMA protocol over CUDA IPC is used to avoid extraneous memory copies between host and device. In a distributed memory setting, GPU data goes through host memory for communication. According to [vandeVaart \(2014\)](#), even though the GPUDirect RDMA allows direct intra-node GPU data communication, it only delivers interesting performance for small messages (less than 30KB), which is not a typical problem size of GPU applications. Instead, when pipelining through host memory and overlapping GPU pack/unpack kernels, the GPU-CPU data movement and inter-node data transfer performs better. Therefore, in a distributed memory environment, we always pipeline through host memory. Based on such a setup, packed GPU data always goes through PCI-Express for communication no matter if it is in a shared or distributed memory environment; thus, PCI-Express bandwidth could be a bottleneck of overall communication in a ping-pong benchmark. Similar to last section, we first evaluate the integration of the GPU datatype engine with OpenMPI by measuring PCI-Express bandwidth achieved by *vector* and *indexed* datatypes, comparing data in contiguous format of the same size, with results shown in Figure 4.10. Thanks to the pipeline mechanism discussed in Section 4.3.1, we achieved 90% and 78% of the PCI-Express bandwidth for *vector* and *indexed* types, respectively, by selecting a proper pipeline size.

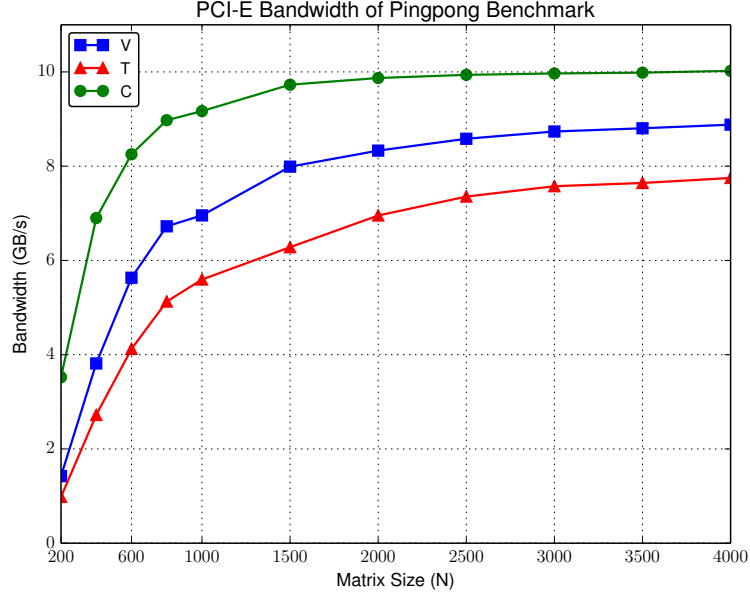


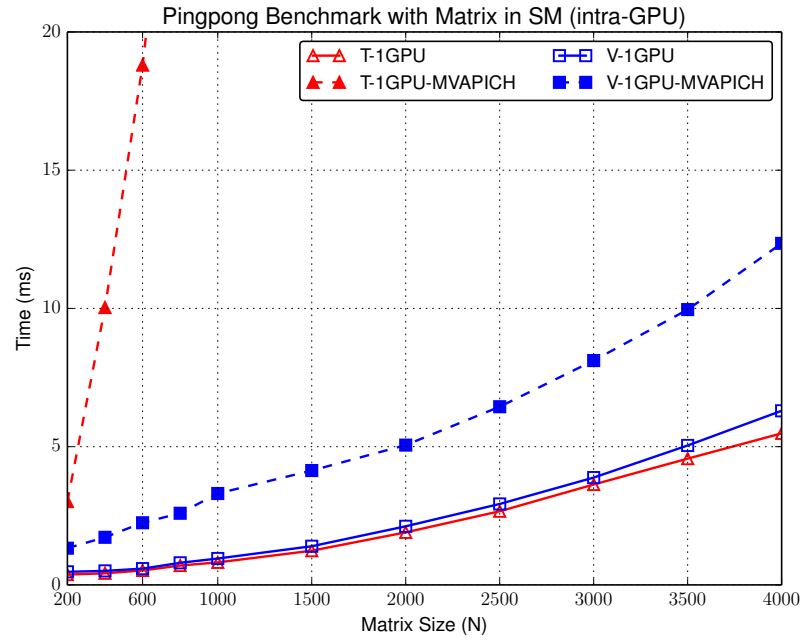
Figure 4.10: PCI-Express bandwidth of vector and indexed data type comparing with contiguous data.

Then, in the following ping-pong benchmarks, we explore both a shared memory (“SM”) and a distributed memory (using InfiniBand “IB”) environment under the following configurations with several commonly used data types, and compare them with the state-of-art MVAPICH2:

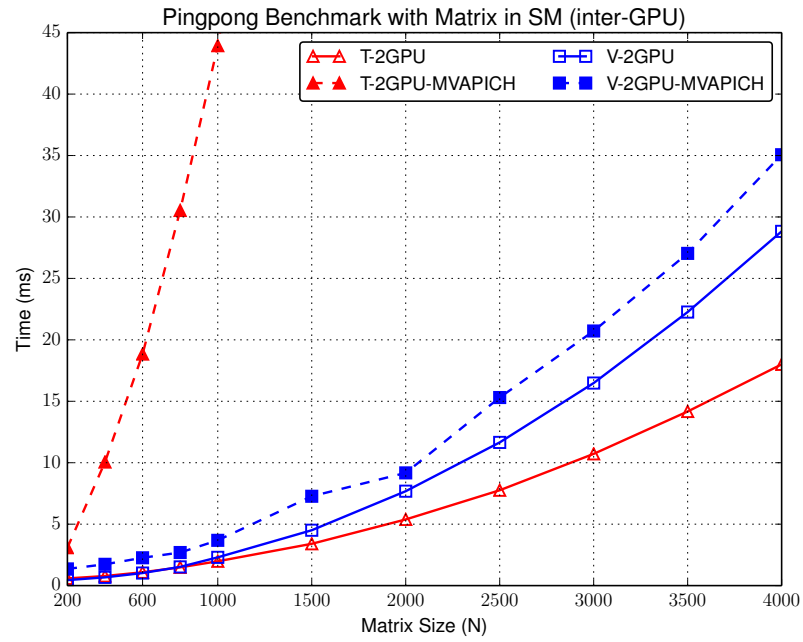
- “1GPU”: both sender and receiver use the same GPU.
- “2GPU”: sender and receiver use different GPUs. Data is sent over network (PCI-Express or InfiniBand) to the receiver process.
- “CPU”: the non-contiguous data is in host memory. This benchmarks the Open MPI CPU datatype engine.

Vector and Indexed Type

Figure 4.11 and Figure 4.12 present the ping-pong benchmark with regard to the matrix size in both “SM” and “IB” environments. As discussed in Section 4.3.1,



(a) Shared Memory Intra-GPU



(b) Shared Memory Inter-GPU

Figure 4.11: Ping-pong benchmark with matrices on shared memory machine. “V” refers to sub-matrix, “T” refers to triangular matrix.

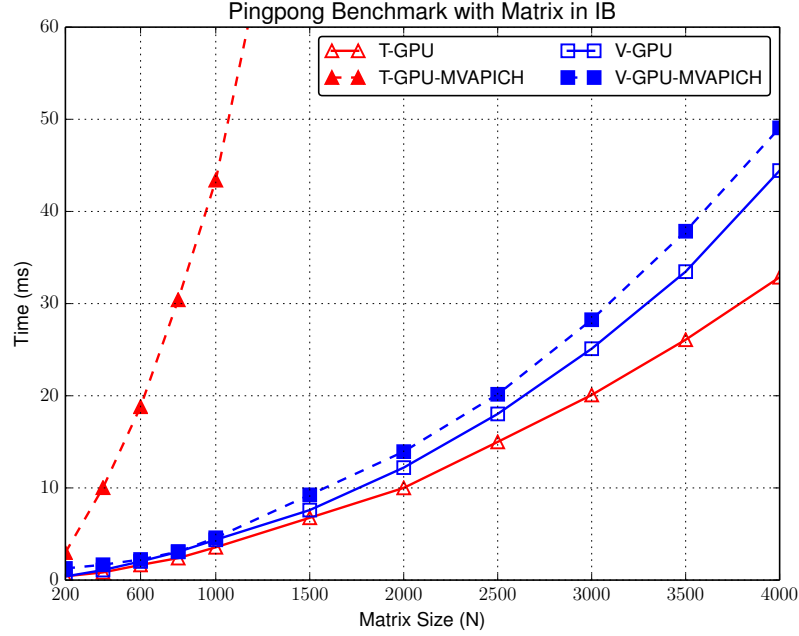


Figure 4.12: Ping-pong benchmark with matrices on distributed memory machine. “V” refers to sub-matrix, “T” refers to triangular matrix.

in the “SM” environment with CUDA IPC support, we provide two options for unpacking in the receiver side: first, the receiver unpacks directly from the packed buffer in the remote GPU memory; second, the receiver process copies the packed buffer into a local GPU buffer prior to unpacking. The first option involves a lot of small chunks of data fetching from remote device memory, generating too much traffic and under-utilizing the PCI-Express. In comparison, the second option groups small data into a big data movement between GPUs, minimizing the traffic on the PCI-Express and becoming faster. Based on our experiment, by using a local GPU buffer, the performance is 5-10% faster than directly accessing remote GPU memory; so limited by the space, we always use the second option in later benchmarks. The “1GPU” case omits the data movement between GPUs, being at least 2x faster than any “2GPU” case. Therefore, even though data is already packed to a contiguous format, the data transfer between GPUs over PCI-Express is still the bottleneck of non-contiguous GPU data communication in an “SM” environment.

Compared with MVAPICH2, our implementation is always significantly faster, independent of the datatype. Because of MVAPICH2’s vectorization algorithm converting any type of datatype into a set of vector datatypes Wang et al. (2014), each contiguous block in such an *indexed* datatype is considered as a single vector type and packed/unpacked separately, resulting in sub-optimal performance. As seen in the figure, their *indexed* implementation is slow, going outside the time range once the matrix size reached 1000.

In an “IB” environment, even though data is transitioned through host memory before being sent over the network, thanks to zero copy, the device-to-host transfers are handled automatically by the hardware, and this transfer is overlapped with the execution of the GPU pack/unpack kernels. In this environment we notice a significantly more desirable behavior from MVAPICH2, at least for the vector type. However, our approach achieves a roughly 10% improvement for the *vector* type. Similar to the *indexed* result of “SM” environment, the MVAPICH2 performance is quickly outside the range for matrices as small as 1500.

Vector-Contiguous

When using MPI datatypes, the sender and the receiver can have different datatypes as long as the datatype signatures are identical. Such features improve the application’s ability to reshape data on the fly, such as in FFT and matrix transpose. In FFT, one side uses a *vector*, and the other side uses a *contiguous* type. Figure 4.13 shows the ping-pong performance with such datatypes of different sizes. As seen in the figure, taking the benefit of GPU RDMA and zero copy, our implementation performs better than MVAPICH2 in both shared and distributed memory environments

Matrix Transpose

Matrix transpose is a very complex operation and a good stress-test for a datatype engine. With column-major storage, each column is contiguous in memory. A

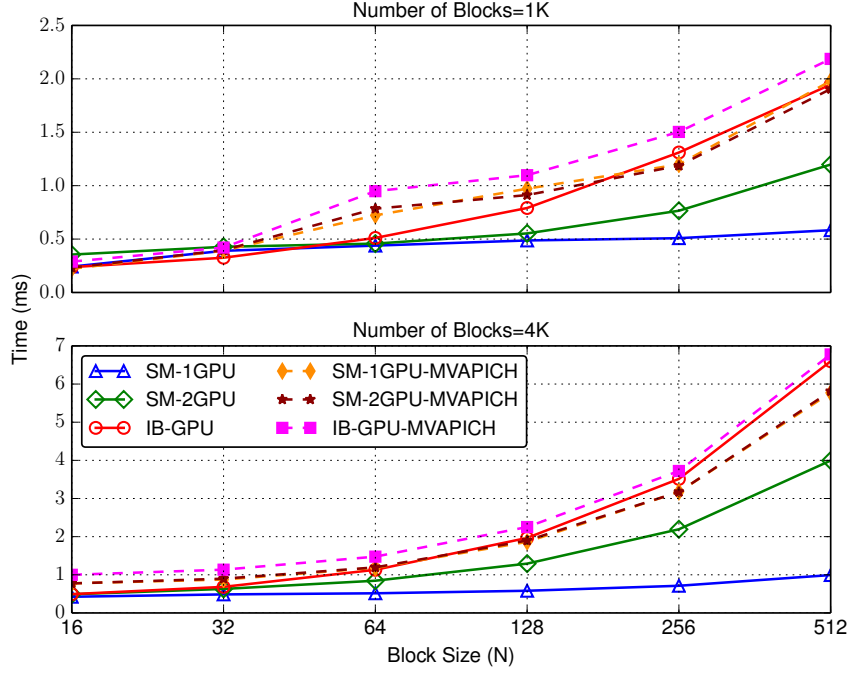


Figure 4.13: Ping-pong benchmark with vector and contiguous data type.

matrix can be described by a *contiguous* type or *vector* type if only accessing the sub-matrix. After the transpose, each column can be represented by a *vector* type with a block length of 1 element; consequently, the whole transposed matrix is a collection of N *vector* types. Figure 4.14 shows the benchmark for a matrix transpose depending on the matrix size. Since there is only 1 element in each block, the memory access is not following the coalesced rule, and the performance is not comparable with the regular *vector* type. However, such difficulty also occurs in the CPU implementation, benefiting from the parallel capability and high memory bandwidth, our GPU datatype implementation is at least 10x faster than the CPU version of Open MPI. Lacking stable support for such a datatype, MVAPICH2 crashed in this experiment and is not included in the figure.

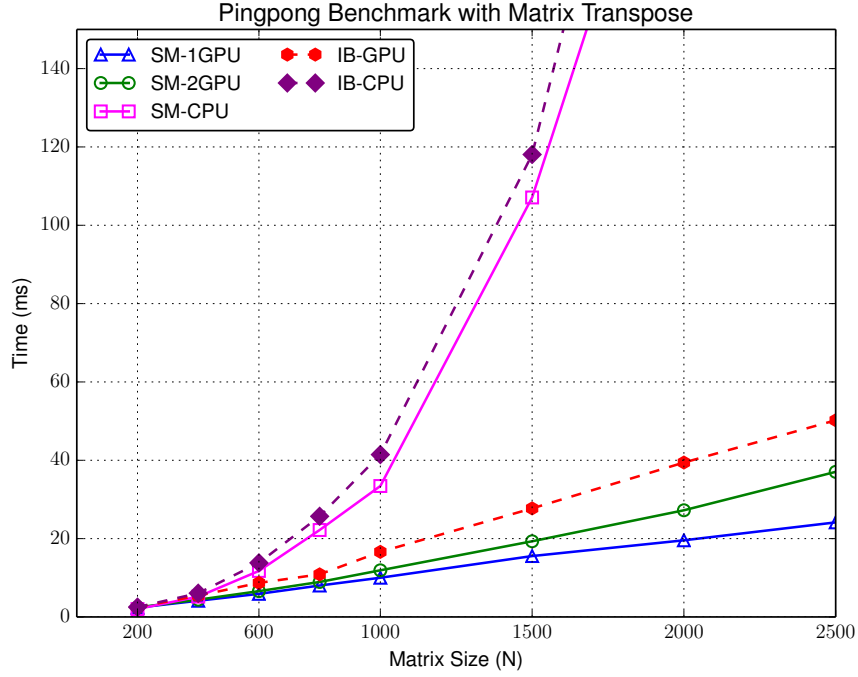


Figure 4.14: Ping-pong benchmark for matrix transpose in both shared and distributed memory environment.

4.4.3 GPU Resources of Packing/Unpacking Kernels

In previous benchmarks, GPU packing/unpacking kernels aggressively used CUDA’s Streaming Multiprocessor (SM). Figure 4.7 shows that by using as many CUDA cores as possible, the kernels are able to achieve more than 80 GB/s of GPU memory bandwidth. However, in most cases, each MPI process is attached to a separate GPU; since GPUs are connected by PCI-Express, then the communication bandwidth is limited to the 10 GB/s available through PCI-Express. In this section, we investigate the minimal resources required to fulfill the PCI-Express bandwidth. The top figures of Figure 4.15 and Figure 4.16 present the GPU memory bandwidth of packing/unpacking kernels for sub-matrix “V” and triangular matrix “T” data types. NVIDIA’s Kepler GPU has four warp schedulers per SM; therefore, in order to achieve the best GPU occupancy, the block size should be a multiple of 128 threads (32 threads per warp). In the benchmark, we use 256 threads per block. As seen in the

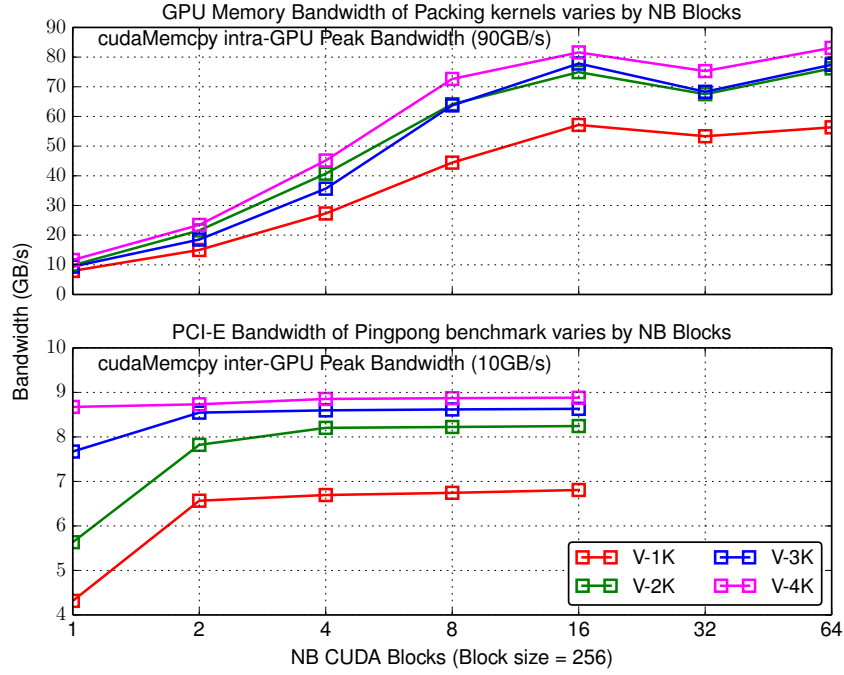


Figure 4.15: GPU memory and PCI-Express bandwidth of pack/unpack sub-matrix “V” data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.

figure, it requires 16 blocks to achieve the peak bandwidth, and achieves 10 GB/s (the peak of PCI-Express bandwidth) by launching only 2 blocks in most cases. Hence, theoretically, by using no more than 2 blocks, the cost of packing/unpacking can be hidden by communication over PCI-Express when pipelining is applied. Similarly, bottom figures of Figure 4.15 and Figure 4.16 illustrates that the PCI-Express bandwidth of the same two data types varies by the number of blocks used for kernel launching. As seen in the figure, as we expected, the bandwidth becomes stable when using at least 2 CUDA blocks. The K40 GPU has 15 SMs, so in the worst case, one seventh of the GPU SMs are required to overlap the cost of packing/unpacking kernels with communications over PCI-Express. In other cases when each MPI process is attached to the same GPU or future NVLink is introduced with higher bandwidth, our GPU datatype engine can be easily adapted by tuning CUDA blocks to fulfill bandwidth.

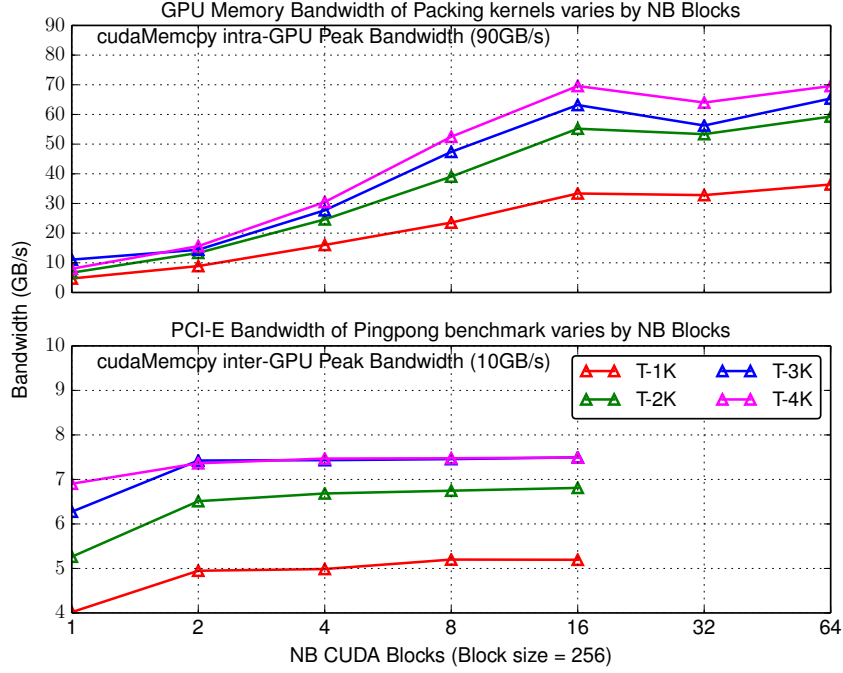


Figure 4.16: GPU memory and PCI-Express bandwidth of pack/unpack triangular matrix “T” data types varies by number of blocks used for kernel launching. Matrix size varies from 1K to 4K.

4.4.4 Pipeline and Resource Contention Effects

All previous benchmarks were executed under the assumption that the GPU resources are readily available for pack/un-pack. As in some cases, overlapping communication with computation is possible, the application might be using the GPU while MPI communications with non-contiguous data-types are ongoing. In this section, we investigate how resource contention affects the pack/unpack performance, as well as the pipelining discussed in Sec 4.3.3.

In this benchmark, we launch a special kernel to continuously occupy a fixed percentage of the GPU while executing the ping-pong benchmark. The grid size of the kernel varies to occupy full, half, or a quarter of the GPU resources; we then measure the ping-pong performance under these scenarios. The datatypes used are (*vector*) sub-matrices of size 1000 by 1000 and 2000 by 2000, since they are typical problem

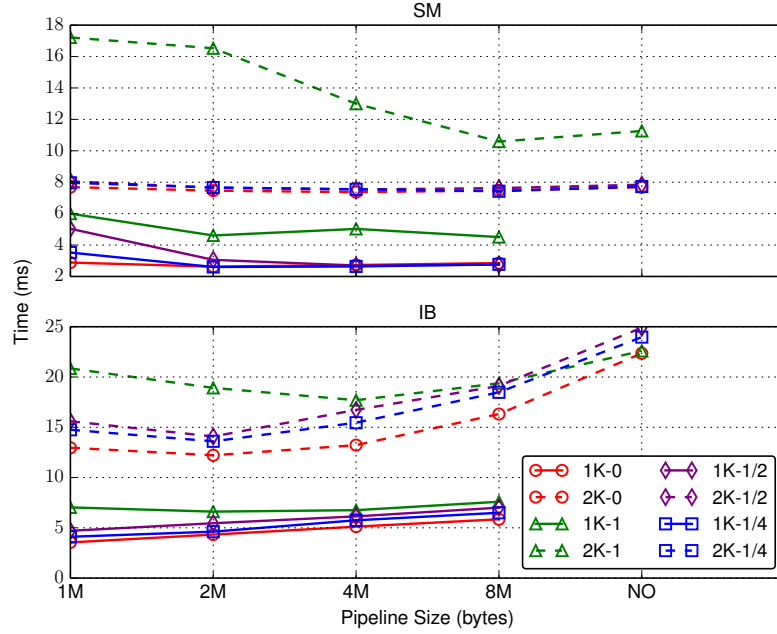


Figure 4.17: Ping-pong benchmark with partial GPU resources available. In the legend, the number after the matrix size is the ratio of GPU resources occupied.

sizes for GPU applications in the linear algebra domain. The results are shown in Figure 4.17. Thanks to the pipelining methodology, a proper pipeline size improves the performance in both shared and distributed memory machines. However, as seen in the figure, with a small pipeline size the pack/unpack operations are divided into many small GPU kernels, and the scheduling of such kernels could be delayed by the CUDA runtime when the occupancy of the GPU is high. Our GPU pack/unpack kernels mainly contain memory operations without floating point operations, and they are memory bound. Therefore, as long as the GPU is not fully occupied, our pack/unpack methodology is not significantly affected. By using a proper pipeline size, we limit the loss of performance to under 10%.

4.5 Summary

In this chapter, we have presented an efficient solution for communication of non-contiguous data resident in GPU memory with the following two detail steps: first, we have designed a GPU datatype engine to efficiently pack/unpack non-contiguous/contiguous data into contiguous/non-contiguous layout before/after communications by taking the advantage of large parallel capability of GPU hardware; second, the datatype engine is integrated into Open MPI infrastructure to provide RDMA-like communication. In additional, we have proposed a pipeline model to overlap pack/unpack operations with data movements. We have showcased the advantage of our non-contiguous GPU data communication by conducting several benchmarks with four common used datatype in scientific computation domains on both shared and distributed memory machines.

Chapter 5

GPU-aware Collective Communication

5.1 Issues of Traditional Collective Communication Algorithms in Heterogeneous System

Collective communication is another set of data movement patterns in which messages are exchanged among a group of processes. In this dissertation, we focus on two widely used collective operations: broadcast and reduce. We take broadcast as an example for algorithm analysis, reduce is similar approach. In Open MPI, there are five typical broadcast algorithms: flat-tree/linear, chain, binomial tree, binary tree and splitted-binary tree [Pješivac-Grbović et al. \(2007\)](#). In flat-tree/linear algorithm, root sends messages to all the other processes in sequence. In chain algorithm, all the processes form a chain and messages are propagated from the root to leaf one by one. In binomial and binary tree algorithms, messages traverse the tree starting at the root and going towards the leaf processes through intermediate processes. In the splitted-binary tree algorithm, the original message is split into two parts, and the first half of the message is sent to the left half of the binary tree, and the second half of the message is sent to the right half of the tree. Later, every node exchanges

message with their partner in the opposite side of the binary tree. For large message, all of these algorithms support message segmentation which divides whole message into small segments and these segments are transferred in pipeline fashion, allowing for overlap of concurrent communications.

Chapter 4 has already shown huge performance improvement of non-contiguous GPU data communication via offloading pack/unpack operations on GPU and applying pipeline to overlap pack/unpack with data movement. The improvement of GPU-aware p2p communication demonstrates that it is very important to integrate the knowledge of GPU (i.e. many core feature and GPUDirect) into MPI runtime. Because of the following issues, directly applying the conventional collective algorithm in heterogeneous system is not efficient; hence, in the context of Open MPI, even though collective communications are built on top of p2p communications, optimal GPU-aware p2p does not guarantee the optimal GPU-aware collective communications.

5.1.1 Multiple Networks

Modern systems trend to be heterogeneous, which contains multi-core CPUs and accelerators such as GPUs. GPUs are attached to host via high speed host to peripheral connections (i.e. PCI-Express and future NVLink). Therefore, message exchanges between processes within a collective communication involve intra-socket, inter-socket and inter-node communications, which can occupy different physical networks. In this dissertation, PCI-Express, Intel OPI and InfiniBand are used as examples of these three types of networks respectively. Because the disparate networks are independent, a smart collective communication framework should be able to efficiently utilize all networks, eventually providing more opportunities for concurrent communication over different networks. Traditional collective operations are targeted to homogeneous systems, which assumes the communication cost between any two processes are the same. Therefore, they do not have the ability to take care

of multiple network links with different latency. In most cases, network links are full-duplex: a process can have two concurrent communications but different directions. The chain algorithm is able to utilize the benefit of full-duplex network by leveraging one direction to receive data and another direction to send data. However, chain algorithm has poor ability to resist network noise: a delay of any process would propagate the noise to the following processes of the chain. Moreover, a process is able to use different networks (i.e. PCI-Express, QPI and InfiniBand) to concurrently communicate with other processes as long as networks are independent. Process in chain algorithm only sends data in one direction, hence, it is not able to efficiently utilize all network resources. A tree algorithm allows concurrent communications if mapping communications of process to children processes to different networks. However, with traditional binary/binomial tree algorithms without taking care of network topology, process may use the same direction of network to propagate data to all of its children processes. It is acceptable for small messages, since the performance is bounded by network latency instead of network bandwidth. For large messages, which is a typical message size of GPU applications, communications of process to its children processes are serialized and leading to sub-optimal performance. Therefore, none of the traditional collective algorithms works efficiently in heterogeneous system. It is desired to leverage the knowledge of GPU network topology to design collective algorithms, which allow different parent-children communication via independent network links and better utilization of all network links.

5.1.2 Process Mapping

A virtual topology represents the way that MPI processes communicate of a collective operation. A physical topology represents network connections between CPU cores, GPUs and nodes in the hardware. How to map virtual topology to physical topology, also called process mapping, is critical for the performance of collective

operation. There are three typical process mapping strategies in MPI: by-core, by-node, by-socket. By-core/socket associates processes with successive core/socket. By-node maps one process on each node, wrapping around the list of available nodes until all processes have been mapped. Because the latency of disparate networks (intra-socket, inter-socket and inter-node) are quite different, it requires MPI application developers to have the capability to select the optimal process mapping to minimize the communication over slow channel. For example: in a traditional chain broadcast algorithm, chain is created based on processes' rank instead of processes' physical location; if the processes are mapped by core, the number of inter-node communications is the minimal; but if processes are mapped by node, every link in the chain becomes an inter-node communication and deliver sub-optimal performance. MPI also provides *MPI_Comm_split* communicators with user defined process mapping. However, as discussed in 5.1.1, networks in heterogeneous systems are much more complicated than homogeneous system, application developers who has little knowledge about the underlying network of systems are unlikely to select the optimal process mapping strategy. Therefore, it is preferred that MPI library is able to automatically rearrange mapping of processes involved in collective operations based on network topology.

In this chapter, we present a topology-aware GPU collective framework by leveraging GPU network information such as topology. By introducing a tight collaboration between multiple layers of collective algorithms, our collective communication achieves concurrent inter-node, intra-socket and inter-socket communication over different independent networks. As a side-effect of using GPUs, we also have the opportunity minimize the cost of the reduction operations by offloading them on the GPUs to further boost performance of collective communications.

5.2 Design of Topology-aware Collective Communication

5.2.1 Topology-aware Tree

To integrate the network topology knowledge into collective communication, topology-aware tree is used to represent topology hierarchy of entire network. The first phase of building topology-aware tree is to gather topology information from all processes participating in collective communication. Each process collects its own hierarchical topology information using Portable Hardware Locality (hwloc) [Broquedis et al. \(2010\)](#) framework. The hwloc software package provides an abstraction of the hierarchical topology of modern architectures, including nodes, sockets and cores. Because of the limitation of hwloc, we cannot get further network topology information such as network switch or router. So in this dissertation, we only consider three network levels: node level, socket level and gpu level. In GPU-aware MPI, each GPU maps to a separate process. Each process uses a topology tuple (*Node_ID*, *Socket_ID*, *GPU_ID*) to represent its location. After every process gathers its topology tuple, all the processes exchange their local information so that all the processes has topology information of all other processes to form a topology table. Later, topology table is cached in memory, so that all the following collective operations within the same or duplicated communicator can directly use the cached topology table instead of gathering it again, no matter what kind of collective operations they are.

With topology table, we can build a topology-aware tree to do collective operations such as broadcast or reduce. Since reduce is just an opposite operation of broadcast with additional reduction operations, we use broadcast as an example in following sections. In the gpu level, all the processes belong to a socket form a group called *Level 1 group*. As in Figure [5.1](#), P0, P1, P2 and P3 are in the same Level 1 group. A socket leader is selected from each Level 1 group. The strategy to select a leader can

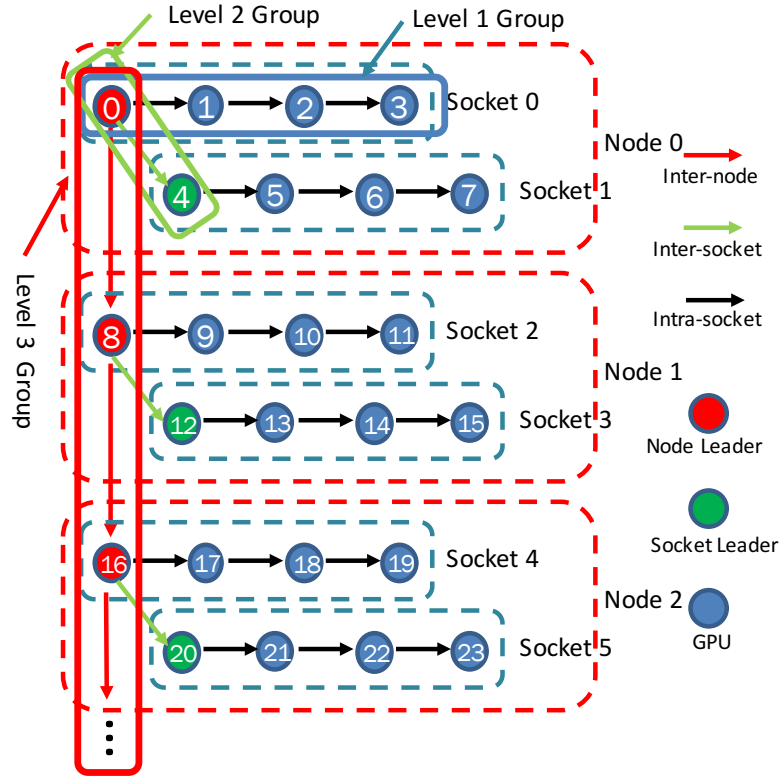


Figure 5.1: Topology-aware tree of broadcast algorithm on multi-GPU cluster. (4 GPUs per socket and 2 sockets per node)

vary. We choose the process with the lowest *Core_ID* on the same socket as socket leader. In the socket level, all the socket leaders in the same node form a group called *Level 2 group*. As shown in Figure 5.1, P0 and P4 belong to the same Level 2 group. Furthermore, a node leader is selected among all the socket leaders in the same node and all the node leaders forms a group called *Level 3 group*, which includes P0, P8 and P16. In a word, each group has a leader, which is a member of upper level group.

Once leaders of all the groups are selected, we can build a topology-aware tree to do collective operations. The topology-aware tree is a combination of sub-trees of every group. For each group, sub-tree contains all the processes within the group and uses the group leader as root of this sub-tree. Because the group leader is a member of upper level group, the root of a sub-tree must appears in other sub-trees and it becomes an intermediate process to combine sub-trees. As in Figure 5.1, P4, P5, P6 and P7 form a sub-tree (chain shape) in its Level 1 group with P4 as root , also P0 and P4 form another sub-tree (chain shape) in its socket group with P4 as root. P4 glues these two sub-trees together. Recording the entire topology-aware tree for each process consumes too much memory and is not scalable. Instead, each process only store partial topology-aware tree related to itself (it's parent and children), and caches it into local host memory for reuse of future broadcast communications of the same communicator. It eliminate the cost of building topology-aware tree.

5.2.2 Algorithm Selection of Sub-trees

Since communication within each group is independent from others (Level 1 2 and 3 groups communications are respected to intra-socket, inter-socket and inter-node communications), each group can have a different sub-tree. Although the three levels of algorithms are tightly integrated, there are still a variety of combinations that are possible, whose performance greatly varies depending on hardware features and properties. In this section, we discuss how to select the proper algorithm for each group. [Pješivac-Grbović et al. \(2007\)](#) introduces decision tree approach to

Table 5.1: Latency of different broadcast algorithms with Hockney model

Algorithms	Latency
Flat tree\Linear	$T = n_s * (P - 1) * (\alpha + \beta m_s)$
Chain	$T = (P + n_s - 2) * (\alpha + \beta m_s)$
Binomial tree	$T = n_s * \lceil \log_2 P \rceil * (\alpha + \beta m_s)$
Binary tree	$T = 2 * (\lceil \log_2(P + 1) \rceil + n_s - 2) * (\alpha + \beta m_s)$
Splitted-binary tree	$T = 2 * (\lceil \log_2(P + 1) \rceil + \lceil \frac{n_s}{2} \rceil - 2) * (\alpha + \beta m_s) + \alpha(\frac{m_s}{2}) + \frac{m_s}{2} * \beta(\frac{m_s}{2})$

select algorithms with the help of performance model of collective communications. This approach does not work for heterogeneous systems since most performance models assume flat network (unique type). However, in our topology-aware collective algorithm, network within each hierarchical group is flat, it is able to employ performance models to guide the selection of collective algorithms for each group. One of the most frequently used model is Hockney Model [Hockney \(1994\)](#). Even though there are more sophisticated models such as LogP [Culler et al. \(1993\)](#), LogGP [Alexandrov et al. \(1995\)](#) and PLogP [Kielmann et al. \(2000\)](#), this model is sufficient for our needs. Hockney model assumes that the time to send a message of size m between two nodes is $T = \alpha + \beta m$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte or reciprocal of network bandwidth. In the case of reduction operation, we assume that the time spent in computation on data in a message of size m is γm , where γ is computation time per byte. This linear model ignores effects caused by memory access patterns and cache behavior, but is able to provide a lower limit on time spent in computation. This cost model assumes that all processes can send and receive one message at the same time while maintain constant latency and bandwidth, regardless of the source and destination. Therefore, the entire time of send a message of size m between two processes is:

$$T = \alpha + \beta m + \gamma m$$

Section 5.1 introduces 5 broadcast algorithms. For large message, all of such 5 algorithms in Open MPI support message segmentation and propagating in pipeline fashion. Table 5.1 shows the performance analysis of these 5 broadcast algorithms

using the Hockney model [Pješivac-Grbović et al. \(2007\)](#) when pipeline is applied. Since α and β represent constant parameters of hardware network, the selection of most efficient algorithm is based on the message size m (equals to size of each pipeline segment m_s times number of segments n_s) and number of processes participated in collective communication P . In the context of communication of each group, P refers to the number of processes with in the group. In a typical GPU clusters shown in [Figure 5.1](#), there are four GPUs per socket and 2 socket in total. In such setup, we select pipeline chain algorithm for Level 1 group since according to the formula in [Table 5.1](#), the cost of chain algorithm of 4 processes can be considered as $n_s * (\alpha + \beta m)$ which is the smallest among all 5 algorithms. Beside, system noise propagation, one of the major side effect of chain algorithm, is unlikely a issue since there are only 4 processes. From the hardware perspective, PCI-Express is bi-directional, it support a process to send and receive message simultaneously, which is the precondition of Hockney Model. Similarly, we also select pipeline chain algorithm for Level 2 group. For Level 3 group, if there are only a few nodes, we select chain algorithm; otherwise, tree-based algorithms can be used for large scale clusters.

5.2.3 Collective Communication with Topology-aware Tree

The traditional way to implement hierarchical collective operations is using communicators [Kandalla et al. \(2010\)](#) [Subramoni et al. \(2012b\)](#) [Awan et al. \(2016\)](#). For each group, a sub communicator is created. The broadcast starts from the top level communicator. The next level can start only when the upper level broadcast is finished. Therefore, this method is not efficient for large messages. To efficiently transfer large messages, messages usually are divided into fixed-size segments and are transmitted in pipeline fashion. But in such implementation, the handling of communicators of different levels does not cooperate tightly, therefore, the lower level broadcast can not start until the higher level finishes sending the whole message, and leads to sub-optimal pipelining.

In this dissertation, we present a fine grain pipeline approach. In our implementation, topology-aware collective is not divided into several smaller collective operations of different communicators, but is treated as a single communicator. A group leader is the transition process between different levels. Our design dissolves the boundaries of different levels and allows fine grain pipeline. Instead of waiting for receiving the whole message from upper level communicator, a transition process posts non-blocking receive request for one segment from its parent in the topology-aware tree; the completion of that request triggers multiple non-blocking send requests to its children and another receive request from its parents for next segment. Therefore, a transition process can start transmitting messages as soon as one message segment has been received. Since we are using non-blocking p2p communications, a transition process can issue multiple communications of different levels simultaneously. Also, because Level 1(intra-socket), Level 2(inter-socket) and Level 3(inter-node) communications can occupy different networks, our collective communication can achieve concurrent communications between different levels.

5.2.4 Minimize Communications Over PCI-Express

As seen in Figure 5.1, node leader is the busiest MPI process of the entire communications because it not only stays in node group, but also socket and core group. In broadcast operation, node leader receives data from previous node leader and sends data to next node leader, next socket leader and next process within the socket. Figure 5.2.a shows such data flow of node leader when using GPUDirect. Inter-GPU communications between node leaders go through Network Interface Controller (NIC) of InfiniBand via PCI-Express. When a node leader sending data to next socket leader, data goes through a implicit intermediate CPU buffer to next socket leader GPU. Such data-flow utilize PCI-Express and QPI bus. When the node leader sending data to next process of core group, data-flow go through PCI-Express as well. Since such three communications occupy the same direction

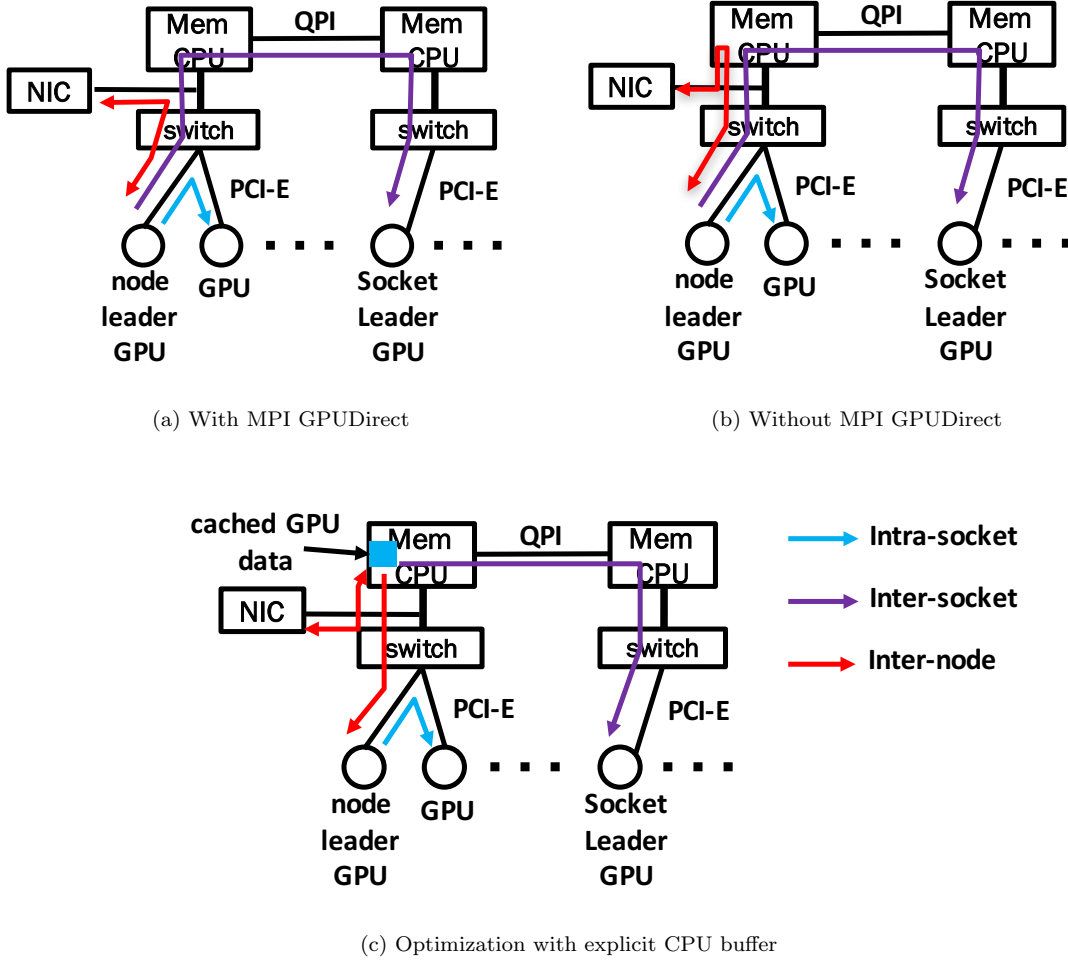


Figure 5.2: Data flow of non-root node leader MPI process

of PCI-Express simultaneously, only one third of PCI-Express bandwidth can be reached for each communication. As shown in Figure 5.2.b, when GPUDirect is disabled, inter and intra socket communications are the same as before, but inter-node communications need to take an extra step to go through implicit intermediate CPU buffer. Since the CPU buffer is implicitly managed by MPI p2p communication, each p2p communication would use different CPU buffers even for transmitting the same data. Therefore, a lot of CPU memory and PCI-Express bandwidth is wasted.

To solve the congestion of PCI-Express in previous implements, we allocate a explicit CPU buffer for node leader process to cache GPU data. None-root node leader caches received data into this CPU buffer, so that it can send the data to next

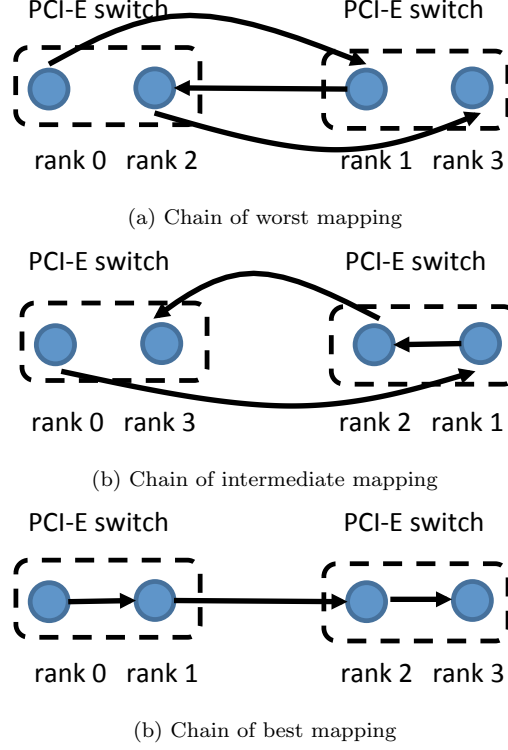


Figure 5.3: Process mapping using hardware topology of PCI-Express switch

node leader and socket leader directly from CPU buffer without pulling data from GPU memory via PCI-Express again. Later, cached data is updated to GPU memory via asynchronous memory movement. Root process also caches data in CPU memory to alleviate the load of PCI-Express. Figure 5.2.c shows the optimized data flow of node leader process. Therefore, as long as NIC and GPUs are not connected to the same PCI-Express switch, communications between NIC and explicit CPU buffer, CPU buffer to GPU and GPU to neighbor GPU use different PCI-Express lane, and therefore can be overlapped. Hence with the design of using explicit CPU buffer, we are able to map intra-socket, inter-socket and inter-node communications to use different physical networks, and achieve communication overlapping.

5.2.5 PCI-Express Switch Level Process Reorder

As discussed in Section 5.2.1, the topology-aware tree are build to minimize heavy communications no matter what kinds of process mapping strategies are chosen. In typical GPU clusters, several GPUs within the same socket are connected to a PCI-Express switch, and inter-switch communications are slower than intra-switch communications. Since PCI-Express switch is also bi-directional, having more than one communications per direction shares the PCI-Express bandwidth, therefore reduce performance. As seen in Figure 5.3, when mapping processes to GPUs and applying the chain algorithm based on their ranks, it is not ensure that how many times the chain crosses PCI-Express switch. In worst mapping, the chain crosses switch three times and two of them are in the same direction so that the bandwidth is cut in half. In intermediate mapping, the chain crosses switch twice but in different direction so the performance would not be affected ideally. In best mapping, the chain crosses switch only once. Therefore, to minimize inter-switch communications, we detect the hardware topology of PCI-Express switches with the help of hwloc, and build a chain of optimal mapping based on GPU locality instead of MPI rank.

5.2.6 Offload Reduction Operation on GPU

Reduction operations are mathematical operations on arrays, which are extremely parallel, so it works more efficiently in GPU by letting each CUDA thread to handle reduction operation on each element of array. However, as discussed in Section 2, the current GPU-aware MPIs still use CPU to do reduction, which is not efficient. Since most MPI implementation is still single threaded, reduction operations in CPU occupies CPU resources and delays the handling of other communications independent with the reduction results. Therefore, in the design of our collective operations, we offload the reduction operations in GPU asynchronously by using multiple CUDA streams, which allows communications overlapping with reduction operations. We

developed CUDA kernels for pre-defined MPI reduction operations. Users can also develop their own CUDA kernel functions to handle user defined reduction operations.

5.3 Performance Evaluation

In this section, we evaluate the performance of topology-aware collective framework using two most widely used collective operations: broadcast and reduce. We investigate GPU-aware collective operations using three types of experiments: first, the total number of processes are fixed, and we measure the performance of different message size; second, we look at the strong scalability, which measures the performance by varying number of processes with fixed message size; third, we measure the performance with different process mapping strategy to demonstrate the performance of our framework is not affected by process mapping. We compare our topology-aware collective operations (shown as “OMPI-topo”) with default Open MPI (shown as “OMPI-tuned”) and MVAPICH2.2-GDR (shown as “MVAPICH2”). The experiment is conducted in Nvidia PSG cluster: each node is equipped with 4 K40 GPUs with CUDA 7.5 and 2 deca-core Intel Xeon E5-2690v2 Ivy Bridge CPUs, nodes are connected by 40Gb/s FDR IB.

In the OMPI-topo, both the broadcast and the reduce operations are pipelining algorithms, in which messages are split into several small segments. A perfect pipeline needs to meet two criterion: large enough segment size and number of segments. If segment size is too small, message latency as α in Hockney model becomes dominate, preventing the full p2p bandwidth from being leveraged. If there are not enough number of segments, pipeline establish time is still significant and the overall performance is hurt. Therefore, it is difficult for small messages to meet both criterion and hence our framework mainly targets for large messages which is also the message size of most GPU applications. To fully utilize the bandwidth of PCI-Express, we use 512 KB as segment size, because p2p communication over PCI-Express is about to reach the peak bandwidth with message size 512 KB.

As discussed in Section 5.2.2, pipeline chain algorithm is used for both Level 1 and Level 2 groups. Because of the limitation of resource allocation, we can only obtain 8 nodes for the experiment. Considering the number of nodes and message size, chain algorithm is also applied in Level 3 group based on the analysis of cost model.

5.3.1 Performance Scalability

Figure 5.4 shows the latency of broadcast and reduce with GPU data varies by message size on 8 nodes (32 GPUs in total). Tuned module is default in OpenMPI, which can switch algorithms based on different message size. Therefore, as shown in this figure, after 4MB/16MB, the broadcast/reduce algorithm of OMPI-tuned is switched and slope of curves changes. As discussed before, pipeline segment size is set to 512KB, hence, OMPI-topo does not have obvious improvement over others when the message is less than 2MB because of the lack of entire segments. For large message, the benefit of fine grain pipelining becomes dominated, so that OMPI-topo over-performs MVAPICH2 and OMPI-tuned 2-3 times on broadcast. Furthermore, by taking the benefit of asynchronized reduction operation in GPU, our reduce is almost 10 times faster than the other two MPI libraries, while their reductions occupy CPU.

5.3.2 Strong Scalability

Scalability is another important factor to evaluate the performance of MPI libraries. Figure 5.5 presents the result of strong scalability experiment of broadcast and reduce with fixed message size and varies number of nodes. By caching data in CPU memory, it reduce the PCI-Express traffic for non-root node leader processes. Hence such processes participated in all 3 hierarchical groups (inter-node, inter-socket and intra-socket) are able to concurrently communicate with members of 3 different groups via independent networks(InfiniBand, CPU memory and PCI-Express). However, since original data is in GPU memory, inter node and intra socket communication

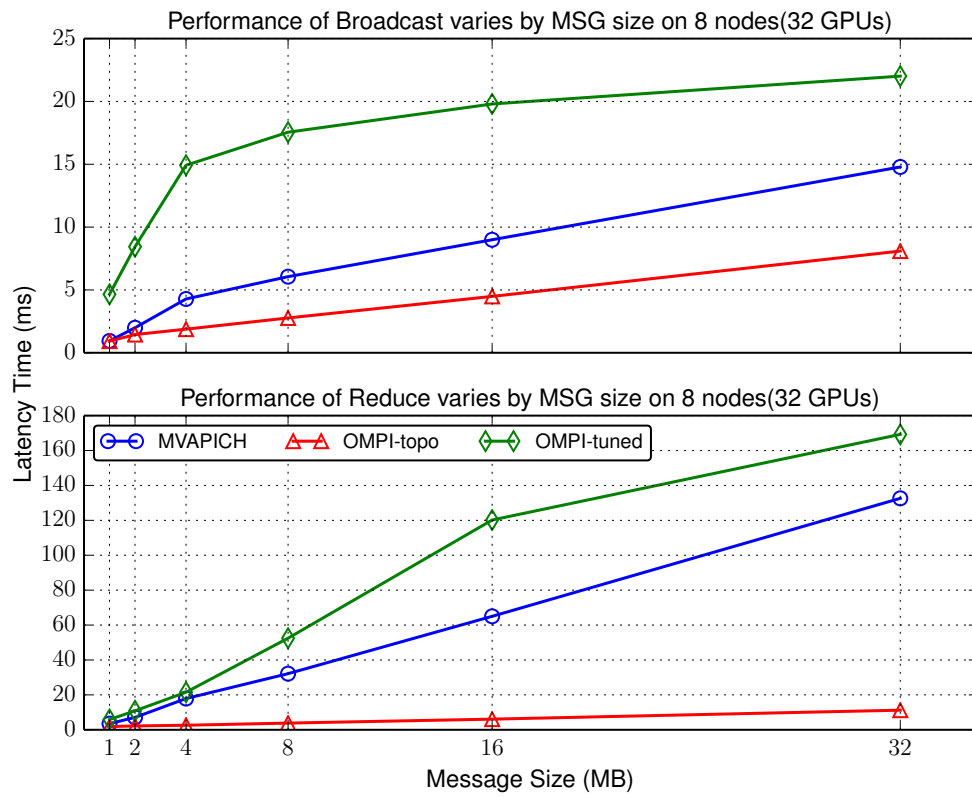


Figure 5.4: Performance of broadcast and reduce with GPU data varies by message size using 8 nodes 32 GPUs

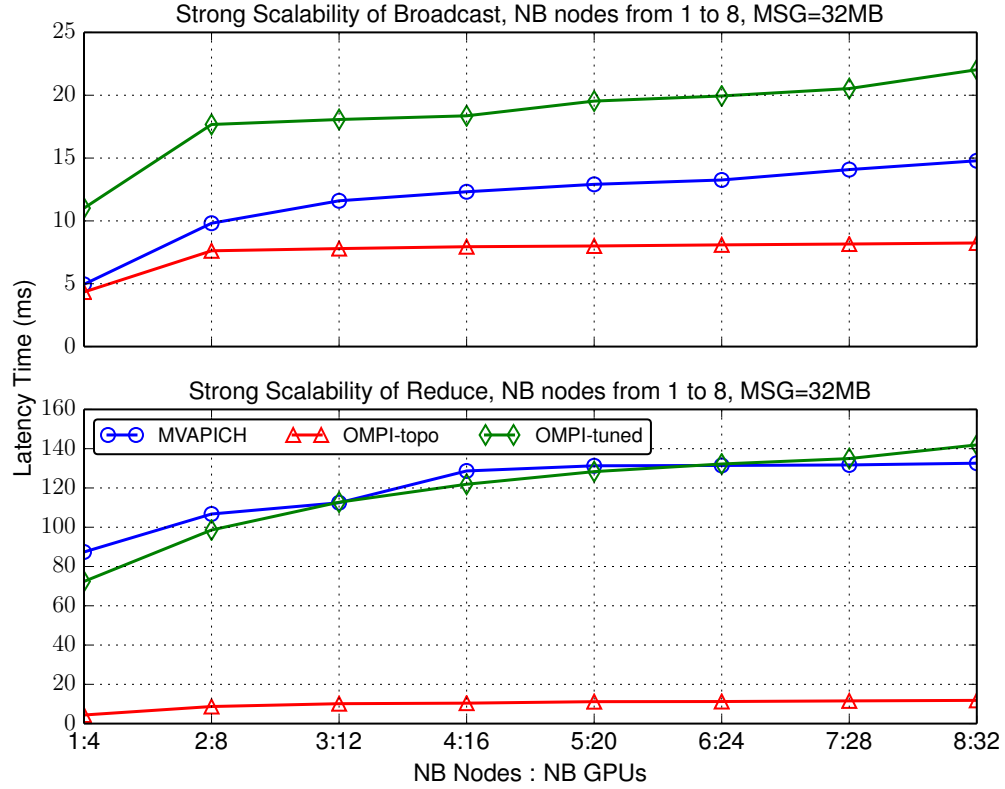


Figure 5.5: Strong scalability of broadcast and reduce with GPU data varies by number of nodes, message size is 32 MB

of root process occupies the same direction of PCI-Express, leading to dropping of performance when more than 2 nodes is engaged, compared with 1 node.

Since chain algorithm is used for each group and groups occupy independent networks for communications, the entire cost of broadcast/reduce can be considered as the cost of propagating data from root to the last node leader in Level 3 group plus the cost of transferring data to the last process of Level 1 group of the last node. Such communication pattern is also a chain. According to the model of chain algorithm shown in Table 5.1, the time of chain algorithm can be treated as $T = n_s * (\alpha + \beta m)$ if the message size is large enough to ignore the cost of establishing pipeline, and hence performance does not depends on the number of processes within the chain. Therefore, as seen in Figure 5.5, the latency of our broadcast and reduce

trend to be stable when increasing the number of nodes, which represents perfect strong scalability. OMPI tuned switches algorithms based on number of processes and message sizes. However, the decision strategy is not designed for homogeneous system, not heterogeneous system with GPUs, hence, it would not select a optimal algorithm, leading to much slower performance than MVAPICH2 and our OMPI-topo. Evidence can be seen that, when only using one node, OMPI-tuned does not use the chain algorithm which should be the optimal one, resulting significant performance drop. Again, because of fine grain pipelining and concurrent communication of different topology groups, we achieve better scalability than OMPI-tuned and MVAPICH2.

5.3.3 Process Mapping

In MPI program, process mapping is crucial for performance. Figure 5.6 demonstrates the performance influence of different mapping strategies. Because 4 GPUs are attached to the same socket, we use two kinds of traditional mapping: by-core and by-node. In addition, we randomly switch ranks in communicator to create a random mapping by *MPI_Comm_split*. We average the result of 50 times random mapping to look at the performance influence of unusual mapping. It is obvious that no matter what kinds of process mapping strategies is used, OMPI-topo always has steady and good performance because process are mapped based on topology not ranking. MVAPICH2 may use a chain-like algorithm as their default algorithms for message size of 32MB. Evidence can be seen when the processes are mapped by node, all communications in chain-like algorithm are inter-node communications, and hence is slowest of other process mapping strategies.

In typical heterogeneous systems, GPUs within a socket are divided to several PCI-Express switches, resulting to heavier inter-switch communications than intra-switch communications. Therefore, a smart collective communication should also be able to take care of the topology of intra-socket GPU locality. Now, we focus on the effect of process mapping based on topology of PCI-Express switch. In the PSG

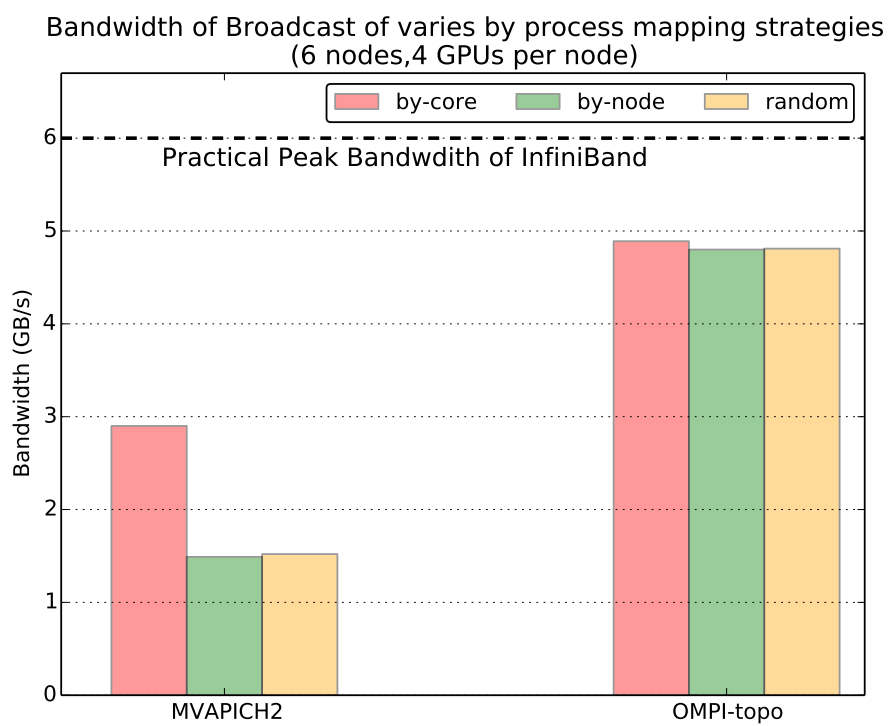


Figure 5.6: Bandwidth of broadcast with GPU data varies by different process mapping with message size 32MB

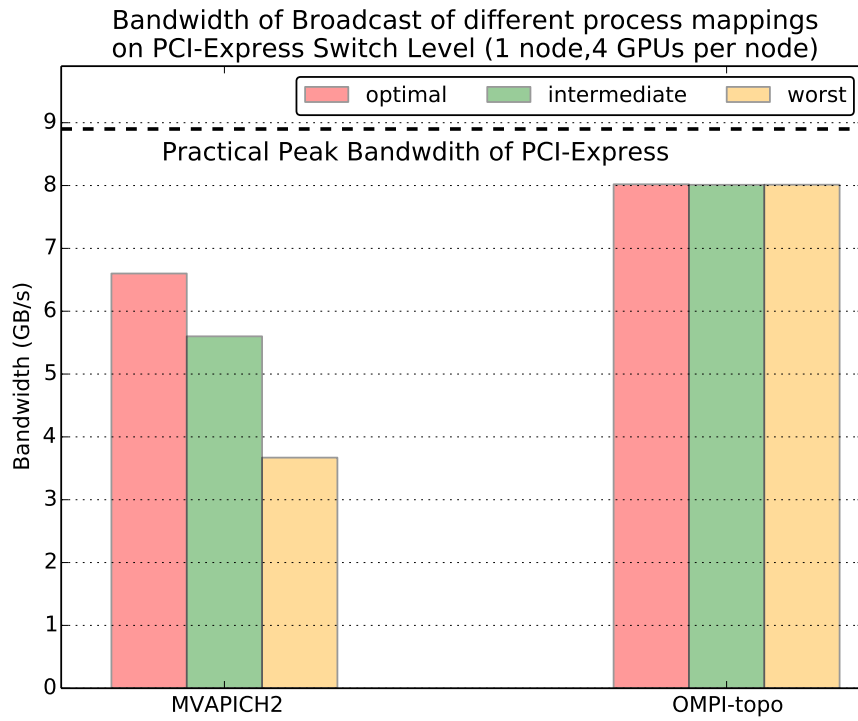


Figure 5.7: Bandwidth of broadcast with GPU data on 1 nodes varies by different process mapping on the level of PCI-Express switch with message size 32MB

K40 cluster, GPU 0, 1 and GPU 2, 3 are connect to different switches. To only investigate the influence of process mapping based on PCI-Express switch without the influence of other networks, we conduct the experiment of process mapping on PCI-Express switch level in single node. As discussed in Figure 5.3 of Section ??, there are 3 types of process mapping (optimal, intermediate and worst mapping) for a chain algorithm in PSG K40 cluster, which deliver different performance. Figure 5.7 presents the bandwidth of broadcast with these three mapping strategies. In the result of MVAPICH2, the worst mapping get half the bandwidth of its optimal mapping since it crosses PIC-Express three times. The intermediate mapping theoretically should performs the same as optimal mapping since PCI-Express is bidirectional. However it is still slower than optimal mapping in MVAPICH2. Because we build the chain based on topology of PCI-Express switch instead of rank, no matter what kind of mapping is used, we are able to achieve the same performance.

5.4 Summary

In this chapter, we have proposed a topology-aware collective framework in Open MPI infrastructure, which is able to minimize the slowest channels in heterogeneous systems and provide a close collaboration between different levels of networks. In addition, we have minimized communications over PCI-Express by caching data in CPU memory, so that inter-node and inter-socket communications are directly use the cached data instead of pulling data from GPU memory. Hence, the traffic over PCI-Express is alleviated. We have also offloaded the reduction operations onto GPU, which is able to take advantage of the embarrassingly parallel nature of the reduction operations and efficiently map them onto GPU threads.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

As heterogeneous compute nodes, featuring different types of processing units such as CPU cores and accelerators, become more pervasive, the need for a programming model capable of providing transparent access to all types of resources and delivering portability and efficiency across a large range of hybrid environments becomes critical. In this dissertation, we demonstrate that by incorporating integrating GPU knowledge such as cores, memory and network topology into conventional programming models including data-flow and message passing models, they are able to incorporate with GPUs tightly and fully utilize all types of computing resources in an efficient way, and therefore become real GPU-aware programming models.

The data-flow programming model, in which the inherent parallelism of the application is expressed as DAG, coupled with a runtime to manage tasks in homogeneous systems, has been proven to outperform legacy folk-join approaches. When adapting it to heterogeneous system, one the of major difficulties is data granularity disagreement of CPU and GPU tasks caused by significant hardware differences (less heavy weight cores vs many light-weight cores). To address this issue, we have proposed a “hierarchical DAG” approach that further improves the

applicability of the data-flow model to accelerated compute nodes. Data granularities of tasks become variable, and the runtime arbitrates depending on the type of the target computing unit. The performance analysis demonstrates that such an approach improves the asymptotic performance for dense linear algebra applications by employing the appropriate task grain on GPUs, while retaining a suitable amount of parallelism for CPU computations. Because GPU memory has separate memory space with CPU memory, and limited size, we have developed cache coherence protocol along with a multi-level memory management strategy to maximize data reuse and minimize data movement between CPU and GPUs by tracking data in CPU and GPU memory. We have also achieved overlapping of communication with data movement by offloading different operations to separate CUDA streams.

As a more generic programming paradigm, message passing programming model focuses on message exchanges among processes without assistance of shared variables. As a widely accepted standard of message passing communication, MPI defines the communication patterns of point-to-point and collective communications, and have been proved efficient and portability in homogeneous systems. In heterogeneous systems, the desire of GPU-aware MPI is urgent. In this dissertation, we have presented a efficient point-to-point communication design of data residing in GPU memory. The GPU datatype engine presented in this dissertation takes advantage of the parallel capability of the GPUs to provide a highly efficient in-GPU datatype packing and unpacking. We integrate the GPU datatype engine into the state-of-the-art Open MPI library, at a level of integration such that all communications with contiguous or non-contiguous datatypes will transparently use the best packing/unpacking approach. The different protocols proposed, RDMA, copy in/out, pipeline, and the use of novel technologies, such as GPUDirect, drastically improve the performance of the non-contiguous data movements, when the source and/or the destination buffers are GPU-resident. Experimental results demonstrate that our design out-performs the state-of-the-art MVAPICH2 for data with both regular and irregular memory layout.

In additional point-to-point communications, our topology-aware collective framework fully exploit the hierarchical network of heterogeneous system, and orchestrates the collaboration between multiple levels of networks. The fine grain pipeline proposed in this dissertation dissolve the the boundary between different levels of networks and allows a smooth transition across different levels. By incorporating the GPU network topology into collective framework, we provide more opportunity for concurrent communications over independent networks of different topology levels. We demonstrate experimentally that 1) our framework is immune to modifications of the underlying process-core binding; 2) it delivers better performance and strong scalability than state-of-the-art MVAPICH2.

6.2 Future Directions

This dissertation proved that conventional programming models can be beneficial from exploiting the knowledge of GPUs, from single GPU hardware level to entire network topology, and are back to glories in heterogeneous systems. Even though this work is done on Nvidia’s GPUs with CUDA, the idea can be extended to other many-core accelerators such as AMD’s GPU with OpenCL and Intel Xeon Phi with OpenMP.

With the development of Nvidia’s GPU, NVLink is likely to replace current PCI-Express to provide higher bandwidth and lower latency. The penalty of GPU Unified Memory Architecture is eliminated with the improvement of latency. The boundary of GPU and CPU memory is trended to dissolve, resulting shared memory between CPU and GPU which is similar to integrated GPU. Our work can be extended to this kind of platform by un-plugging the data movement component. Another trend is the topology change of NVLink, which allows a GPU to communicate with multiple GPUs simultaneously, and therefore affects our topology-aware collective framework. Since NVLink is only in the level of intra-node communications, the flexibility of the plug-in/plug-out design in our framework allows for easy embedding of new intra-node

collective operation algorithm specific for new NVLink topology into our framework. Undoubtedly, a faster intra-node collective module will further boost the throughputs of our topology-aware collective framework.

Bibliography

- Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing. [4](#), [12](#), [15](#)
- Aji, A. M., Dinan, J., Buntinas, D., Balaji, P., Feng, W.-c., Bisset, K. R., and Thakur, R. (2012). MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems. In *HPCC'12*, pages 647–654, Washington, DC, USA. [23](#)
- Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1995). LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA. ACM. [95](#)
- AMD (2010). *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Advanced Micro Devices, Inc. Publication number 24593, revision 3.17. [39](#)
- Awan, A. A., Hamidouche, K., Venkatesh, A., and Panda, D. K. (2016). Efficient Large Message Broadcast Using NCCL and CUDA-Aware MPI for Deep Learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 15–22, New York, NY, USA. ACM. [26](#), [96](#)
- Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. J. (2012). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38:37–51. [4](#), [12](#)
- Bosilca, G., Bouteiller, A., Herault, T., Lemarinier, P., Saengpatana, N. O., Tomov, S., and Dongarra, J. J. (2011). Performance portability of a gpu enabled factorization

- with the dague framework. In *2011 IEEE International Conference on Cluster Computing*, pages 395–402. 4, [30](#)
- Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. [92](#)
- Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., and Tomov, S. (2013). clMAGMA: High Performance Dense Linear Algebra with OpenCL. In *International Workshop on OpenCL, IWOCL 2013*, Atlanta, Georgia, USA. [15](#)
- Chu, C. H., Hamidouche, K., Venkatesh, A., Awan, A. A., and Panda, D. K. (2016). CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 726–735. [26](#)
- Cosnard, M., Jeannot, E., and Yang, T. (1999). Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 413–421. [12](#)
- Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '93*, pages 1–12, New York, NY, USA. ACM. [95](#)
- Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU-3 Workshop*, pages 63–74, New York, NY, USA. [19](#)

- Eicken, T., Culler, D., Goldstein, S., and Schauser, K. (1992). Active Messages: A Mechanism for Integrated Communication and Computation. In *ISCA '92*, pages 256–266. [66](#)
- Fogue, M., Igual, F. D., Quintana-Orti, E. S., and van de Geijn, R. A. (2010). Retargeting PLAPACK to clusters with hardware accelerators. In *High Performance Computing and Simulation (HPCS)*, pages 444–451. IEEE. [15](#)
- Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. [17](#)
- Graham, R., Venkata, M. G., Ladd, J., Shamis, P., Rabinovitz, I., Filipov, V., and Shainer, G. (2011). Cheetah: A Framework for Scalable Hierarchical Collective Operations. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 73–83. [9](#), [25](#)
- Gropp, W. (2002). Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK. Springer-Verlag. [17](#)
- Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789 – 828. [17](#)
- Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. (2011). Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321. [12](#)

- Hockney, R. W. (1994). The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.*, 20(3):389–398. [95](#)
- Huang, W., Santhanaraman, G., Jin, H., Gao, Q., and Panda, D. (2007). Design and implementation of high performance mvapich2: Mpi2 over infiniband. [17](#)
- Jenkins, J., Dinan, J., Balaji, P., Peterka, T., Samatova, N., and Thakur, R. (2014). Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2627–2637. [24](#)
- Kandalla, K., Subramoni, H., Vishnu, A., and Panda, D. K. (2010). Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. [9](#), [25](#), [96](#)
- Karonis, N. T., de Supinski, B. R., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. (2000). Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 377–384. [25](#)
- Kielmann, T., Bal, H. E., and Verstoep, K. (2000). Fast Measurement of LogP Parameters for Message Passing Platforms. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 1176–1183, London, UK, UK. Springer-Verlag. [95](#)
- Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. (1999). MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '99, pages 131–140, New York, NY, USA. ACM. [24](#)

- Kim, K., Eijkhout, V., and van de Geijn, R. A. (2012). Dense matrix computation on a heterogenous architecture: A block synchronous approach. Technical Report 63, FLAME Working Note. 16
- Lawlor, O. (2009). Message passing for GPGPU clusters: CudaMPI. In *CLUSTER'09.*, pages 1–8. 23
- Lima, J., Broquedis, F., Gautier, T., and Raffin, B. (2013). Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In *Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 105–112. 16
- Ltaief, H., Tomov, S., Nath, R., Du, P., and Dongarra, J. (2011). A scalable high performant cholesky factorization for multicore with gpu accelerators. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science, VECPAR'10*, pages 93–101, Berlin, Heidelberg. Springer-Verlag. 13
- Ma, T., Bosilca, G., Bouteiller, A., and Dongarra, J. (2012). HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 970–982. 25
- MPI Forum (1995). MPI: A Message Passing Interface Standard. <http://www.mpi-forum.org/>. 6, 17, 19, 20, 21, 22
- NVIDIA (2015). NVIDIA GPUDirect. <https://developer.nvidia.com/gpudirect>. 3
- NVIDIA (2016a). NVIDIA CUDA Basic Linear Algebra Subroutines. <https://developer.nvidia.com/cublas>. 30
- NVIDIA (2016b). NVIDIA CUDA Toolkit Documentation. <http://docs.nvidia.com/cuda/index.html>. 3, 61
- NVIDIA (2016). NVIDIA NCCL. <https://github.com/NVIDIA/nccl>. 26

- Oden, L., Klenk, B., and Frning, H. (2014). Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 483–492. [26](#)
- Parsons, B. S. and Pai, V. S. (2014). Accelerating MPI Collective Communications Through Hierarchical Algorithms Without Sacrificing Inter-Node Communication Flexibility. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14*, pages 208–218, Washington, DC, USA. IEEE Computer Society. [25](#)
- Pješivac-Grbović, J., Angskun, T., Bosilca, G., Fagg, G. E., Gabriel, E., and Dongarra, J. J. (2007). Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143. [88](#), [94](#), [96](#)
- Plaat, A., Bal, H. E., and Hofman, R. F. H. (1999). Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 244–253. [21](#)
- Potluri, S., Bureddy, D., Wang, H., Subramoni, H., and Panda, D. (2013). Extending OpenSHMEM for GPU Computing. In *IPDPS'13*, pages 1001–1012. [23](#)
- Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. (2009). Solving dense linear systems on platforms with multiple hardware accelerators. *ACM Sigplan Notices*, 44(4):121–130. [15](#)
- Ross, R., Miller, N., and Gropp, W. (2003). Implementing Fast and Reusable Datatype Processing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer Berlin Heidelberg. [7](#), [19](#)

- Sawa, Y. and Suda, R. (2010). *Autotuning Method for Deciding Block Size Parameters in Dynamically Load-Balanced BLAS*, pages 33–48. Springer New York, New York, NY. [47](#)
- Schneider, T., Gerstenberger, R., and Hoefer, T. (2012). Micro-Applications for Communication Data Access Patterns and MPI Datatypes. In *EuroMPI'12*, pages 121–131. [19](#)
- Singh, A. K., Potluri, S., Wang, H., Kandalla, K., Sur, S., and Panda, D. K. (2011). MPI Alltoall Personalized Exchange on GPGPU Clusters: Design Alternatives and Benefit. In *2011 IEEE International Conference on Cluster Computing*, pages 420–427. [25](#)
- Song, F., Tomov, S., and Dongarra, J. (2012). Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA. ACM. [16](#)
- Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., and Panda, D. K. (2012a). Design of a Scalable InfiniBand Topology Service to Enable Network-topology-aware Placement of Processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 70:1–70:12, Los Alamitos, CA, USA. IEEE Computer Society Press. [25](#)
- Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., and Panda, D. K. (2012b). Design of a Scalable InfiniBand Topology Service to Enable Network-topology-aware Placement of Processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 70:1–70:12, Los Alamitos, CA, USA. IEEE Computer Society Press. [96](#)

- Tipparaju, V., Nieplocha, J., and Panda, D. (2003). Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS '03*, pages 84.1–, Washington, DC, USA. IEEE Computer Society. 25
- Top500 (2016). <http://www.top500.org>. 1
- vandeVaart, R. (2014). Open MPI with RDMA support and CUDA. In *NVIDIA GTC'14*. 73, 77
- Wang, H., Potluri, S., Bureddy, D., Rosales, C., and Panda, D. (2014). GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2595–2605. 23, 81
- Wang, H., Potluri, S., Luo, M., Singh, A., Ouyang, X., Sur, S., and Panda, D. (2011a). Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. In *CLUSTER'11*, pages 308–316. 23
- Wang, H., Potluri, S., Luo, M., Singh, A. K., Sur, S., and Panda, D. K. (2011b). MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Computer Science - Research and Development*, 26(3-4):257–266. 23
- Wang, L., Wu, W., Xu, Z., Xiao, J., and Yang, Y. (2016). BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing. In *ICS'16*, Istanbul, Turkey. 65
- Yu, D., Eversole, A., Seltzer, M., Yao, K., Kuchaiev, O., Zhang, Y., Seide, F., Huang, Z., Guenter, B., Wang, H., Droppo, J., Zweig, G., Rossbach, C., Gao, J., Stolcke, A., Currey, J., Slaney, M., Chen, G., Agarwal, A., Basoglu, C., Padmilac, M., Kamenev, A., Ivanov, V., Cypher, S., Parthasarathi, H., Mitra, B., Peng, B., and Huang, X. (2014). An introduction to computational networks and the computational network toolkit. Technical report. 21

Zee, F. G. V., Chan, E., v. d. Geijn, R. A., Quintana-Ort, E. S., and Quintana-Ort, G. (2009). The libflame library for dense matrix computations. *Computing in Science Engineering*, 11(6):56–63. [16](#)

Zhu, H., Goodell, D., Gropp, W., and Thakur, R. (2009). *Hierarchical Collectives in MPICH2*, pages 325–326. Springer Berlin Heidelberg, Berlin, Heidelberg. [24](#)

Vita

Wei Wu was born in Jintan, Jiangsu Province, China, on January 18, 1986. After completing his education at Hualuogeng High School, in 2004, he entered Beijing Institute of Technology, and received his Bachelor's degree in Software Engineering in July 2008. After finishing his Master's degree in Computer Engineering at Purdue University Calumet in December 2010, he enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville in August 2011. Meanwhile, he worked as a Graduate Research Assistant at the Innovative Computing Laboratory (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His research interests are focused on high performance computing, with a concentration on the GPUs with variety of programming model, e.g. data-flow and message passing (MPI) models. While at ICL, he actively participated in several projects including PaRSEC and Open MPI. He completed a summer internship at Oak Ridge National Laboratory in 2014 and another one at AMD Research in 2015. He was also very active at serving in the high performance computing communities: he served as a student volunteer in ACM/IEEE Supercomputing conferences from 2013 to 2014. Wei Wu is expected to receive his Doctor of Philosophy degree in April, 2017. After graduation, he will be pursuing his academic career at Los Alamos National Laboratory as a Research Scientist.