Tiling-Based Programming Model for GPU Clusters Targeting Structured Grids

by

Burak Bastem

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

April 2, 2019

Tiling-Based Programming Model for GPU Clusters Targeting Structured Grids

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Burak Bastem

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Asst. Prof. Didem Unat

_____

Asst. Prof. Ayşe Yılmazer

_____

Asst. Prof. Alptekin Küpçü

Date: _____

To my family

ABSTRACT

Currently, more than 25% of supercomputers in TOP500 list employ GPUs due to their massively parallel and power-efficient architectures. However, programming GPUs efficiently in a large-scale system is a demanding task not only for computational scientists but also for programming experts because it requires generating GPU-specific code, managing distinct address spaces and handling communication. While these requirements reduce productivity, they also limit the portability of the application. There are pragma-based programming models such as OpenACC offering portability and productivity with code annotations. However, they require performance tuning and lack support for programming GPU clusters. Handling communication efficiently is essential in the cluster environment. There are related works such as Cluster-SkePU handling communication and offering skeleton-based abstractions to ease GPU programming, but such abstractions arguably limit programming flexibility. Moreover, most of the works in the literature do not consider the bandwidth bottleneck of the interconnect that links GPUs to hosts. Some task-based approaches provide overlapping data transfers with computation as a solution to the interconnect bandwidth limit, but they hold programmer responsible for GPU-specific code and task scheduling.

To ease the programming effort, increase the portability and optimize communication, we propose a tiling-based programming model for structured grid problems running on a GPU cluster. The model implicitly applies data decomposition with TiDA and manages distinct address spaces with CUDA. It automatically generates GPU-specific code itself by leveraging OpenACC annotations behind a uniform interface for CPUs and GPUs. Furthermore, it handles data transfers and communication and overlaps them with computation by exploiting CUDA streams and non-blocking

MPI routines. We demonstrate the effectiveness of the programming model on a heat simulation and a real-life cardiac modeling. The results show that it successfully overlaps communication and achieves good speedup.

ÖZETÇE

Günümüzde, TOP500 listesindeki süper bilgisayarların % 25'inden fazlası, büyük ölçüde paralel ve güç açısından verimli mimarileri nedeniyle GPU'ları kullanmaktadır. Ancak, büyük ölçekli bir sistemde GPU'ların verimli bir şekilde programlanması, yalnızca hesaplamalı bilim dalındaki insanlar için değil, aynı zamanda programlama uzmanları için de zorlu bir iştir çünkü bu işte GPU'ya özel kod üretilmeli, farklı bellekler yönetilmeli ve iletişim ele alınmalıdır. Bu gereksinimler üretkenliği azaltırken, uygulamanın taşınabilirliğini de sınırlar. Kod yönergeleriyle taşınabilirlik ve üretkenlik sunan OpenACC gibi direktif tabanlı programlama modelleri bulunmaktadır. Ancak bu modeller performansı iyileştirme amaçlı ayarlamalar gerektirir ve GPU'lu kümelenmiş sistemlerin programlanmasını desteklemezler. Kümelenmiş sistemlerde iletişimin verimli bir şekilde ele alınması gerekir. İletişimi ele alan ve GPU programlamayı kolaylaştırmak için iskelet tabanlı soyutlamalar sunan çalışmalar bulunmaktadır, ancak bu soyutlamalar programlamada yeterli esnekliği sağlayamazlar. Üstelik, literatürdeki çalışmaların çoğu, GPU'ları ana işlemciye bağlayan ara bağlantının bant genişliği kısıtını dikkate almamaktadır. Bazı görev tabanlı yaklaşımlar, bu kısıta çözüm olması amacıyla hesaplama ile örtüşen veri aktarımlarını kullanmaktadır fakat bu yaklaşımlarda programlayıcı GPU'ya özgü kodun yazılmasından ve görev zamanlamasından sorumludur.

Programlama çabasını kolaylaştırmak, taşınabilirliği artırmak ve transferleri optimize etmek amacıyla GPU'lu kümelenmiş sistemlerde yürütülecek yapısal çözüm alanı problemleri için bloklamaya dayalı programlama modeli sunuyoruz. Model, üstü kapalı bir şekilde, TiDA ile veriyi ayrıştırır ve CUDA ile farklı bellekleri yönetir. CPU ve GPU'lar için tek tip bir arayüzün arkasında OpenACC'nin yönergelerinden yararlanarak otomatik bir şekilde GPU'ya has kod üretir. Ek olarak, veri aktarımlarını ve

iletişimi yönetir ve bunları, CUDA akışları ve bloklamayan MPI rutinlerini kullanarak hesaplamalarla örtüştürür. Programlama modelinin etkinliğini bir ısı simülasyonu ve gerçek hayatta kullanılan bir kardiyak modellemesi üzerinde gösterdik. Sonuçlar, programlama modelinin iletişimi başarıyla örtüştürdüğünü ve iyi hızlanma sağladığını göstermektedir.

ACKNOWLEDGMENTS

First of all, I would like to express my deepest thanks and gratitude to my advisor Didem Unat for her great help and understanding and for the countless opportunities she has provided. She has become a precious source for my personal and professional development. I am indebted to my thesis committee members Ayşe Yılmazer and Alptekin Küpçü. Alptekin Küpçü is also the one who has never turned me down when I asked for his help. I would like to sincerely thank my advisor Didem Unat, John M. Shalf, Ann S. Almgren and Fikri Karaesmen for making my visit to Lawrence Berkeley National Laboratory possible. I appreciate the funds for the visit from U.S. Department of Energy and Koç University College of Engineering. More importantly, I am truly thankful to John M. Shalf, Ann S. Almgren and Weiqun Zhang for the collaboration and experience there. As for the experience at Koç University ParCore-Lab, my special thanks go to Muhammad Nufail Farooqi, Hassan Salehe Matar and my other labmates. I would also like to thank Seçil Arslan who helped me to continue my research along with my work in the last phase of my Master's. Most importantly, these would not be possible without the aid and support of my family.

I am grateful to TÜBİTAK and Koç University for funding my Master's. I conducted my research on TiDA machine at Lawrence Berkeley National Laboratory, Summitdev at Oak Ridge National Laboratory and Piz Daint at Swiss National Supercomputing Center, so I am thankful to these institutions as well.

Finally, I acknowledge that this thesis is based on [Bastem et al., 2017] and another paper which I have co-authored with my advisor and is under review at the moment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

HPC	-	High Performance Computing
CPU	-	Central Processing Unit
GPU	-	Graphics Processing Unit
Host	-	CPU and its memory
Device	-	GPU and its memory
CUDA	-	Compute Unified Device Architecture
OpenACC	-	Open Accelerators
OpenMP	-	Open Multi-Processing
MPI	-	Message Passing Interface
TiDA	-	Tiling Data Abstractions
RDMA	-	Remote Direct Memory Access
P2P	-	Peer to peer

Chapter 1

INTRODUCTION

Heterogeneous systems that combine accelerators and general-purpose multicores are no longer considered to be an exotic architecture but constitutes more than 25% of TOP500 systems [top, 2019]. GPU-based heterogeneous systems have provided high performance for particularly structured grid problems that are used for solving PDE equations as well as image processing applications [Micikevicius, 2009, Kim et al., 2013, Zhou et al., 2012]. One of the difficulties of GPU programming for attaining good performance is that application developers need to have deep understanding of disjoint host and device address spaces. CUDA being the most prominent GPU programming model unfortunately requires nontrivial programming effort. Although it provides unified memory to give the illusion of a shared address space for convenience, it is necessary to use hints and prefetching for a good performance which defeats its purpose. Ideally, a programmer would like to write a single version of an application that will perform well on both homogenous and heterogeneous multicore systems. Pragma-based programming models such as OpenMP or OpenACC promise to achieve this goal with code annotations [Lee and Eigenmann, 2010, Wolfe, 2010, Kim et al., 2016]. These accelerator models are built on existing concepts (e.g. parallel region, parallel for loop) to provide a unified programming model for CPUs and GPUs. However, the achieved performance is typically suboptimal without aggressive performance tuning [Hoshino et al., 2013].

PCI Express (PCIe) link within the host has a considerably lower bandwidth than CPU or GPU memory bandwidths. NVLink is a high-bandwidth communication protocol between the CPU and the GPU, and between GPUs [NVDIA, 2014], which

allows faster transfer speed compare to PCIe Gen3. While NVLink technology improves the data transfer rate, the compute capability of GPUs continues to improve as well. As a result, in order not to lose the performance benefits of GPUs, application developers should hide data transfer latency.

Yet, providing solutions to aforementioned issues does not suffice for today's applications and systems. Developers need to program the whole system containing many GPUs for a larger scale. Communication is critical for such scalability; therefore, hiding it becomes necessary which requires even more nontrivial programming effort.

We propose a tiling-based high-level asynchronous programming model for structured grid problems to simplify programming on GPU clusters by providing a uniform programming interface for CPUs and GPUs and to optimize performance by hiding on-node and off-node transfers. We combine the concept of tiling, which is one of the techniques commonly used for optimizing applications for data locality [Rivera and Tseng, 2000, Unat et al., 2017], with GPU programming in a single model to overlap data transfers with execution of tiles. The model offers a simple interface for data decomposition with tiles and implicitly handles memory management on both CPUs and GPUs, transfers for on-node and off-node communication and kernel code generation for GPUs.

The programming model is implemented as a library that leverages existing mature software for maintainability. The library extends TiDA [Unat et al., 2016] for tiling and employs CUDA streams for overlapping on-node memory transfers with computation. It supports non-blocking MPI transfers for communication across the execution units and is able to directly transfer memories when GPUDirect is available. It relies on OpenACC for the kernel code generation and CUDA pinned memory for address space management.

We demonstrate the effectiveness of the programming model on two different applications: a heat simulation and a real-life cardiac modeling. Heat simulation is a memory-intensive application in which transfers and communication dominate the computation time. Cardiac modeling is a compute-intensive application where our

library hides the transfer and communication costs greatly.

The outline of this thesis is as follows. We present related works in Chapter 2, give background in Chapter 3, and introduce our GPU-cluster programming model in Chapter 4. Chapter 5 gives details of the library that implements asynchronous computation and overlapping communication. In Chapter 6, we evaluate the performance with two applications on a GPU cluster. Finally, we conclude the thesis in Chapter 7.

Chapter 2

RELATED WORK

In the literature, there are various works developing abstractions and optimizations for GPU programming. Only small number of them propose all-in-one solution while most of them proposing partial solutions.

CUDA, OpenCL and OpenACC are the most complete GPU programming models. CUDA and OpenCL are low-level and require programmer to manage both host and device memories and implement kernels. To abstract memory management, CUDA provides unified memory. With unified memory, one can even use multiple GPUs on a single host without having to manage distinct host and device memories. However, unified memory degrades the application performance. For better performance, programmer needs to use explicit hints and prefetching with unified memory, which is kind of memory management. Most importantly, CUDA and OpenCL do not support programming GPU clusters. Compare to CUDA and OpenCL, OpenACC provides some convenience to programmer with its pragma-based directives. Nevertheless, programmer should have deep knowledge about directives and explicitly manage memory to get a comparable performance to CUDA. Similar to CUDA and OpenCL, OpenACC also lacks support for programming GPU clusters.

Programming GPU clusters requires programmer to use communication primitives in addition to GPU programming primitives. Furthermore, programmer needs to handle interaction between these two kinds of primitives in an optimized manner for a better performance. With these in mind, some works extend GPU programming models to have inter-GPU communication support, or extend communication tools to have support for optimized GPU memory operations across the nodes in clusters. CUDASA [Strengert et al., 2008], CudaMPI [Lawlor, 2009] and dCUDA [Gysi et al., 2016]

modify CUDA to have MPI-like features for inter-node communication, so that using only CUDA would be sufficient. SnuCL [Kim et al., 2012] and dOpenCL [Kegel et al., 2012] modify OpenCL for the same purpose. Alternatively, XACC [Nakao et al., 2014] combines PGAS and OpenACC to enable GPU execution in a cluster without means of communication. [Potluri et al., 2013, Wang et al., 2014, Wu et al., 2016, Aji et al., 2016] modify MPI to have optimized inter-node GPU memory operations. However, these works focus on convenient and optimized ways just for communication. Our programming model focuses on all aspects of programming GPU clusters including distinct memory management and kernel generation while employing communication optimizations.

There are works which offer only partial abstractions and propose improvements to OpenACC. [Komoda et al., 2013, Matsumura et al., 2018] provide multi-GPU execution for OpenACC, but the execution is limited to multiple GPUs which share the same host. [Cui et al., 2017] introduces partitioning and pipelining to OpenACC for better performance. Nevertheless, necessity of managing memory with pragmas and lack of support for programming GPU clusters still remain. Similar to OpenACC, OpenMP and [Unat et al., 2011, Lee and Eigenmann, 2010, Bueno et al., 2012] use annotation based approach for programming GPUs. In addition to the fact that annotation is not a complete abstraction, they also have the same drawbacks of OpenACC. Moreover, [Unat et al., 2011, Lee and Eigenmann, 2010] use in-house compilers, which makes it difficult to maintain in long term. Others provide high-level means instead of a complete abstraction for memory management. In Thrust [Bell and Hoberock, 2012], Kokkos [Edwards et al., 2014] and C++ AMP [Gregory and Miller, 2012], programmer specifically defines where the data should reside. For example, a Thrust programmer uses *thrust::device_vector* to place data on a device [Bell and Hoberock, 2012]. [Jablin et al., 2011] and many others only automate memory transfers between host and device, and leave the implementation of GPU kernels to the programmer. As a recent work, Groute [Ben-Nun et al., 2017] proposes high-level abstractions for inter-GPU communication on a single host. In [Augonnet et al., 2009, Augonnet et al.,

2012, Agullo et al., 2018, Song and Dongarra, 2012, Gautier et al., 2013, Wen et al., 2014, Grasso et al., 2014], authors consider memory transfers and kernels of an application as tasks and focus on task scheduling. By doing so, they provide some level of abstraction on memory management, but hold the programmer responsible for GPU kernels. More importantly, task-based programming models introduce new responsibilities for programmer. There are various other works which propose GPU programming abstractions or implementations for specific areas or applications like matrix computation [Song et al., 2012], dense linear algebra problems [Song and Dongarra, 2015], real-time applications [Schaetz and Uecker, 2012], etc. Lastly, aforementioned works target a single GPU or multiple GPUs on a single host except for [Bueno et al., 2012, Edwards et al., 2014, Augonnet et al., 2012, Agullo et al., 2018]. Compared to mentioned studies, our programming model offers all the necessary abstractions while targeting multiple GPUs on a cluster.

Recent literature abstracts memory management with custom data types and GPU kernels with skeletons. Skeletons are pre-implemented primitive kernels like plus, minus, map, reduce, etc. SkePU [Enmyren and Kessler, 2010], SkelCL [Steuwer et al., 2011, Steuwer and Gorlatch, 2013, Steuwer et al., 2012] and Marrow [Marques et al., 2013] are all skeleton based libraries for GPU programming. Additionally, [Dastgeer et al., 2011] and [Dastgeer and Kessler, 2016] are incremental works to SkePU, respectively providing auto-tuning and asynchronous execution for skeleton calls. PSkel [Pereira et al., 2015] is another skeleton based library focusing on stencil computation. All, except [Marques et al., 2013], support multiple GPUs, but on a single host. Muesli [Ernsting and Kuchen, 2012] and Cluster-SkePU [Majeed et al., 2013] integrate means of communication from MPI and propose programming GPU clusters with skeletons. Even though skeleton based approaches provide abstraction for GPU kernels, they arguably complicate even simple mathematical operations. Moreover, programmers need to implement skeletons themselves for complex operations that cannot be implemented using existing skeletons. In our programming model that we developed as a library, programmers implement computation inside lambda expres-

sions from which kernels are automatically generated. Moreover, instead of offering means for communication, our programming model handles communication itself.

Chapter 3

BACKGROUND

In this chapter, we first describe graphics processing unit (GPU), then explain how to program a GPU using two popular programming models, CUDA and OpenACC. We also compare convenience and performance of these programming models. Finally, we discuss communication.

3.1 GPU

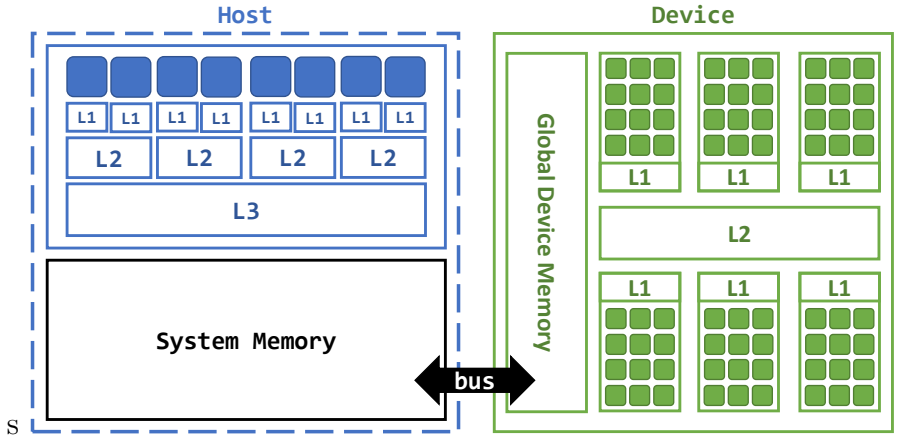

Figure 3.1: Abstract heterogeneous architecture

CPUs and GPUs are two distinct devices each having its own memory. A CPU employs powerful cores with more sophisticated cache hierarchy and offers low latency, making it superior for general usage. A GPU employs many more cores that are less complex and powerful but more efficient. Its cores and parallel architecture enable producing higher throughput for data parallel tasks. However, GPUs cannot be employed for general usage like running an operating system and are in need of

a host to which they are connected via PCI-e or NVLink. Figure 3.1 illustrates a modern heterogeneous architecture in which a GPU is interconnected to a host.

In terms of CUDA capable GPUs, multiple cores are grouped into streaming multiprocessors (SM) as demonstrated in Figure 3.1. Each SM has its own cache, registers and warp schedulers. A warp is a set of threads working in single-instruction-multiple-thread (SIMT) fashion. Multiple warps form a thread block and multiple thread blocks form a grid on which GPU code executes. Lastly, thread blocks can run concurrently and be assigned to different SMs.

3.2 GPU Programming

GPU programming requires a programmer to define functions called kernels that are off-loaded to the GPU. The programmer also implements a host-side code that manages data transfers between host and device, sets up kernel parameters (block and grid size), and launches kernels. CUDA and OpenACC offer slightly different programming interfaces for these operations. CUDA is essentially C/C++ API with few extensions and gives the programmer greater control with detailed interface. OpenACC uses a pragma-based interface, which arguably provides a high level interface and high productivity to the programmer.

3.2.1 Kernels

In CUDA, programmer explicitly implements parallel kernels. Each kernel contains the code that is executed by a single thread. When invoking a kernel, the programmer specifies the number of threads in a block and number of blocks in a grid that concurrently executes the kernel. Tuning these parameters greatly affects the performance. The programmer can also tune the performance on the GPU by using on-chip memories.

In OpenACC, compiler generates the kernel code. Programmer starts a parallel section with a pragma as in OpenMP. Parallel sections are annotated with the *parallel* construct and inside a parallel section loops are annotated with the *loop* construct for

GPU parallelization. If a loop is a tightly nested loop, programmer can compliment the *loop* construct with a *collapse* clause. The *collapse* clause takes an integer which indicates how many loops are tightly nested and can be parallelizable. For clarity, programmer can combine *parallel* with *loop* just before a tightly nested loop. There is also *kernels* construct in OpenACC to identify loop parallelism. While compiler relies on programmer's hints to map parallelism in a *parallel* construct, it automatically determines parallelism in a *kernels* construct by analyzing loops in the scope.

Depending on pragmas and their attributes, OpenACC compiler generates an optimized kernel with default kernel parameters. With the help of pragma attributes, one can control these kernel parameters. For example, *num_gangs*, *num_workers* and *vector_length* correspond to number of CUDA blocks in a grid, number of CUDA warps in a block and number of CUDA threads in a warp, respectively.

3.2.2 Memory Management

Main memories of the host and the device are physically disjoint and separated by a bus. Data that is shared by the host and the device must be allocated in both memories, and explicitly transferred between them by the application.

In a default memory transfer between host and device, CUDA copies data from pageable host memory to pinned host memory and then to device memory which is allocated with *cudaMalloc* or vice versa. The copy may seem unnecessary but CUDA requires page-locking (pinning) the data during the transfer. Figure 3.2 illustrates the behavior on the system memory in a default memory transfer. Even though CUDA runtime handles copy between pageable memory and pinned memory itself, this operation lowers the transfer performance. To improve the performance of transfers, CUDA provides an option, where it lets programmer allocate pinned host memory with *cudaMallocHost* function and use this memory for data. This way application data can directly be copied from/to device. Apart from pinned memory, NVIDIA introduced unified memory in CUDA 6 to alleviate some of the memory management complexity. In case of use of unified memory, programmer sees a single address space

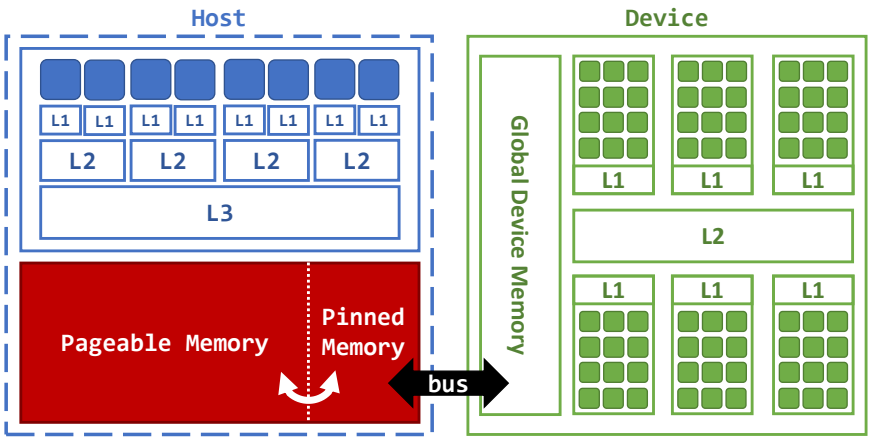

Figure 3.2: In a memory transfer between host and device CUDA implicitly copies from pageable memory to pinned memory or vice versa.

instead of two distinct address spaces and is not responsible for memory transfers. Figure 3.3 demonstrates how a programmer sees memory in a heterogeneous system when unified memory is used. Programmer allocates unified memory that can be accessed by the host and device with a single data pointer using *cudaMallocManaged*. CUDA runtime handles necessary data transfers internally where migration between two memories takes place on demand.

In OpenACC, memory management by the programmer is optional. Compiler automatically generates code for memory transfers before and after each generated kernel. However, this causes extremely low performance. Since compiler cannot have as much information as the programmer does about which data is needed and when on the GPU, memory management by the programmer becomes necessary for good performance. OpenACC provides memory management in two ways; with structured data lifetime and unstructured data lifetime. Programmer can create a scope with the *data* construct, which specifies the lifetime of the data in that scope. With the data scope, s/he can specify a memory copy or indicate that data is already in device or declare a pointer already pointing to device memory etc. For unstructured data lifetime, *enter data* and *exit data* directives are used. In this case, specified data is

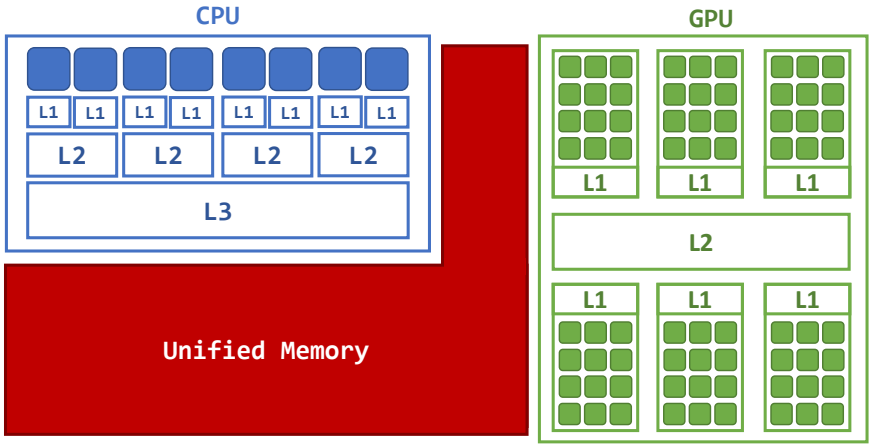

Figure 3.3: A programmer sees a single address space with unified memory.

not limited to an enclosed scope. It can live on device from the beginning of the *enter data* until the *exit data*.

OpenACC also provides options to use pinned or managed memory instead of pageable memory with compiler flags. While *-ta=tesla:pinned* flag allows usage of pinned host memory, *-ta=tesla:managed* flag allows usage of managed memory which corresponds to unified memory in CUDA.

3.2.3 Performance of Programming Models

We have implemented a stencil kernel which solves heat equation using CUDA only, OpenACC only and combination of both; then, compared their performance, convenience and functionality for two main GPU operations: data transfers and kernels. In the implementation of combined CUDA and OpenACC, we used CUDA for memory management and OpenACC for kernel code generation. We also tested pageable, pinned and unified memory for each execution model. All implementations run with data of size 384^3 for 100 iterations on a NVIDIA Tesla K40m.

Figure 3.4 shows the running times of GPU implementations. Timing includes both data transfer and compute times. The CUDA-only pinned memory version has the best performance with the lowest running time. In all cases, pageable memory and

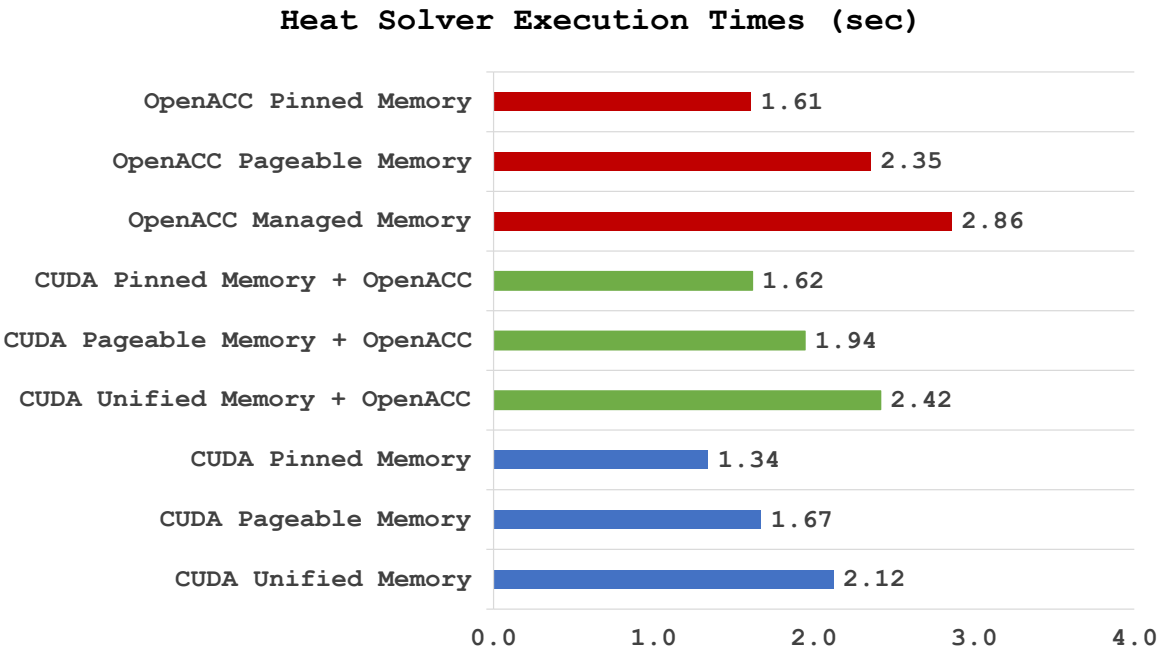

Figure 3.4: Performance of different execution model implementations of heat solver. CUDA + OpenACC indicates implementations that rely on CUDA for memory management and OpenACC for kernel code generation.

virtual memory versions achieve lower performance than that of pinned as expected. OpenACC achieves lower performance than CUDA for the each memory management versions. When we manage the memory with CUDA pinned memory but rely on OpenACC for kernel code generation, the performance of OpenACC improves and gets much closer to that of CUDA. There are two reasons why CUDA still performs slightly better. Firstly, the pure CUDA implementation launches one kernel per time step to both calculate heat equation and update data boundaries. However, OpenACC implementations launch one kernel to calculate heat equation and multiple kernels to update data boundaries due to loop level parallelism. Therefore, there is an overhead for each kernel launch. Second, we tune the grid and block geometry for the pure CUDA version; however, we let compiler decide the geometries for the OpenACC kernels. Note that none of the implementations use on-chip shared device memory.

Based on the results, we decided to use pinned host memory with CUDA for

explicit memory management and data transfers, but leverage OpenACC for kernel code generation in our library implementation. This frees the users from implementing low-level CUDA kernels or relying on in-house compiler. However, the user still benefits from higher performance that comes with CUDA pinned memory. Pinned host memory is also necessary for the library to increase concurrency with CUDA streams by overlapping operations such as memory transfers and kernel executions, which will be discussed shortly.

3.3 Communication

People in the field of High Performance Computing (HPC) predominantly use processes and MPI for their communication to program clusters and supercomputers. MPI is a library standard for message passing and stands for Message Passing Interface. Such a standard provides portability which allows a code to run on different architectures without change. There are libraries implementing MPI specifications like OpenMPI, MVAPICH, etc.

MPI provides point-to-point and collective communication routines for message passing. Point-to-point communication routines involve two processes. One sends a message and the other receives it. Collective communication routines involve all processes in the communication environment. The routines have blocking and non-blocking versions. In blocking version, a routine does not return until message passing is complete. On the other hand, non-blocking versions immediately return, providing opportunity to hide communication behind computation. To ensure message passing is complete in non-blocking communication, programmer can check the status of message or wait with routines MPI also provides.

Normally, message passing takes place on the host memory and messages involving GPU data needs to be moved to/from GPU. In a send operation, the data should be first transferred to host memory before it is sent to another process. Similarly, in a receive operation, a message from another process is first received at the host memory from where it needs to be transferred to device. GPUDirect technology allows message

passing directly from GPUs, preventing intermediate transfers. There are different types of GPUDirect support for different cases. If communicating processes are on the same host and employ different GPUs that are interconnected to each other, the processes can take advantage of GPUDirect Peer-to-Peer (P2P) Transfer. GPUDirect P2P Transfer allows direct memory transfers between GPUs. Figure 3.5 illustrates such a case.

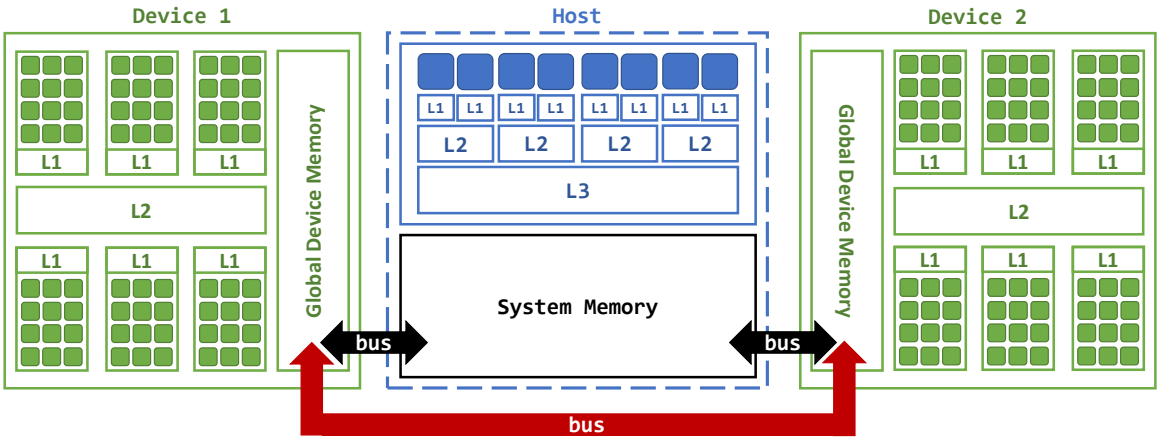

Figure 3.5: GPUDirect P2P Transfer and an interconnect between GPUs allow direct memory transfers between GPUs.

If communicating processes are on different hosts and hosts employ network interfaces supporting GPUDirect Remote Direct Memory Access (RDMA), the processes can send/receive messages directly from GPUs. In other words, a compatible network interface is able to access device memory with GPUDirect RDMA. Figure 3.6 demonstrates this capability.

Finally, there are MPI implementations that adapt these GPUDirect technologies to prevent unnecessary transfers and are referred to as CUDA-aware MPI.

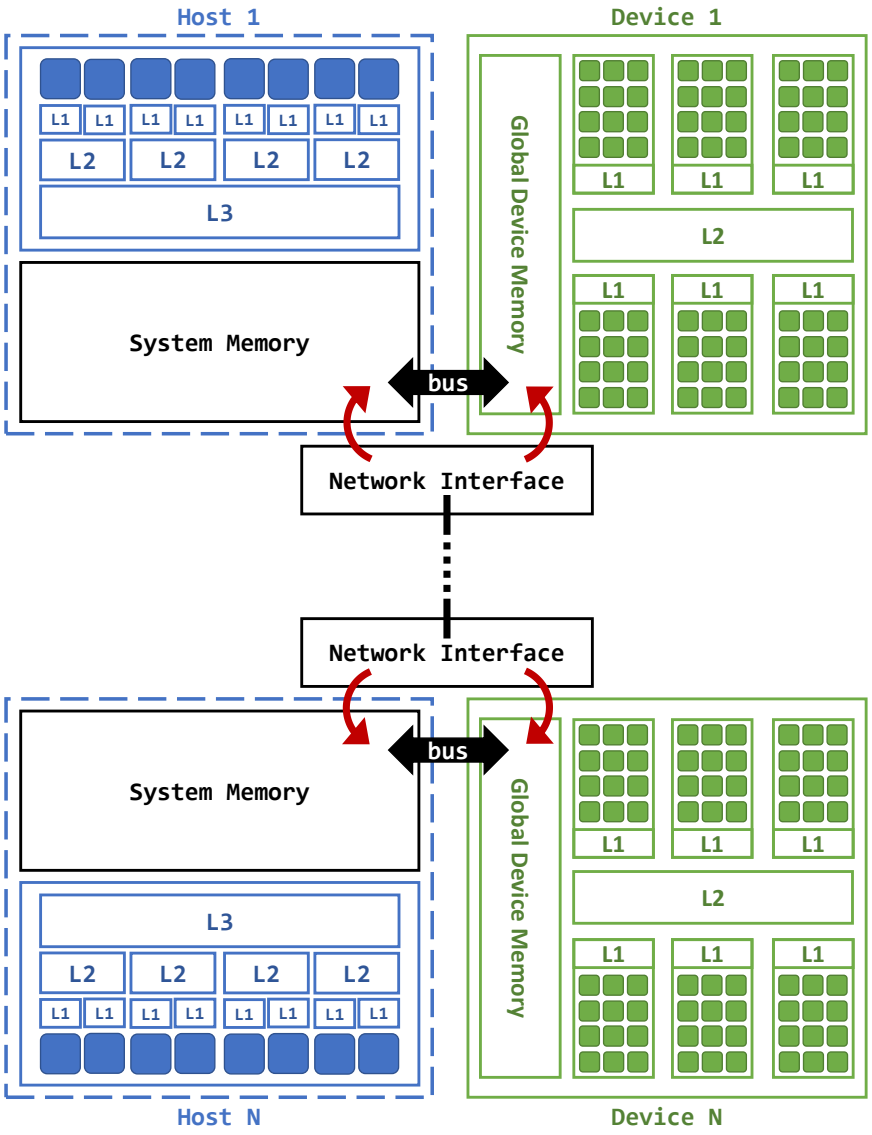

Figure 3.6: GPUDirect RDMA and a compatible network interface allow direct communication from/to GPU memory.

Chapter 4

TILING-BASED PROGRAMMING MODEL

The goal of the tiling-based programming model is to provide productivity and performance. We support productivity with a complete abstraction for data decomposition, memory management, communication and GPU kernels, and provide performance with asynchrony. In this section, we first explain the execution model that the programming model employs, then give the design principles and show its simple user interface with an example.

4.1 Execution Model

The execution model relies on partitioning the application data into tiles. Tiles are handy for many reasons. Firstly, they offer data locality for executions taking place on CPUs because they can provide cache blocking optimizations [Unat et al., 2016, Unat et al., 2017]. Secondly, they allow us to easily distribute work across the GPUs and CPUs. Moreover, tiles enable overlapping as some are being computed, some others

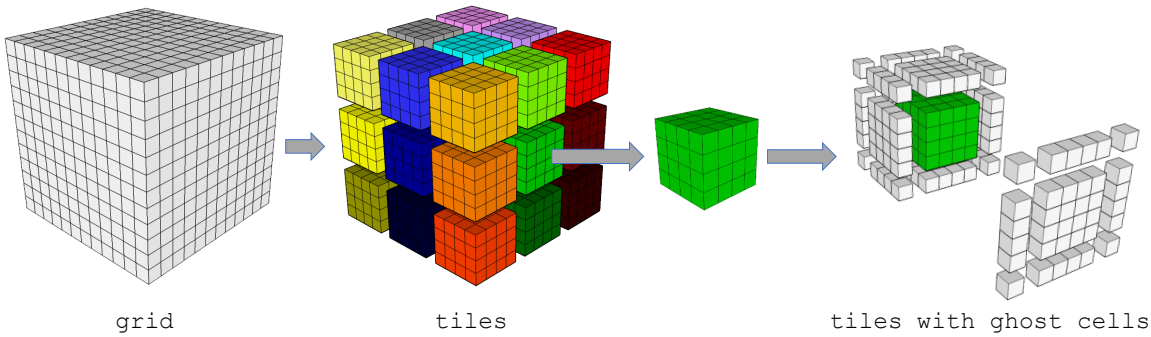

Figure 4.1: Partitioning application data into tiles and representing neighbor cells with ghost cells

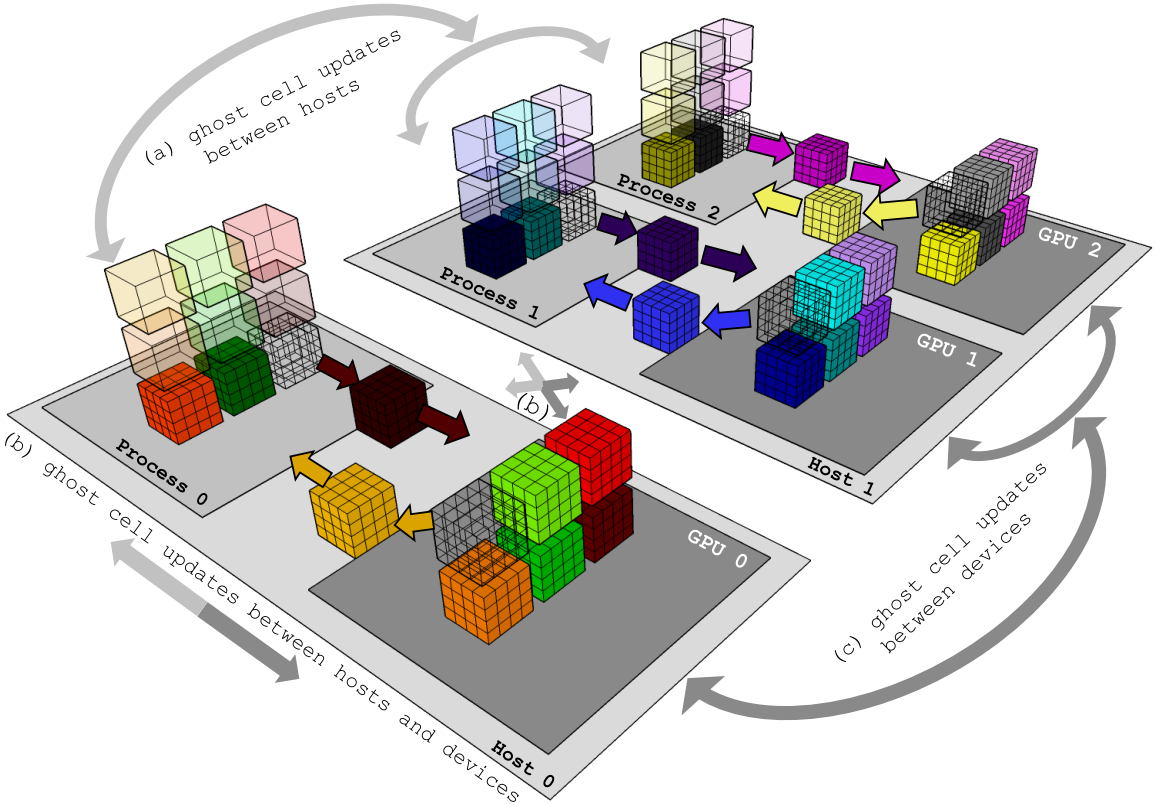

Figure 4.2: Both CPUs and GPUs participate in computing tiles while some tiles are asynchronously being transferred between them.

are transferred. Lastly, tiles eliminate the memory limitation on the device caused by the fact that a GPU typically has smaller memory than a host has. For each tile, the execution model allocates separate physical address spaces in hosts and devices. For the cases where data from other tiles is necessary for computation, the execution model can allocate the tiles with additional cells which are called ghost cells, or halos. Figure 4.1 illustrates the application data partitioned into tiles.

The execution model employs asynchronous computation on CPU and GPU, as well as asynchronous intra-node and inter-node direct memory transfers. As illustrated in Figure 4.2, both CPUs and GPUs participate in computing tiles while some tiles are being transferred between them. At the same time, ghost cell updates can take place asynchronously. No matter where the source and destination tiles are lo-

cated, execution model is capable of transferring ghost cells directly or indirectly. In other words, (a) ghost cell updates between hosts, (b) ghost cell updates between hosts and devices, (c) ghost cell updates between devices are asynchronously and transparently handled. For example, in Figure 4.2, a tile on Process 2 can receive its ghost cells from a tile on GPU 0.

4.2 Programming Interface

The programming model provides three fundamental data structures that are exposed to the programmer: `Tile`, `tile array` and `tile iterator`. `Tile` defines an iteration space on a partition with a lower and an upper bound. `Tile array` holds both the data and metadata regarding the partition. It is responsible for managing host and device memories and updating of ghost cells. `Tile iterator` helps to construct loops, generate iteration spaces on tiles, and traverse them on GPUs and/or CPUs. The programming model makes use of lambda expressions to abstract code generation.

```
1 TileArray ta(app_data, data_dim, tile_dim, ghost_cell_dim);
2 for(TileArray::Iterator i = ta.begin(gpu_exec_ratio); i != ta.end(); ++i) {
3  Tile t = *i;
4  compute(t, [](double* data,int depth,int height,int width,int index){
5   data[index] = ...//computation
6  });
7 }
8 ta.updateGhostCells();
```

Listing 4.1: Programming interface with tiles

In Listing 4.1, Line 1 creates a `TileArray` with application data, its size, tile size, and ghost cell size. Our runtime system automatically partitions and distributes the data among available devices and hosts. This is useful when application data has already been created and processed. A programmer can also start from scratch and create a `TileArray` without any data. Line 2 starts an iteration with for-loop. The syntax of this is the same as the syntax of iterating through a standard C++ list. The iterator `i` is created with `ta.begin(gpu_exec_ratio)` at the loop initialization. With

`gpu_exec_ratio`, programmers can specify what fraction of tiles will be executed on GPUs. They can execute all tiles on CPUs by specifying the ratio as `0` (or `false`), or by simply providing no argument. Setting the ratio to `1` (or `true`) enables a complete GPU iteration, making all tiles run on GPUs. Programmers can take advantage of CPU and GPU executions together by setting a ratio between `0` and `1`. For example, a value of `0.5` results in equal distribution of tiles among GPUs and CPUs. Incidentally, CPU executions leverage multiple threads to take advantage of multiple cores on CPUs. In the loop body, Line 3 accesses a tile by dereferencing the iterator. Then, `compute` function takes the tile and a lambda expression as its arguments. The lambda expression hides the kernel code generation from the programmer and takes a pointer representing the data of a tile, three integers representing the dimensions of data, and another integer representing the iteration index on data. More than one data pointer in the parameter list can be specified if computation requires multiple tiles. Finally, Line 8 updates the ghost cells.

Chapter 5

IMPLEMENTATION

The programming model is developed as a C++ library[1]. which makes use of various existing software to provide abstraction and performance. Next, we give the details of its implementation.

5.1 Memory Allocations

The library uses processes as workers, where each process is responsible for one GPU. Each host process allocates memory space both on the host and device for each tile assigned to it. If the GPU does not have enough memory, it allocates memory for as many partitions as possible in which case partitions share allocations on the device. For all the allocations, including host allocations, the library uses CUDA pinned memory because pageable memory would require an additional copy in a data transfer. Directly using pinned memory eliminates this overhead and yields to better performance [Bastem et al., 2017]. In addition, we employ asynchronous device operations, which are not applicable with pageable memory. Another option would be to use unified memory, which offers a convenient way to manage address spaces with some overhead. With the hardware support for page faults since the Pascal architecture, its performance, as well as capabilities, has increased. Nevertheless, page faults degrade performance due to synchronizations and page table updates [Sakharnykh, 2017]. To avoid this penalty, CUDA provides explicit hints and prefetching, but these limit the concurrency [Sakharnykh, 2017]. Since our library already abstracts distinct memories from the programmer, we choose pinned memory for its superior performance.

[1]Code is available at parcorelab.ku.edu.tr.

5.2 Data Transfers with Streams

Our implementation leverages CUDA streams for the transfers between host and device, which allows us to overlap the transfers with executions taking place on host and device. Streams are queues to which GPU operations are submitted, and run concurrently with one another, so operations on concurrent streams overlap. The programming model assigns a stream to each device tile and submits operations of a tile to its assigned stream. In this way, tiles can be transferred between memories while some others get executed. This is particularly useful when tiles of a process cannot all fit into a single GPU's memory.

Even though the library overlaps memory transfers, it is crucial to avoid unnecessary ones because applications may exhibit memory-intensive behavior in which case there would not be enough computation to hide the transfers. To prevent redundant transfers, we employ a caching mechanism. With the caching mechanism, unnecessary memory transfers between hosts and devices are avoided in two ways. First is lazy initialization of data on GPU. If the programmer chooses to create a tile array without processed data, then we directly initialize the data of tiles on the GPU. Secondly, the programming model keeps tiles on where they are last executed and monitors them so that tiles are not transferred unnecessarily if they are going to be processed on the same device or host again.

5.3 Kernels

The library leverages OpenACC directives to convert lambda expressions provided through the *compute* interface in Listing 4.1 into GPU kernels. The programmer does not need to have knowledge about GPU architecture and how to write code for it while implementing the lambda function. The *compute* function implicitly iterates over tiles and performs the implicit iteration with a loop annotated with OpenACC directives if the tile is intended to run on GPUs. If not, it performs the implicit iteration with a loop annotated with OpenMP directives to be executed on the host.

As a result, the programmer does not need to implement kernels or deal with compiler annotations.

5.4 Communication

Optimizing performance of communication layer is important because it affects the scalability of the applications to take full advantage of a GPU cluster. For this reason, we employ asynchronous communication execution not only between host and device but also between all execution units on-node and off-node with asynchronous CUDA transfers and non-blocking MPI routines. We support GPUDirect technologies for direct transfers which do not need intermediate steps such as a copy between host and device on an inter-node device transfer or on an inter-device transfer. We support direct RDMA transfers from/to a device and direct peer-to-peer transfers among multiple devices via CUDA aware MPI. Since many Top500 supercomputers and GPU workstations have necessary hardware and software for GPUDirect, supporting direct transfers appeals to a wide range of users. However, the library has also support for indirect transfers in case a system does not have the hardware and software available for GPUDirect.

The programming model initiates communication when the programmer requests an update of ghost regions. Briefly, it packs necessary cells of a tile into pre-allocated buffers, sends them to neighbor tiles, and unpacks the ones received from the neighbors. We refer to a group of cells that is from a neighbor tile or needed by a neighbor tile as a `ghost zone`. In a ghost cell update, if the neighbor tiles are on the same process and execution unit, they simply exchange zones. If not, they pack and transfer zones to each other. From a ghost zone point of view, each has a tile they are located on and a source tile they are received from. For example, the green cells on the left tile in Figure 5.1 form a ghost zone and its source tile is the middle tile. When initializing a tile array, the library computes source tile ID for each ghost zone of each tile. It also computes the indices of ghost zones on tiles and their corresponding indices on source tiles. Moreover, it allocates receive/send buffers for ghost zones of each tile at

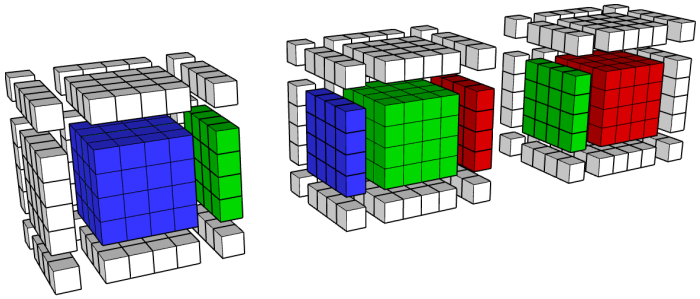

Figure 5.1: Ghost cells from a tile form a ghost zone.

the initialization. These allocations take place on host memories when a tile array is created. The device buffer allocations happen with the device tile allocations at the first GPU iteration.

The library divides ghost cell update into seven phases to take advantage of concurrency by hiding packing/unpacking and communication.

Phase I Each process checks how many zones will be received from other processes and how many zones will be transferred from device to host. Then, it creates MPI receive requests and CUDA events (referred to as *transfer events*) based on these numbers. Each process also creates additional CUDA events (referred to as *stream events*) and pushes one to each stream. All the CUDA events are created with *cudaEventDisableTiming* flag to increase performance and to avoid synchronization issues.

Phase II handles zones that require communication with other processes. Each process first initiates a non-blocking MPI receive for ghost zones coming from other processes using MPI requests created in the previous phase. Then, each packs the zones that will be sent. If a zone belongs to a tile located on a GPU, the process submits a packing kernel to the stream of that tile and continues packing other zones. If the system does not have support for GPUDirect, the process also submits an asynchronous transfer of the package to the host before continuing with other zones. If a zone belongs to a tile located on a host, the process packs the zone and initiates a non-blocking MPI send for its transfer.

Phase III Each process synchronizes streams to which zone packing kernels were submitted in Phase II. This ensures streams have completed packing and the packages are ready to be sent. It also ensures transfers to the host is complete, which were submitted in Phase II for systems not supporting GPUDirect. Then, each process initiates non-blocking MPI sends for the packages.

Phase IV Each process transfers the ghost zones between host and device. If a zone belongs to a tile located on a host, the process packs the zone and submits its asynchronous transfer to the stream of that tile. If a zone belongs to a tile located on a device, the process submits a packing kernel to the stream of that tile. Then, the submission of transfer of the zone to host and a *transfer event* created in Phase I respectively follows the kernel submission. Since these packages are transferred from device to host, CUDA events will be used to ensure transfers are completed before unpacking them on host in a later phase.

Phase V exchanges the ghost zones of tiles that are on the same process and execution unit. The exchange takes place on the host if neighbor tiles reside on the host and takes place on the device if they reside on the device. For each zone on device, an exchange kernel will be submitted to the stream of the destination tile. Before submitting the exchange kernel, each process submits an event synchronization to that stream using *stream event* that was pushed into the stream of source tile in Phase I to ensure that the computation of the source tile is completed before its data is copied by the neighbor tile.

Phase VI Each process unpacks packages sent by other processes in Phase II and III in a first come first serve basis. As packages arrive, each process checks the *MPI_tag*s and discovers which ghost zone they are destined to. Based on the location of the destination tile, the process directly unpacks the package on the host or submits an unpacking kernel for the device. If the destination tile is on device and there is no GPUDirect support, the process submits an asynchronous transfer of the package to the device before the kernel submission.

Phase VII unpacks packages transferred between host and device. It submits an

unpacking kernel to the stream of the destination tile for packages sent from host to device. For a package sent in the reverse direction, it first synchronizes CPU with the *transfer event* from Phase IV, then unpacks the package on host.

Hereby dividing ghost cell update into phases provides concurrency by which the programming model is able to hide packing/unpacking and communication.

Chapter 6

PERFORMANCE

We evaluate the performance of our programming model on the *Heat* and *Cardiac* simulations[1]. The reason we chose these applications is to see how our programming model performs in memory-intensive (*Heat*) and computation-intensive (*Cardiac*) cases. *Heat* computes heat transfer equation for each point in 3D space by performing 7-point stencil in each timestep. *Cardiac* simulation simulates the propagation of electrical signals in the cardiac tissue in 2D space using the Aliev-Panfilov model [Aliev and Panfilov, 1996], which is a reaction-diffusion system. The model comprises both ODE and PDE parts where the ODE part describes the kinetics of reactions which are the cellular exchanges of various ions across the cell membrane during the electrical impulse. The PDE part describes the spatial diffusion of reactants.

Baseline: We report all speedups against the baseline which manages memory manually with CUDA, uses pinned memory for host allocations and generates GPU kernels with OpenACC annotations. In all experiments, the reported speedups are based on average of execution times of 4 runs. Execution times include computation time and the time required for data transfers between hosts and devices. *Heat* and *Cardiac* run for 100 and 1350 timesteps, respectively.

GPU Cluster: Experiments are conducted on Summitdev at the Oak Ridge National Laboratory. Summitdev employs 54 nodes each having 2 10-core POWER8 CPUs and 4 NVIDIA P100 GPUs. GPUs are connected via NVLink 1.0 and nodes are connected in a full fat-tree via 2 Mellanox EDR Infiniband adapters. We used `nvcc` version 9.2 and PGI version 18.7 for the rest.

---

[1]Code for different implementations of both simulations is available at parcorelab.ku.edu.tr.

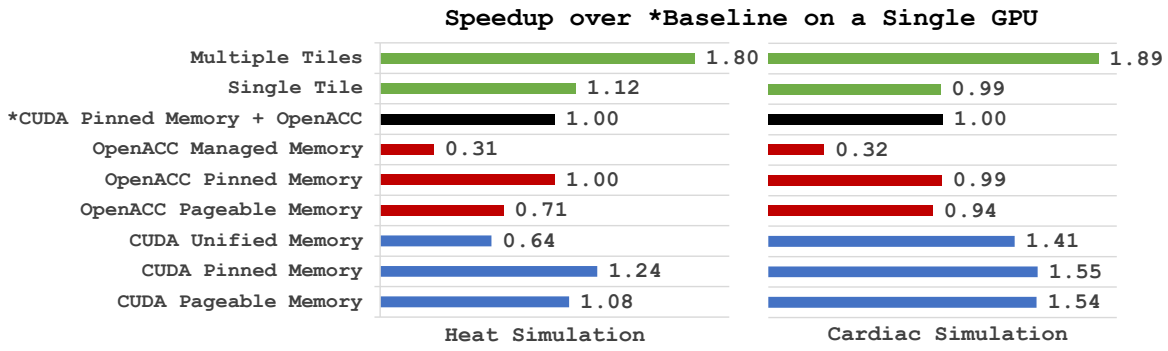

Figure 6.1: Performance comparison of various implementations of *Heat* and *Cardiac*. Single and multiple tile versions are based on our programming model.

Single GPU Performance: Fig.6.1 shows performance of different implementations on a single GPU for the *Heat* and *Cardiac* simulations. The grid size for both applications is chosen so that nearly all the GPU memory (96%) is consumed. The figure presents two results for which our programming model is used: *Single Tile* and *Multiple Tiles*. As the name suggests, in the *Single Tile* execution, data is not partitioned, instead it is executed as one. For the *Multiple Tiles* case, we experimented executions with different number of tiles and picked the best performing one, which is 128 tiles for *Heat* and 81 tiles for *Cardiac*.

Our single tile execution for *Heat* outperforms the baseline by 12% due to overlapping of memory transfers and computation. Using multiple tiles increases overlapping and results in 80% and 89% better performance compared to the baseline for *Heat* and *Cardiac*, respectively. In addition, for comparison, we show the results for different memory options of CUDA and OpenACC. The CUDA versions use hand-written GPU kernels whereas the OpenACC versions use auto-generated kernels by the PGI compiler. The pinned memory versions are clearly the winner for *Heat*. Since *Cardiac* is more compute-intensive, the performance benefit of pinned over pageable memory is negligible. The hand-written CUDA kernels with pinned memory are better than the OpenACC variants (baselines) for both applications. Our library leverages pinned memory for memory allocations due to their superior performance and utilizes

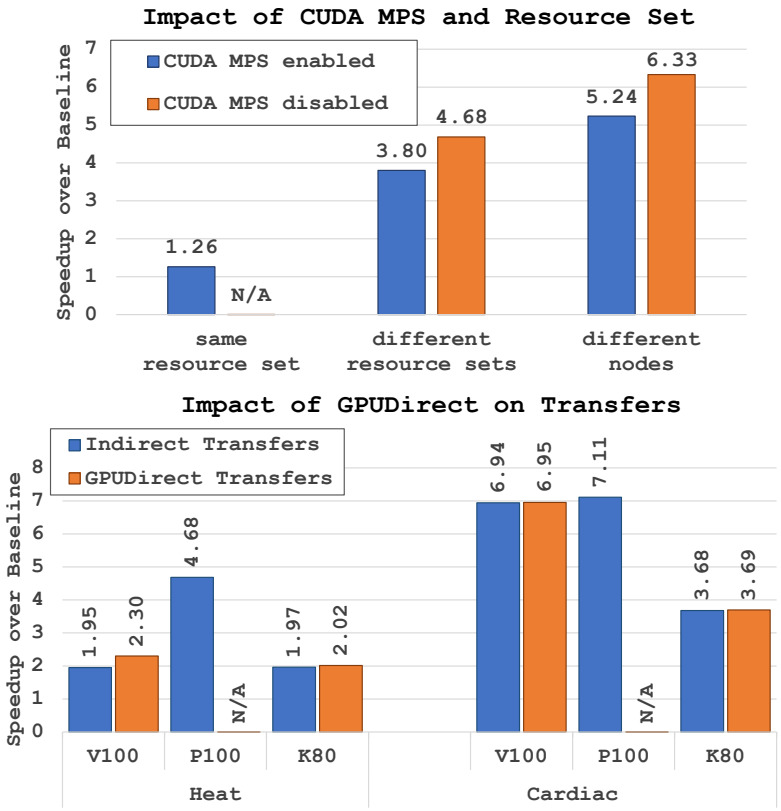

Figure 6.2: Performance impact of MPS, resource sets and GPUDirect on 4 GPUs

OpenACC directives for kernel code generation due to their convenience. The performance loss for the OpenACC kernels is compensated with the multi-tile execution whose performance surpasses the hand-written CUDA kernels in both applications.

Single Node Performance (4 GPUs): CUDA compute mode on Summitdev is configured as exclusive-process, which allows only one process per GPU. This mode also prevents having multiple processes per host if they use pinned host memory. In order to employ multiple processes in such cases, CUDA multi-process service (MPS) should be enabled. While submitting jobs, Summitdev provides a resource set configuration, which allows what resources on a node are visible to a job; each GPU can have its own resource set or share the resource set with the other GPUs in the node. Fig.6.2 (left) shows the performance impact of MPS and resource set configurations for 4 GPUs using *Heat*. One can disable MPS and use a separate resource set for

each GPU, which provides the best single node performance on Summitdev. As a comparison, we also provide the performance when four GPUs are selected from four different nodes.

Even though Summitdev supports GPUDirect technologies and has CUDA-aware MPI, a long-time issue about the GPU Inter-Process Communication in the system prevented us from conducting experiments with direct communication across devices and hosts. Instead, we used indirect transfers on Summitdev. However, we demonstrate the library's GPUDirect support with K80 and V100 workstations with 4 GPUs in Fig.6.2 (right). While *Cardiac* performs almost the same whether direct transfers are used or not, *Heat* shows a slight increase with direct transfers especially on V100 workstation which employs high-bandwidth NVLink 2.0 between GPUs.

GPU Cluster Performance (48 Nodes - 192 GPUs): We performed strong scaling studies with *Heat* and *Cardiac* and weak scaling study with *Heat* for the evaluation. On strong scaling, we kept the data size fixed and increased the number of GPUs. On weak scaling, we increased the data size proportional to the number of GPUs employed so that each GPU in each execution has the same amount of work.

We also analyzed the impact when we employ GPUs in the system differently on our strong scaling studies. In one scheme, we employed free GPUs on a host before employing the ones on different hosts. We refer to this scheme as *Compact*. On the other scheme, we employed free GPUs on the system as evenly as possible. For example, n GPUs will be employed from n different hosts even though a host has multiple GPUs. If there are n/2 hosts, then 2 GPUs will be employed from each host even though there are more than 2 GPUs on a host. We refer to this scheme as *Scatter*.

Figure 6.3 shows the strong scaling studies for *Heat* and *Cardiac* on multiple GPUs. On *Heat*, our programming model achieves up to 63x speedup with 128 GPUs, then starts to throttle because there is not enough computation left per GPU to hide the transfers and kernel-launch overheads dominate such computation. *Cardiac* being more compute-intensive creates the opportunity of better scaling as the number of

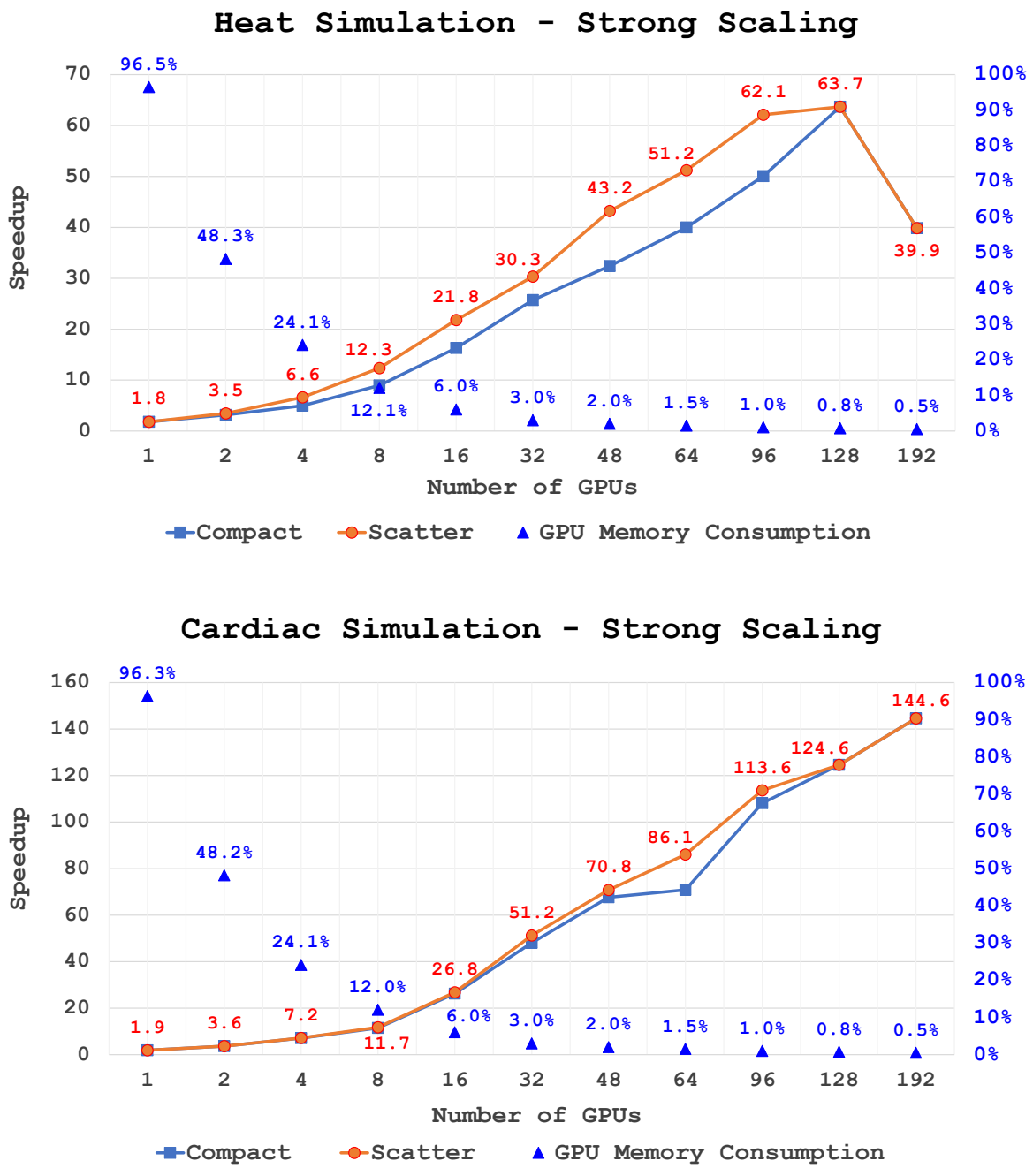

Figure 6.3: Strong scaling studies for Heat and Cardiac simulations.

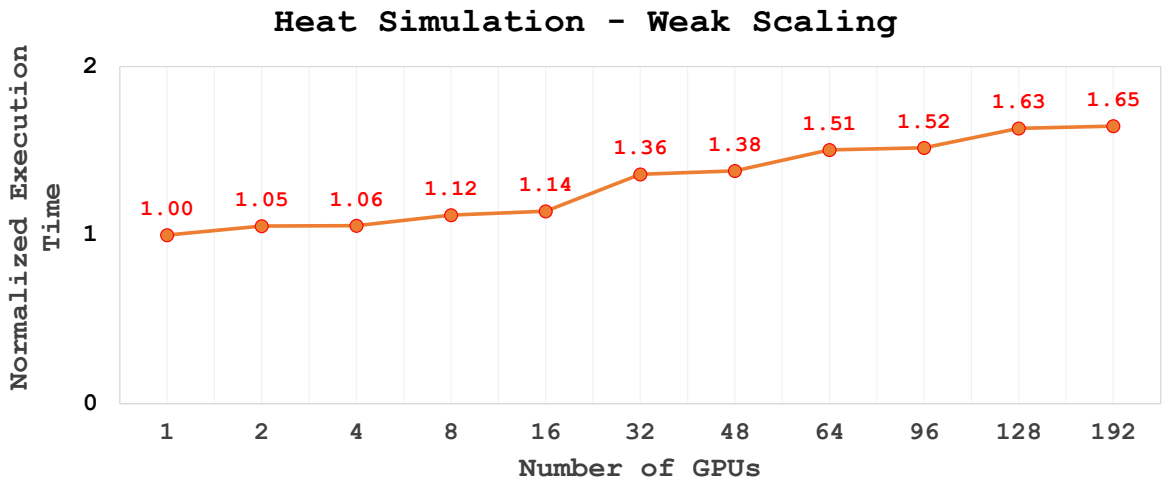

Figure 6.4: Weak scaling study for Heat Simulation

GPUs increases; nearly 144x speedup is achieved with 192 GPUs. Note that on 192 GPUs, both simulations only consume 0.5% of the GPU memory.

From Figure 6.3, we can also observe that *Scatter* performs better than *Compact* in *Heat* due to memory-intensive behavior of the application and high network bandwidth in the cluster. However, *Cardiac* shows similar performance in *Scatter* and *Compact* due to its compute-intensive behavior.

Finally, Figure 6.4 presents weak scaling with *Heat* which consumes 96.3% memory in each GPU. The reason for the increase in normalized execution times is that communication takes more time as more GPUs are employed. As the number of GPUs increases, overall data size as well as number of messages increases. More messages require more time in communication. In the meantime, since each GPU has the same amount of work in each execution, the time necessary for computation on a GPU stays the same. Therefore, there is more communication to overlap with the same amount of computation each time number of GPUs increases. Moreover, *Heat* is a memory-intensive application, so time for memory transfers already exceed time for computation. Hence, normalized execution time increases as the number of GPUs increases.

Chapter 7

CONCLUSION

In this thesis, we propose a high-level programming model for GPU clusters targeting structured grid problems. The model partitions data into tiles, distributes them to execution units, manages host and device memories, generates GPU kernels, and handles communication. It leverages non-blocking MPI calls for asynchronous communication and takes advantage of CUDA streams for overlapping device transfers and computation on GPUs. We evaluate the programming model on a heat simulation and a real-life cardiac modeling. The evaluation on two applications shows that tiling helps accelerate GPU executions by increasing the overlapping of transfers and computation. The evaluation also shows the scalability of the programming model on a GPU cluster.

BIBLIOGRAPHY

[top, 2019] (2019). Top500. `https://www.top500.org/`.

[Agullo et al., 2018] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. P. (2018). Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1.

[Aji et al., 2016] Aji, A. M., Panwar, L. S., Ji, F., Murthy, K., Chabbi, M., Balaji, P., Bisset, K. R., Dinan, J., Feng, W.-c., Mellor-Crummey, J., et al. (2016). Mpi-acc: Accelerator-aware mpi for scientific applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1401–1414.

[Aliev and Panfilov, 1996] Aliev, R. R. and Panfilov, A. V. (1996). A simple two-variable model of cardiac excitation. *Chaos, Solitons & Fractals*, 7(3):293–301.

[Augonnet et al., 2012] Augonnet, C., Aumage, O., Furmento, N., Namyst, R., and Thibault, S. (2012). Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer.

[Augonnet et al., 2009] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer.

[Bastem et al., 2017] Bastem, B., Unat, D., Zhang, W., Almgren, A., and Shalf, J. (2017). Overlapping data transfers with computation on gpu with tiles. In *46th International Conference on Parallel Processing*, pages 171–180. IEEE.

[Bell and Hoberock, 2012] Bell, N. and Hoberock, J. (2012). Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier.

[Ben-Nun et al., 2017] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. (2017). Groute: An asynchronous multi-gpu programming model for irregular computations. In *ACM SIGPLAN Notices*, volume 52, pages 235–248.

[Bueno et al., 2012] Bueno, J., Planas, J., Duran, A., Badia, R. M., Martorell, X., Ayguade, E., and Labarta, J. (2012). Productive programming of GPU clusters with OmpSs. In *IEEE 26th Intl. Symposium on Parallel & Distributed Processing*, pages 557–568.

[Cui et al., 2017] Cui, X., Scogland, T. R., de Supinski, B. R., and Feng, W.-c. (2017). Directive-based partitioning and pipelining for graphics processing units. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 575–584. IEEE.

[Dastgeer et al., 2011] Dastgeer, U., Enmyren, J., and Kessler, C. W. (2011). Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 25–32. ACM.

[Dastgeer and Kessler, 2016] Dastgeer, U. and Kessler, C. (2016). Smart containers and skeleton programming for gpu-based systems. *International journal of parallel programming*, 44(3):506–530.

[Edwards et al., 2014] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216.

[Enmyren and Kessler, 2010] Enmyren, J. and Kessler, C. W. (2010). Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM.

[Ernsting and Kuchen, 2012] Ernsting, S. and Kuchen, H. (2012). Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138.

[Gautier et al., 2013] Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. (2013). Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE.

[Grasso et al., 2014] Grasso, I., Pellegrini, S., Cosenza, B., and Fahringer, T. (2014). A uniform approach for programming distributed heterogeneous computing systems. *Parallel and Distributed Computing*, 74(12):3228–3239.

[Gregory and Miller, 2012] Gregory, K. and Miller, A. (2012). *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press.

[Gysi et al., 2016] Gysi, T., Bär, J., and Hoefler, T. (2016). dcuda: hardware supported overlap of computation and communication. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 609–620. IEEE.

[Hoshino et al., 2013] Hoshino, T., Maruyama, N., Matsuoka, S., and Takaki, R. (2013). Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143.

[Jablin et al., 2011] Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I. (2011). Automatic cpu-gpu communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151.

[Kegel et al., 2012] Kegel, P., Steuwer, M., and Gorlatch, S. (2012). dopencl: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 174–186.

[Kim et al., 2013] Kim, H. S., Unat, D., Baden, S. B., and Schulze, J. P. (2013). A new approach to interactive viewpoint selection for volume data sets. *Information Visualization*, 12(3-4):240–256.

[Kim et al., 2016] Kim, J., Lee, Y.-J., Park, J., and Lee, J. (2016). Translating openmp device constructs to opencl using unnecessary data transfer elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 51:1–51:12, Piscataway, NJ, USA. IEEE Press.

[Kim et al., 2012] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM.

[Komoda et al., 2013] Komoda, T., Miwa, S., Nakamura, H., and Maruyama, N. (2013). Integrating multi-gpu execution in an openacc compiler. In *2013 42nd International Conference on Parallel Processing*, pages 260–269.

[Lawlor, 2009] Lawlor, O. S. (2009). Message passing for gpgpu clusters: Cudampi. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. IEEE.

[Lee and Eigenmann, 2010] Lee, S. and Eigenmann, R. (2010). Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society.

[Majeed et al., 2013] Majeed, M., Dastgeer, U., and Kessler, C. (2013). Cluster-skepu: A multi-backend skeleton programming library for gpu clusters. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, page 468. Citeseer.

[Marques et al., 2013] Marques, R., Paulino, H., Alexandre, F., and Medeiros, P. D. (2013). Algorithmic skeleton framework for the orchestration of gpu computations. In *European Conference on Parallel Processing*, pages 874–885. Springer.

[Matsumura et al., 2018] Matsumura, K., Sato, M., Boku, T., Podobas, A., and Matsuoka, S. (2018). Macc: An openacc transpiler for automatic multi-gpu use. In *Asian Conference on Supercomputing Frontiers*, pages 109–127. Springer.

[Micikevicius, 2009] Micikevicius, P. (2009). 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM.

[Nakao et al., 2014] Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., Bokut, T., and Sato, M. (2014). Xcalableacc: Extension of xcalablemp pgas language using openacc for accelerator clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, pages 27–36. IEEE Press.

[NVDIA, 2014] NVDIA (2014). NVDIA NVLINK High-Speed Interconnect: Application Performance. Technical report.

[Pereira et al., 2015] Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). Pskel: A

stencil programming framework for cpu-gpu systems. *Concurr. Comput. : Pract. Exper.*, 27(17):4938–4953.

[Potluri et al., 2013] Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., and Panda, D. K. (2013). Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *42nd International Conference on Parallel Processing*, pages 80–89. IEEE.

[Rivera and Tseng, 2000] Rivera, G. and Tseng, C.-W. (2000). Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, SC '00. IEEE Computer Society.

[Sakharnykh, 2017] Sakharnykh, N. (2017). Maximizing unified memory performance in cuda. `https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda`.

[Schaetz and Uecker, 2012] Schaetz, S. and Uecker, M. (2012). A multi-gpu programming library for real-time applications. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 114–128. Springer.

[Song and Dongarra, 2012] Song, F. and Dongarra, J. (2012). A scalable framework for heterogeneous gpu-based clusters. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 91–100. ACM.

[Song and Dongarra, 2015] Song, F. and Dongarra, J. (2015). A scalable approach to solving dense linear algebra problems on hybrid cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27(14):3702–3723.

[Song et al., 2012] 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, pages 365–376. ACM.

[Steuwer and Gorlatch, 2013] Steuwer, M. and Gorlatch, S. (2013). Skelcl: Enhancing opencl for high-level programming of multi-gpu systems. In *International Conference on Parallel Computing Technologies*, pages 258–272. Springer.

[Steuwer et al., 2011] Steuwer, M., Kegel, P., and Gorlatch, S. (2011). Skelcl-a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE.

[Steuwer et al., 2012] Steuwer, M., Kegel, P., and Gorlatch, S. (2012). Towards high-level programming of multi-gpu systems using the skelcl library.

[Strengert et al., 2008] Strengert, M., Müller, C., Dachsbacher, C., and Ertl, T. (2008). Cudasa: Compute unified device and systems architecture. In *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '08, pages 49–56.

[Unat et al., 2011] Unat, D., Cai, X., and Baden, S. B. (2011). Mint: Realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 214–224.

[Unat et al., 2017] Unat, D., Dubey, A., Hoefler, T., Shalf, J., Abraham, M., Bianco, M., Chamberlain, B. L., Cledat, R., Edwards, H. C., Finkel, H., Fuerlinger, K., Hannig, F., Jeannot, E., Kamil, A., Keasler J., Kelly, P. H. J., Leung, V., Ltaief, H., Maruyama, N., Newburn, C. J., and Pericas, M. (2017). Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1.

[Unat et al., 2016] Unat, D., Nguyen, T., Zhang, W., Farooqi, M. N., Bastem, B., Michelogiannakis, G., Almgren, A., and Shalf, J. (2016). Tida: High-level programming abstractions for data locality management. In *International Conference on High Performance Computing*, pages 116–135. Springer.

[Wang et al., 2014] Wang, H., Potluri, S., Bureddy, D., Rosales, C., and Panda, D. K. (2014). Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605.

[Wen et al., 2014] Wen, Y., Wang, Z., and O'boyle, M. F. (2014). Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE.

[Wolfe, 2010] Wolfe, M. (2010). Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50.

[Wu et al., 2016] Wu, W., Bosilca, G., Vandevaart, R., Jeaugey, S., and Dongarra, J. (2016). Gpu-aware non-contiguous data movement in open mpi. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 231–242. ACM.

[Zhou et al., 2012] Zhou, J., Unat, D., Choi, D. J., Guest, C. C., and Cui, Y. (2012). Hands-on performance tuning of 3d finite difference earthquake simulation on gpu fermi chipset. *Procedia Computer Science*, 9:976 – 985.