Scalable Heterogeneous Supercomputing: Programming Methodologies and Automated Code Generation

by

Mohammed Sourouri

Thesis submitted for the degree of Philosophiae Doctor
Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
November 2015

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1703

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: John Grieg AS, Bergen.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Manycore processors such as Graphics Processing Units (GPUs) and Xeon Phis have remarkable computational capabilities and energy efficiency, making these units an attractive alternative to conventional CPUs for general-purpose computations. The distinct advantages of manycore processors have been quickly adopted to modern heterogeneous supercomputers, where each node is equipped with manycore processors in addition to CPUs.

This thesis takes aim at developing methodologies for efficient programming of GPU clusters, from a single compute node equipped with multiple GPUs that share the same PCIe bus, to large supercomputers involving thousands of GPUs connected by a high-speed network. The former configuration represents a peek into future node architecture of GPU clusters, where each compute node will be densely populated with GPUs. For this type of configuration, intra-node communication will play a more dominant role. We present programming techniques specifically designed to handle intra-node communication between multiple GPUs more effectively. For supercomputers involving multiple nodes, we have developed an automated code generator that delivers good weak scalability on thousands of GPUs.

While GPUs are improving rapidly, they are still not general-purpose, and depend on CPUs to act as their host. Consequently, GPU clusters often feature powerful multi-core CPUs in addition to GPUs. Despite the presence of CPUs, the focal point of many GPU applications has so far been on performing computations exclusively on the GPUs, keeping CPUs sidelined. However, as CPUs continue to advance, they have become too powerful to ignore. This gives rise to heterogeneous computing where CPUs and GPUs jointly take part in the computations.

The potentially achievable performance of heterogeneous computing codes can be very large, but requires careful attention to many programming details. We explore resource-efficient programming methodologies for heterogeneous computing where the CPU is an integral part of the computations. The experiments conducted demonstrate that by careful workload-partitioning and communication orchestration, our heterogeneous computing strategy outperforms a similar GPU-only approach on structured grid and unstructured grids.

Although our work demonstrates the benefit of heterogeneous computing, the painstaking programming effort required is holding back its wider adoption. We address this issue through the development and implementation of a programming model and source-to-source compiler called Panda, which automatically parallelizes serial 3D stencil codes originally written in C to heterogeneous CPU+GPU code for execution on GPU clusters. We have used two applications to assess the performance of our framework. Experimental

results show that the Panda-generated code is able to realize up to 90% of the performance of corresponding handwritten heterogeneous CPU+GPU implementations, while always outperforming the handwritten GPU-only implementations.

Compared to the more established GPU-only approach, the methodologies presented in this thesis contribute to harnessing the computational powers of GPU clusters in a more resource-efficient way that can substantially accelerate simulations. Moreover, by providing a user-friendly code generation tool, the tedious and error-prone process associated with programming GPU clusters is alleviated, so that computational scientists can concentrate on the science instead of code development.

Preface

This thesis has been submitted to the Faculty of Mathematics and Natural sciences at the University of Oslo in fulfillment of the requirements for the degree of Philosophiae Doctor (Ph.D.). It is the result of more than three years of research conducted at Simula Research Laboratory and University of California, San Diego. This work has been supervised by Professor Xing Cai, Professor Scott B. Baden and Dr. Johan Simon Seland. Furthermore, this work was supported by the FRINATEK program of the Research Council of Norway, through grant No. 214113/F20.

Acknowledgements

I would like to thank my advisors: Professor Xing Cai for always being there for me with his leadership, patience, guidance and invaluable help throughout my Ph.D., Professor Scott B. Baden for a great collaboration and for inspiring me to commit to the highest standards, and Dr. Simon Seland for valuable discussions and advice. It has been a privilege of a lifetime to work with all of you.

It has also been a pleasure to work at Simula Research Laboratory, as it provides an excellent work environment for conducting research. However, I would never in a million years end up at Simula had it not been for Dr. Tor Gillberg, whose encouragement and belief in me convinced me to pursue an academic career.

As a member of the High-Performance Computing group, I had the great opportunity to interact with fellow colleague Dr. Johannes Langguth with whom I had enjoyable and interesting collaboration. I also wish to thank Dr. Huayou Su at the National University of Defense Technology for his enthusiasm and input during his visit to Simula, also for making my stay in China pleesent.

During my enriching time at University of California, San Diego, I had the opportunity to work with many talented researchers. In particular, I wish to thank Dr. Nhat Tan Nguyen and Tatenda Chipeperekwa. I would also like to thank Natalie Lynn Larson for the countless hours they spent on proofreading my papers.

This Ph.D. has truly been a remarkable journey that has forged new collaborations across multiple countries. I wish to extend my gratitude to Filippo Spiga at the University of Cambridge for providing help and administrative support on the Wilkes cluster. Moreover, I would also like to thank Robert French, Adam Simpson, and Dr. Jack Wells at the Oak Ridge Leadership Computing Facility for their help on the Titan supercomputer. I would like to thank Ryan Davis at Princeton University and Assistant Professor Didem Unat at Koç University for interesting discussions and suggestions.

None of this work would be possible without the continuous patience, love and support of my dear friends and family, in particular, my parents, Sima and Reza, and my brother Saher.

Finally, I would like to express my sincere thankfulness to my beloved Frøydis, to whom I dedicate this thesis, for her love, patience and understanding.

Thank you.

Mohammed Sourouri, Oslo, September 2015

Contents

List of Papers

- **Paper I**

  Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads
  Mohammed Sourouri, Tor Gillberg, Scott B. Baden, Xing Cai
  Published in the proceedings of the *20th IEEE International Conference on Parallel and Distributed Systems, 2014, Pages 981-986*

- **Paper II**

  CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters
  Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B. Baden, Xing Cai
  Accepted for publication in the proceedings of the *18th IEEE International Conference on Computational Science and Engineering, 2015*

- **Paper III**

  Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes
  Johannes Langguth, Mohammed Sourouri, Glenn T. Lines, Scott B. Baden, Xing Cai
  Published in *IEEE Micro, Volume 35, Issue 4, Pages 6-15, 2015*

- **Paper IV**

  Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers
  Mohammed Sourouri, Scott B. Baden, Xing Cai
  Submitted for publication.

Introduction

Figure 1: The Titan supercomputer consists of 18688 compute nodes and was used in connection with Paper IV. The total compute capacity of the machine is 27 Petaflops. Image courtesy of Oak Ridge National Laboratory.

Mesh based simulations constitute one important motif of High Performance Computing (HPC) and are used in a wide range of scientific applications such as earthquake simulations [75], weather prediction [66], new materials discovery [57], and cardiac modeling [28]. For many computational scientists and engineers, computer simulations have become an irreplaceable tool, as they offer a fast, safe and an affordable way of conducting scientific experiments.

A common trait for many scientific applications is the need for more computing power to solve larger problems or to solve a problem faster. The continuous need for more computing power is by and large the main driving force behind the HPC market today. Historically, the most traditional way of providing computational scientists with more processing power has been by the release of newer processing units with higher computing capacity, that is, units capable of delivering more Floating-Point Operations Per Second (FLOPS). Occasionally, better and faster algorithms have also played a vital role.

The primary source of greater processing capacity in processing units has been attributed to *Moore's law* [24], which states that the number of transistors roughly doubles every two years. An important detail in the evolutionary history of processing units is the slower improvement rate of memory bandwidth. The disparity between a processing unit's processing power and memory bandwidth, better known as the *memory wall* [70], is a growing concern for many computational scientists. Because applications that are limited by the memory bandwidth are prevented from fully utilizing the system's compute capacity, this leads to a waste of resources. In general, applications that are limited by the system's memory bandwidth are categorized as *memory bound* applications, while applications that are bound by the system's processing power are called *compute-bound* applications [72]. Prime examples of the former are stencil computations, while dense matrix multiplication serves as a good example of the latter. This thesis concentrates solely on mesh-based applications that are normally memory bound.

When scientific applications solve large problems that are too big to fit in the memory of a single machine or demand more processing power than a single machine can deliver or both, the application is usually written for clusters or very large clusters called *supercomputers*. Both of these systems are composed of multiple computers, called *compute nodes*, connected by a high-speed interconnect to aggregate the computational power and memories of each individual compute node.

Supercomputers have predominantly been homogeneous systems powered by conventional CPUs. However, we have lately witnessed a shift towards heterogeneous clusters. The nodes of these systems are equipped with manycore processors such as Graphics Processing Units (GPUs) or Xeon Phis, in addition to CPUs. By looking at the latest Top 500 list [63] of supercomputers published in June 2015, it is evident that the interest in heterogeneous systems is growing. For example, on the November 2009 list, only two of the systems were heterogeneous, while on the latest list, 88 systems are classified as heterogeneous systems. Figure 1 shows a picture of the Titan supercomputer, which is a heterogeneous supercomputer where each compute node is equipped with an NVIDIA Tesla K20x GPU and one 16-core AMD Opteron 6274 CPU. Titan consists of more than 18600 compute nodes and is at the moment of writing the second fastest supercomputer in the world [63].

One possible explanation for the increasing interest in heterogeneous clusters might be that manycore processors, such as GPUs, deliver higher theoretical rates of FLOPS with a greater power-efficiency than traditional multi-core CPUs. High FLOPS performance is regarded as a key attribute in many fields of scientific computing, especially with respect to numerical applications. The focus of this thesis is on mesh-based simulations on heterogeneous supercomputers equipped with GPUs developed by NVIDIA.

Currently, the most powerful supercomputers are Petascale systems, which means that they are capable of performing more than one quadrillion (10^{15}) FLOPs. Moreover, the largest and the fastest supercomputers today are distributed memory systems interconnected by ultra-fast Infiniband technology [63]. In distributed memory systems, the different compute nodes are physically separated by the network so access to each other's memory requires explicit inter-node communication.

1 Distributed Memory Parallelization

The *de facto* programming model for scientific applications that targets distributed memory systems is message passing. The Message Passing Interface (MPI) [34] is a standardized library interface that developers are encouraged to follow. Examples of well-known, vendor neutral MPI libraries are MVAPICH2 [37] and OpenMPI [45].

Compared to sequential applications, writing MPI applications is regarded a challenge for many computational scientists, as it introduces the programmer to a parallel programming model called Single Program Multiple Data (SPMD), where the same application is executed on unique MPI processes, but with different data [11]. Other complicating details of MPI programming are domain decomposition, process layout, data sharing and explicit communication between different processes, which require calls to MPI routines.

HPC applications are judged by their ability to scale with the computing power provided by the cluster they are executed on. The two most common scaling methodologies in HPC are *strong* and *weak* scaling. In strong scaling studies the emphasis is generally on the solution time. Hence, the problem size is kept fixed, and more computational resources are added to obtain a faster solution time. Weak scaling studies are a type of experiments where the problem size is increased proportionally with the number of compute resources because the application can benefit by increasing the problem size and/or resolution. Under both of the scaling studies, communication becomes quickly a bottleneck as the number of compute nodes taking part in the computations grows. Usually, this is because the speed of communication is much slower than the speed of computations, but other reasons such as sequential communication patterns and network traffic congestions could also lead to poor scalability.

Hiding communication is considered by many as one of the most challenging aspects of developing MPI applications. The most widespread way of hiding latency overheads is by overlapping computations and communication [39]. Typically, this is done by adding a layer of *ghost cells* or *halo points* [25] around the problem domain so that the boundary points are separated from the interior points. Before computation of the interior points is started, the halo boundary points are computed first. During the computation of the interior points, halo boundary data are exchanged between neighboring domains using MPI routines such as `MPI_Irecv` and `MPI_Isend`. However, a much used strategy to improve performance is by using non-blocking MPI routines, such as `MPI_Irecv` and `MPI_Isend`, to build efficient pipelines. Other latency-hiding techniques include message aggregation [53], data compression [53] and virtualization [18], but are outside the scope of this thesis.

2 GPU Programming using CUDA

The increasing popularity of GPU-based computing poses a great challenge for computational scientists because programming GPUs is radically different than programming CPUs. This is primarily due to GPUs are complicated to program than CPUs is primarily due to the inherently different hardware architectures. For example, CPUs and GPUs differ in terms of how memory is handled. GPUs are designed to prioritize memory bandwidth over latency since latency can be hidden by parallel computation. CPUs, however, are designed around large cache coherent memories to increase (single threaded) application performance. GPUs are not general-purpose processing units and must be installed in a system with a CPU that can act as the *host*. NVIDIA GPUs are programmed using the CUDA [40] programming API.

CUDA exposes the developer to a parallel programming model based on SPMD [7]. Moreover, in CUDA lightweight processes called *threads* are organized into *thread blocks*, which are used to carry out computations in special functions called *kernels*. Every thread block launches the same kernel, but each thread within a thread block processes its designated data elements. However, before the GPU can process the data, the programmer must explicitly transfer data from the host CPU to the GPU across the slow PCIe bus because GPUs and CPUs do not share the same memory space. The CUDA API provides functions for realizing such data transfers. Furthermore, independent of the direction, each and every transfer is incurred with a performance penalty when data is moved across the high-latency PCIe bus.

The descriptions unveiled above highlight only *some* of the details that require a programmer's attention. Although a small number of programmers manage to overcome the obstacles of GPU programming, realizing high performance is possibly the most challenging part of GPU programming, which sometimes requires that existing algorithms are redesigned so that they better map to the GPU's architecture [10, 13, 14, 26, 32, 54, 71]. Unless the algorithms are reworked, CPUs could potentially outperform GPUs [8, 38].

GPUs are designed to exhibit parallelism by incorporating thousands of simplistic cores operating at low frequencies < 800 MHz. CPUs on the other hand incorporate typically 8-18 more powerful cores running at frequency close to 3 GHz. The inclusion of thousands of cores demands a memory bandwidth that is capable of handling the traffic generated by all of the cores. In order to cope with the increased memory traffic, GPUs utilize faster and more expensive GDDR5 memory, while CPUs use the much slower and less expensive DDR3/DDR4 technology. Because GDDR5 is more expensive, GPUs come with a very limited memory capacity. A typical compute node today is equipped with 128-256 GB of DDR3/DDR4 CPU memory, while the fastest GPUs are equipped with only 12 GB of memory. Hence, the limited memory capacity offered by GPUs becomes quickly a bottleneck when performing simulations involving large datasets/compute heavy kernels. However, by distributing a simulation across multiple GPUs, computational scientists are able to access more memory and computational capacity.

Multi-GPU programming follows the same principles as ordinary multi-CPU codes, that is, MPI is used for inter-node communication and ghost cells are used to hide communication overheads. One important difference is that in multi-GPU applications, computations are typically executed by CUDA kernels on the GPU and not on the CPU. The role of the CPU in multi-GPU applications is mostly to perform administrative tasks such as intra and inter-node communication. Since the computations are done on the GPU, the CPU is mostly idling and thus left underutilized.

Processing Unit	2× Intel Xeon E5-2680	Nvidia Tesla K20
Peak DP, GFLOPs	345.6	1170
Peak BW [GB/s]	102.4	208
STREAM [GB/s]	77	151

Table 1: An overview of key hardware specifications of the GPU-equipped compute nodes of Stampede.

3 Heterogeneous CPU+GPU Computing

One recent development in scientific computing has focused on combining CPUs and manycore processors for improved performance and energy efficiency [36]. The main purpose of a CPU+GPU implementation is to fully utilize the entire pool of processing units to solve a given problem as fast as possible. Table 1 displays the specifications for the GPU and the CPUs installed in some of the compute nodes of the Stampede [62] supercomputer. Judging by the performance numbers shown in Table 1, the GPU's theoretical peak FLOPS rate is approximately 3.4× higher than the two CPUs'. The realistic memory bandwidth obtained using the STREAM benchmark [33] is merely 1.94× higher. We concentrate on memory bound applications and therefore on the memory bandwidth numbers.

In short, the numbers from Table 1 tell us that the GPU is close to 2× faster than the two CPUs. So the workload division must reflect this performance difference. If the workload division is not appropriate, the application will most likely run into workload balancing issues that will degrade the performance because the fast GPU will continuously wait for the CPU. Thus, load balancing is an essential component of any heterogeneous CPU+GPU implementation.

There are multiple ways to load balance heterogeneous CPU-GPU implementations. Previous attempts at developing prediction models for heterogeneous CPU+GPU codes include [3, 6, 55, 65, 67] both *static* and *dynamic* load balancing have been proposed in the past.

Static load balancing means that the workload is partitioned before the computations. Typically, the entire or a small portion of the application is profiled first. Then, the acquired profiling data is used as a guiding measure to determine the workload division. There are significant advantages with static load balancing. Since the load balancing is performed before the actual computation takes place, the overhead associated with this strategy is virtually non-existent. The disadvantage of static load balancing is that it cannot be applied to computational problems in which the optimum workload division cannot be determined by profiling or where the optimum workload division varies during the execution.

Dynamic load balancing means that a special load balancer or scheduler automatically adjusts the workload division between the CPU and the GPU during the computation. This is especially useful for volatile workloads. Dynamic load balancers are usually domain specific, and can thus be difficult to generalize. The main disadvantage of dynamic load balancing is the relatively high overhead arising from the need to continuously reevaluate and adjust the workload division. The workload of the applications that we focus on do not change during execution, therefore we consider a static load balancing scheme as the most viable approach

In connection with this thesis, we have developed a simple static model for predicting the CPU's workload ratio for memory-bound applications. As opposed to other models [3, 6, 55, 65, 67] our model is not dependent on instrumenting, sampling or profiling of the target application on multiple nodes. The only dependency introduced in our model is the STREAM memory benchmark, which is open-source software that can be freely downloaded. It is only necessary to run the STREAM memory benchmark on a single compute node. We relate the workload ratio of a given processing unit to the its bandwidth and divide it by the aggregated memory bandwidth of all the processing units, as shown in (1).

$$\frac{CPU_{Bw}}{(GPU_{Bw}+CPU_{Bw})} \tag{1}$$

In (1), CPU_{Bw} and GPU_{Bw} represent the actual memory bandwidths obtained using the STREAM memory benchmark. As an illustrative example, we use results from Table 1 and insert these numbers into (1) to get an appropriate CPU workload division ratio, which is 33%. Additionally, if the peak theoretical results from Table 1 were used, the suggested CPU workload division ratio would be 32%, which could leave the CPUs slightly underutilized. However, achieving the peak theoretical memory bandwidth is a naive assumption, which is similar to what other researchers have observed [27, 72]. Thus, for more accurate predictions, we use the realistic memory bandwidth obtained using the STREAM memory benchmark.

4 A Framework for Heterogeneous CPU+GPU Computing

It is hypothesized that the collaboration of CPUs and manycore processors will play an even more important role in near-future, especially as future HPC will adopt fused CPU+manycore processor chips [1, 36]. A number of studies have demonstrated the benefit of concurrent CPU+GPU execution in for example stencil computations [28, 49, 57, 65, 73]. Despite the advantages of this approach, the number of tools that can reap the benefit of this strategy is rather limited.

Many scientists already find code development for a single GPU challenging, in particularly an entire cluster of CPUs+GPUs. This challenge is further complicated by the lack of a high-level unified programming model that enables developers to exploit different levels of parallelism. Despite the proliferation of programming models such as CUDA and OpenCL [23], developing clean code with high performance in a productive manner remains a big task. The lack of productivity is tightly coupled with the fact that current programming models require low-level knowledge of the underlying architecture. This type of knowledge is often difficult to grasp for computational scientists. Moreover, current programming models also expose the developer to far too many complex programming details.

Another complexity that is often neglected is portability. Developers face at least two challenges with respect to portability. The first challenge is tied to new GPU architectures. GPUs, like CPUs, are also updated at the rate of Moore's law, resulting in a new generation of architecture every two years. Traditionally, with every new generation of architecture, certain architecture-specific optimizations become obsolete. The second daunting challenge arises when developers try to port code between different types of clusters e.g. between two heterogeneous clusters using different types of GPUs or even worse, between a homogeneous and a heterogeneous cluster.

The difficulties described above have given rise to a variety of approaches such as compiler directives, libraries and Domain Specific Languages (DSLs). One developer friendly approach, advocated by some experts, is the use of compiler directives to guide the compiler in generating parallelized code. Thanks to the backing of numerous vendors, OpenACC [43] and OpenMP [44] have rapidly established themselves as the most popular solutions for directive-based code development. Despite delivering acceptable performance [31, 69] in a broad range of applications, neither of these two solutions is capable of producing code that can target an entirely homogeneous or an entirely heterogeneous cluster. As a result, developers must write code that deals with MPI.

DSLs constitute a compromise between language generality and performance. Depending on the framework, DSLs may support distributed memory systems. Since DSLs' knowledge are limited to a particular domain, they can leverage on this knowledge to deliver excellent performance. In contrary to a directive-based approach, DSLs [17, 74] require that both novice and expert programmers invest a considerable amount of time and effort in code development. A similar investment in code *redevelopment* is also required, if the programmer already has a parallel or a serial implementation.

Unlike DSLs, but like directives, libraries [56] offer the opportunity to stay within the boundaries of a general purpose programming language, but at the expense of performance. The common trait of libraries and DSLs is that they both require explicit changes to the code, which can easily cause programmers unnecessary difficulties. Portability is another issue that libraries often fail to address, as they are traditionally optimized for a specific architecture or cluster.

The different programming models presented so far highlight the lack of a developer-friendly model that is capable of realizing high-performance on modern heterogeneous clusters using a general purpose programming language. This is especially a challenge for domain scientists who wish to write code that can harness the computational provided by heterogeneous clusters.

5 Summary of papers

During the course of this PhD project, two papers were published in international peer-reviewed conferences [58, 59], one in an international peer-reviewed journal [28] and another one is submitted to an international peer-reviewed conference.

The focal point of this thesis has been a bottom-up approach to heterogeneous computing on GPU clusters. Paper I describes an effective communication scheme for 3D stencil computations on compute nodes equipped with multiple GPUs. Papers II and III detail advanced hybrid programming models for implementing scalable HPC applications on GPU clusters. The hybrid programming model outlined in Papers II and III consists of MPI, CUDA and OpenMP, making it possible to combine the computing power of CPUs and GPUs to achieve high performance on both structured and unstructured grids. Paper IV presents Panda, a novel programming model and its adherent compiler framework for automated generation of 3D stencil codes on structured grids incorporating the hybrid programming model detailed in Paper II and III. Details regarding the computational resources used in the thesis are presented in Appendix I.

5.1 Paper I: Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads

Future heterogeneous supercomputers such as ORNL Summit [42] and ANL Aurora [2] will feature compute nodes that are equipped with multiple GPUs or Xeon Phis. Installing multiple devices per node has advantages with respect to space and energy. We are already witnessing a shift towards this architectural change. For example, the world's current No. 1 supercomputer, Tianhe-2 (see [63]), is already equipped with three Xeon Phi coprocessors on each node.

The most widespread method for developing HPC code with multiple GPUs per node in mind is by spawning a unique MPI process for each GPU. The clear advantage of this approach is versatility, since the same code will work flawlessly regardless of the number of devices installed per node. However, in this approach, intra-node and inter-node communication are not differentiated and as a result unnecessary overhead is induced due to ineffective and redundant memory copies between the GPU, CPU and the MPI communication subsystem [21], and the creation of a process context for each GPU [48]. When an MPI process is controlling a GPU, it is encapsulated by the process' context, which means that e.g. two neighboring GPUs on the same node cannot exchange data directly unless message-passing or inter process communication is practiced. On the other hand, when one or more threads are controlling one or multiple GPUs, data can be more directly exchanged between the different GPUs using functions from the standard CUDA API.

This paper introduces an efficient intra-node communication scheme designed for computations on compute nodes that are equipped with multiple GPUs. In the presented scheme, the domain is decomposed, whilst one OpenMP thread is spawned to control each GPU, as opposed to one MPI process per GPU. The benefit of using threads is that the GPUs can effectively communicate using shared-memory and the ability to perform concurrent kernel launches. Since the GPUs stay within the same process context, the GPUs can benefit from fast intra-node GPUDirect v2 Peer-to-Peer [41], which is not possible if MPI is used.

Another optimization, called multi-streaming, is used to increase performance by placing the CUDA streams, which are responsible for sending computed halo boundaries and unpacking the halo boundaries, in separate OpenMP threads. In addition to the thread responsible for controlling the GPU, two additional threads are spawned per GPU, one for sending computed halo boundaries and one for receiving computed halo boundaries. The benefit of this strategy is that CUDA kernels responsible for unpacking halo boundary data can start immediately after the data from a neighboring device has been received. On the contrary, if only one thread was used to control multiple GPUs, the running thread could be blocked by for example another function, which would prevent the CUDA unpack kernels from being launched.

The performance of the proposed scheme is compared to a state-of-the-art MPI imple-

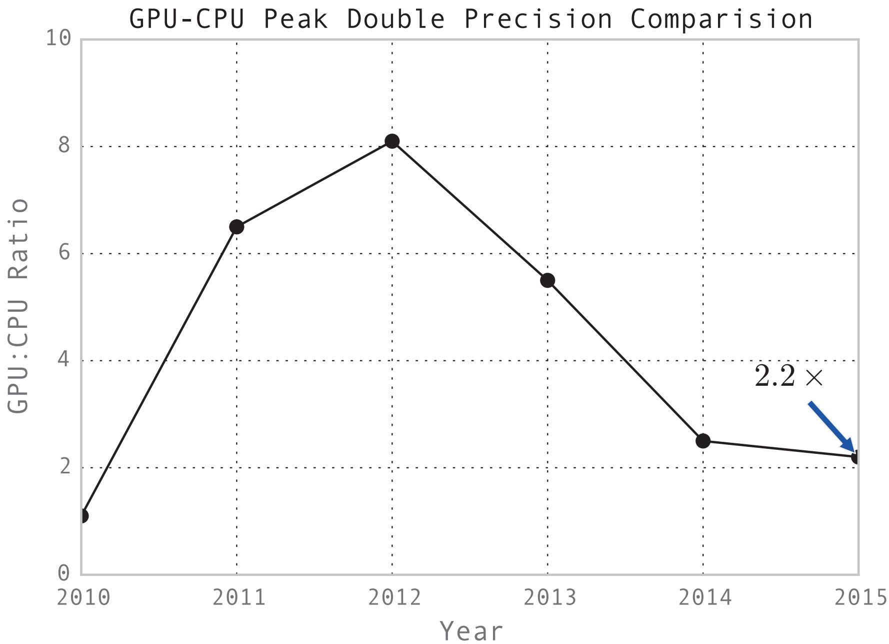


Figure 2: Since 2010, the difference in peak double precision floating point between GPUs and CPUs has become smaller, from 8× in 2010 to 2.2× in 2015.

mentation [5, 20, 35, 46, 47, 50, 56] where an MPI process is spawned per GPU and two CUDA streams are created. The first CUDA stream is used for the halo boundaries, while the second CUDA stream is used for computation of the interior points. Strong scaling experiments are conducted using a simple 7-point 3D Laplacian kernel. This particular compute kernel is chosen because it rapidly becomes communication bound. Our proposed scheme outperforms the MPI implementation and is up to 1.85× faster.

5.2 Paper II: CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters

High computational throughput and energy efficiency have placed GPUs at the heart of many clusters. GPUs are not general-purpose and depend on a CPU to operate, which is why GPU clusters are populated with CPUs. A recent surge of microarchitectural enhancements [15] such as the integration of more cores, advanced vector extensions, and fused multiply-add, has made it possible for CPUs to deliver an impressive amount of processing power, as Figure 2 displays. Furthermore, CPUs also provide fast and large last level caches, which can increase performance substantially if properly exploited, as a numerous studies [4, 10, 12, 51, 60] have shown. Additionally, modern CPUs nowadays provide a good memory bandwidth and a bytes-per-FLOP ratio that is within the vicinity of GPUs [30]. However, in many GPU applications, computations that once were performed on a CPU are now offloaded to the GPU, leaving the powerful CPUs underutilized.

In Paper II, two different CPU+GPU implementation techniques are developed and compared with a corresponding GPU-only implementation where the computations are performed exclusively on the GPU. The implementations developed employ a workload-partitioning strategy, which enables concurrent CPU+GPU execution to increase performance by exploiting the CPU's strength.

The first CPU+GPU implementation is a naive version that augments an existing state-of-the-art multi-GPU application [5, 35, 46] based on MPI and CUDA, by performing computations on the CPU using OpenMP. More specifically, the domain is decomposed, followed by a separation of halo boundary and interior points on each GPU. By processing the interior points and the boundary points separately in different CUDA streams, communication can be overlapped with computation.

Similar to state-of-the-art multi-GPU applications, asynchronous MPI routines are posted at the very beginning to build efficient communication pipelines, followed by CUDA kernel launches on the GPU. The computations on the CPU can only start once the different CUDA kernels have been launched. The naive implementation trades ease of use with only moderate speedups compared to the GPU-only version. The main drawback of this version stems from its inability to overlap CPU+GPU computations with inter-node MPI communication. Although asynchronous CUDA routines are used to ensure that intra-node communication is overlapped, inter-node communication is rarely overlapped since CUDA kernel launches and CUDA data transfers can not be launched because the CPU is busy computing. For example, the unpacking of halo boundary data can not start on the GPU until the CPU has completed the computations of the interior points.

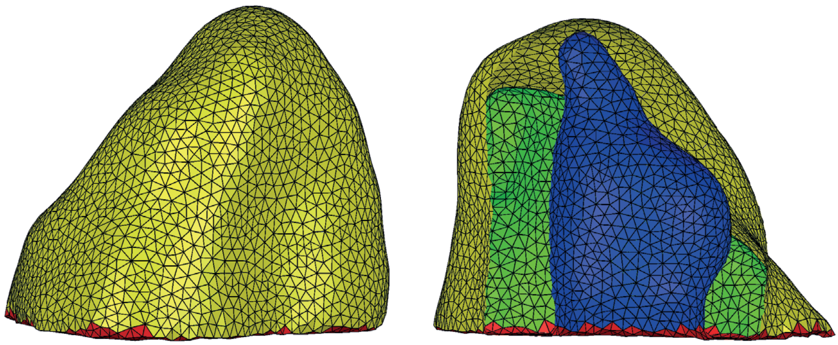
The naive implementation inability to overlap CPU+GPU computations with communication is addressed in an improved implementation called *nested*. OpenMP's nested parallelism capability is used to separate computations of the interior points and inter-node communication as distinct tasks. Moreover, two different thread groups are then created to concurrently process the different tasks. The first thread group is responsible for MPI communication, launching CUDA kernels and computations of halo boundaries on the CPU. Furthermore, the second group is dedicated to computing the interior points on the CPU.

One of the challenges that developers are facing when dealing with CPU+GPU codes is to find the optimal workload division ratio for the processing units, that is, the appropriate compute portion that gives the highest performance. Paper II presents a performance model for predicting the load balance between the CPU and the GPU in memory bound applications. With the aid of the STREAM memory benchmark [33] the realistic memory bandwidth of each processing unit is surveyed. The obtained memory bandwidth results are then used to determine the CPU workload ratio by dividing the CPU's memory bandwidth by the total aggregated bandwidth of the CPU and the GPU.

Strong and weak scaling experiments on the Stampede [62] and the Wilkes [64] clusters were conducted to assess the performance of the two implementations. Additionally, the results were compared to a corresponding handwritten GPU-only implementation. Both of the proposed implementation strategies outperformed the GPU-only implementation on the two clusters. In order to evaluate the accuracy of our performance model, a series of CPU workload sensitivity experiments were conducted by varying the CPU's workload ratio. The results from this experiment aligned well with the results predicted by our performance model.

Despite the accuracy of our performance model, projecting a perfect workload division ratio for CPU+GPU codes remains a complicated matter because the workload ratio can be very sensitive to various parameters such as problem size, performance difference between the processing units, etc. Another important finding of this experiment is that a CPU workload ratio that is too high will degrade the performance, but a too low CPU workload ratio is acceptable. In other words, in situations where the CPU workload can not be predicted accurately, it is better to lower the predicted inaccurate CPU workload.

5.3 Paper III: Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes



(a) Side view of the mesh (b) Inside view of the mesh

Figure 3: The mesh used in an unpublished version of Paper III, models a healthy male human heart acquired by MRI. Image courtesy of Johannes Langguth.

Paper III investigates heterogeneous CPU+GPU computations on an unstructured tetrahedral mesh by solving the diffusion equation using a cell-centered Finite Volume Method. The tetrahedral mesh representing a healthy male human heart served as a test instance, consisting of 115 million tetrahedrons. The mesh is illustrated in Figure 3. Additionally, some best practices for developing heterogeneous CPU+GPU codes that can be of help to other scientists are also presented.

We give detailed advice including how to handle multiple GPUs per node, how the different tasks on the CPU should be programmed and how to statically adjust the CPU workload ratio in conjunction with an increasing number of GPUs per node.

The methodologies and ideas presented in Paper III are similar to those presented in Paper II, but applied to another scientific domain. Moreover, if a compute node is equipped with multiple GPUs per node, the technique from Paper I, where one CPU thread is created for each GPU is used. A stiff challenge that many computational scientists face when working on unstructured meshes arises from indirect and irregular memory accesses. The irregular nature of the problem also poses a challenge with respect to workload-partitioning, load balancing and the inherently more complex communication pattern. Another complicating factor with the irregular accesses is that they dramatically reduce the computational intensity, which quickly limits the scalability because of a low compute-to-communication ratio.

Both a heterogeneous CPU+GPU and a corresponding GPU-only implementation were investigated on the Stampede [62] and Wilkes [64] clusters using up to 128 GPUs. A homogeneous CPU-only version was also implemented to better assess the CPU's performance, and thus its contribution. Additionally, to establish an upper bound of the achievable performance, the MPI calls were commented out so that both inter and intra-node communication were disabled. Before the experiments were conducted, the CPU workload ratio was computed statically by using the performance model presented in Paper II.

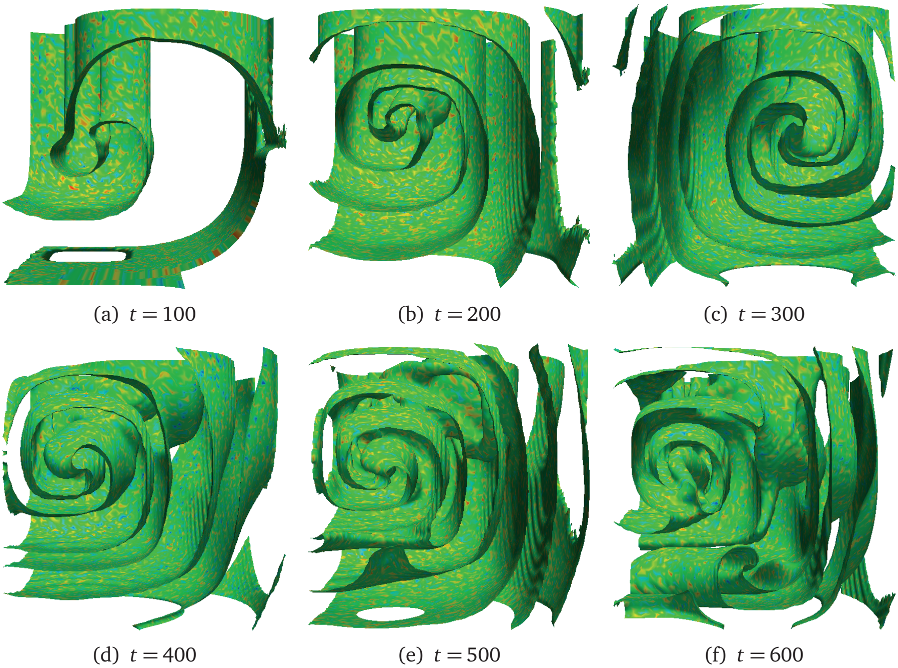
Strong scaling experiments on 128 nodes of Stampede showed that the heterogeneous CPU+GPU implementation consistently outperformed the GPU-only implementation, while realizing 95% of the upper bound. Similar results were also observed for 64 nodes on the Wilkes cluster when a single GPU was used per node. However, when both GPUs on each Wilkes node were used, the GPU-only implementation was faster than the heterogeneous CPU+GPU implementation. In the dual GPU configuration, one MPI process was spawned for each GPU. A consequence of this process layout was that the number of available CPU cores was divided equally between the two MPI processes, which significantly weakened the CPU's contribution.

Our investigations showed that when both GPUs on each Wilkes node were used, the access to fewer CPU cores and higher intra-node communication overhead became the performance limiter. Like in Paper II, the workload ratio predicted by the performance model was within the vicinity of the observed best results. Similarly, the experimental performance results presented in Paper III validate the viability of heterogeneous CPU+GPU computing even on unstructured grids.

5.4 Paper IV: Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers

A distinct drawback of the heterogeneous CPU+GPU computing technique demonstrated in Papers II and III is the tedious and often error-prone implementation process associated with it. Heterogeneous CPU+GPU codes require that the same computation and communication functions are replicated on both of the processing units. In other words, the same functions on the CPU must be implemented for the GPU and vice versa. Another complicating factor is the complex intra-node communication that takes places between the two processing units and the workload-partitioning strategy employed to divide the computational workload between the CPU and the GPU. This partitioning requires careful attention to many programming details.

Paper IV introduces a novel programming model and a domain-specific source-to-source compiler called Panda, which automatically parallelizes 3D stencil codes written in sequential C to a heterogeneous CPU+GPU form for execution on GPU clusters. The programming model provides a set of new compiler directives that serves as an interface,



(a) $t = 100$ (b) $t = 200$ (c) $t = 300$

(d) $t = 400$ (e) $t = 500$ (f) $t = 600$

Figure 4: The Cardiac Electrophysiology Simulator visualized at different time steps, t. The figures show how electrical signals propagate through the cardiac tissue and create spiral shaped patterns.

which lets the user annotate parts of a serial C code that deal with time consuming 3D stencil computations. The annotations implicitly capture parallelism that guides the compiler to perform appropriate transformations for auto-generation of CPU+GPU code. Moreover, by keeping the number of directives to a minimum, the Panda programming model offers not only a simple, but yet a highly user-friendly interface that promotes productivity. Furthermore, general-purpose compilers that do not implement Panda directives will simply ignore them and as a result, users only need to maintain a single code base for their sequential and their parallel code.

The Panda framework is implemented in C++ using the ROSE [29] compiler infrastructure and targets 3D stencils. Furthermore, the Panda framework employs a modular design where the different parts are compartmentalized. An overview of the Panda source-to-source compiler is illustrated in Figure 5. For brevity, several modules are excluded from Figure 5.

The `Directive Manager` module ensures that the input source file is correctly annotated. In addition, the role of the `Directive Manager` module is to extract information about the user specified compute arrays and their sizes. Based on the extracted information a `Partitioner` module will decompose the domain into smaller cuboids. Furthermore, a special `Stencil Analyzer` module will then analyze the annotated loop nests and search for nearest neighbor compute patterns. The result of the `Stencil Analyzer` module is then written into a `Stencil` object that is passed to the different generator modules that

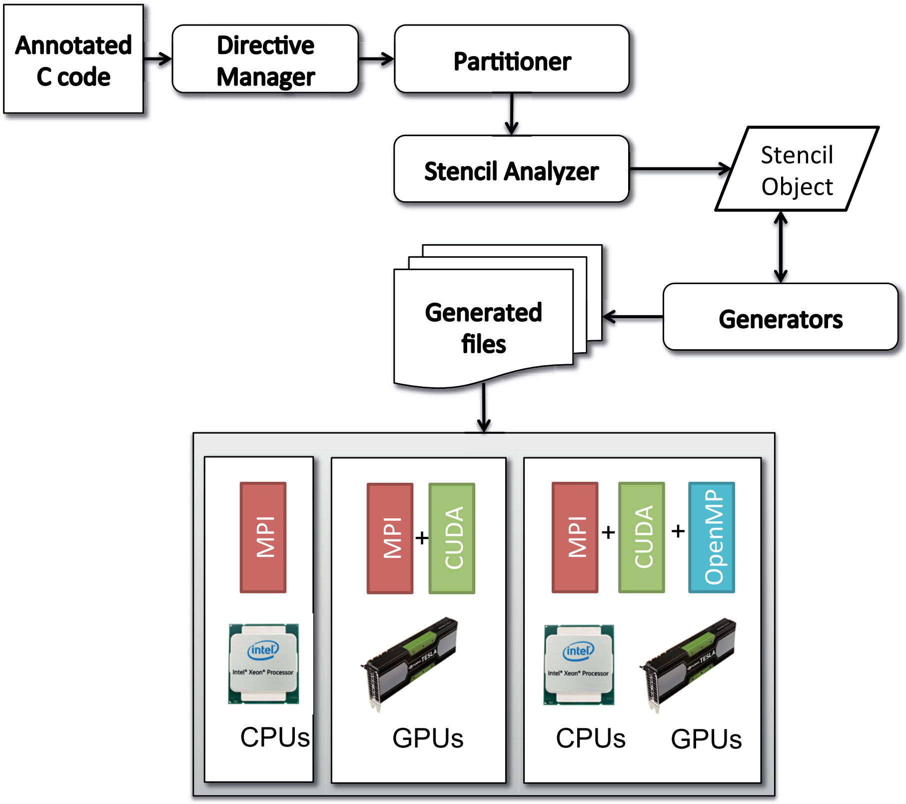


Figure 5: A high-level overview of the Panda framework.

are responsible for generating the actual source files.

Two applications were used to assess the performance of our compiler framework. As the first application we used the well-known 3D Laplacian stencil kernel from Papers I and II, while the second application was a real-world 3D Cardiac Physiology Simulator, as illustrated in Figure 4. The former application was used for its interesting computation-to-communication characteristics, while the latter application was used to demonstrate Panda's ability to tackle more realistic code, including computations on the physical boundaries. In addition to the Panda auto-generated codes, highly optimized handwritten versions of the two applications mentioned above were also developed for the purpose of evaluating the effectiveness of Panda codes. Depending on the cluster configuration, the Panda generated code was able to realize close to 90% of the performance of the handwritten heterogeneous CPU+GPU code for both applications. Although the Panda generated code is not as fast as the handwritten code, our results indicate that the Panda generated code is still faster than the aggressively optimized handwritten codes where the computations are performed exclusively on the GPU. We thus believe that our auto-generated CPU+GPU code provides a satisfactory alternative to implementations that ignore the computational power of CPU and exclusively offload computations to the GPU.

Panda's area of operation is currently limited to stencil computations on arrays that are logically represented as 3D. Moreover, it is also assumed that the annotated compute loops are parallelizable in such a way that the computed values can be updated concurrently. The narrowed domain of operation makes it possible to carry out effective optimizations at the expense of generality.

6 Discussion

The programming methodology presented in Paper I highlights intra-node communication bottlenecks that arise within a compute node equipped with multiple GPUs. San Diego Supercomputer Center's latest HPC system, Comet [52], is an example of a Petascale machine that adopts this node configuration. Future systems such as ORNL Summit [42] indicate that this trend will continue. On the basis of the experimental results presented in Paper I, we believe that intra-node communication and the complex interactions between multiple GPUs per node will play an important role in both current and future systems. Hence, extending the methodology presented in Paper I could be worth pursuing.

A clear limitation with the programming methodology presented in Paper I is that it is currently limited to GPUs that are located on the same node. In other words, inter-node communication between multiple nodes equipped with multiple GPUs is not taken into consideration. An obvious extension would therefore be to have an MPI process wrap the programming technique presented in Paper I so that inter-node communication is realized.

In the context of heterogeneous CPU+GPU computations, the use of multiple threads to control each GPU, such as in Paper I could possibly impede the CPU's performance. An important finding in Papers II and III was that the number of CPU threads spent on computation was crucial for achieving high performance. Hence, it could be worth pursuing the use of logical threads such as Intel Hyper-Threading [19] technology as an alternative to threads that are each mapped to a physical CPU core. We are also aware that later CUDA versions now support non-blocking CUDA events, which could potentially mimic some of the functionality of the methodology presented in Paper I. However, the use of CUDA events comes at the expense of increased code complexity and reduced code readability because CUDA events require calls to at least three additional CUDA functions. Moreover, the use of CUDA events does not automatically address issues such as concurrent kernel launches.

We acknowledge that the methodology presented in Paper I requires attention to many intriguing programming details, which can be difficult to grasp. Hence, in order to make the techniques presented in Paper I accessible to more scientists, it could be worth investigating different ways to abstract its complexity. One possible idea could be to provide a library or C++ template that automatically hides the more tedious programming details.

The technique presented in Papers II and III uses a fine-grained approach to utilizing both CPU and GPU for computations. The experiments conducted in both of the papers demonstrate that a conjoined CPU+GPU approach increases the overall computational speed. If the difference in the realistically achievable performance between CPUs and GPUs stays at the current level, combining CPUs and GPUs will remain an attractive alternative, and thus worth exploring.

Although the fine-grained threading approach of Papers II and III leads to a good overlap of computations and inter-node communication, intra-node communication is still an unresolved issue, as described in Paper III. Another weakness of the approach is when compute nodes are equipped with multiple GPUs. As both Paper II and III show, the introduction of additional GPUs widens the performance gap between the CPUs and the GPUs even further. To support two GPUs per node in Papers II and III, an MPI process was created for each GPU and the CPU cores were divided equally between the different MPI processes. The downside of this strategy is that the CPU's performance is substantially degraded, because each MPI process will have only access to half of the memory bandwidth and half of the shared cache [9].

A natural extension of the programming methodology presented in Papers II and III would be to implement the strategy presented in Paper I, as it minimizes the use of MPI processes and reduces intra-node communication overhead. One key feature of the methodology presented in Paper I was to use multiple CPU threads to control multiple GPUs. This would mean that a single MPI process would control multiple GPUs.

Although this thesis has focused solely on GPUs, the findings of Paper I-IV are also applicable to Xeon Phis. Currently, the difference in peak floating point capability between a CPU and a Xeon Phi is similar to the difference between a CPU and a GPU. Thus, a similar speedup should be expected in a heterogeneous CPU+Xeon Phi implementation too.

The first and foremost limitation of our work in Paper IV is its being domain-specific. Although such a limitation restricts the outreach of our work, we believe that our domain choice is large enough to carry out meaningful translations that would not be possible with a more generic approach.

Performance wise, one of the biggest performance limiters of the work presented in Paper IV is the lack of highly optimized CPU code when hybrid CPU+GPU code is generated. Fast CPU code is a necessity in order to narrow the computational performance gap between the CPU and the GPU. In the work presented so far, the CPU's straightforward compute loops are not modified, and as one of the conclusions of Paper II, the CPU's workload ratio must be lowered to catch up with the GPU. On the other hand, an aggressively handwritten CPU code that performs 3D cache blocking [38] in combination with optimal block sizes will mean that the CPUs will handle more computational work in a CPU+GPU implementation.

CPU optimization techniques have been an on-going research topic for many years and numerous works show that cache blocking [12, 30, 38, 51, 60, 68] is an effective strategy to improve the performance of stencil codes on the CPU. There are already many impressive frameworks [22, 61] and code-generators [4, 9, 16] that are capable of generating high-quality CPU code. Hence, instead of writing a new module in Panda, an alternative would be to review the possibility of adding support from an existing tool to generate optimized CPU code.

Another limitation of Panda is that it is unable to recognize and translate code on subscripted multi-dimensional arrays (e.g. `U[i][j][k]`). There are many reasons why

Panda only supports flat arrays. First of all, serial C/C++ performance programmers tend to prefer flattened arrays. Moreover, flat arrays map perfectly to how (linear) memory is allocated in CUDA, which creates a 1:1 mapping, and thus simplifies the process of code generation. The use of multi-dimensional arrays complicates the CUDA translation, as it requires that the compiler flattens the arrays or that special CUDA data structures such as `cudaPitchPtr` are used. Furthermore, in many scientific codes, data are often not laid out contiguously in memory. Flat arrays rely on special *incrementors* that automatically compute the array index. We support non-contiguous data layout by identifying and transforming the incrementor. Panda is capable of automatically identifying these incrementors, and thus supporting arrays that are laid non-contiguous in memory.

Despite some of its drawbacks, subscripted multi-dimensional arrays are more widespread in codes written by domain scientists, as it is syntactically closer to the actual mathematical notation. Adding support for subscripted multi-dimensional arrays should not pose a major problem, but requires an additional flag so that critical translator modules such as the *Stencil Analyzer* module are made aware of the new data layout. A benefit of supporting multi-dimensional arrays is that it will make the process of performing stencil analysis less complicated.

Panda is able to recognize and analyze stencils with a wider reach than 6 points to its neighbors. However, code generation of MPI communication and halo boundary computation of the corners that have more than 6 neighbors, has not been implemented yet. So far, our focus has been on laying the foundation for a framework capable of auto-generating MPI, MPI+CUDA, and MPI+CUDA+OpenMP code.

One and two-dimensional codes are not supported because we have only focused on 3D problems, which pose the biggest challenge with respect to both communication and computations. However, there are many real-world applications that are one or two-dimensional, such as spherulitic crystallization and channel crystallization, two common problems in the field of polymer physics. In order to support one and two-dimensional problems, the *Directive Manager* must communicate the dimension of the problem, which can be detected by looking at the number of parameters passed to the size clause, to the *Stencil Analyzer* module. Once the *Stencil Analyzer* module has been made aware of the problem's dimension, it can perform analysis within an appropriate space.

Another limitation in Panda is the lack of support for parallel I/O and checkpointing. Parallel I/O is an important component in HPC applications when it comes to tasks such as visualization or reading user-input. The limitation of handling parallel I/O can be addressed by introducing a directive specifically for dealing with I/O, and a clause that lets the user to specify the rank identifier of one or a range of ranks. Currently, Panda does not support application-level checkpointing primarily due to the lack of parallel I/O support. In other words, before checkpointing can be supported, the limitations of parallel I/O must be resolved first. Once parallel I/O is supported, special directives can be developed to let the user indicate areas of interest for checkpointing.

7 Conclusion

The main goal of this thesis is to contribute to the improvement and development of novel programming methodologies and tools for computational scientists. Paper I focused on the complex interactions and intra-node communication between multiple GPUs that are located on the same node. It is highly anticipated that both upcoming heterogeneous Petascale [2, 42] and future Exascale [1] systems will adopt a node architecture where each node is densely populated with multiple manycore processors such as GPUs. In these systems, reducing the cost associated with intra-node communication will become crucial. We expect that the programming techniques detailed in Paper I will make an important contribution towards reducing intra-node communication costs, which arise when multiple GPUs are installed on the same node.

The focus of Paper II was on achieving higher compute performance by taking advantage of the increasing computational power offered by modern CPUs by performing concurrent CPU+GPU computations. A big challenge in heterogeneous CPU+GPU computing is to find an appropriate CPU workload ratio that is neither too high nor too low. We have derived a simple performance model for predicting balanced CPU workloads with CPU+GPU computing in mind. Experimental results of a simple 7-point 3D stencil benchmark application on a structured grid showed that our heterogeneous CPU+GPU codes were able to outperform a corresponding GPU-only implementation by a large margin and that our performance model did a good job of predicting a balanced CPU workload ratio. The contributions of Paper II are detailed insights into an advanced programming technique where task parallelism was used to make efficient use of CPUs and GPUs and an ancillary performance model to predict an appropriate CPU workload ratio.

Motivated by the performance results in Paper II, a more challenging application for performing heterogeneous CPU+GPU computations was chosen. The chosen application solves the diffusion equation using the finite volume method on tetrahedral meshes. Experimental results using up to 128 GPUs on the Stampede supercomputer showed that the heterogeneous CPU+GPU version was on average 43% faster than the GPU-only version. Paper III confirmed our findings from Paper II that conjoining the computational capacity of the CPU with the GPU increases the application performance. Moreover, Paper III contributes in giving detailed insights into the development of heterogeneous CPU+GPU applications for unstructured meshes.

Paper IV makes contributions in the development of a novel automated code generator for performing heterogeneous CPU+GPU computations. The tool, called Panda, is currently at the proof-of-concept stage and has many limitations, but is nonetheless capable of parallelizing simple 7-point 3D stencil codes written in sequential C. In order to assess the performance of the auto-generated code, a series of experiments were conducted using the 3D stencil benchmark from Paper II and a real-world application in cardiac modeling. For evaluation purposes, aggressively optimized versions of the two applications were handwritten. The first version performed heterogeneous CPU+GPU computations, while the second version performed computations exclusively on the GPU. Experiments showed that the Panda-generated code was able to realize 90% of the performance of the handwritten versions. However, an important finding was that the Panda-code was always able to outperform the handwritten GPU-only code. The promising results are achieved primarily because Panda implements many generalized versions of the programming techniques unveiled in Papers II and III.

This thesis has thus shed some light on increasing the efficiency of memory-bound HPC applications by performing concurrent CPU+GPU computations and by providing computational scientists with a tool that can automatize the development of such applications.

Bibliography

1. Ang, J., R. Barrett, R. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. Hammond, K. Hemmert, S. Kelly, H. Le, V. Leung, D. Resnick, A. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. Wright (2014, November). Abstract machine models and proxy architectures for exascale computing.

2. Argonne Leadership Computing Facility (2015). Aurora. `http://aurora.alcf.anl.gov/`. [Online; accessed 1-June-2015].

3. Augonnet, C., S. Thibault, R. Namyst, and P.-A. Wacrenier (2011, February). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.: Pract. Exper. 23*(2), 187–198.

4. Bandishti, V., I. Pananilath, and U. Bondhugula (2012, November). Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 40:1–40:11.

5. Bernaschi, M., M. Bisson, and D. Rossetti (2013, February). Benchmarking of communication techniques for GPUs. *Journal of Parallel and Distributed Computing 73*, 250–255.

6. Bosilca, G., A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra (2012). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing 38*(1–2), 37–51.

7. Brodtkorb, A. R., C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli (2010, January). State-of-the-art in heterogeneous computing. *Sci. Program. 18*(1), 1–33.

8. Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron (2008). A performance study of general-purpose applications on graphics processors using {CUDA}. *Journal of Parallel and Distributed Computing 68*(10), 1370–1380.

9. Christen, M., O. Schenk, and B. Burkhart (2011, May). PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 676–687.

10. Datta, K., S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick (2009). Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review 51*(1), 129–159.

11. Diaz, J., C. Munoz-Caro, and A. Nino (2012, August). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Trans. Parallel Distrib. Syst. 23*(8), 1369–1386.

12. Frigo, M. and V. Strumpen (2005). Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pp. 361–366.

13. Gillberg, T., M. Sourouri, and X. Cai (2012, June). A new parallel 3d front propagation algorithm for fast simulation of geological folds. In *Proceedings of the International Conference on Computational Science, {ICCS} 2012*, pp. 947–955.

14. Grosser, T., A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege (2014, February). Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 66:66–66:75.

15. Hammarlund, P., A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton (2014). Haswell: The fourth-generation intel core processor. *IEEE Micro 34*(2), 6–20.

16. Henretty, T., R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan (2013, June). A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pp. 13–24.

17. Holewinski, J., L.-N. Pouchet, and P. Sadayappan (2012). High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 311–320.

18. Huang, C., G. Zheng, L. Kalé, and S. Kumar (2006, March). Performance evaluation of adaptive mpi. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 12–21.

19. Intel Corporation (2015). Intel Hyper-Threading Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`. [Online; accessed 27-September-2015].

20. Jacobsen, D., J. Thibault, and I. Senocak (2010, January). An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting*.

21. Ji, F., A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. Feng, and X. Ma (2012, May). Efficient intranode communication in GPU-accelerated systems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 1838–1847.

22. Kamil, S., C. Chan, L. Oliker, J. Shalf, and S. Williams (2010, April). An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12.

23. Khronos Group (2015). OpenCL - the open standard for parallel programming of heterogeneous systems. `https://khronos.org/opencl/`. [Online; accessed 25-May-2015].

24. Kim, N., T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan (2003, Dec). Leakage current: Moore's law meets static power. *Computer 36*(12), 68–75.

25. Kjolstad, F. B. and M. Snir (2010, March). Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pp. 4:1–4:9.

26. Krishnasamy, E., M. Sourouri, and X. Cai (2015, June). Multi-GPU implementations of parallel 3D sweeping algorithms with application to geological folding. In *Proceedings of the International Conference on Computational Science, {ICCS} 2015*, pp. 1494–1503.

27. Langguth, J. and X. Cai (2014, December). Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 191–199.

28. Langguth, J., M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai (2015, July). Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. *Micro, IEEE 35*(4), 6–15.

29. Lawrence Livermore National Laboratory (2015). ROSE compiler infrastructure. `http://rosecompiler.org`. [Online; accessed 04-June-2015].

30. Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey (2010). Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 451–460.

31. Levesque, J. M., R. Sankaran, and R. Grout (2012, November). Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 15:1–15:11.

32. Maruyama, N. and T. Aoki (2014, January). Optimizing stencil computations for nvidia kepler GPUs. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pp. 89–95.

33. McCalpin, J. D. (1995, December). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.

34. Message Passing Interface Forum (2012, September). *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS).

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

36. Mittal, S. and J. S. Vetter (2015). A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv. 47*(4).

37. MVAPICH (2015). MPI over InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). http://mvapich.cse.ohio-state.edu. [Online; accessed 25-September-2015].

38. Nguyen, a., N. Satish, J. Chhugani, C. K. C. Kim, and P. Dubey (2010, November). 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13.

39. Nguyen, T., P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden (2012, November). Bamboo: translating MPI applications to a latency-tolerant, data-driven form. In *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 39:1–39:11.

40. NVIDIA (2015a). CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Online; accessed 25-May-2015].

41. NVIDIA (2015b). NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. [Online; accessed 04-March-2015].

42. Oak Ridge Leadership Computing Facility (2015). Summit. https://olcf.ornl.gov/summit/. [Online; accessed 29-May-2015].

43. OpenACC - Directives for Accelerators (2015). The OpenACC Application Program Interface. http://openacc-standard.org. [Online; accessed 23-May-2015].

44. OpenMP Architecture Review Board (2015). OpenMP Application Program Interface. http://openmp.org. [Online; accessed 23-May-2015].

45. OpenMPI (2015). Open Source High Performance Computing. http://http://www.open-mpi.org. [Online; accessed 25-September-2015].

46. Phillips, E. H. and M. Fatica (2010, April). Implementing the Himeno benchmark with CUDA on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–10.

47. Playne, D. P. and K. A. Hawick (2011, July). Asynchronous communication for finite-difference simulations on GPU clusters using CUDA and MPI. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 169–174.

48. Potluri, S., H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda (2012, May). Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 1848–1857.

49. Ragan-Kelley, J., C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 519–530.

50. Rietmann, M., P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini (2012, November). Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 38:1–38:11.

51. Rivera, G. and C.-W. Tseng (2000). Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*.

52. San Diego Supercomputer Center - HPC Systems (2015). `http://www.sdsc.edu/services/hpc/hpc_systems.html#comet`. [Online; accessed 25-September-2015].

53. Satish, N., C. Kim, J. Chhugani, and P. Dubey (2012, November). Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 14:1–14:11.

54. Schäfer, A. and D. Fey (2011, June). High performance stencil code algorithms for GPGPUs. In *Proceedings of 2011 International Conference on Computational Sciences (ICCS)*, Volume 4, pp. 2027–2036.

55. Scogland, T., B. Rountree, W. chun Feng, and B. de Supinski (2012, May). Heterogeneous task scheduling for accelerated OpenMP. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 144–155.

56. Shimokawabe, T., T. Aoki, and N. Onodera (2014, January). A high-productivity framework for multi-GPU computation of mesh-based applications. presented at the International Workshop on High-Performance Stencil Computations, Vienna, Austria.

57. Shimokawabe, T., T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka (2011, November). Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 3:1–3:11.

58. Sourouri, M., T. Gillberg, S. Baden, and X. Cai (2014, December). Effective multi-GPU communication using multiple CUDA streams and threads. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 981–986.

59. Sourouri, M., J. Langguth, F. Spiga, S. B. Baden, and X. Cai (2015, October). CPU+GPU programming of stencil computations for resource-efficient use of GPU clusters. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*.

60. Strzodka, R., M. Shaheen, D. Pajak, and H.-P. Seidel (2010, June). Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 49–59.

61. Tang, Y., R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson (2011, June). The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 117–128.

62. Texas Advanced Computing Center Stampede (2015). `https://www.tacc.utexas.edu/stampede/`. [Online; accessed 12-February-2015].

63. Top500.org (2015). June 2015 | TOP500 Supercomputer Sites. `http://top500.org/lists/2015/06/`. [Online; accessed 04-Sept-2015].

64. University of Cambridge - HPC Service - The Wilkes Cluster (2015). `http://www.hpc.cam.ac.uk/services/wilkes.html`. [Online; accessed 12-February-2015].

65. Venkatasubramanian, S. and R. W. Vuduc (2009, June). Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd International Conference on Supercomputing*, pp. 244–255.

66. Wahib, M. and N. Maruyama (2013, September). Highly optimized full GPU-acceleration of non-hydrostatic weather model SCALE-LES. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pp. 1–8.

67. Wang, Z., L. Zheng, Q. Chen, and M. Guo (2014, February). CPU+GPU scheduling with asymptotic profiling. *Parallel Computing 40*(2), 107–115.

68. Wellein, G., G. Hager, T. Zeiser, M. Wittmann, and H. Fehske (2009, July). Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, pp. 579–586.

69. Wienke, S., P. Springer, C. Terboven, and D. an Mey (2012, August). OpenACC — first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing - 18th International Conference*, Volume 7484, pp. 859–870.

70. Wilkes, M. V. (2001, March). The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News 29*(1), 2–7.

71. Williams, S., D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker (2012, November). Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 96:1–96:11.

72. Williams, S., A. Waterman, and D. Patterson (2009, April). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*(4), 65–76.

73. Yang, C., W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng (2013, February). A peta-scalable CPU-GPU algorithm for global atmospheric simulations. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12.

74. Zhang, Y. and F. Mueller (2012). Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 155–164.

75. Zhou, J., Y. Cui, E. Poyraz, D. J. Choi, and C. C. Guest (2013, June). Multi-GPU implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers. In *Proceedings of the International Conference on Computational Science*, pp. 1255–1264.

Paper I:
Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads

Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads

Mohammed Sourouri[1,2], Tor Gillberg[1], Scott B. Baden[3], Xing Cai[1,2]

[1] Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] University of California, San Diego,
La Jolla, CA 92093 USA

Manycore processors such as Graphics Processing Units (GPUs) and Xeon Phis have remarkable computational capabilities and energy efficiency, making these units an attractive alternative to conventional CPUs for general-purpose computations. The distinct advantages of manycore processors have been quickly adopted to modern heterogeneous supercomputers, where each node is equipped with manycore processors in addition to CPUs.

This thesis takes aim at developing methodologies for efficient programming of GPU clusters, from a single compute node equipped with multiple GPUs that share the same PCIe bus, to large supercomputers involving thousands of GPUs connected by a high-speed network. The former configuration represents a peek into future node architecture of GPU clusters, where each compute node will be densely populated with GPUs. For this type of configuration, intra-node communication will play a more dominant role. We present programming techniques specifically designed to handle intra-node communication between multiple GPUs more effectively. For supercomputers involving multiple nodes, we have developed an automated code generator that delivers good weak scalability on thousands of GPUs.

While GPUs are improving rapidly, they are still not general-purpose, and depend on CPUs to act as their host. Consequently, GPU clusters often feature powerful multi-core CPUs in addition to GPUs. Despite the presence of CPUs, the focal point of many GPU applications has so far been on performing computations exclusively on the GPUs, keeping CPUs sidelined. However, as CPUs continue to advance, they have become too powerful to ignore. This gives rise to heterogeneous computing where CPUs and GPUs jointly take part in the computations.

The potentially achievable performance of heterogeneous computing codes can be very large, but requires careful attention to many programming details. We explore resource-efficient programming methodologies for heterogeneous computing where the CPU is an integral part of the computations. The experiments conducted demonstrate that by careful workload-partitioning and communication orchestration, our heterogeneous computing strategy outperforms a similar GPU-only approach on structured grid and unstructured grids.

Although our work demonstrates the benefit of heterogeneous computing, the painstaking programming effort required is holding back its wider adoption. We address this issue through the development and implementation of a programming model and source-to-source compiler called Panda, which automatically parallelizes serial 3D stencil codes originally written in C to heterogeneous CPU+GPU code for execution on GPU clusters. We have used two applications to assess the performance of our framework. Experimental results show that the Panda-generated code is able to realize up to 90% of the performance of corresponding handwritten heterogeneous CPU+GPU implementations, while always outperforming the handwritten GPU-only implementations.

Compared to the more established GPU-only approach, the methodologies presented in this thesis contribute to harnessing the computational powers of GPU clusters in a more resource-efficient way that can substantially accelerate simulations. Moreover, by providing a user-friendly code generation tool, the tedious and error-prone process associated with programming GPU clusters is alleviated, so that computational scientists can concentrate on the science instead of code development.

1 Introduction

Heterogeneous systems have lately emerged in the supercomputing landscape. Such systems are made up of compute nodes that contain, in addition to CPUs, non-CPU devices such as Graphics Processing Units (GPUs) or Many Integrated Core (MIC) co-processors. An expected feature of future heterogeneous systems is that each compute node will have more than one accelerating device, adding a new level of hardware parallelism. Some of the current heterogeneous supercomputers have already adopted multiple devices per compute node. The most prominent example is the world's current No. 1 supercomputer, Tianhe-2 (see [13]), where each node is equipped with three MIC co-processors. Having multiple devices per node has its advantages with respect to space, energy and thereby the total cost of computing power.

When multiple accelerating devices are used per node in a cluster, data exchanges between the devices are of two types: inter-node and intra-node. MPI [4] is the natural choice for the first type. However, when it comes to intra-node data exchanges, MPI might not be the best solution. First, most MPI implementations do not have a hierarchical layout, meaning that intra-node communication is inefficiently treated as inter-node communication. Second, one MPI process per device will increase the overall memory footprint, in comparison with using one MPI process per node. Third, using multiple MPI processes per device requires the creation of additional process contexts, thus additional overhead on top of the enlarged memory footprint.

In this work, we explore the intra-node communication between multiple GPUs that share the same PCIe bus. To improve the state-of-the-art communication performance, we make use of multiple CUDA streams together with multiple OpenMP threads.

The primary contributions of this paper are as follows:

- We propose an efficient intra-node communication scheme, which lets multiple OpenMP threads control each GPU to improve the overlap between computation and communication. Moreover, for each pair of neighboring GPUs, four CUDA streams are used to enable completely simultaneous send and receive of data.

- We quantify the performance advantage of our new intra-node communication scheme by using a representative 3D stencil example, for which inter-GPU data exchanges have a dominating impact on performance.

2 Background

Apart from aggregating the computation capacity, using multiple GPUs is also motivated from a memory perspective. Compared with the host memory, a GPU's device memory is considerably smaller. For example, 32-64 GB is a typical size for the system memory, whereas a single GPU has between 4 and 12 GB device memory, thus becoming a limiting factor. One way of overcoming this barrier is to use multiple GPUs, by either interconnecting multiple nodes equipped with a single GPU or by using a node with multiple GPUs. The latter configuration is the focus of this paper.

The current generation of GPUs targeted at the HPC market does not share the same memory space with its CPU host. A programmer thus has to explicitly transfer data between the device and host. Although recent CUDA versions can abstract the data transfer calls, the physical data motion still takes place.

Using multiple GPUs per node adds to the complexity. Independent of the direction, data transfers incur a performance penalty when moving across the high-latency PCIe bus that connects a GPU with its host. Therefore, one of the main objectives of our new communication scheme is to better hide the costs of data transfers.

Due to the inefficiencies connected with MPI in the context of intra-node communication, recent research such as [11] has therefore focused on utilizing a single MPI process per node while adopting multiple OpenMP threads per node. For example, in [11], a single thread is spawned per device. Despite these efforts, the underlying methodologies are essentially the same and imperfect in efficiency. Hence, we believe that a new approach is needed.

3 State of the Art

This section describes how boundary data exchanges (communication) and computation are handled in the current state-of-the-art intra-node communication scheme, which uses an asynchronous solution as exemplified in [1, 2, 6, 8–11]. The state-of-the-art scheme uses one MPI process or OpenMP thread to control each device, and two CUDA streams per device to overlap communication with computation.

As shown in Figure 1, a *subdomain* is the responsible computation area of a GPU. The data values that are needed by the neighbors constitute the so-called *boundary region*, whereas the data values that are to be provided by the neighbors constitute the so-called *ghost region*.

Between each pair of neighboring GPUs, the *data exchange* process consists of first copying data from the "outgoing" buffer of a GPU to the host, and then from the host into the "incoming" buffer of a neighboring GPU. Alternatively, P2P [5] can be used to directly

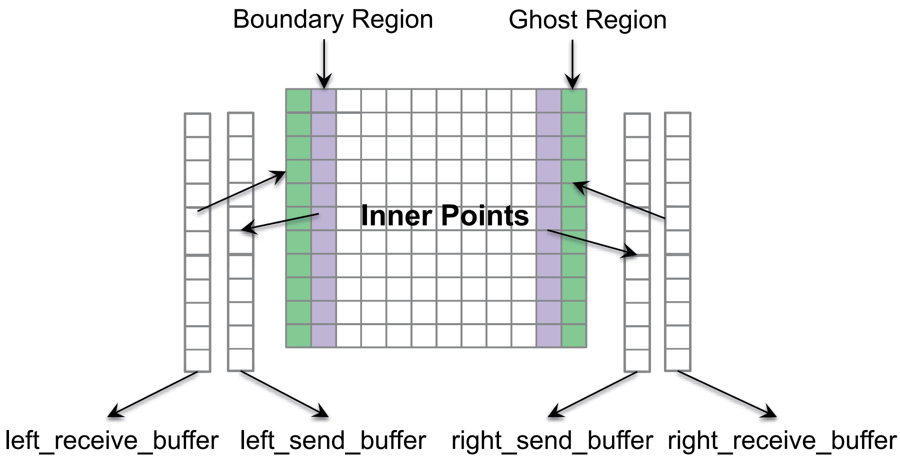


Figure 1: A simple two-neighbor example. The send buffers correspond to the boundary region (purple), and the receive buffers match the ghost region (green).

transfer the content of each outgoing buffer to a matching incoming buffer on a receiving GPU.

Computation is launched first in the boundary region, followed by data exchange of the boundaries. Concurrently with the data exchange, computation of the inner points is performed.

There are two variations of data exchanges in the state-of-the-art scheme. They are presented in [5] as the *left-right* and *pairwise* approaches. In both approaches, data exchange is done in two phases. During the first phase of the left-right approach, data is sent to the right neighbor and received from the left neighbor. The direction of data movement is reversed during the second phase. In the pairwise approach, the first phase consists of exchanging border data between some pairs of GPUs. In the second phase, data is exchanged between the remaining neighboring pairs. Another difference between these two approaches is that communication is *uni-directional* in the left-right approach, while it is *bi-directional* in the pairwise approach.

CUDA streams can be used to overlap communication and computation on Nvidia GPUs. Streams are sequence of operations that are executed in the order they are issued. The operations are queued. A queue manager picks an operation from the front of the queue, before feeding it to the device. The overall idea of streams is to increase concurrency by executing multiple operations simultaneously.

The state-of-the-art communication scheme relies on using two CUDA streams per GPU to overlap communication and computation. As Figure 2 reveals, the first stream is dedicated to computing the boundary points and exchanging data, while the second stream is dedicated to computing the inner points. In the Send Boundary Data block of Figure 2, data from the GPU memory is copied to a data buffer on the host, followed by a CUDA stream or device synchronization. Synchronization is necessary to ensure that the data transfer from the GPU to the host are indeed completed, before data can be copied

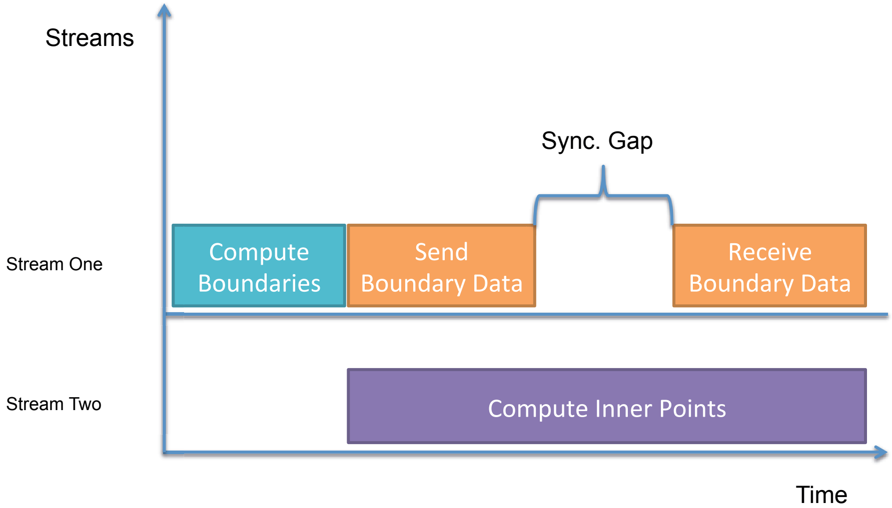


Figure 2: Use of two CUDA streams to overlap communication and computation.

from the host to the receiving buffers on the GPUs. If MPI is used for intra-node data exchange, an additional layer of process communication and synchronization is added. As a result, the synchronization gap depicted in Figure 2 is widened further.

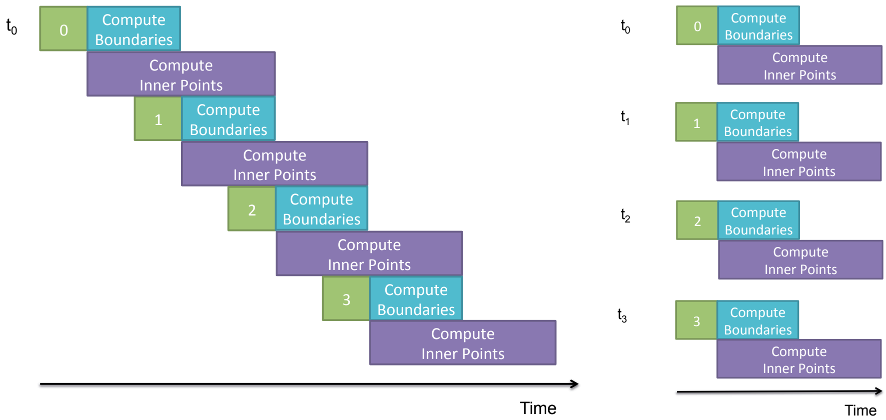
4 A New Communication Scheme

We propose a new intra-node communication scheme that targets multiple GPUs sharing the same PCIe bus. In this section, we describe the two fundamental components of our scheme, namely multiple OpenMP threads per GPU and multiple CUDA streams per pair of neighboring GPUs, and how these two components can be combined to outperform the state-of-the-art communication scheme.

4.1 Multi-Threading

Our context of study is intra-node communication between multiple GPUs that share the same PCIe bus. When a thread-based programming model is used, a single or multiple threads can be used to control the GPUs. Although the single-threaded approach is easier to implement, it has distinct disadvantages, because the application execution is serialized, including the kernel launches. Figure 3(a) depicts the situation if all the actions of the host are executed serially. That is, the kernels on GPU 0 is launched first, then on GPU 1, and so on. As the figure shows, the overhead of using a single thread is high. For this reason, we choose not to use a single thread to control multiple GPUs in our scheme. Instead, we choose to use multiple OpenMP [7] threads.

Similar to the state-of-the-art scheme, we let one OpenMP thread control each GPU as the *main thread*. The benefit of the one-thread-per-GPU approach is that the different kernels



(a) Single-threaded execution sequence

(b) Multi-threaded execution sequence

Figure 3: Using a single host thread or multiple host threads to launch kernels on multiple GPUs, where t_i denotes the thread number. A green bar represents the device number, a purple bar corresponds to computing the boundary region, and a blue bar corresponds to computing the inner points.

can be launched in parallel, and thus, eliminating kernel launch overheads. Figure 3(b) illustrates this in greater detail. However, unlike the state-of-the-art scheme, we take one step further and make use of three OpenMP threads per GPU for improving the communication performance. The boundaries of a subdomain are independent of each other. If decoupled, each boundary can be processed in parallel. To decouple we split each boundary of the subdomain into two parts: a send and a receive part. Next, we spawn one *assistant* OpenMP thread to handle all the outgoing buffers, and another thread to handle all the incoming buffers, meaning that the total number of OpenMP threads per GPU equals three times the number of GPUs.

4.2 Multi-Streaming

We have just shown how a group of threads can increase scheme concurrency, and realize bi-directional communication. Next, we show how multiple threads in tandem with multiple streams can be used to reduce additional overheads.

In the state-of-the-art scheme, two CUDA streams are created per GPU to overlap communication with computation. The first stream computes the boundaries and performs communication, while the second stream is responsible for computing the inner points. Despite being independent, the second stream is launched after the computation of the boundaries has finished in the first stream [10]. Even if the boundary kernels were launched simultaneously as the inner points kernel, the use of a single thread/process per device would result in a delay of the start of the inner points kernel.

We repeat the subdomain splitting process mentioned in Section IV-A. In addition we create a group of streams per thread. These streams are created for the send, receive and compute phases, as shown in Figure 4. In other words, the number of neighbor-handling

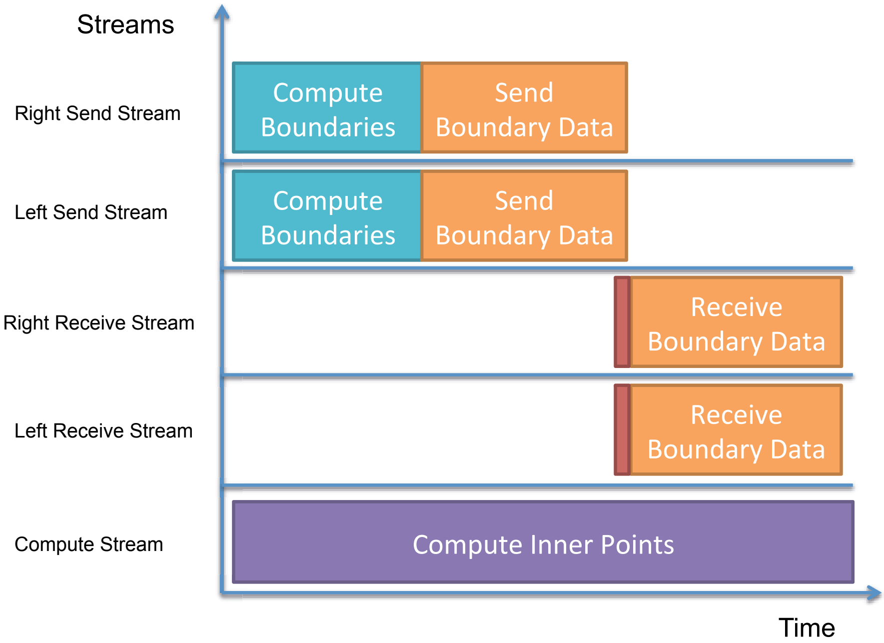


Figure 4: Five streams used in our scheme to achieve better overlap of communication and computation.

streams per GPU is twice the number of neighbors of the GPU.

The different assistant OpenMP threads are associated with each group of streams. The use of multiple streams has two major advantages. First, it enables us to decouple the data transfers going in different directions within each subdomain. As a result, the two phases can now be merged into a single phase. Second, by creating a group of streams and attaching the assistant threads to each stream group, we are able to mitigate the delays and overheads that arise in the state-of-the-art scheme.

There is another motivation for letting two assistant OpenMP threads separately control the sending and receiving streams. In the state-of-the-art scenario a single thread is used to control a device, synchronization will stall the master thread. By using additional threads, we avoid stalling the master thread. This is especially important for real-world applications where the master thread needs to attend to other tasks as well.

Multiple streams also express the independence that exists between tasks more clearly. We found this property to be especially useful on the Fermi GPUs, where placing multiple streams inside a loop can lead to false dependencies.

4.3 Discussion

Our scheme is built upon two principles: multi-threading and multi-streaming. These two techniques are combined to create a more efficient intra-node communication scheme that increases the overall concurrency.

Table 1: An overview of the GPUs used. SM denotes streaming multi-processor, whereas storage/SM means shared memory plus L1 cache per SM.

GPU	Tesla K20m	Tesla C2050
Architecture	Kepler	Fermi
# SMs	13	14
# cores/SM	192	32
register file/SM	65K	32K
storage/SM	64KB	64KB
memory size	5GB	3GB
bandwidth (GB/s)	208	144
SP, GFLOPs	3520	1030
DP, GFLOPs	1170	515

Multiple threads are used to reduce kernel launch overhead, avoid stalling the running master thread and improve application performance by reducing the gap between computation and communication. We also believe that multiple threads can also be useful for architectures that do not support streams.

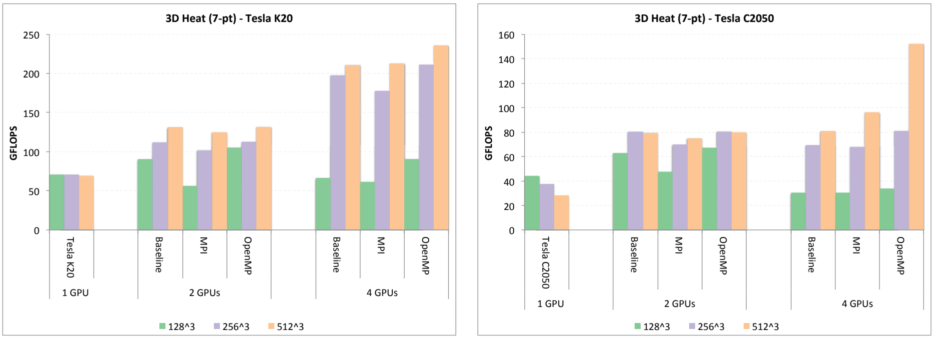
Multiple streams are used to stack communication so that data exchange can occur simultaneously on all boundaries of a subdomain. The use of multiple streams gives us a more fine-grained control of the different operations, enabling us to launch all groups of streams at the same time, resulting in bi-directional data transfers in a single phase. Moreover, the combination of multiple threads and multiple streams reduces the synchronization overhead that is needed between the send and receive streams. Additionally, P2P can be adopted for the purpose of reducing the overhead directly related to data transfer.

5 Experimental Setup and Measurements

All tests were conducted on two systems. The first machine is a dual-socket server containing two Intel Xeon E5-2650 CPUs with four Nvidia Tesla K20 GPUs. The second machine is a special multi-GPU node from NERSC's GPU testbed, Dirac. This node is equipped with two Intel Xeon 5520 CPUs with four Nvidia Tesla C2050 GPUs. A more detailed technical overview of the different GPUs is shown in Table 1. All calculations used double precision floating point with CUDA version 5.5, and OpenMPI 1.6.5.

5.1 Benchmark Stencil Computation

Stencil computations constitute one fundamental tool of scientific computing. They are typically used to discretely solve a differential equation using finite difference methods, which in turn give rise to a stencil calculation. For this paper, we have chosen the following



(a) Performance results on Tesla K20 (b) Performance results on Tesla C2050

Figure 5: The sustained performance results measured in GFLOPs on the two experimental platforms.

7-point stencil that sweeps over a uniform 3D grid:

$$U_{i,j,k}^{n+1} = \alpha U_{i,j,k}^{n} + \beta \left(U_{i+1,j,k}^{n} + U_{i,j+1,k}^{n} + U_{i,j,k+1}^{n} + U_{i-1,j,k}^{n} + U_{i,j-1,k}^{n} + U_{i,j,k-1}^{n} \right)$$

where α and β are two scalar constants. This 3D stencil computation can arise from discretizing the Laplace equation over a box and solving the resulting system of linear equations by Jacobi iterations. In the literature, this 3D stencil is thus widely referred to as the 3D Laplace stencil. (The same stencil can also arise from solving a 3D heat equation by a forward-Euler time stepping method.)

We chose, for simplicity reasons, to decompose our problem domain along the z direction, resulting in a 1D decomposition. One benefit of using a 1D decomposition is that kernels developed for a single-GPU can be used without any additional modification. Nevertheless, we acknowledge that using a 1D domain could be considered as suboptimal due to the communication overhead that arises when working on extremely large problem sizes or across many nodes. However, we believe it is sufficient for our study, as we deal with neither extremely large problem sizes nor many nodes. Previous studies such as [3] have shown that 1D decomposition with similar problem size, provides linear scaling up to 16 nodes.

Each result is divided into three scenarios, determined by the type of implementation used: Baseline, MPI, and OpenMP. The baseline version constitutes a naïve implementation where a single host thread is used to control all GPU devices. MPI represents the state-of-the-art scheme, while OpenMP represents our scheme.

The size of the 3D grid ranges from 128^3 to 512^3 inner points.

5.2 Experiments on Kepler

As Figure 5(a) shows, our scheme is able to outperform the state-of-the-art scheme in every case by a considerable margin. Interestingly, our scheme has a clearer advantage for smaller problem sizes when using two Kepler GPUs, and for the larger problem sizes when using four Kepler GPUs. However, if the computation part is big enough, communication can be entirely hidden. This explains the good performance of the state-of-the-art scheme for the largest problem sizes.

5.3 Experiments on Fermi

The performance results on the Fermi system, shown in Figure 5(b) are similar to the results observed on the Kepler platform. We are also able to outperform the state-of-the-art scheme on the Fermi GPUs. The performance gaps are especially large when four GPUs are involved. For example, our scheme is 58 % faster for the largest problem size.

By comparing the results from Figure 5(a), and Figure 5(b), we note that our scheme is able to outperform the state-of-the-art scheme quite considerably when two GPUs are used. On the other hand, when four GPUs are used, the difference between the two different schemes is not visible until we reach the larger problem sizes. For larger problem sizes, our scheme is better at hiding communication than the state-of-the-art scheme.

6 Related Work

Paulius Micikevičius [6] has investigated how a fourth-order wave equation can be solved on a single compute node with up to four GPUs using MPI. Micikevičius reports linear scaling in the number of GPUs used for all but one case. The domain is decomposed along the z-axis, and computation is overlapped with communication.

We decompose the domain in the same manner, and overlap computation with communication. However, we rely on the use of threads and not MPI processes to control multiple GPUs. We also make use of a new communication scheme to increase the overlap. We have observed the same linear and superlinear speedup that Micikevičius reports. The superlinear speedup observed can be explained by reduced TLB miss rate.

Thibault et al. [12] developed a multi-GPU Navier-Stokes solver in CUDA that targets incompressible fluid flow in 3D. In their study, one Pthread is spawned per GPU. The code runs on a single Tesla S870 GPU server with four GPUs. Depending on the number of GPUs and problem size, the speedup is between $21-100\times$ compared to a single CPU core. The implementation was observed not to overlap computation with communication, and CUDA streams are not used.

Thibault's work was extended by Jacobsen et al. [2]. Major changes include the use of CUDA streams to overlap computation with communication, and the use of MPI processes in preference to Pthreads. One MPI process was created per device. All experiments were conducted on a cluster containing 128 GPUs. Each node was equipped with two Tesla C1060 GPUs, putting the number of compute nodes at 64.

In Bernaschi et al. [1], a CUDA-aware MPI implementation is used to study the inter-node communication performance by measuring the time it takes to update a single spin of the 3D Heisenberg spin glass model. The scheme used in Bernaschi et al. uses two streams, one for compute and one for data exchange. P2P is used between two nodes (each equipped with two different Fermi GPUs). Inter-node P2P is possible thanks to the use of APEnet+, a custom 3D Torus interconnect that can access the GPU memory without going through the host.

Our proposed scheme used up to five streams per GPU, four streams for data exchange, and one for the inner points. The computations of the inner points and the boundaries are scheduled to run at the same time. Moreover, since our scheme uses OpenMP threads, we are able to easily pass pointers between GPUs with minimal overhead, whereas exchanging GPU pointers using MPI processes involves explicit message passing.

7 Conclusions

We have proposed a new intra-node communication scheme that is faster than the existing state-of-the-art approach. The main ingredient of our scheme is to combine multiple OpenMP threads with multiple CUDA streams for a more efficient overlap of communication with computation.

First, we make use of multiple OpenMP threads per device to increase the scheme concurrency. This is in stark contrast to the state-of-the-art where each device is controlled by a single thread or process. Then, we create a group of CUDA streams for each stage of the communication, and computation, whereas the state-the-art approach uses only two CUDA streams. Finally, we combine the two techniques together to create a more efficient intra-node communication scheme that is able to perform bi-directional communication with lower synchronization overhead.

Depending on the test platform, results indicate that our scheme is able to outperform the state-of-the-art scheme quite noticeably. The best observed speedup on the C2050 platform was 1.6×, and 1.85× on the K20 platform.

We have three immediate extensions planned for the future. The first is to study the effect of larger ghost regions. Currently, our implementations use the thinnest possible ghost region width. A thicker ghost region can potentially benefit the computation of wider stencils such as a 19-point stencil or in combination with time unrolling where two sweeps are performed per time step. Time unrolling trades off redundant computation for a reduced number of boundary data exchanges.

In this study we have looked at a traditional stencil code, however, our scheme is by no means limited to stencil method. We are in the process of exploring the use of our strategies in a real world application involving cell-centered finite volume method on a 3D unstructured tetrahedral mesh. Finally, work is also underway to extend our scheme to other architectures such as Intel's Xeon Phi.

Acknowledgments

This work was supported by the FriNatek program of the Research Council of Norway, through grant No. 214113/F20. This work used the Extreme Science and Engineering Discovery Environment, which is supported by National Science Foundation grant No. OCI-1053575. Computer time on the Dirac system was provided by the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Bibliography

1. Bernaschi, M., M. Bisson, and D. Rossetti (2013, February). Benchmarking of communication techniques for GPUs. *Journal of Parallel and Distributed Computing 73*, 250–255.

2. Jacobsen, D., J. Thibault, and I. Senocak (2010, January). An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting*.

3. Maruyama, N., T. Nomura, K. Sato, and S. Matsuoka (2011). Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 11:1–11:12.

4. Message Passing Interface Forum (2012, September). *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS).

5. Micikevicius, P. Multi-GPU programming. `http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf`.

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

7. OpenMP Architecture Review Board (2015). OpenMP Application Program Interface. `http://openmp.org`. [Online; accessed 23-May-2015].

8. Phillips, E. H. and M. Fatica (2010, April). Implementing the Himeno benchmark with CUDA on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–10.

9. Playne, D. P. and K. A. Hawick (2011, July). Asynchronous communication for finite-difference simulations on GPU clusters using CUDA and MPI. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 169–174.

10. Rietmann, M., P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini (2012, November). Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 38:1–38:11.

11. Shimokawabe, T., T. Aoki, and N. Onodera (2014, January). A high-productivity framework for multi-GPU computation of mesh-based applications. presented at the International Workshop on High-Performance Stencil Computations, Vienna, Austria.

12. Thibault, J. and I. Senocak (2010, January). CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*.

13. Top500.org (2015). June 2015 | TOP500 Supercomputer Sites. http://top500.org/lists/2015/06/. [Online; accessed 04-Sept-2015].

Paper II:
CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters

CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters

Mohammed Sourouri[1,2], **Johannes Langguth**[1], **Filippo Spiga**[4], **Scott B. Baden**[3], **Xing Cai**[1,2]

[1] Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] University of California, San Diego,
La Jolla, CA 92093 USA

[4] University of Cambridge, UK,
The Old Schools, Trinity Ln, Cambridge CB2 1TN, United Kingdom

Manycore processors such as Graphics Processing Units (GPUs) and Xeon Phis have remarkable computational capabilities and energy efficiency, making these units an attractive alternative to conventional CPUs for general-purpose computations. The distinct advantages of manycore processors have been quickly adopted to modern heterogeneous supercomputers, where each node is equipped with manycore processors in addition to CPUs.

This thesis takes aim at developing methodologies for efficient programming of GPU clusters, from a single compute node equipped with multiple GPUs that share the same PCIe bus, to large supercomputers involving thousands of GPUs connected by a high-speed network. The former configuration represents a peek into future node architecture of GPU clusters, where each compute node will be densely populated with GPUs. For this type of configuration, intra-node communication will play a more dominant role. We present programming techniques specifically designed to handle intra-node communication between multiple GPUs more effectively. For supercomputers involving multiple nodes, we have developed an automated code generator that delivers good weak scalability on thousands of GPUs.

While GPUs are improving rapidly, they are still not general-purpose, and depend on CPUs to act as their host. Consequently, GPU clusters often feature powerful multi-core CPUs in addition to GPUs. Despite the presence of CPUs, the focal point of many GPU applications has so far been on performing computations exclusively on the GPUs, keeping CPUs sidelined. However, as CPUs continue to advance, they have become too powerful to ignore. This gives rise to heterogeneous computing where CPUs and GPUs jointly take part in the computations.

The potentially achievable performance of heterogeneous computing codes can be very large, but requires careful attention to many programming details. We explore resource-efficient programming methodologies for heterogeneous computing where the CPU is an integral part of the computations. The experiments conducted demonstrate that by careful workload-partitioning and communication orchestration, our heterogeneous computing strategy outperforms a similar GPU-only approach on structured grid and unstructured grids.

Although our work demonstrates the benefit of heterogeneous computing, the painstaking programming effort required is holding back its wider adoption. We address this issue through the development and implementation of a programming model and source-to-source compiler called Panda, which automatically parallelizes serial 3D stencil codes originally written in C to heterogeneous CPU+GPU code for execution on GPU clusters. We have used two applications to assess the performance of our framework. Experimental results show that the Panda-generated code is able to realize up to 90% of the performance of corresponding handwritten heterogeneous CPU+GPU implementations, while always outperforming the handwritten GPU-only implementations.

Compared to the more established GPU-only approach, the methodologies presented in this thesis contribute to harnessing the computational powers of GPU clusters in a more resource-efficient way that can substantially accelerate simulations. Moreover, by providing a user-friendly code generation tool, the tedious and error-prone process associated with programming GPU clusters is alleviated, so that computational scientists can concentrate on the science instead of code development.

1 Introduction

GPU clusters are becoming increasingly popular, because they deliver excellent performance and high power efficiency in many types of scientific applications [16, 27]. However, the current generation of GPUs are not general-purpose, which means they cannot act as standalone devices. GPUs are therefore dependent on general-purpose CPUs that can act as hosts. Even if future generations of GPUs could operate as standalone devices, a virtually CPU-free cluster is not advisable since GPUs are not suitable for certain HPC workloads. Future GPU clusters such as LLNL Sierra and ORNL Summit [11] suggest that heterogeneous CPU-GPU clusters will remain widespread, which prompts the development of CPU+GPU computing.

In heterogeneous CPU+GPU applications, several programming models, such as MPI, OpenMP and CUDA are combined to exploit the strengths of CPUs and GPUs to achieve high performance. However, three principal problems arise when going from homogeneous to heterogeneous CPU+GPU computations.

First, an additional work partitioning between the CPUs and GPUs needs to be introduced. Unlike the standard partitioning among the compute nodes of a homogeneous cluster, this new workload division is asymmetric, which requires a division ratio that is proportional to the relative compute speed of the two processing units. Second, inter- and intra-node data exchanges involving both the CPUs and GPUs, must be properly programmed in order to attain high communication efficiency. Third, a scheme for controlling intra-node communication, synchronization and computation must be implemented.

The challenge here lies in establishing an effective overlap between the computation and the inter- and intra-node communication. This requires introducing proper task parallelism in addition to the usual data parallelism.

To tackle the first problem, we make use of the fact that many scientific computations are memory-bound. Thus, we can roughly predict the CPU-only and GPU-only performance based on realistic memory bandwidth values obtained by e.g. the STREAM benchmark [8]. Such a simple modeling can suggest a reasonable workload division between the CPU and the GPU(s) on each compute node. At the same time, care should be taken to make a heterogeneous implementation flexible enough for fine-tuning the workload division.

We address the second problem by using a single MPI process to control the CPU and the GPU, instead of two separate MPI processes. The resulting CPU-GPU data exchanges are performed via `cudaMemcpyAsync`. For GPU-GPU data exchanges, we use a portable approach that lets the respective CPU relay the messages. In other words, MPI messages between CPUs cover both CPU ⟷ CPU and GPU ⟷ GPU interactions. The key to efficient data exchanges lies in obtaining a good overlap with various computing activities, which are controlled by the CPU. Direct GPU ⟷ GPU data exchanges across nodes can easily be adopted to improve our approach, but this requires that CUDA-aware MPI with GPUDirect v3 [10] is available.

To overlap computations with the various intra-node and inter-node data exchanges, we adopt a programming style that involves MPI, OpenMP and CUDA. In such an approach, task parallelism is key. Some of the OpenMP threads are dedicated to the main part of the computation, while the remaining OpenMP threads handle other tasks, such as data movement and boundary computations.

To illustrate the programming details, we choose a widely used 3D 7-point stencil. Our programming approach is by no means limited to the motivating example, and is applicable to many other numerical computations. This paper makes the following contributions:

- We develop an optimized heterogeneous CPU+GPU implementation for computing the stencil over a uniform 3D grid with detailed illustrations of advanced programming techniques (Section 3).

- We show that fine-grained use of CPU threads increases parallelism and thus application performance.

- Insight into a simple performance model for heterogeneous CPU+GPU applications (Section 4).

- Experiments showing that concurrent CPU+GPU computations can outperform a highly optimized GPU-only implementation, even on GPU clusters where each compute node is equipped with two GPUs per node (Section 4).

2 GPU-only Implementations

In this section, we give a detailed description of the GPU-only, MPI+CUDA implementation used.

For this paper, we have chosen a representative numerical kernel that sweeps over a uniform 3D grid. The size of the grid is defined as $N_x \times N_y \times N_z$, and a 7-point stencil is used to alternatingly update the two arrays, `u_new` and `u_old`, which are stored in row-major order. The corresponding GPU versions of the two arrays are d_unew and d_uold.

2.1 Single-GPU Implementation

To secure good single-GPU performance, we use the pipelined wavefront technique, as described in [15] and [18]. This technique consists of introducing a for-loop to compute values column-wise along the z axis. In addition, we make use of the Kepler GPUs' fast read-only cache. In our example application, data from d_uold is only read, making it an ideal candidate for the read-only cache. Moreover, a small portion of the constant memory is used to store constants in order to free registers.

2.2 Multi-GPU Implementation

It suffices to use a conventional 3D domain decomposition to break the global domain into smaller 3D subdomains, one per GPU. In this implementation, all the computational work is done by the GPUs, while the required GPU $\longleftrightarrow$ GPU interactions are realized as MPI messages that are relayed through the CPU.

To enable overlap between computation and communication, the entire computation is split into interior points and boundary points, and further, each subdomain is extended with a layer of ghost cells [4].

Although CUDA-aware MPI libraries with GPUDirect v3 such as MVAPICH2-GDR [23] can simplify multi-GPU programming, we have assumed for the sake of generality and portability that GPU-GPU interactions are relayed through the CPU. In other words, CPU $\longleftrightarrow$ GPU intra-node data movement is always required, in addition to inter-node MPI communication. Since each subdomain's boundary points are computed first, the overhead related to the intra-node data transfer and inter-node MPI can be overlapped with the remaining computation on the interior grid points.

In addition to a CUDA kernel that computes the interior points, we use supplementary CUDA kernels to compute the boundary points. We create one CUDA stream per side of the subdomains in order to allow simultaneous execution of the compute kernels for the boundary points and the interior points. Furthermore, asynchronous data transfer functions such as cudaMemcpyAsync are used to perform concurrent CPU $\longleftrightarrow$ GPU data transfers. As a result, computation and communication can be overlapped.

```
for (int t = 0; t < iterations; t++) {
  for (auto i: direction_vector)
    MPI_Irecv(recv_buf);
    ComputePackBoundary<<<dir_stream[i]>>>;
    cudaMemcpyAsync(cudaMemcpyDeviceToHost,dir_stream[i]);

  ComputeInteriorPoints<<<inner_stream>>>;

  for (auto i: direction_vector)
    cudaStreamSynchronize(dir_stream[i]);
    MPI_Isend(send_buf);

```

```
  MPI_Waitall();

  for (auto i: direction_vector)
    cudaMemcpyAsync(cudaMemcpyHostToDevice,dir_stream[i]);
    UnpackBoundary<<<dir_stream[i]>>>;

  // cudaDeviceSynchronize and swap pointers
}
```

Listing 1: An MPI+CUDA implementation using only GPUs for computation.

Listing 1 outlines the pseudocode of our multi-GPU implementation. The different sides are stored as enumerated types in a C++ vector, iterated using the C++11 `auto` keyword. Before every boundary exchange, points in the boundary region need to be computed and then stored in a buffer. The process of storing values in a buffer is referred to as *packing*, and the reverse is called *unpacking*. In our implementation, the `ComputePackBoundary` kernel performs computation of boundary points as well as packing, thus eliminating redundant global memory accesses.

Each `ComputePackBoundary` kernel is placed in its own CUDA stream. The same streams are reused for unpacking, similar to the approach presented in [17]. In Listing 1, `cudaStreamSynchronize` ensures that data from the GPU has been successfully copied to the CPU before `MPI_Isend` is called.

In the multi-GPU version presented in this section, computation and communication are overlapped since the computation of interior points is decoupled from the computation of boundary points through the use of separate CUDA streams, and thus occurs simultaneously with the MPI communication.

3 Heterogeneous CPU+GPU Implementations

In this section we describe two CPU+GPU implementations that extend the multi-GPU implementation presented in Section 2.2. The drawback with the GPU-only implementation is that it leaves the CPU unused most of the time. Hence, both implementations described in this section aim to exploit the CPU for further performance improvements. The first implementation does so in a naive manner, while the second implementation adopts further optimizations including task parallelism, realized through OpenMP's nested parallel regions.

Both of our CPU+GPU implementations mix MPI+CUDA with OpenMP. We have already introduced the MPI+CUDA part, but not the OpenMP part. Thus, we start by describing a multi-core CPU implementation.

3.1 Multi-Core CPU Implementation

Our CPU kernel uses pencil shaped cache blocking along the y axis in combination with non-temporal store instructions, as described in [20]. This technique can be considered a special case of the original quadrilateral cache blocking technique presented in [14]. We

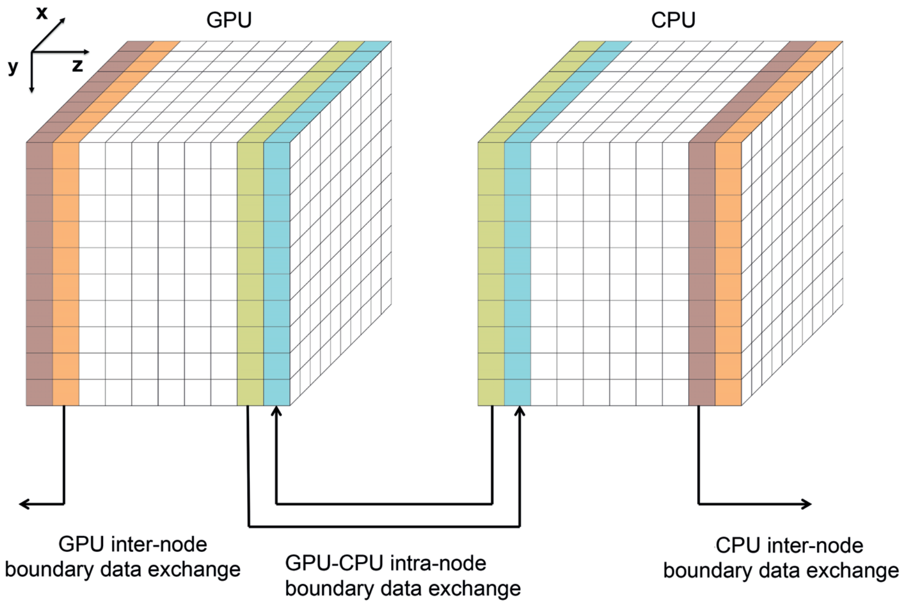


Figure 1: Workload division between the CPU and GPU on a compute node.

have also implemented a simple auto-tuner, which ensures that the optimum cache block size is chosen when the code is moved from one cluster to another.

3.2 Naive Implementation

We decompose the global domain as described in Section 2.2. Moreover, each subdomain is decomposed again, this time using a 1D decomposition along the z axis. This means that each subdomain is divided into two parts, as illustrated in Figure 1. The reason for decomposing each subdomain along the z axis is that a xy plane is contiguous in memory.

The total volume of inter-node data movement remains the same, but the number of MPI messages in the x and y directions are doubled, because CPUs are now also responsible for a part of the computation. For this reason, separate send and receive buffers are created for the CPU and GPU parts. These are noted as `gpu_send_buf`, `cpu_send_buf`, etc. in the following pseudocode.

In order to overlap computation with communication, we continue to use the same technique as in our multi-GPU implementation, where interior points and boundary points are treated separately. We reuse the CUDA kernels from our multi-GPU code, but write new functions for boundary point computation on the CPU side.

To better mask the intra-node boundary data exchange between the GPU and CPU, we create two additional CUDA streams called *data_stream* and *intra_stream*. These streams are used for CPU $\longleftrightarrow$ GPU intra-node boundary data transfers.

The basic strategy of the naive implementation is to augment the existing MPI+CUDA implementation using OpenMP, as outlined in Listing 2.

```
#pragma omp parallel default(shared)
for (int t = 0; t < iterations; t++) {
  #pragma omp master
  {
    for (auto i: direction_vector ex. intra boundary)
      MPI_Irecv(gpu_recv_buffer);
      MPI_Irecv(cpu_recv_buffer);
      ComputePackBoundary<<<dir_stream[i]>>>;
      cudaMemcpyAsync(DeviceToHost,dir_stream[i]);

    ComputeIntraBoundary<<<intra_stream>>>;
    cudaMemcpyAsync(DeviceToHost,intra_stream);

    ComputeInteriorPoints<<<inner_stream>>>;
  }

  for (auto i: direction_vector)
    HostComputePackInterBoundary();
    HostComputeIntraBounddary();

  #pragma omp barrier
  #pragma omp master
  {
    cudaMemcpyAsync(HostToDevice,data_stream);

    for (auto i: direction_vector ex. intra boundary)
      MPI_Isend(cpu_send_buffer);
      cudaStreamSynchronize(inter_stream);
      MPI_Isend(gpu_send_buffer);

    MPI_Waitall();

    for (auto i: direction_vector ex. intra boundary)
      cudaMemcpyAsync(HostToDevice,dir_stream[i]);
      UnpackBoundary<<<dir_stream[i]>>>;
  }

  HostComputeInteriorPoints();

  for (auto i: direction_vector ex. intra boundary)
    HostUnpackBoundary();

  #pragma omp barrier
  #pragma omp master
  {
    cudaDeviceSynchronize();
    std::swap(d_uold,d_unew);
    std::swap(u_uold,u_unew);
  }
  #pragma omp barrier
}}
```


Listing 2: A naive MPI + OpenMP + CUDA implementation.

Using OpenMP, we spawn a number of CPU threads equal to the number of physical CPU cores, and declare the stencil compute loop as a parallel OpenMP region. First, we use the master thread to launch the different CUDA kernels and GPU → CPU data copies before proceeding to computation of the boundary points on the CPU. Upon completion of the last boundary on the CPU, we start the communication of the different boundaries using MPI, creating a sequential pipeline of MPI messages.

Before data from the GPU can be communicated, a `cudaStreamSynchronize` is necessary to ensure that data has arrived on the CPU. Moreover, `MPI_Waitall` is required before launching the unpacking kernels on the GPU. Next, computation of the interior points on the CPU, `HostComputeInteriorPoints`, is started.

Unpacking of the received boundary data on the CPU, `HostUnpackBoundary`, is not dependent on the computation of the interior points on the CPU, but in order to avoid delaying this computation, we call `HostUnpackBoundary` as soon as the computation has finished. Furthermore, before an entire iteration is concluded, the master thread, guarded by two barriers (one before and after) swaps the data pointers.

The naive implementation can be further simplified by using the multithreaded thread safety level (MPI_THREAD_MULTIPLE) provided by many MPI libraries. Although this mode greatly simplifies the actual code, it comes at the expense of performance. Because when the thread safety level is set to MPI_THREAD_MULTIPLE, the MPI library makes extensive use of synchronization to keep internal data structures safe, resulting in high communication overhead.

3.3 Nested Implementation

There are two drawbacks with the naive CPU+GPU implementation: a) MPI communication is not guaranteed to happen simultaneously with computation of the interior points on the CPU. b) Computation of the interior points and boundary points do not happen simultaneously on the CPU due to the lack of task parallelism. The use of the CPU's resources is too coarse-grained, leading to much idling, and thus, performance degradation.

The drawbacks to the naive implementation are addressed in an alternative implementation which we call *nested*, due to the use of nested parallelism. We insert task parallelism by dividing the OpenMP threads into two groups. This is done using nested parallel OpenMP regions. To enable nested parallelism support in OpenMP, the `omp_set_nested` flag must be set to `true`. Moreover, each parallel region uses the `num_threads` clause to explicitly specify the number of threads to use within each parallel region. The pseudocode for the nested implementation is shown in Listing 3.

First, two OpenMP threads are spawned in the outer parallel region. Then, each of the two threads starts an inner parallel region with its own thread group. The total number of threads equals the number of physical CPU cores.

The first group is responsible for computation of boundary points, controlling the GPU, and MPI communication, while the second group is responsible for computing the interior points on the CPU. Within each thread group, a new parallel region is created.

In the first thread group, `MPI_Irecv` and the CUDA kernels for computing boundary points are posted by the master thread, while the other threads perform computation of boundary points on the CPU. Once the master thread has completed its duties, it will join the other threads in its group in computing the boundary points.

Simultaneously, threads in the second group are busy with computing the interior points. Although the use of additional parallel regions implies extra overhead, this approach has the major advantage that thread synchronization in one thread group will not affect the threads in the other group. This means that the necessary synchronization required before MPI communication in the first thread group will not stall the computing threads in the other group.

```
omp_set_nested(true);

#pragma omp parallel default(shared) num_threads(2)
{
  int tid = omp_get_thread_num();

  for (int t = 0; t < iterations; t++) {
    if (tid == 0) {
      #pragma omp parallel num_threads(x)
      {
      #pragma omp master
      {
        for (auto i: direction_vector ex. intra boundary)
          MPI_Irecv(gpu_recv_buffer);
          MPI_Irecv(cpu_recv_buffer);
          ComputePackBoundary<<<dir_stream[i]>>>;
          cudaMemcpyAsync(DeviceToHost,dir_stream[i]);

        ComputeIntraBoundary<<<intra_stream>>>;
        cudaMemcpyAsync(DeviceToHost,intra_stream);

        ComputeInteriorPoints<<<inner_stream>>>;
      }

      for (auto i: direction_vector)
        HostComputePackInterBoundary();
        HostComputeIntraBoundary();

      #pragma omp barrier
      #pragma omp master
      {
        cudaMemcpyAsync(HostToDevice,data_stream);

        for (auto i: direction_vector ex. intra boundary)
          MPI_Isend(cpu_send_buffer);
          cudaStreamSynchronize(dir_stream[i]);
          MPI_Isend(gpu_send_buffer);
```

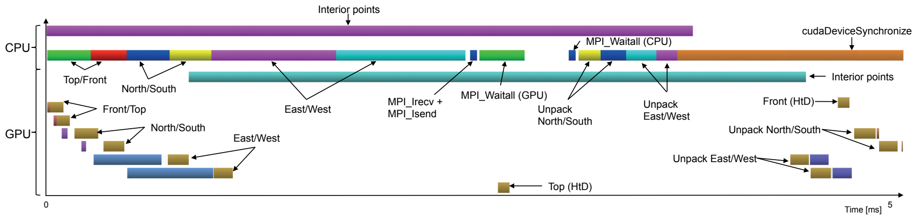


Figure 2: An actual profiling snapshot for the nested heterogeneous implementation, focusing on a single compute node that has six neighbors.

```

        MPI_Waitall();

        for (auto i: direction_vector ex. intra boundary)
          cudaMemcpyAsync(HostToDevice,dir_stream[i]);
          UnpackBoundary<<<dir_stream[i]>>>;
        }

        #pragma omp barrier
        for (auto i: direction_vector ex. intra boundary)
          HostUnpackBoundary();
        } // end omp region for the first thread group
      } else { // tid == 1
        #pragma omp parallel for num_threads(y)
        HostComputeInteriorPoints();
      }

    #pragma omp master
    {
      // cudaDeviceSynchronize and swap pointers
    }
    #pragma omp barrier
    }}
```

Listing 3: A nested MPI + OpenMP + CUDA implementation.

By using the Nvidia Nvprof profiling application, we acquired vital profiling data that is rendered in Figure 2. As the illustration shows, in the nested implementation, computation of the interior points on the CPU is overlapped with computation of the boundary points and communication. The profiling also reveals that while the different functions are overlapped on the CPU, this is not the case on the GPU.

Neither the packing nor the unpacking kernels are overlapped with computation of the interior points on the GPU. Because the computation of the boundary points occupies all the SMs on the GPU, the execution of the kernel that computes the interior points is suspended until enough resources become available. Likewise, the computation of the interior points occupies all the SMs and thus delays the start of the unpacking kernels until it relinquishes some SMs.

Table 1: An architectural overview of the clusters used.

Cluster	**Stampede**	**Wilkes**
Processor	Xeon E5-2680	Xeon E5-2630v2
Architecture	Sandy Bridge-EP	Ivy Bridge-EP
Cores	8	6
Sockets	2	2
# GPUs per node	1	2
Clock freq. [GHz]	2.7	2.6
L3 cache/chip	20 MB	15 MB
Memory size	32 GB	64 GB
Peak DP, GFLOPs	345.6	249.6
Peak BW [GB/s]	102.4	119.4
STREAM [GB/s]	77	72.95
Compiler	icc 13.1	icc 13.1
MPI	mvapich-2 1.9	mvapich-2 2.1

4 Performance Projections

The overall goal of a CPU+GPU implementation is to exploit the entire pool of hardware resources on a compute node in order to solve a given problem as quickly as possible. However, as Tables 1 and 2 show, the CPU and the GPU are not equally fast. Thus, a good CPU+GPU implementation must take the different computational speeds into account. Failing to do so will generally lead to a severe load imbalance because the fast GPU will constantly wait for the slow CPU to complete its workload, and thus to poor performance.

Because our numerical kernel is memory-bound, we balance the load according to the realistic memory bandwidth values, CPU_{Bw} in Table 1 and GPU_{Bw} in Table 2, which are measured by the STREAM benchmark.

We use $CPU_{Bw}/(GPU_{Bw}+CPU_{Bw})$ to determine the workload ratio assigned to the CPU, and assign the remaining work to the GPU.

5 Experimental Setup And Results

In this section we present experimental results obtained using the GPU-only implementation described in Section 2.2, and the two CPU+GPU implementations described in Section 3. We used two supercomputers, Stampede and Wilkes to perform our experiments.

TACC Stampede is a primarily Xeon Phi cluster, but a small number of the nodes are equipped Tesla K20m GPUs instead (one per node). While 128 GPU nodes are available, they have been partitioned in such a way that it is not possible to access more than 32 GPU nodes at a time.

Wilkes is a GPU cluster at the University of Cambridge, UK. The cluster consists of 128

Table 2: An architectural overview of the accelerator present in both systems.

Accelerator	**Nvidia Tesla K20**
Architecture	Kepler
# of SMs	13
Clock freq. [GHz]	0.7
On-chip memory	64 kB
Memory size	5 GB
Peak DP, GFLOPs	1170
Peak BW [GB/s]	208
STREAM [GB/s]	151
Compiler	nvcc 6.0

nodes, where each node is equipped with two Tesla K20c GPUs. The CPUs on Wilkes are weaker than those found in Stampede. We were unable to use all of the 128 nodes due to several non-operational nodes.

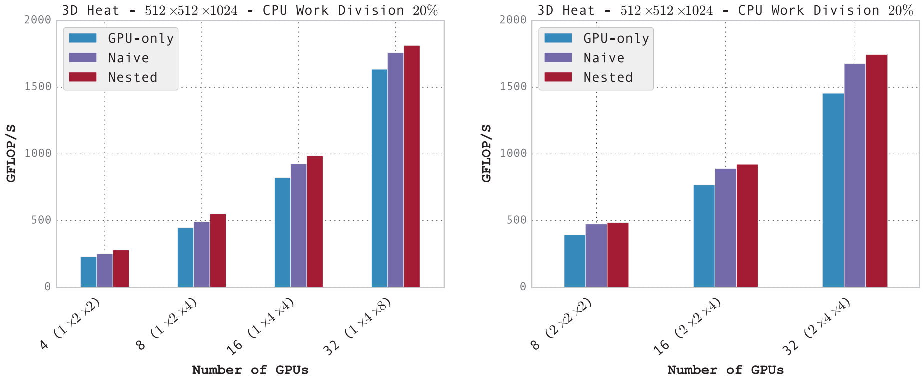
5.1 Strong Scaling

For the strong scaling experiments, we fix the problem size at $512 \times 512 \times 1024$, while varying the CPU workload ratio from 10% to 25% using a step size of 5%. With the availability of 16 CPU threads and only a single GPU per node on Stampede, one needs to pay extra attention to the nested implementation so that the two thread groups are balanced correctly. In our experiments, the optimum thread distribution was 10 CPU threads on computation of the interior points, and the remaining six threads in the other thread group.

Figures 3(a) and 3(b) show a comparison of the different implementations using 2D and 3D domain decompositions on Stampede, while Figures 4(a) — 4(d) display the same information for the Wilkes cluster. Both of our CPU+GPU implementations scale well, and they are able to outperform the GPU-only implementation. The benefit of the nested implementation is more evident when 16 or more nodes are used.

The difference between the two CPU+GPU implementations is smaller on Wilkes. This is due to the availability of fewer CPU cores, which means a smaller contribution from the CPU segment. Moreover, a prerequisite for achieving good performance using the nested implementation is that the CPU cores in the different thread groups are not overutilized. For example, if one thread group simply has too much work to do, it might lead to threads in the opposite group to idle excessively. Ideally, we would like both thread groups to be perfectly balanced so that they complete their tasks simultaneously.

Results where both GPUs on each Wilkes node are used, are shown in Figures 4(c) and 4(d). In order to make use of both GPUs of Wilkes, the number of MPI processes is doubled on each compute node for the GPU-only and the CPU+GPU implementations. When both GPUs and 2D decomposition are used, the GPU-only implementation is faster than the naive version. The performance degradation observed in both of the CPU+GPU



(a) Performance using 2D domain decomposition (b) Performance using 3D domain decomposition

Figure 3: Strong scaling performance results measured in GFLOPs on the GPU nodes of Stampede.

implementations can be explained by the increased number of MPI processes per node. On Wilkes, using two MPI processes per compute node means that each process gets access to only six OpenMP threads, which easily leads to a CPU workload that is too high. In spite of this, we observe that both CPU+GPU implementations are marginally faster than the GPU-only implementation when using 3D decomposition.

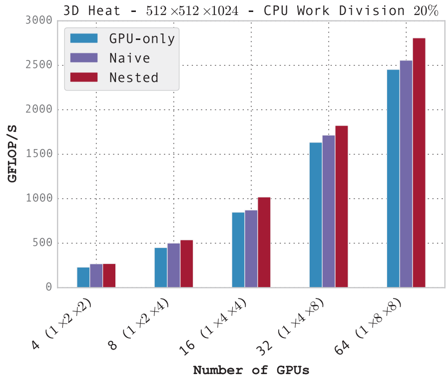
5.2 Weak Scaling

For our weak scaling experiments, we keep the problem size fixed at 512^3 for each MPI process. Figures 5(a) — 3(b) show the weak scaling performance on Stampede and Wilkes using 3D domain decomposition. We observe that both CPU+GPU implementations are faster than the GPU-only version, illustrating that the CPU can increase the overall compute performance, and thus improve the overall scalability. Moreover, as the number of nodes increases, the nested version increases its lead compared to the naive implementation. This is due to the nested version's ability to better overlap computation with communication.

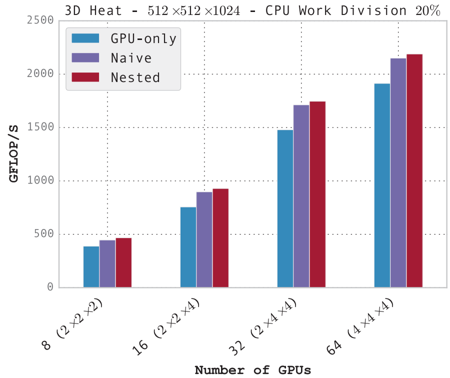
5.3 Sensitivity to Workload Division

For this particular study, we conduct strong scaling experiments, but vary the CPU's workload ratio from 10% to 40%, using a step size of 1%. The same experiment is repeated with an increasing number of nodes. The goal of this study is to find the best CPU workload ratio for a given problem size.

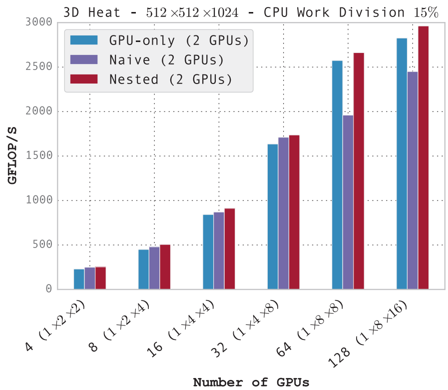
Figure 5(d) shows the impact of the workload ratio on the two evaluation platforms. We observe different optimum workload ratios for different numbers of nodes. This means that predicting an optimum workload ratio beforehand can be quite challenging since the workload ratio is heavily influenced by multiple factors such as problem size, decomposition, and node count. These factors highlight the importance of implementations in which the workload division can be easily adjusted.



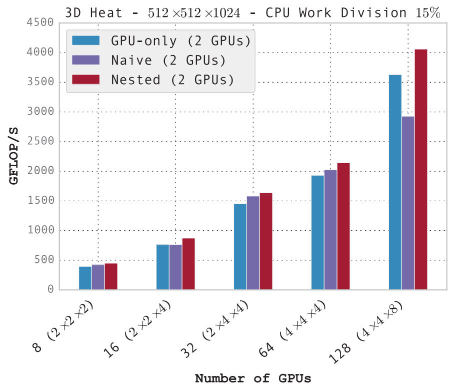
(a) Performance using 2D domain decomposition and one GPU per node



(b) Performance using 3D domain decomposition and one GPU per node



(c) Performance using 2D domain decomposition and two GPUs per node

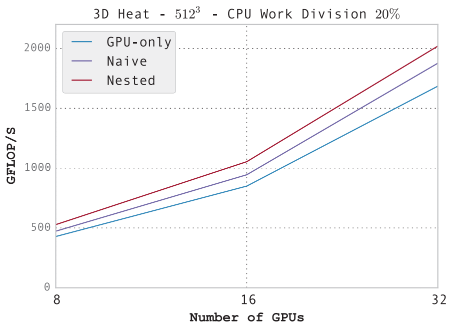


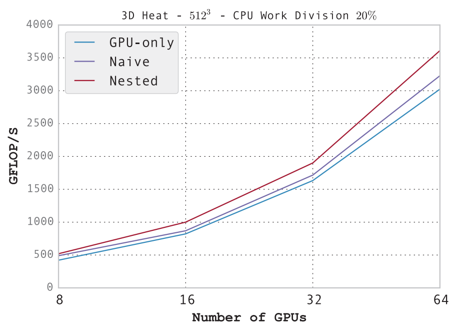
(d) Performance using 3D domain decomposition and two GPUs per node

Figure 4: Strong scaling performance results measured in GFLOPs on the GPU nodes of Wilkes.

The underlying architecture also plays a pivotal role with respect to the workload ratio. For example, we observed that in the strong scaling studies when the number of subdomains is large, the CPU's share of computation might fit into its L3 cache, which creates a workload imbalance between the CPU and the GPU. In such a scenario, the CPU will wait on the GPU. To address this problem, we increased the CPU's workload.

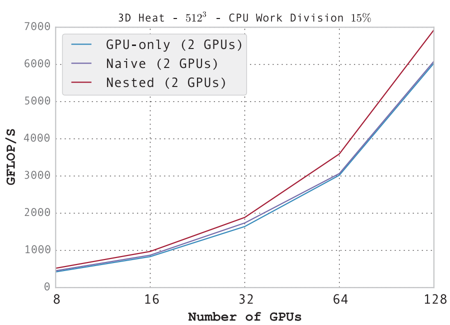
Based on the performance model discussed in Section 4, we can project the optimum workload ratio. As shown in Figure 5(d), peak performance for Stampede is achieved when the CPU workload division ratio is 23% and 25%, for 8 and 32 nodes, respectively. On Wilkes, the optimal performance is achieved at CPU workload division ratios of 19% and 17%, indicating that our simple performance model is indeed capable of predicting a reasonable initial workload ratio (29% for Stampede and 22% for Wilkes). Moreover, we observe that a good work division strategy is to always give the CPU a slightly smaller

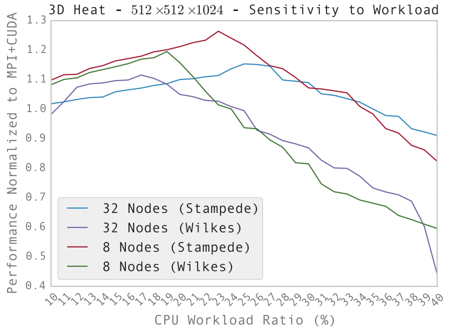




(a) Weak scaling on Stampede using 3D domain decomposition

(b) Weak scaling on Wilkes, one GPU per node, 3D decomposition





(c) Weak scaling on Wilkes, two GPUs per node, 3D decomposition

(d) CPU workload division sensitivity using 2D domain decomposition

Figure 5: Weak scaling results on Stampede and Wilkes.

workload than suggested by the performance model. The result of a small CPU workload might leave the CPUs underutilized, but it does not degrade the performance as much as a workload that is too big. As mentioned previously, an oversized CPU workload can lead to catastrophic consequences since vital parts of the application such as MPI communication and launching of GPU kernels might be substantially delayed.

6 Related Work

Although stencil applications using both single- and multi-GPU programming have by now been thoroughly studied [9, 12, 28], fewer works have considered the topic of large-scale stencil CPU+GPU computing. At the broader scale, the topic of CPU+GPU computing has been discussed in many scietific works. For example, Horton et al. [3] updated the MAGMA library so that Cholesky, QR, and LU factorizations can be performed conccurently on the CPU and the GPU. Liu and Luk [7], and Wang et al. [22] have looked at scientific CPU+GPU computation in the context of power efficiency, while Rahimian et al. have developed a large-scale CPU+GPU blood-flow simulator for real-world use [13]. Moreover, a lot of research activities have also focused on dynamic scheduling algorithms for CPU+GPU computations. Prominent examples are DAGuE [2], ClusterSs [19], StarPU [1], and CAP [24]. While being successful in reaching their stated goals, none of these works describes performance projections, domain decomposition, communication handling, and other fundamental programming details with respect to stencil computation.

Langguth and Cai [5] have studied CPU+GPU finite volume computations on unstructured grids using a single compute node. Similarly, Wang and JaJa [25] have focused on accelerating an FFT-based Poisson solver on a single compute node. Both report that their CPU+GPU implementations outperform the corresponding GPU-only implementations.

Venkatasubramanian and Vuduc [21] present a small-scale CPU+GPU implementation together with a performance model for a 2D Poisson equation on a square domain using Jacobi's method. Thanks to algorithms such as chaotic relaxation and asynchroneous iteration, CPU-GPU synchronization are minimized. The authors report that their single node CPU+GPU implementations are able to outperform the corresponding GPU-only implementation by 8% and 11% (depending on the system). However, the authors are unable to observe the same performance gain for their CPU+GPU implementation when moving on to multiple nodes. The big performance difference between the GPU and the CPU is given as the reason. Despite a slower multi-node CPU+GPU implementation, the authors conclude that CPU+GPU implementations will play an important role in future CPU-GPU architectures.

To the best of our knowledge, only the works by Shimokawabe et al. [16], Yang et al. [26], and Langguth et al. [6] compare directly with our work, as they perform CPU+GPU computation at a larger scale.

Our work differs from [16] in multiple ways. First of all, in [16], due to problems with the memory accesses on the GPU, only a global 2D domain decomposition is used. Our work, on the other hand has demonstrated good scaling for both 2D and 3D decompositions. Another crucial difference between our work and the study by Shimokawabe et al. is CPU utilization. In Shimokawabe et al., the CPU is used only for lightweight boundary computations, while interior point computations are left to the GPU. When Shimokawabe et al. attempted to allow the CPUs to handle a larger part of the computation they observed that the CPU became a bottleneck. In our strategy, the CPU computes a slice of the total domain that is commensurate with its computational speed.

In [26], a CPU+GPU implementation is developed to perform atmospheric simulations on a cubed-sphere domain. Similarly to [16], the implementation presented in [26] uses the CPU for boundary computation only. This means that the powerful CPU cores are only used for a very small amount of computation, while all the remaining computations occur on the GPU. Moreover, the implementation presented in [26] is unable to overlap CPU→GPU and GPU→CPU data transfers.

Our task-based approach is more fine-grained because the CPU threads are divided into two separate groups so that CPU idling is minimized. We use only a handful of threads for boundary point computation, and the remaining CPU threads are used for computing the interior points. Because communication happens in one thread group, we are also able to completely mask intra-node CPU ⟷ GPU data transfers.

The work originally presented in [5], was further extended by Langguth et al. in [6], so that CPU+GPU computations can be executed across multiple nodes. Because the computational domain is different to the one represented in this paper, it is difficult to make a fair and direct comparision. However, we note that the implementation presented in [5] performs a manual thread to core assignment, which effectively means that OpenMP directives can no longer be used. This means that the abstraction layer that OpenMP otherwise provides, is peeled off, and as a concequence, the user is exposed to many low-level programming details.

Due to the use of OpenMP's own nested parallelism capabilities, we are able to use OpenMP's directives in our code, without the need to for example manually divide the iteration space when performing computations. This is not only easier, but that also provides better portability when moving the code from one cluster to another.

7 Conclusions

In this paper, we have presented and evaluated two CPU+GPU implementations. We have demonstrated that by letting the CPU take part in the computations, the overall solution time for a stencil application on two different GPU clusters is reduced. At the same time, we have made effective use of all the resources available. We have also introduced a simple performance model for CPU+GPU implementations, which can provide guidance for workload division.

Acknowledgments

This work was supported by the FriNatek program of the Research Council of Norway, through grant No. 214113/F20. The authors thank High Performance Computing Service at the University of Cambridge, UK, and TACC at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

Bibliography

1. Augonnet, C., S. Thibault, R. Namyst, and P.-A. Wacrenier (2011, February). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.: Pract. Exper. 23*(2), 187–198.

2. Bosilca, G., A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra (2012). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing 38*(1–2), 37–51.

3. Horton, M., S. Tomov, and J. Dongarra (2011, July). A class of hybrid LAPACK algorithms for multicore and GPU architectures. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pp. 150–158.

4. Kjolstad, F. B. and M. Snir (2010, March). Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pp. 4:1–4:9.

5. Langguth, J. and X. Cai (2014, December). Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 191–199.

6. Langguth, J., M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai (2015, July). Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. *Micro, IEEE 35*(4), 6–15.

7. Liu, Q. and W. Luk (2012, March). Heterogeneous systems for energy efficient scientific computing. In *Proceedings of the 8th International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*, Volume 7199, pp. 64–75.

8. McCalpin, J. D. (1995, December). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.

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

10. NVIDIA (2015). NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`. [Online; accessed 04-March-2015].

11. Oak Ridge Leadership Computing Facility (2015). Summit. `https://olcf.ornl.gov/summit/`. [Online; accessed 29-May-2015].

12. Phillips, E. H. and M. Fatica (2010, April). Implementing the Himeno benchmark with CUDA on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–10.

13. Rahimian, A., I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros (2010, November). Petascale Direct Numerical Simulation of Blood Flow on 200K Cores

and Heterogeneous Architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11.

14. Rivera, G. and C.-W. Tseng (2000). Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*.

15. Schäfer, A. and D. Fey (2011, June). High performance stencil code algorithms for GPGPUs. In *Proceedings of 2011 International Conference on Computational Sciences (ICCS)*, Volume 4, pp. 2027–2036.

16. Shimokawabe, T., T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka (2011, November). Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 3:1–3:11.

17. Sourouri, M., T. Gillberg, S. Baden, and X. Cai (2014, December). Effective multi-GPU communication using multiple CUDA streams and threads. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 981–986.

18. Su, H., N. Wu, M. Wen, C. Zhang, and X. Cai (2013, June). On the GPU performance of 3D stencil computations implemented in OpenCL. In *Proceedings of the 28th International Supercomputing Conference*, Volume 7905, pp. 125–135.

19. Tejedor, E., M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta (2011, June). ClusterSs: A task-based programming model for clusters. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pp. 267–268.

20. Treibig, J., G. Wellein, and G. Hager (2011, May). Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science 2*(2), 130–137.

21. Venkatasubramanian, S. and R. W. Vuduc (2009, June). Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd International Conference on Supercomputing*, pp. 244–255.

22. Wang, G. and X. Ren (2010, September). Power-efficient work distribution method for CPU-GPU heterogeneous system. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pp. 122–129.

23. Wang, H., S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda (2011, April). MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development 26*(3-4), 257–266.

24. Wang, Z., L. Zheng, Q. Chen, and M. Guo (2014, February). CPU+GPU scheduling with asymptotic profiling. *Parallel Computing 40*(2), 107–115.

25. Wu, J. and J. JaJa (2013, May). High performance FFT based poisson solver on a CPU-GPU heterogeneous platform. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 115–125.

26. Yang, C., W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng (2013, February). A peta-scalable CPU-GPU algorithm for global atmospheric simulations. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12.

27. Zhou, J., Y. Cui, E. Poyraz, D. J. Choi, and C. C. Guest (2013, June). Multi-GPU implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers. In *Proceedings of the International Conference on Computational Science*, pp. 1255–1264.

28. Zumbusch, G. (2013, June). Vectorized higher order finite difference kernels. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, pp. 343–357.

Paper III:
Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes

Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes

Johannes Langguth[1], **Mohammed Sourouri**[1,2], **Glenn T. Lines**[1], **Scott B. Baden**[3], **Xing Cai**[1,2]

[1] Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] University of California, San Diego,
La Jolla, CA 92093 USA

Manycore processors such as Graphics Processing Units (GPUs) and Xeon Phis have remarkable computational capabilities and energy efficiency, making these units an attractive alternative to conventional CPUs for general-purpose computations. The distinct advantages of manycore processors have been quickly adopted to modern heterogeneous supercomputers, where each node is equipped with manycore processors in addition to CPUs.

This thesis takes aim at developing methodologies for efficient programming of GPU clusters, from a single compute node equipped with multiple GPUs that share the same PCIe bus, to large supercomputers involving thousands of GPUs connected by a high-speed network. The former configuration represents a peek into future node architecture of GPU clusters, where each compute node will be densely populated with GPUs. For this type of configuration, intra-node communication will play a more dominant role. We present programming techniques specifically designed to handle intra-node communication between multiple GPUs more effectively. For supercomputers involving multiple nodes, we have developed an automated code generator that delivers good weak scalability on thousands of GPUs.

While GPUs are improving rapidly, they are still not general-purpose, and depend on CPUs to act as their host. Consequently, GPU clusters often feature powerful multi-core CPUs in addition to GPUs. Despite the presence of CPUs, the focal point of many GPU applications has so far been on performing computations exclusively on the GPUs, keeping CPUs sidelined. However, as CPUs continue to advance, they have become too powerful to ignore. This gives rise to heterogeneous computing where CPUs and GPUs jointly take part in the computations.

The potentially achievable performance of heterogeneous computing codes can be very large, but requires careful attention to many programming details. We explore resource-efficient programming methodologies for heterogeneous computing where the CPU is an integral part of the computations. The experiments conducted demonstrate that by careful workload-partitioning and communication orchestration, our heterogeneous computing strategy outperforms a similar GPU-only approach on structured grid and unstructured grids.

Although our work demonstrates the benefit of heterogeneous computing, the painstaking programming effort required is holding back its wider adoption. We address this issue through the development and implementation of a programming model and source-to-source compiler called Panda, which automatically parallelizes serial 3D stencil codes originally written in C to heterogeneous CPU+GPU code for execution on GPU clusters. We have used two applications to assess the performance of our framework. Experimental results show that the Panda-generated code is able to realize up to 90% of the performance of corresponding handwritten heterogeneous CPU+GPU implementations, while always outperforming the handwritten GPU-only implementations.

Compared to the more established GPU-only approach, the methodologies presented in this thesis contribute to harnessing the computational powers of GPU clusters in a more resource-efficient way that can substantially accelerate simulations. Moreover, by providing a user-friendly code generation tool, the tedious and error-prone process associated with programming GPU clusters is alleviated, so that computational scientists can concentrate on the science instead of code development.

1 Introduction

General-purpose GPUs as hardware accelerators have made their successful entrance into the high-performance computing landscape over the past few years. In a GPU-enhanced cluster, a compute node typically has significantly higher performance than a node of a homogeneous CPU-based cluster, thus resulting in a denser packing of the heterogeneous clusters. This allows significant savings in cost and power due to smaller interconnects.

Due to the large difference between a GPU and a CPU with respect to the theoretical floating-point capability, in compute-bound applications there is little incentive to include the CPUs for sharing the computational work on a GPU-enhanced cluster, since this invariably increases the complexity of the implementation. However, for computations whose performance is limited by data traffic, rather than floating-point operations, the computing capability of CPUs should not be overlooked. This is because the GPU-CPU difference in memory bandwidth is considerably smaller.

If CPUs should indeed join GPUs in the computations, three important questions will arise:

(i) How much computational work should be assigned to the CPUs? If there are different types of operations, which should be placed on the GPU?

(ii) How should the different tasks on the CPU side be programmed?

(iii) How much performance improvement can we realistically expect from heterogeneous CPU-GPU computing?

Extending our earlier work on single GPU-enhanced compute nodes [4], we will try to shed some light on the three questions for heterogeneous clusters in this paper. Of course it is impossible to answer the above three questions in general. The answers depend on the specific computational problem to be solved and the hardware configuration of a heterogeneous system. By choosing a representative case of solving the diffusion equation using the cell-centered finite volume method over unstructured meshes, we aim to provide our advice on good programming practices, as well as some important OpenMP and CUDA programming details that carry over to many similar problems.

Moreover, we will also discuss the important issue of CPU-GPU workload partitioning. Last but not least, we report performance measurements on up to 128 GPU-enhanced compute nodes, demonstrating the actual performance benefit due to heterogeneous CPU-GPU computing.

For structured meshes, heterogeneous CPU-GPU computation has been studied in several publications, e.g. [3, 10, 13, 14]. However, the unstructured nature of our problem poses significant additional challenges with respect to partitioning, communication and load balancing.

2 Solving diffusion equations with finite volumes

As a representative computational problem, we use the following *diffusion equation* that describes a very common phenomenon in nature:

$$\frac{\partial u(x,t)}{\partial t} = \mathrm{div}(\vec{K}(x)\mathrm{grad}\ u), \tag{1}$$

where $u(x,t)$ is typically some concentration modeled as a function of space and time, and $\vec{K}(x)$ denotes a spatially varying tensor field that, together with the concentration gradient, determines the speed and direction in which high concentration spreads towards low concentration. Being one of the basic building blocks of many sophisticated mathematical models, the diffusion equation (1) is an important research topic for fast numerical solvers and efficient software implementations.

In this paper, we will consider a finite-volume approach for numerically solving (1) in 3D, using an unstructured tetrahedral mesh.

Without going into the mathematical details, it suffices to say that the actual computation per time step can be represented by a matrix-vector multiply:

$$\boldsymbol{u}^{\ell} = Z\boldsymbol{u}^{\ell-1}, \tag{2}$$

where superscript ℓ denotes the time level, the $\boldsymbol{u}$ vector contains the numerical approximations at the center of all tetrahedra. Matrix Z is sparse and has, in addition to a nonzero main diagonal, up to 16 nonzero values per row. These 16 off-diagonal nonzeros correspond to each tetrahedron's four immediate neighbors and 12 second-level neighbors.

Throughout this paper, we assume that the main diagonal of Z will be stored in a separate 1D array D, whereas the off-diagonal entries are stored in a padded, dense $N \times 16$ array A with N being the total number of tetrahedra in the mesh. In an unstructured tetrahedral mesh, the column positions of these off-diagonal entries do not follow any easily predictable pattern. Thus, they have to be stored separately in I, a $N \times 16$ array of integer index values.

A plain implementation of (2) is given in the code segment below:

```
for (i=0; i<N; i++) {
  double value = D[i]*u_old[i];
  for (j=0; j<16; j++)
    value += A[i,j]*u_old[I[i,j]];
  u_new[i] = value;
}
```

To calculate u_new for each tetrahedron, 33 floating-point operations (17 multiplications and 16 additions) are needed in every time step.

At least 208 bytes per tetrahedron need to be read from memory (i.e., 128 bytes for the 16 A[i,j] values, 64 bytes for the 16 I[i,j] values, 8 bytes for D[i] and 8 bytes for u_old[i]). In case cache data reuse is not perfect, more data must be loaded, depending on the access pattern of the off-diagonal u_old values. Moreover, 8 bytes (due to u_new[i]) are written to memory per tetrahedron. Thus, computational intensity is at most 33 FLOP per 216 bytes and thus 0.15. This is far lower than the ratio between FLOPS and memory bandwidth in GB/s of modern processors, which starts at 2 and can be higher than 5 for GPUs. Therefore, the theoretical upper limit of the performance for this computation on any compute device is:

$$P = \frac{33\text{ FLOP} \times \text{memory bandwidth}}{216\text{ bytes}}. \tag{3}$$

3 Partitioning and Problem Setup

3.1 Hierarchical Partitioning

During each time step of the computation, using asynchronous MPI communications, node n receives updated values for these ghost cells from neighboring nodes and sends updated values of its separator cells in return. The elements to be sent to a specific node must be packed into a contiguous buffer during every round. The ghost cells are organized such that their values can be received in a contiguous fashion.

In order to use heterogeneous computing, a second tier of partitioning is required to split the CPU part from the accelerator part on each node.

Here, an asymmetric partitioning is needed since the GPUs will generally receive a higher workload than the CPUs. On each node, we generate one subpartition per GPU, plus one CPU subpartition. In the case of multiple CPU sockets, the CPUs can work on a shared subpartition using OpenMP, although care must be taken to obtain their full performance [6]. Note that although our test systems nodes have at most two GPUs and CPUs, our code can deal with almost any configuration as long as enough CPU threads are available.

Thus, the global mesh is broken into k parts, each of which is subdivided into the MPI separator, a CPU-GPU separator, a CPU interior part, and for each GPU a GPU-CPU separator and a GPU interior part, as illustrated in Figure 1. All the resulting separators are packed together in order to allow contiguous access for computation and communication. Note that there is no explicit GPU-GPU separator since this communication is performed by transferring data via the CPU.

While there exist several techniques for direct communication between accelerators, e.g. GPUdirect [8], we do not make use of them here because they tend to be very hardware specific and thus offer little portability.

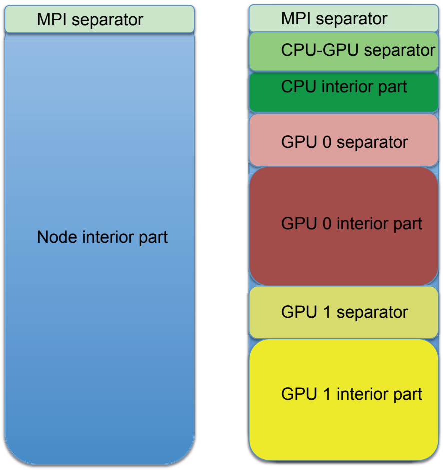


Figure 1: Left: Workload per compute node after the initial symmetric partitioning. The cells have been permuted such that the MPI separator forms a contiguous block. Right: Division of the workload after the intra-node partitioning and appropriate permutations.

3.2 Implementation Fundamentals

The CPU part of a compute node, which can comprise several physical sockets working on shared memory, handles all communication between the GPUs as well as the MPI communication with other nodes using a single MPI process. The advantage of this technique is that the entire complexity of running the heterogeneous computation is encapsulated in the intra-node code. Thus, existing inter-node communication schemes can be reused when transforming conventional codes into heterogeneous implementations. In our case, inter-node communication is handled by a simple set of `MPI_Isend` and `MPI_Irecv` instructions. In addition to its simplicity, this approach has the advantage of keeping the number of MPI ranks, and thus the total size of the separators low, which ensures that communication is unlikely to become a bottleneck.

Finally, to ensure good cache data reuse for the off-diagonal `u_old` values, we use the partitioner for a third time to reorder the tetrahedra in the interior computation parts. The goal is to create blocks of tetrahedra that have as many neighbors as possible within each block, and thus as few neighbors as possible outside the block. Doing so regularizes accesses to `u_old[i]`, which has a dramatic effect on performance, as discussed in [5, 6]. We obtained good performance for a blocksize of 512 tetrahedra on the CPU and 64 on the GPU. Note that only the tetrahedra in the interior computation part are reordered in this way. Reordering the tetrahedra in the separators is not worthwhile, since they invariably access neighbors outside the current device - and thus outside their block.

Our GPU kernel processes elements of A in a column-major ordering, as suggested in [12]. This means that for every thread block of size b, every b contiguous rows are turned into a $b \times 16$ submatrix and transposed. This allows coalesced accesses to values of A and I, i.e. threads in a thread block access the elements in a contiguous manner, thus attaining full memory bandwidth. We found that a thread block size $b = 128$ yielded the best performance, as smaller sizes limit the device occupancy. Tetrahedra beyond the last block of size b are computed using a row-major kernel. Due to their small number, this has a negligible effect on performance.

The core of our heterogeneous code is the assignment of different tasks to different hardware CPU threads. In our implementation, this is done by directly assigning a type to a thread based on its OpenMP thread number. We use one *control thread* per accelerator. Each such control thread starts the computation of the separator on its accelerator, copies the result asynchronously to the CPU, and starts the computation on the interior part. Meanwhile, all the remaining threads work on the MPI separator elements. When this is done, a single thread diverges and communicates the `u_new` values belonging to the MPI separator to the neighboring nodes via MPI, while receiving corresponding values in return. In our experiments, using more than one *MPI communication thread* did not pay off.

The remaining threads then compute the CPU-GPU separator, and upon completion one *copy thread* per accelerator diverges in order to start copying the result to its accelerator, while the remaining threads compute the interior CPU part which means they are pure *compute threads*. Each copy thread also transfers `u_new` values belonging to the separators from other accelerators to its own accelerator once they have been transferred to the CPU memory. Since these transfers are asynchronous, the copy thread can then rejoin the compute threads working on the interior CPU part. When all these tasks have been executed, the threads are gathered at a barrier. The array pointers of `u_old` and `u_new` are then swapped on all devices, and a new timestep begins.

Note that we only use physical cores to run the threads. Hyperthreading and similar techniques may make the threads less responsive, and thereby can reduce performance. Thus, for a given number of accelerators, an equal number of `control` and `copy threads` must be available in addition to the compute threads. If too few compute threads remain, the CPU performance will be low, which invalidates the entire approach. As a rule of thumb, the total number of cores should be at least four times the number of accelerators. An overview of the threads for a typical test node is shown in Figure 2.

In addition to this, the GPU control threads (*id* 14 and 15 in Figure 2) and the GPU communication threads (*id* 1 and 2 in Figure 2) use multiple CUDA streams to overlap communication and computation on the GPU. Thanks to its two copy engines, a modern GPU such as the K20 can send its separator while receiving the CPU-GPU separator from its copy thread at the same time, as indicated in Figure 2. An example of streams that overlap communication with computation can be found in Figure 3. It is derived from the output of the *nvprof* GPU profiler, although details have been modified for visibility, e.g. the computation of the interior tetrahedra takes far longer time than all other operations combined, but has been shortened here. The interior computation cannot be overlapped with the separator computation since both use the same compute resources, but it does overlap with communication. While the kernel launches are relatively fast compared

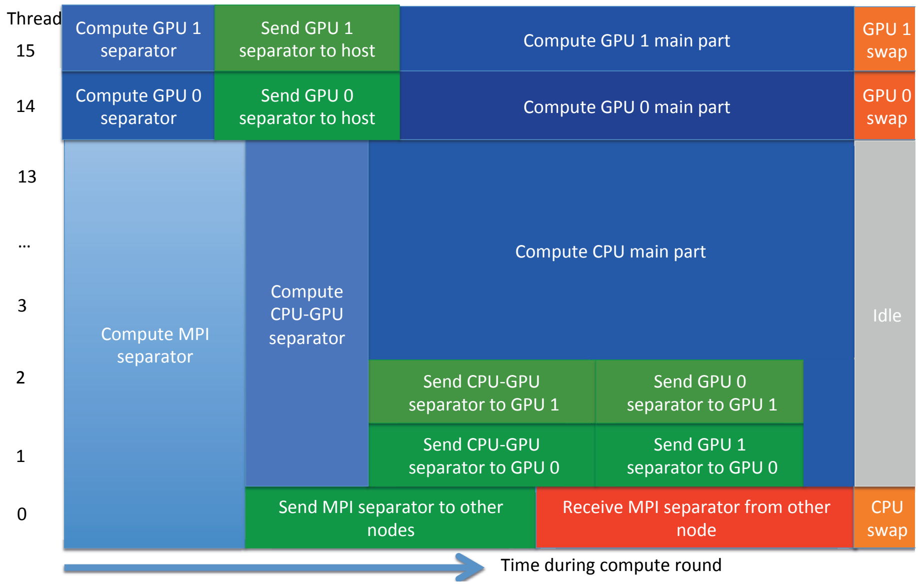


Figure 2: Example of the task parallel thread assignment using 16 cores and 2 GPUs. Threads 14 and 15 serve as *control threads* for the GPUs, and threads 1 and 2 as their *copy thread*. Threads 3 through 13 are *compute threads* and perform only computation, and thread 0 is the *MPI communication thread*.

with the kernel running time, initializing CPU - GPU data transfer incurs a significant overhead in the calling thread, even though the transfer itself is very fast. We also observe gaps due to synchronization, i.e. periods in which no computation takes place. These synchronization costs represent a significant challenge for achieving high performance.

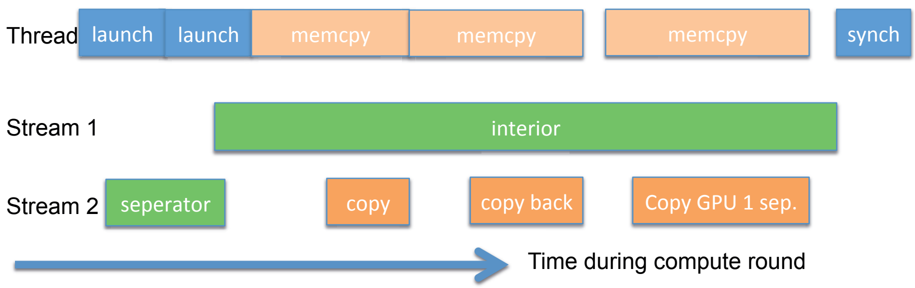


Figure 3: Example of using multiple streams to overlap communication and computation. Instructions issued in the CPU thread are assigned to different GPU streams to overlap communication and computation on the GPU.

4 Experimental Setup

All experimental instances are derived from a 3D mesh of a healthy male human cardiac geometry acquired by MRI. We employ *tetgen* [11] to generate the initial global mesh. For our experiments, we set a target resolution via a maximum volume constraint per tetrahedron of $2.8 * 10^{-6}$, thereby generating more than 115 million tetrahedra. This test instance is large enough for a moderate number of compute nodes. However, we need at least five GPUs to store the partitioned data in device memory at this instance size. To obtain measurements on smaller node counts, we created a second instance of 6.8 million tetrahedra, which can be run using a single GPU. All experiments using fewer than 8 GPUs are run using this smaller instance instead.

The PaToH [1, 2] and Kaffpa [9] partitioning software are then used to generate the initial k-way partitioning of the global mesh. Kaffpa generally takes less time to partition than PaToH, and produces better quality (if high quality setting is used). Since Kaffpa is currently only able to generate symmetric partitions, we use PaToH for the intra-node partitioning and then Kaffpa again for the reordering.

Finally, to maintain generality, we do not exploit the effects of fitting the entire CPU workload in the L3 cache. As discussed in [4], small CPU workloads can lead to very high CPU performance when all required data fits in cache. Thus, in case of an extremely small CPU workload, it is worthwhile to expand it up to the maximum cacheable size, thereby speeding up the overall computation. While we do not make use of this, in the future, this effect guarantees that the CPU can remain useful for memory bound computations even when a large number of very fast accelerators are available. Another potential way to benefit from large CPU caches might be to explicitly load the MPI separators into cache in order to compute them quickly at the start of each round.

As test hardware systems, we use two heterogeneous machines having slightly different characteristics, which lead to different ratios between CPU and GPU workload in these systems. This is important for benchmarking our heterogeneous code.

As our primary test system we use the GPU part of TACC's *Stampede*. It is primarily a Xeon Phi machine, but we use its GPU partition for our experiments. Each of its 128 GPU nodes posses a single NVIDIA K20 GPU and two powerful Intel Xeon E5-2680 processors with 8 cores each, which gives it a strong CPU to GPU performance ratio.

As the secondary machine we use the *Wilkes* system operated by the University of Cambridge. We use up to 64 nodes on *Wilkes*, and each node has two CPUs and two NVIDIA K20 GPUs, only one of which can be accessed at full PCIE bus speed from a given CPU. Thus, each CPU has one preferred GPU, while the second GPU is accessed through the other CPU on the node. The CPUs are Intel Xeon E5-2630v2, i.e. Ivy bridge processors, which are very similar to the Sandy Bridge processors used in *Stampede*. However, they have only 6 cores each and lower attainable memory bandwidth, which reduces the system's CPU to GPU performance ratio. On both machines we use the Intel *icc* compiler 13.1.0, Intel MPI 4.1.3.049, and CUDA 6.0. Hyperthreading is deactivated in all instances, and one OpenMP thread per core is used. OpenMP thread affinity is set to "scatter". We use up to 64 nodes on *Wilkes*.

5 Experimental Results

5.1 Single Device Computation Performance Test

A crucial ingredient of our heterogeneous implementation is the static workload ratio, which is obtained using performance predictions that are based on the memory bandwidth of the compute devices. For any device, its workload ratio is obtained by dividing its predicted performance P by the sum of the predictions for all devices. For convenience, we denote the total GPU workload ratio as r, which means the CPU workload ratio will be $1-r$ and each individual GPU will have a workload of r divided by the number of GPUs.

Now, given the peak memory bandwidth provided by the vendors and the fact that the maximum flop to byte ratio is 0.157, we obtain P_{peak}, i.e. the predicted performance based on these values and thus the appropriate workload ratio r_{peak}. Table 1 shows the results. We compare this to the actual measured performance P_{real}, and the resulting optimal workload ratio r_{opt}. We can use P_{real} to obtain an upper limit on the performance of the heterogeneous code by multiplying the corresponding P_{real} values with the number of compute devices used.

The discrepancy between P_{peak} and P_{real} is significant, which implies that might not be a good performance prediction. We improve it by using P_{stream}, which is the performance estimate based on bandwidth measured using the STREAM benchmark [7]. Table 1 clearly shows that P_{stream} is a much better prediction for P_{real} and r_{stream} is closer to r_{opt}.

	Stampede	Wilkes
CPU peak bandwidth	102.4	102.4
CPU stream bandwidth	77.8	72.9
CPU P_{peak}	16.11	16.11
CPU P_{stream}	12.24	11.47
CPU P_{real}	11.46	8.78
GPU peak bandwidth	208	2×208
GPU stream bandwidth	151.1	2×151.1
GPU P_{peak}	32.74	2×32.74
GPU P_{stream}	23.78	2×23.78
GPU P_{real}	21.46	2×21.46
r_{peak}	0.67	0.80
r_{stream}	0.66	0.80
r_{opt}	0.65	0.83

Table 1: Computational performance estimates (P_{peak} and P_{stream}) and measurements (P_{real}) of a single device in each of the test systems (in GFLOPs). The r values denote workload partitioning ratios computed on that basis. Unlike the GPU peak bandwidth, the GPU stream bandwidth is based on activated ECC.

Interestingly, the difference between the workload ratios is small in most cases. However, overestimating CPU performance even by a small amount has a comparatively large impact on the overall performance when the CPU contribution is small. See [4] for more details on this effect. For example, on *Wilkes*, the fact that both P_{peak} and P_{stream} overestimate the CPU performance leads to CPU workloads that are 17% higher than optimal, i.e. from 0.17 to 0.2. This could in turn lead to roughly 17% higher execution time and thus 15% lower performance. We thus conclude benchmarking the actual performance P_{real} to obtain r_{opt} can pay off and we use it in this study, but it might not be worthwhile in practice. In general, it is advisable to reduce the CPU workload a bit since erring in the direction of high CPU workloads is much more costly than vice versa.

5.2 Homogeneous Node Scaling Experiment

In the previous experiments we obtained a theoretical upper bound on performance. Now we bound it from below by running the full communication and computation on the test systems, but use only CPUs or GPUs, thereby establishing the maximum performance attainable without using heterogeneous computing. This is necessary to assess the performance gain - and thus the potential payoff in using heterogeneous CPU-GPU computation. Figure 4 shows the attained performance on both *Stampede* and *Wilkes*.

These results include MPI communication, and are thus significantly lower per node than the P_{real} values from Table 1 would indicate. Summing up the CPU and GPU values gives us an estimate for the performance upper limit we can expect from heterogeneous computing. Interestingly, despite having the same GPUs and using only one GPU per node in on both machines, we observe a noticeably lower GPU performance on *Wilkes*.

5.3 Heterogeneous Node Scaling Experiment

In this subsection we establish the performance gain obtained from using heterogeneous computation. Figure 5 shows the results for our heterogeneous implementation and compares the attained performance to using only GPUs, and to an instance of the heterogeneous code where all communication is disabled. On *Stampede*, the difference between the heterogeneous and the pure GPU results is quite pronounced, which validates the usefulness of our technique. Furthermore, the communication-free performance is only slightly higher, which indicates that communication is overlapped with computation to a large extent. The speedup for 128 nodes is 98.7.

For *Wilkes*, the *GPU only* value is obtained by running the pure GPU code with two MPI processes on each node. The process placement is such that it matches each process with its preferred GPU, thus optimizing communication performance. The more complex node layout, along with the fact that the CPUs are weaker on this machine, reduce the performance lead of the heterogeneous code. The speedup is 27.7 for 32 and 42.7 for 64 nodes. Also, for 64 nodes, the heterogeneous performance is actually lower than the pure GPU result, while the communication-free performance is significantly higher. This indicates that in this setup, intra-node communication is in fact a bottleneck.

We assume that this is due to limitations in strong scaling, i.e. workloads per compute device become so small that communication becomes an issue in this case. In addition, the CPUs on *Wilkes* essentially have a NUMA access to the GPUs, which our assignment of control threads does not take into account. Furthermore, our one MPI thread communi-

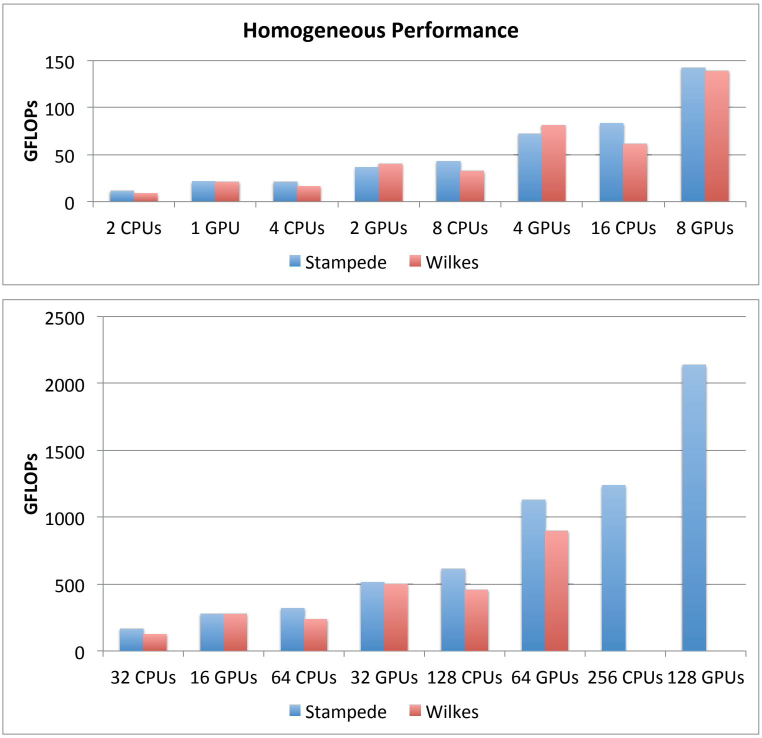


Figure 4: Performance obtained using only CPUs or GPUs. Note that CPUs refers to the number of sockets, not cores.

cation model is not ideally suited to make full use of the two InfiniBand adapter cards per node. This suggests that for very complex compute nodes, the scheme needs to be adapted to obtain full communication performance.

When considering parallel efficiency, which can be derived by multiplying the single-device performance results from Table 1 with the number of devices involved, we find that the value remains quite stable at about 80%. However, it starts to drop when at least 64 GPUs are involved. We assume that this is the general limit for strong scaling at this instance size since the communication-free results show a similar behavior which indicates that on *Stampede* the principal limitations to strong scaling are caused by load-balancing and synchronization issues, rather than communication overhead.

6 Conclusions

We have developed a high performance cell-centered finite volume code for 3D unstructured tetrahedral meshes aimed at exploiting all computational resources on GPU-enhanced clusters. Using both MPI and OpenMP, we have obtained fine grained control

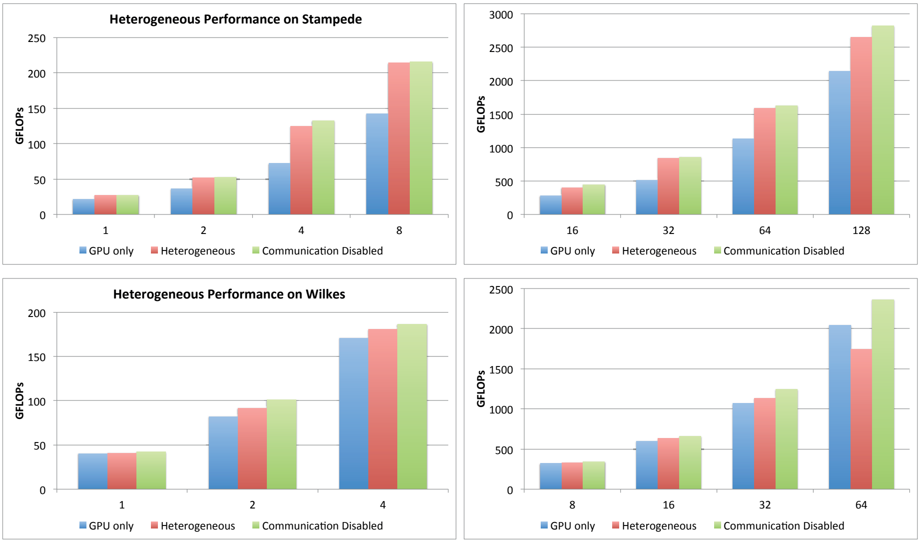


Figure 5: Performance of the heterogeneous simulation on *Stampede* (top row) and *Wilkes* (bottom row) for varying number of nodes and in relation to GPU-only and communication-free performance.

over the inter- and intra-node communication, thereby achieving a high degree of communication/computation overlap. The resulting MPI communication pattern is a traditional symmetric structure using one MPI process per node, while OpenMP is used in a task parallel manner.

Our experiments on *Stampede* show that the strong scalability is very good when using up to 32 GPUs. At 128 nodes, we still attain 95% of the communication-free upper bound. Efficient communication is a concern on the complex nodes of *Wilkes* though.

Although the chosen diffusion equation and the explicit finite-volume numerical strategy are simple, the obtained experiences with hierarchical mesh partitioning, CPU-GPU workload division and OpenMP/CUDA programming readily extend to more advanced real-world applications. One possible direction of future work is to apply our findings to the monodomain model of computational electrocardiology, which consists of the diffusion equation and a set of ordinary differential equations that describe the electrical behavior of cardiac cells.

Even though we have focused on a single application, the programming techniques described in this paper are not specific to it. Assuming that a static load balancing is suitable for the problem, and interior cells outnumber separator cells by a significant factor, the techniques described here can be used to efficiently incorporate CPU and GPU operations on many kinds of mesh-based computations.

Acknowledgements

We would like to acknowledge Prof. Julius Guccione at UCSF for providing segmented surfaces from cMRI images, which we use as our test cases. We would like to thank Filippo Spiga at University of Cambridge for his support in running experiments on *Wilkes*. This work used the Wilkes GPU cluster at the University of Cambridge High Performance Computing Service, provided by Dell Inc., NVIDIA and Mellanox, and part funded by STFC with industrial sponsorship from Rolls Royce and Mitsubishi Heavy Industries. Finally, we acknowledge John Cazes' help in running experiments on *Stampede* that include all 128 GPU nodes. Scott Baden dedicates his portion of this work to Ingela Brising (1943-2015).

Bibliography

1. Çatalyürek, U. V. and C. Aykanat (1995, December). A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. In *Proceedings of International Conference on High Performance Computing*.

2. Çatalyürek, U. V. and C. Aykanat (2001, April). A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS)*.

3. Humphrey, A., Q. Meng, M. Berzins, and T. Harman (2012). Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*, pp. 4:1–4:8.

4. Langguth, J. and X. Cai (2014, December). Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 191–199.

5. Langguth, J., N. Wu, J. Chai, and X. Cai (2013, November). On the GPU performance of cell-centered finite volume method over unstructured tetrahedral meshes. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, pp. 7:1–7:8.

6. Langguth, J., N. Wu, J. Chai, and X. Cai (2015). Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes. *Journal of Parallel and Distributed Computing 76*(0), 120–131.

7. McCalpin, J. D. (1995, December). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.

8. NVIDIA (2015). NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`. [Online; accessed 04-March-2015].

9. Sanders, P. and C. Schulz (2013). Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Volume 7933, pp. 164–175.

10. Shimokawabe, T., T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka (2011, November). Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 3:1–3:11.

11. Si, H. (2007). TetGen. a quality tetrahedral mesh generator and three-dimensional delaunay triangulator. `http://tetgen.berlios.de`. [Online].

12. Vázquez, F., G. Ortega, J. Fernández, and E. Garzón (2010, June). Improving the performance of the sparse matrix vector product with GPUs. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 1146–1151.

13. Wen, M., H. Su, W. Wei, N. Wu, X. Cai, and C. Zhang (2014). High efficient sedimentary basin simulations on hybrid CPU-GPU clusters. *Cluster Computing 17*(2), 359–369.

14. Yue, X., S. Shu, and C. Feng (2013, July). UA-AMG methods for 2-D 1-T radiation diffusion equations and their CPU-GPU implementations. In *2013 21st International Conference on Nuclear Engineering*, pp. V005T14A015–V005T14A015.

Paper IV: Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers

Appendix I: Computational Resources

Computational Resources

1 Dirac

Dirac was a small experimental GPU testbed at the National Energy Research Scientific Computing Center (NERSC). Each of the 44 compute nodes were equipped with two 8-core Intel Xeon 5520 (Nehalem) CPUs, 24GB DDR3 memory, and four Nvidia Tesla C2050 (Fermi) GPUs. Moreover, all of Dirac's compute nodes were connected by 4x QDR Infiniband technology configured in a fat tree topology with a global 2D mesh. Dirac was retired in December 2014.

Source code on Dirac was compiled using CUDA [3] version 5.0 and OpenMPI [4] version 1.6.5. The following compiler flags were used for CUDA's *nvcc* compiler: `-arch=sm_20 -m64 -O3`

2 Stampede

TACC Stampede [5] is primarily an Intel Xeon Phi cluster consisting of 6400 compute nodes. However, a small fraction of the nodes are equipped Tesla K20m GPUs instead (one per node). While 128 GPU nodes are available, they have been partitioned in such a way that it is not possible to access more than 32 GPU nodes at a time. The GPU compute nodes are equipped with two 8-core Intel Xeon E5-2680 (Sandy Bridge-EP) CPU, 32 GB of memory and a single Nvidia Tesla K20m (Kepler) GPU. All Stampede nodes communicate via a Mellanox FDR Infiniband interconnect configured in a fat-tree topology.

GPU-only source code was compiled using CUDA [3] version 6.0 and MVAPICH2 [2] 1.9. The heterogeneous CPU+GPU source code was compiled using CUDA version 2.0, Intel IMPI [1] version 4.1.3.049, and Intel C compiler version 13.0.2.146.

The following compiler flags were used for CUDA's *nvcc* compiler: `-arch=sm_35 -m64 -O3`, while the following flags were used for Intel's *icpc* compiler: `-O3 -openmp -xHOST -fomit-frame-pointer -fno-alias -ip`

3 Wilkes

Wilkes [6] is a GPU cluster at the University of Cambridge, UK. The cluster consists of 128 nodes, where each node is equipped with two Tesla K20c (Kepler) GPUs. Moreover, each compute node is equipped with two 6-core Intel Xeon E5-2630v2 (Ivy Bridge-EP) CPU, 64GB of RAM and two Mellanox FDR Infiniband adapters. Due to several non-operational nodes, we were unable to use all of the 128 nodes.

All GPU-only source code was compiled using CUDA [3] version 6.0 and MVAPICH2 [2] 2.0. The heterogeneous CPU+GPU source code was compiled using CUDA version 2.0, Intel IMPI [1] version 4.1.3.049, and Intel C compiler version 13.0.2.146.

The following compiler flags were used for CUDA's *nvcc* compiler: `-arch=sm_35 -m64 -O3`, while the following flags were used for Intel's *icpc* compiler: `-O3 -openmp -xAVX -fomit-frame-pointer -fno-alias -ip`

4 Titan

Titan is a Cray XK7 system located at Oak Ridge National Laboratory, and currently ranked the second fastest supercomputer on the TOP500 list. In total, Titan consist of 18 688 compute nodes and a theoretical peak performance of 27 Petaflops. Each Titan node consists of a single 16-core AMD Opteron 6274 (Interlagos) CPU, 32GB of host memory and a single Nvidia Tesla K20X (Kepler) GPU. The compute nodes are connected via Gemini interconnect, configured in a 3D torus topology.

All source code on Titan was compiled using the default Cray Compiling Environment in combination with the provided CC wrapper. The CUDA version used was version 6.5.

The following compiler flags were used for CUDA's *nvcc* compiler: `-arch=sm_35 -m64 -O3`

Bibliography

1. Intel (2015). Intel MPI Library. https://software.intel.com/en-us/intel-mpi-library. [Online; accessed 26-September-2015].

2. MVAPICH (2015). MPI over InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). http://mvapich.cse.ohio-state.edu. [Online; accessed 25-September-2015].

3. NVIDIA (2015). CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Online; accessed 25-May-2015].

4. OpenMPI (2015). Open Source High Performance Computing. http://http://www.open-mpi.org. [Online; accessed 25-September-2015].

5. Texas Advanced Computing Center Stampede (2015). https://www.tacc.utexas.edu/stampede/. [Online; accessed 12-February-2015].

6. University of Cambridge - HPC Service - The Wilkes Cluster (2015). http://www.hpc.cam.ac.uk/services/wilkes.html. [Online; accessed 12-February-2015].