Masteroppgave

Aiming at Heterogeneous Parallel Computing for Bioinformatics: To Overlap or Not to Overlap?

Vidar Lunde Olaisen

Computational Science

60 studiepoeng

Institutt for informatikk

Det matematisk-naturvitenskapelige fakultet

Våren 2024

Vidar Lunde Olaisen

Aiming at Heterogeneous Parallel Computing for Bioinformatics: To Overlap or Not to Overlap?

Veiledere:

Xing Cai and Johannes Langguth

Contents

1 Abstract **6**

2 Introduction **7**

3 Harnessing the Power of Heterogeneous Computing **9**

3.1 Central Processing Unit(CPU) 11

3.2 Graphics Processing Unit(GPU) 12

3.3 Memory Hierarchy and Data Communication 13

4 Data Transfer **16**

4.1 Latency, Bandwidth and Throughput 17

4.1.1 Latency 17

4.1.2 Bandwidth and Throughput 18

4.2 Transfer Rate 19

4.3 Methods for Communication 19

4.3.1 Point-to-Point 20

4.3.2 Direct Memory Access 20

4.4 Data Transfer Technologies 21

4.4.1 Peripheral Component Interconnect Express(PCIe) 21

4.4.2 NVLink 23

4.4.3 NVSwitch 25

4.4.4 Message Passing Interface(MPI) 26

4.4.5 GPUDirect 27

4.5 Overlapping Communication and Computation 29

4.6 CPU-free execution path 31

5 Compute Unified Device Architecture(CUDA) 33

5.1 Programming Model 33

5.1.1 Threads 34

5.1.2 Blocks 35

5.1.3 Grid 35

5.2 Computing on CUDA 36

5.3 Cooperative Groups 39

5.3.1 Grouping Types 40

5.4 CUDA Stream 43

5.5 Asynchronous Operations 45

6 Implementation 48

6.1 Partitioning Computation 61

6.2 CPU to CPU on Different Nodes 66

6.3 GPU to GPU Within a Node 69

6.4 GPU to GPU on Different Nodes 73

6.4.1 Communicating Through CPU 74

6.4.2 Direct GPU Communication 79

7 Results and Discussion **83**

7.1 Profiling 84

7.2 Heatmaps 86

7.2.1 Expected results 87

7.3 CPU to CPU on Different Nodes Results 88

7.4 GPU to GPU Within a Node Results 90

7.5 GPU to GPU on Different Nodes Results 93

7.5.1 Communicating through CPU memory 93

7.5.2 Through RDMA 95

8 Conclusion **97**

8.1 Limitations and Problems 98

8.2 Future Improvements 98

List of Abbreviations

Acronyms

Bps Bytes per second. 18

CPU Central Processing Unit. 9

CUDA Compute Unified Device Architecture. 15

DDR Double Data Rate. 14

DMA Direct Memory Access. 20

GDDR Graphical Double Data Rate. 14

GPU Graphics Processing Unit. 9

HBM High Bandwidth memory. 14

HPC High Performance Computing. 12

IPU Intelligence Processing Unit. 9

MPI Message Passing Interface. 26

P2P Peer to Peer. 28

PCIe Peripheral Component Interconnect Express. 14

PtP Point to Point. 19, 20

RDMA Remote Direct Memory Access. 28

SIMD Single Instruction, Multiple Data. 34

SIMT Single Instruction, Multiple Threads. 34

VRAM Video Random Access Memory. 14

Listings

1 Thread groups . . . 36
2 C example . . . 37
3 CUDA example . . . 38
4 CUDA without Cooperative Groups . . . 42
5 CUDA with Cooperative Groups . . . 42
6 CUDA Stream . . . 44
7 Single Core CPU implementation . . . 51
8 Dividing 3d Grid For Nodes . . . 54
9 Dividing Subgrids For GPUs . . . 55
10 Allocating GPUs Subgrid . . . 56
11 GPU Computation Implementation . . . 57
12 MPI Communication example . . . 59
13 GPUDirect Communication . . . 60
14 inter-node GPU Communication . . . 61
15 Inter-node CPU Computation Partition . . . 62
16 GPU Computation Partition . . . 64
17 Inter-node CPU Communication . . . 67
18 Inter-node CPU Communication With Overlap . . . 68
19 Intra-node GPU Communication . . . 70
20 Intra-node GPU Communication With Overlap . . . 72
21 Multi-node Communication . . . 75
22 Node GPUs overlap Example . . . 78
23 Node GPUs Cuda-aware MPI Example . . . 80
24 Node GPUs CUDA-aware MPI overlap Example . . . 82

1 Abstract

In the realm of high-performance computing (HPC), improving existing code can take many forms. This thesis explores the implementation of overlapping communication with computation in heterogeneous parallel computing environments, focusing on scenarios involving both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). By analyzing various configurations and workloads, the research demonstrates that performance gains from overlapping can range between 0 and 50 percent, with higher values when computation and communication times are balanced. The findings provide valuable insights for enhancing the efficiency of parallel applications on heterogeneous computing platforms, offering a pathway to more effective utilization of computational resources in scientific and engineering domains.

2 Introduction

In the era of advancing technology, high-performance computing (HPC) has become more and more important for tackling complex scientific and engineering problems. HPC systems utilizes the power of parallel processing to perform large-scale computations more efficiently than traditional computing systems. Among the various architectures used in HPC, heterogeneous computing has emerged as a prominent approach, combining different types of processors, such as Central Processing Units and Graphic Processing Units, to optimize performance for diverse workloads.

The motivation for this thesis stems from the need to address the growing computational demands in scientific research and industry applications. Modern scientific simulations, data analytics, and machine learning tasks often involve massive datasets and intricate computations that require substantial computational resources. Heterogeneous computing platforms, particularly those integrating GPUs, have shown remarkable potential in accelerating these tasks due to their parallel processing capabilities.

Despite the significant advantages of heterogeneous computing, one of the challenges lies in managing the communication between different processors. The performance of parallel applications can potentially be severely hindered by the overhead associated with data transfer between CPUs and GPUs. Overlapping communication with computation has been proposed as a strategy to mitigate this bottleneck, allowing data transfer to occur concurrently with computational tasks. However, the effectiveness of this approach is highly dependent on the specific characteristics of the computational workload and the data transfer patterns.

This thesis aims to explore the impact of overlapping communication with computation in heterogeneous parallel computing environments. By implementing and analyzing various configurations on a state-of-the-art supercomputing infrastructure, this research seeks to provide a comprehensive understanding of the performance benefits and limitations of this approach. The primary objectives of the study are to:

1. Evaluate the performance improvements achieved through overlapping communication with computation for different grid sizes and processor configurations.

2. Identify the optimal balance between computation and communication to maximize the utilization of heterogeneous computing resources.

3. Investigate the potential drawbacks and challenges associated with implementing overlap, particularly in scenarios with varying grid sizes and data transfer requirements.

The findings from this research will contribute to the broader knowledge of HPC optimization techniques, offering valuable insights for developers and researchers aiming to enhance the performance of parallel applications on heterogeneous platforms. Through a detailed analysis of profiling data and experimental results, this thesis will elucidate the conditions under which overlapping communication with computation is most effective, ultimately guiding future efforts in the design and implementation of high-performance computing systems.

3 Harnessing the Power of Heterogeneous Computing

To see the potential benefits of overlapping computation and communication one first has to select the platform to work on. In this thesis the selection of heterogeneous architectures over homogeneous architectures comes from the situation where one has to communicate between the different processors types it contains to utilize each one. This is compared to homogeneous architectures where usually most of the intra-node data transfers is being sent between Central Processing Unit (CPU) and its off-chip memory. The development of improving the processor types has also been going generally faster when it comes to computational power compared to the development of data transfer improvements, leading to the "memory wall" concept.[1] The combination of heterogeneous architecture's many different components which requires communication to work and the memory wall which explains that communication can become a bigger part of the total computation time[1] provides us with a good framework with exploring the potential benefits of introducing overlap which will hopefully improve these potential problems by hiding the communication behind computations.

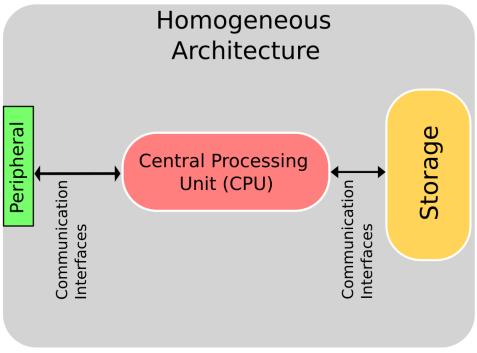

Figure 1: Homogeneous Architecture

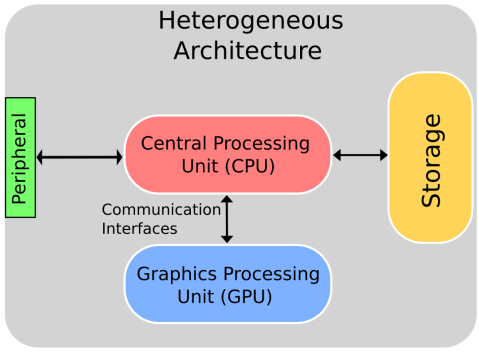

Figure 2: Heterogeneous Architecture

A simplistic example of the difference between 2 architectures

In modern computing, the concept of heterogeneous architecture has emerged as a formidable strategy. It capitalizes on the synergy between various processing units, such as CPU[2], Graphics Processing Unit (GPU)[3] or even Intelligence Processing Unit (IPU)[4], all integrated within a single computing system. In contrast to the

traditional homogeneous architecture `fig. 1`, where identical processing units labor in unison, heterogeneous architectures `fig. 2` unite different processor types, each designed to excel at specific tasks. By utilizing an architecture which combines these different specialized processors we can compute several different types of programs and then selectively decide which processor does what task[5].

Heterogeneous architectures work best when entrusted with the acceleration of specific computations[6]. This implies that one wants to assign each specific task to be done at the most adept processing units for the situation. Consider for example the GPU, a master in parallel tasks like graphics rendering and machine learning training. Its multitude of cores is tailor-made for simultaneous computations in parallel. When coupled with the traditional CPU, known for its versatility in general-purpose computing, an example of heterogeneous architecture is created, harnessing the strengths of each component to gain peak performance[6].

This architecture type, while solving many potential problems, does however induce its fair share of challenges that demand additional attention and resolution for optimal performance[7]. Some of these challenges include:

Firstly, in the context of heterogeneous architectures, one prominent challenge lies in managing communication among the multitude of processors effectively[7]. As the number of processors increases, so does the complexity of data exchange and communication pathways. Each additional processor introduces the need to transfer data, whether it's a one-time transfer or a continuous stream. This heightened communication demand can lead to varying degrees of bottlenecks hindering overall system performance. However, the efficient utilization of heterogeneous architecture shines when these data transfers are managed adeptly. The concept of overlapping computation and communication can often be implemented as a way of solution in these situations. By coordinating tasks in such a way that communication tasks overlap with computation tasks, system efficiency may be enhanced, mitigating the impact of communication overhead and ensuring smoother operation.

Crafting software for heterogeneous architectures can be a complex endeavor[7]. Programmers must manage a number of processing units, memory hierarchies and

control the transfers of data between them. This typically necessitates the use of specialized programming languages and libraries capable of utilizing the full potential of the diverse hardware components. One also has to take into consideration the different architectures, where one either has to go for a wide approach which may work on a lot of different architectures but might be slower, or one can go for a more specialized approach which works on a limited number of architectures but is faster.

So heterogeneous architectures tempts the programmer with the promise of superior computational performance compared to their homogeneous counterparts, it is on the cost of having to potentially having to pour more of ones own time to overcome the challenges which heterogeneous architecture introduces. While these challenges persists advancement in hardware design and software tools hold the potential to ease the programmer's journey, unlocking the architectures full capabilities across a broad spectrum of applications.

In conclusion, heterogeneous computing architecture is made up of several vital components operating in parallel, leveraging the unique strengths of each processing unit to optimize performance across a diverse range of workloads. These components collectively constitute the architecture's functionality and capabilities. In the context of my thesis I will delve more deeply into parts of these potential communication problems, more specific, the potential rewards of managing to overlap the communication between these processors by trying to keep the processors active while at the same time performing parallel communication between them.

3.1 Central Processing Unit(CPU)

At the core of both heterogeneous and homogeneous architectures lies the CPU, serving as a versatile worker often entrusted with the most serial part of the code thanks to its high clock-frequency. The CPU is specifically tailored for sequential processing, control flow management and system-level management[2]. The CPU plays an important role in being the processor which allows the programmer to distribute computational operations within a heterogeneous architecture environment.

The reason the CPU is used in both heterogeneous and homogeneous architectures is due to the adaptability and broad applicability across various computational tasks, where its proficiency in sequential processing allows it to manage tasks with interdependencies, where the order of execution is critical. In the context of heterogeneous systems, the CPU is given the responsibility of performing task allocation and assigning tasks to different specialized processing units based on what the programmer specifies.

Moreover, the CPU assumes a central role in system-level management, overseeing critical functions such as memory management, hardware peripheral communication and efficient resource allocation. This involvement ensures a cohesive and controlled environment for the entire computing system. Additionally, the CPU handles low-level input/output operations, facilitating interactions with hardware peripherals and managing file systems.

The inherent capabilities of CPUs to manage control flows, encompassing decision-making processes, conditional loops and function calls, further solidifiy their significance in a wide array of computational applications.

3.2 Graphics Processing Unit(GPU)

In heterogeneous architecture, the GPU takes center stage as a processing unit, specifically designed for parallel computation. It seems close to the CPU in terms of architecture, terminology and function, but with some component changes allowing for its excelling in tasks characterized by massive data parallelism. GPUs are known for their applications in graphics rendering, scientific simulations and increasingly in machine learning and artificial intelligence.[3]

The identity of GPU functionality resides in its exceptional capability for parallel processing, achieved through housing hundreds or thousands of cores compared to a CPU's amount which can usually be found in quantities of tens or hundreds for machines used in High Performance Computing (HPC)[8]. These cores work collectively, executing simultaneous operations on numerous data elements. This

parallelism proves highly efficient for inherently parallelizable tasks, allowing GPUs to excel in executing massively parallel computations.

Additionally, GPUs boast an architecture optimized for high-speed data processing, resulting in exceptional throughput. This innate ability enables GPUs to process vast volumes of data at remarkable speeds, making them invaluable for handling data-intensive computational tasks. Their high throughput architecture extends beyond mere data processing, contributing significantly to their efficiency across various applications.

Another distinguishing feature of GPUs lies in their tailored instructions sets, encompassing operations such as matrix manipulations, texture mapping and shaders. These specialized instructions significantly enhance their efficiency in graphics rendering and multimedia processing. Moreover, modern GPU architectures have evolved to support computing outside of graphical pipelines, enabling their versatility in non-graphical tasks like machine learning and artificial intelligence, leveraging their parallel processing capabilities beyond traditional graphics-related functions.

In summary, the GPU's distinct features include its prowess in parallel processing, high throughput architecture, specialized instruction sets and evolving capabilities beyond graphical functions. The synergy between CPUs and GPUs within heterogeneous architecture systems amplifies their computing abilities, enabling the efficient execution of diverse applications across various workloads.

3.3 Memory Hierarchy and Data Communication

Efficient memory management and communication between processing units can be important for optimizing computer code within heterogeneous architectures. Addressing these dual challenges involves working with memory hierarchy and data communication to streamline data accessibility and exchange.

The memory hierarchy, a fundamental part of processor components plays a pivotal role in distributing data accessibility and minimizing bottlenecks[9]. This hierarchical system governs the speed and accessibility of various memory compo-

nents, including registers, caches, and device memory. By intelligently organizing data storage and access, the memory hierarchy ensures that diverse computational tasks can efficiently retrieve the data they require. The different memory hierarchies often follow a similar structure where it is made up of registers and cache memories, but depending on which processor it implemented into, the memory hierarchy will be designed a bit differently depending on the processors strengths.

For instance, CPU computing can be more operated around hierarchical memory structures like caches, prioritizing quick data access and focusing on potential cache hits while maintaining low latency between memories and processor cores. On the other hand, GPUs emphasize more on high memory bandwidth than low latency, enabling them to efficiently handle massive amounts of data for parallel processing tasks. The differences in memory hierarchy implementations, such as the presence of Video Random Access Memory (VRAM)[10] types like Graphical Double Data Rate (GDDR) or High Bandwidth memory (HBM) in GPUs instead of traditional Double Data Rate (DDR) [11], highlight some of the unique optimization strategies employed by different processor types.

Data Communication refer to the different types of hardware and software used to establish communication pathways between different components within a computing system[12]. It includes both physical components like cables, connectors and switches, as well as software libraries and communication protocols. Communication interconnects form a crucial foundation within heterogeneous architectures, enabling efficient and user friendly code which enables cooperation and data sharing among diverse processing units like CPUs and GPUs.

In architectures with several components having an efficient network of data interconnects can therefore be crucial in efficient data components, which also makes it more decisive in impacting computing time when working on heterogeneous architecture than on homogeneous architecture thanks to the higher number of internal components the heterogeneous architecture has. These interconnects, such as Peripheral Component Interconnect Express (PCIe)[13], Infiniband[14] or NVLink[15], serve as the pathways of the architecture, enabling swift and seamless cooperation among CPUs, GPUs, and other processors. Efficient data communication are poten-

tially important for maximizing the parallel processing capabilities of GPUs while supporting the versatile computational tasks handled by CPUs.

Protocols and communication standards like Compute Unified Device Architecture (CUDA)[16], which are programming models that adds additional functionality to programming languages like C or C++, can also be used to improve data communication and computation. These programming models how CPUs and GPUs communicate, share memory and coordinate tasks. They provide a unified foundation for developers to create applications that leverage the distinct strengths of both units, ensuring efficient cooperation.

When it comes to software which can direct the data transfers there are several to choose from. A notable example is GPUDirect, an innovation developed by NVIDIA[17]; enabling direct communication between GPUs, bypassing the traditional role of the CPU as an intermediary for data transfers between GPUs. This advancement significantly reduces data transfer overhead, elevates transfer speeds and crucially minimizes CPU utilization, particularly beneficial for specific workloads that heavily rely on inter-GPU communication.

Understanding and effectively utilizing the data communication can accelerate program execution. By identifying potential bottlenecks in code and optimizing data access patterns, programmers can leverage the full capabilities of heterogeneous architectures. Furthermore, programming around the concept of concurrent communication and computation can further enhance performance, allowing processors to efficiently execute tasks while seamlessly exchanging data. Overall, a comprehensive understanding of memory management and communication mechanisms is essential for maximizing the performance and efficiency of code running on heterogeneous architectures.

4 Data Transfer

Data transfer is essential to any computing architectures which require collaboration between different components, from memory storage to processors like CPUs and GPUs[18]. If enabled through a combination of hardware and software its implementation can be essential to fully utilize the components without being to much affected by the necessary communication. A potential method of performing data transfers in a way which benefits the computing architecture is implementing a method of overlapping communication with computation. We will through this chapter try to grasp how data transfers work and how managing it more detailed can potentially improve the overall computation time.

Data transfers are in essence what the name implies, it is the process of transferring data between two components in a system with the help of both software and hardware. It allows for the transfer of binary data, a sequence of 0s and 1s, from one component memory to another, transcending physical boundaries between diverse elements of a computing system.

To get a starting impression of how data transfers work consider the following equation, which estimates the total time required for performing data transfer:

$$T = T_l + \frac{N}{B} \tag{1}$$

Here, (T) denotes the total time required for data transfer. (T_l) accounts for latency—the time it takes to initiate communication and dispatch the first data packet. (N) signifies the amount of data to be transferred, while (B) represents the available bandwidth. The calculation is straightforward, necessitating only three variables to estimate the transmission time between two points.

The importance of data transfers has also grown with time as more and more components is being implemented into architectures to complement each other, which leads to the situation where the communication between these components becomes more frequent and essential. Examples of this will be shown in this thesis

where we are working on several nodes, each made up of an heterogeneous architecture, where the CPUs and GPUs are required to communicate effectively. The focus of data transfers has also grown not just because of more components which require more communication, but because of the quantity of data which has become available and is worked on as the world wide web has grown.

This has lead to an increased focus on optimizing data transfers, as an example by trying to transfer data to components before they are required there, also called data prefetching[19]. It therefore has become an area which has grown in importance as it often could lead to bottlenecks when left ignored, especially in applications which are often focused around handling data efficiently and without delays, from scientific simulations to artificial intelligence.

4.1 Latency, Bandwidth and Throughput

Efficient data transfer mechanisms are essential in heterogeneous architectures, and understanding latency, bandwidth, and throughput is key to optimizing these processes. These factors influence the quickness and effectiveness of data movement between system components. Let's clarify each term:

4.1.1 Latency

Latency measures the delay experienced by a packet of data as it travels from the source to the destination. It includes all the time lags due to processing, propagation, queuing, and any other delays that occur along the path. While certain factors can introduce variability, the latency between two endpoints can be relatively predictable under consistent network conditions. Latency is most evident when initiating data transfers, often marked from when data starts to be transferred to when it begins to be received by the target component. Reducing latency is beneficial for processors as it lowers their idle time waiting for data, and is particularly critical for systems requiring rapid interactions between components, such as real-time applications where any delay could be detrimental.

The importance of focusing on latency is therefore dependent on what type of memory operations one performs on which memory location, where when working on local cache usually indicates low latency, communicating with other devices can make latency noticeable in certain scenarios. Running a program which requires frequent access to different memories from different components will therefore often be affected by latency in a bigger degree than a program running on 1 CPU which has its data located in memory.

4.1.2 Bandwidth and Throughput

Bandwidth and throughput, while related, refer to different aspects of data transfer capacity. Bandwidth is the theoretical maximum capacity of a transmission medium to transfer data. Think of it as the width of a highway—the wider it is (i.e., the higher the bandwidth), the more lanes of traffic (data) it can support at any given time. It sets the upper limit on how much data could be sent per unit of time, typically measured in Bytes per second (Bps), without accounting for any real-world conditions

Throughput, on the other hand, is the actual amount of data successfully transferred from one place to another over a given period. It's more akin to the average speed and volume of cars (data) that successfully make it from point A to B on the highway, considering the traffic, road conditions, and speed limits. Throughput is affected by various practical factors such as the available bandwidth, network congestion, and data loss, which could slow down transfers. It is also measured in Bps but reflects the real-world performance rather than the theoretical maximum.

The distinction between bandwidth and throughput is crucial: bandwidth represents potential performance while throughput represents achieved performance. High throughput is vital for data-intensive tasks, as it ensures that data reaches its destination efficiently without significant delays.

Balancing latency and throughput is important for effective data transfers. Low latency means that data transfer can begin quickly, while high throughput ensures

the data continues to flow efficiently once started. The importance of each depends on the specific demands of applications and system architectures. Some require rapid initiation of data transfers (low latency), while others need fast and continuous data movement (high throughput). Typically, strategies to improve both are deployed to achieve the best possible data transfer performance in heterogeneous architectures.

4.2 Transfer Rate

The efficiency of data exchange within a computing architecture relies on various elements that collectively impact the transfer rate. Ranging from bandwidth, channel capacity, encoding techniques to the manner which data is packed significantly influence the pace at which information flows between different processing units, devices and memory spaces. Understanding these components will allow us to estimate how to optimize data transfer rates within a complex computing system.

Gigatransfers per Second (GT/s)

One of the most frequently used parameters in estimating data transfer rates is the measurement of GigaTransfers per Second(GT/s). GT/s quantifies the number of data transfers that occur in one second, providing a key insight into the pace of which data can be transmitted between two devices. This forms the basis for understanding the potential speed of a data transfer, particularly when using high-speed interfaces like Peripheral Component Interconnect Express.

4.3 Methods for Communication

The world of computer networking is a dynamic landscape, driven by an large range of different components and methods that enable communication on both inter-node and intra-node scales. The method of communicating between these can also be different, as an example there exist Point to Point (PtP), which will be used in this thesis and gone more thoroughly underneath, or even broadcast communication which allows for sending the same data from one component to many receivers.

4.3.1 Point-to-Point

A normal method of network communication is the PtP connection. This configuration establishes a direct link through interconnects between two endpoints, enabling them to communicate without intermediary devices. The directness of PtP connections not only simplifies the communication process but also often results in a reduction of potential points of failure, making it an essential concept for those new to networking.

PtP connections are characterized by their exclusivity of use between the connected parties. This means that the entire bandwidth of the connection is dedicated to the two endpoints, which can lead to increased reliability and performance, particularly for applications that require consistent and predictable communication rates.

4.3.2 Direct Memory Access

Direct Memory Access (DMA) is a vital technique in computer architecture that enables efficient data transfers between devices without requiring constant CPU involvement. DMA is commonly employed in scenarios where high-speed data movement is necessary, such as inter-node communication, network communication, and peripheral device interactions.

At its core, DMA allows devices to directly access the system's memory, bypassing the CPU for data transfer operations. This capability significantly reduces CPU overhead and latency, as the CPU is freed from managing every data transfer, allowing it to focus on executing other tasks concurrently.

DMA operates by utilizing specialized DMA controllers within devices, such as CPUs or GPUs. These DMA controllers are responsible for initiating and managing data transfers between the device and the system memory.

One of the key advantages of DMA is its ability to perform data transfers asynchronously, decoupling the data transfer process from the CPU's execution flow. Asynchronous DMA operations allow the CPU to continue executing computations

or processing tasks while data transfers occur in parallel. This parallelism enhances system performance and overall efficiency, especially in environments where concurrent processing is critical.

4.4 Data Transfer Technologies

4.4.1 Peripheral Component Interconnect Express(PCIe)

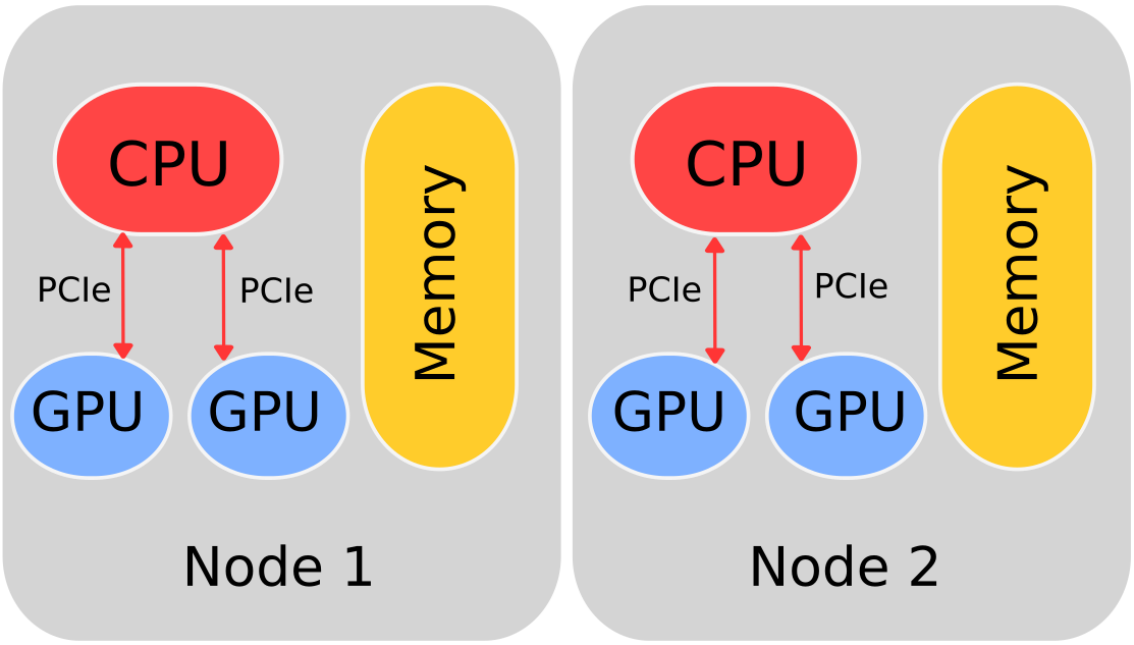

Figure 3: Visual of how PCIe is normally implemented as a communication interconnection between CPUs and GPUs

PCIe[13] is a fundamental communication interface that underpins the efficient data movement within modern heterogeneous architectures. Unlike earlier developed bus architectures that shared communication paths, PCIe employs a point-to-point topology. This mean that each component, whether it is a GPU, CPU or other device, connects directly to the system's central switch, usually integrated into the chipset or CPU `fig.` 3. This direct connection minimizes contention for bandwidth and enables devices to communicate simultaneously while reducing the chance of creating bottlenecks.

PCIe communication occurs through bus lanes, with each lane capable of transmitting data bidirectionally. The number of lanes a device uses determines its data

transfer bandwidth. For example, a PCIe x1 connection employs one lane, while PCIe x16 connection uses sixteen lanes which often results in significantly higher data throughput.

The amount of data rates over a single PCIe lane in each direction is depending on what generation one chooses, where later generations usually improve's the previous generation by performing at twice the speed. A basic overview can be looked at in `fig. 3`. But do notice that we are viewing their data rate in Gigatransfers per second(GT/s) and not Gigabytes per second(GB/s).

PCIe 1 - 2.5GT/s
PCIe 2 - 5GT/s
PCIe 3 - 8GT/s
PCIe 4 - 16GT/s
PCIe 5 - 32GT/s

So to get the value of bytes per second, we also have to know how wide the channel width is or word length.

Data transmission in PCIe happens in packets, with each packet containing not only data but also control and synchronization information. These packets are transmitted serially over pairs of differential wires, ensuring high-speed communication and mitigating electromagnetic interference.

One of the remarkable features of PCIe is its dynamic negotiation of link widths and data rates. Devices can automatically determine the optimal configurations for communication based on their capabilities. This flexibility allows components with varying performance requirements to coexist on the same bus.

PCIe technology has seen multiple generation, with each iteration offering increased data transfer rates. PCIe Gen 3 and 4 for example deliver's progressively higher speeds. These advancements in data rates are pivotal in data-intensive applications such as graphics rendering, video editing and scientific simulations where rapid data exchange is paramount.

PCIe often forms the backbone of communication within heterogeneous architectures. It empowers components to share data efficiently, providing the crucial connectivity needs for tasks like GPU acceleration, AI processing and high-performance computing.

4.4.2 NVLink

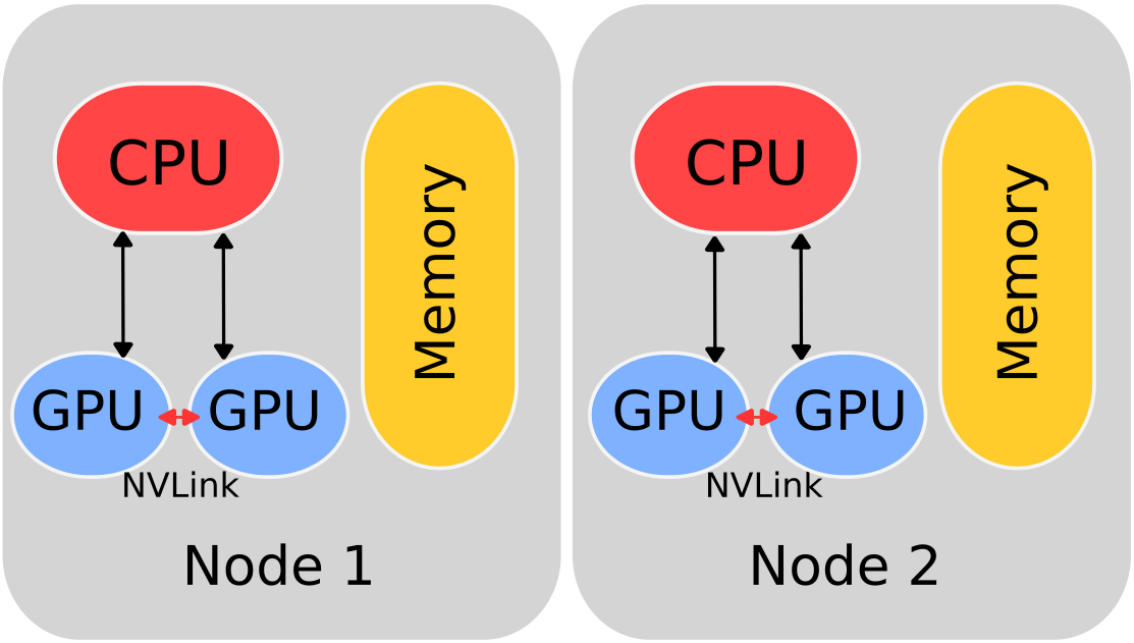

Figure 4: NVLink is especially made for communication between NVIDIA GPUs

NVLink is an interconnect technology developed by NVIDIA[15] to improve communication between GPUs within a heterogeneous architecture. And unlike traditional bus lines like the PCIe connections, NVLink is specifically designed to address the data transfer requirements of high-performance computing and deep learning workloads. It therefore has higher bandwidth reaching as high as 900 GB/s in its 4th-generation. A channel between two devices over NVLink is also called a Brick or NVLink Brick. It can also be used between CPUs and GPUs as it still allows for larger bandwidth.[15]

NVLink operates as a high-speed, point-to-point, serial interconnect that enable's direct serial communication between NVIDIA GPUs `fig.` 4. This direct connection minimizes latency and maximizes data bandwidth, making it well suited for applications that demand rapid and seamless data exchanging between GPUs.

NVLink's architecture allows GPUs to share data at rates significantly faster than those achievable with traditional PCIe connections.[15]

NVLink is particularly advantageous in multi-GPU setups, such as those used for scientific simulations, AI training and complex simulations. It enables GPUs to collaborate closely on tasks that involve parallel processing, large-scale data analysis and deep learning model training.[15]

One of the standout features of NVLink is its ability to scale. The technology supports configurations with multiple NVLink links, allowing for increased data transfer bandwidth and enabling large scale parallel processing across multiple GPUs. This scalability is essential for tackling workloads that require massive computational power.[15]

NVIDIA has introduced multiple generations of NVLink, each offering improved data rates and capabilities. These generations enhance the bandwidth and latency characteristics of NVLink, enabling it to keep up with the demands of evolving applications and architectures.[15]

While NVLink provides remarkable benefits for inter-GPU communication, it is important to note that it is designed for use with NVIDIA GPUs. Therefore it is most commonly found in systems that deploy NVIDIA GPUs for tasks like high performance computing, AI training and data intensive simulations.[15]

When sending data between two GPUs over NVLink it goes through several stages to end up at its destination.

1. Before we even start sending the data we need to prepare it for transmit. This means the processor needs to find the wanted data and organize, format and encode it before starting. The GPU also needs to establish communication through the NVLink to see if it is even possible to send.

2. It then encodes the serial data using an NVIDIA encoding program. This is done to help with clock synchronization and other considerations in serial data

transmission. It does this by adding additional bits to the data it sends over, as these bits contain information used later by the receiving end.

3. Thirdly we start sending the encoded data over the physical NVLink using high-speed signaling. Depending on the version of NVLink the encoded data will be send through several serial lanes concurrently to achieve higher bandwidth.

4. When the data arrives at its destination the processor will start by decoding and it checks if the additional bits give any error messages or any other information.

5. If nothing is wrong it will then deserialize the original data and the receiving end will then have stored the data.

4.4.3 NVSwitch

NVSwitch is a relatively recent addition to NVIDIA's hardware portfolio, playing a relatively important role in architectures that harness multiple GPUs requiring direct communication among themselves for tasks, including HPC. NVSwitch can be viewed as dedicated hardware integrated alongside GPUs, with each GPU connected to one or more NVSwitches via NVLink connections. The NVSwitch ports can operate at a remarkable speed of 50GB/s each, contributing to each NVSwitch's overall total bandwidth of 900GB/s. This impressive bandwidth allows up to nince devices to route data directly to any other nine devices within the system connected to the same NVSwitch[20]. An example of usage can be found in the NVIDIA developed system called DGX-2[21], here they created a system which connects 16 GPUs together through NVSwitches giving a total of 800 Gb/s of bandwidth `fig. 5`.

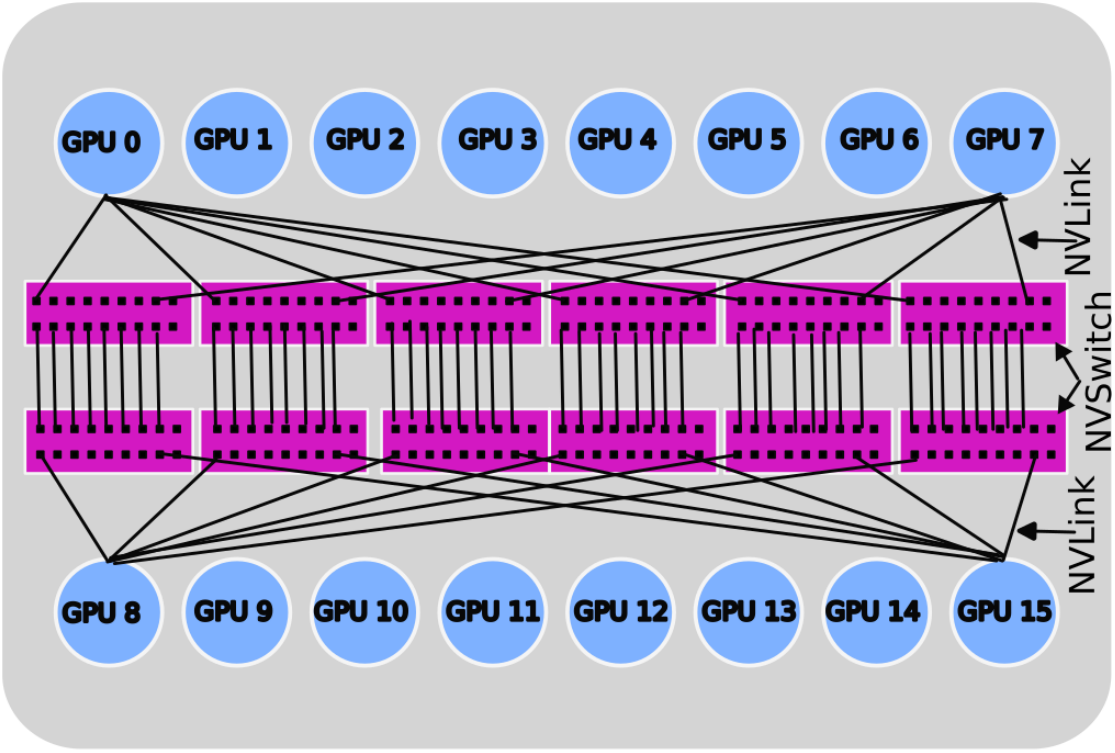

Figure 5: Example of NVSwitches being used in a V100 DGX-2. Here the NVSwitch connect 16 GPUs together through 12 NVSwithces, leaving 2 ports free on each NVSwitch.

4.4.4 Message Passing Interface(MPI)

Message Passing Interface (MPI) is an technology which is widely used communication protocol and library for parallel computing. This means that it defines a set of rules, conventions and standards that allow different processes running on separate compute nodes to communicate with each other. It specifies how data is exchanged, how processes are identified and how communication is managed in parallel computing environment. In addition it introduces a collection of functions and procedures that the programmers can utilize to implement parallel algorithms and applications.

It can also provide the users with some key features:

1. PtP Communication: MPI supports point-to-point communication between individual processes, allowing them to send and receive messages asynchronously. This enables processes to coordinate their activities and exchange data efficiently.

2. MPI also provides collective communication operations, such as broadcast, scatter, gather, and reduce, which allow groups of processes to synchronize and exchange data collectively. These collective operations are crucial for implementing parallel algorithms and computations.

3. MPI is designed to scale efficiently across a large number of nodes, making it suitable for high-performance computing clusters and supercomputers. It allows developers to harness the computational power of distributed systems effectively.

CUDA-aware MPI is an extension of the MPI library that enhances communication between CPUs and NVIDIA GPUs in heterogeneous computing environments. It enables MPI processes running on CPUs to directly access GPU memory and perform data transfers without intermediate copying, reducing overhead and improving performance.

By leveraging CUDA-aware MPI, developers can seamlessly integrate GPU acceleration into their MPI-based applications, taking advantage of the massive parallel processing capabilities of GPUs while maintaining efficient communication with CPU processes. This integration enhances the overall performance and scalability of parallel applications in heterogeneous architecture environments.

4.4.5 GPUDirect

GPUDirect is not a single technology itself, but rather a collective term used to refer to the suite of technologies developed by NVIDIA to enhance data transfers and communication efficiency involving GPUs and other devices in various scenarios, such as memory, inter-GPU communication and cluster communication. The individual technologies under the GPUDirect umbrella, such as GPUDirect Storage, GPUDirect Peer-to-Peer and GPUDirect RDMA are each a technology implemented to address different communication and data transfer challenges which often require

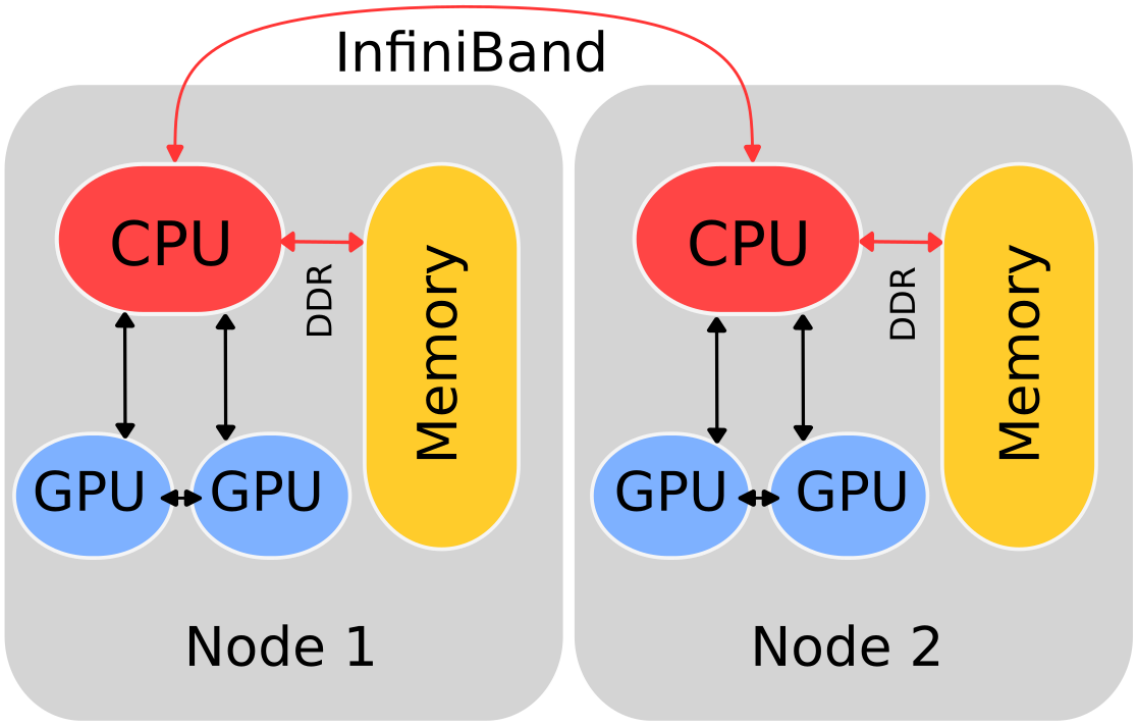

Figure 6: An overview of possible connections between rest of the components, 2 new connections have been added since figure 4. 1 Between the CPUs which can be an InfiniBand technology, and 1 bus between each CPU and its memory called which in this case is Double Data Rate.

transfer of data between locations without having to involve the CPU[17].

GPUDirect P2P GPU Peer to Peer (P2P) communication refers to the ability of multiple GPUs within the same node to directly exchange data with each others memory without involving the CPU. This capability is especially useful in multi-GPU setups, such as those used for deep learning training or scientific simulations. By avoiding the need to transfer data through the CPU and system memory, GPU P2P communication can reduce latency and improve data throughput between GPUs. This results in enhanced collaboration and data sharing among GPUs, leading to more efficient parallel processing and improved overall node performance[17].

GPUDirect RDMA GPU Remote Direct Memory Access (RDMA) extends the concept of traditional RDMA to GPUs, allowing GPUs to perform direct memory-to-memory transfers across cluster-connected nodes without CPU involvement. This technology is particularly beneficial for distributed computing environments where multiple GPUs are spread across different nodes. By enabling GPUs to exchange

data directly with minimal CPU intervention, GPU RDMA accelerates data communication and synchronization between GPUs, enabling efficient parallel processing and interconnect tasks across the cluster. This capability is especially crucial for high-performance computing and data-intensive applications that require fast and low-latency communication between GPUs in different nodes[17].

4.5 Overlapping Communication and Computation

Some situations where the benefits of implementing overlapping communication and computation have been mentioned previously in the thesis. But what is overlapping when it comes to communication and computation? Overlapping simply means performing data transfers and processing tasks simultaneously, rather than sequentially. This strategy optimizes the usage of computational resources and can potentially lead to faster task completion.

When discussing overlapping communication and computation, we are essentially referring to multitasking for computers. The computer performs several actions at once, transferring data from one component of the architecture to another while also performing some processing on the data. This approach helps keep the computer busy and can reduce the downtime of components waiting for transfers or computations to complete.

To illustrate this, let's examine an illustration `fig. 7`, which simplifies a typical result of overlapping. Here, the time taken to transfer data and perform computations is shown separately. When these tasks are performed without overlap, they are executed sequentially, and when one task performs, the other is just waiting patiently before it can start. However, when we overlap these tasks, we can see that the total time required is less than the sum of individual times. This reduction in time comes not from running tasks any faster but rather from being able to start them faster.

For effective overlapping, programmers can carefully plan data transfers and computations around overlapping. This may include the programmer prioritizing

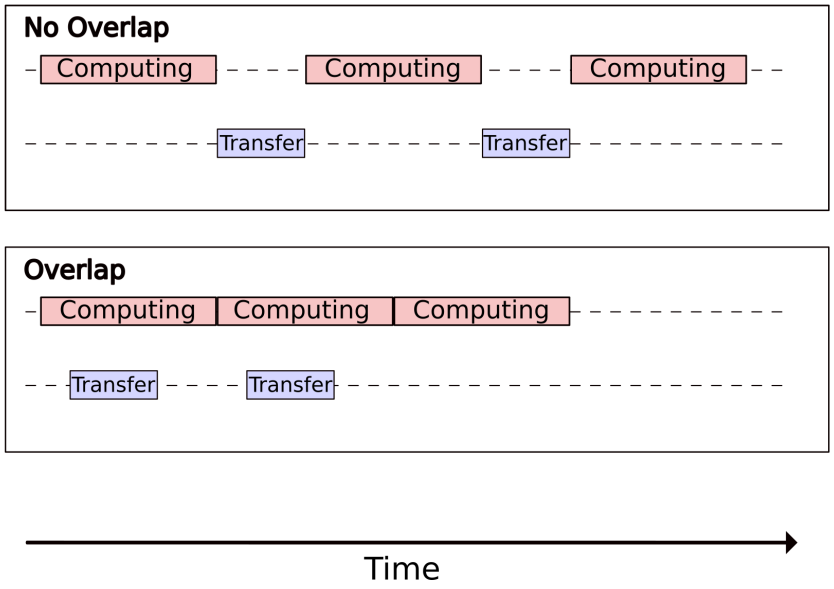

Figure 7: Difference in total computation time between no overlap and overlap implementation

the computation of data that will be transferred first or prioritizing sending the most important data first. One essentially want to achieve the equal time between data transfer times ($T_{transfer}$) and computation times ($T_{compute}$). The balance one should aim for between these two operations can be represented by the following equation:

$$T_{compute} = T_{transfer} \tag{2}$$

$$n \cdot C = T_l + \frac{N}{B} \tag{3}$$

Where n is the number of elements to compute, C is the computational time per element, T_l is latency, N represents the amount of data to be transferred, and B is the bandwidth. The optimal amount of data N to begin overlapping can be solved by equating $T_{compute}$ with $T_{transfer}$ and rearranging for N:

$$N = B \cdot (n \cdot C - T_l) \tag{4}$$

This equation informs us of how much data we should aim to transfer to utilize

the period of computation optimally. Implementing an effective overlap requires precise timing, allowing data transfer to conclude as computation on the transferred data completes.

fig. 8 depicts an idealized scenario of overlapping, showing up to a 50% reduction in the total time by perfectly aligning the communication and computation. Achieving this level of overlap assumes minimal latency and requires that the computation on each data packet begins as soon as it is received, which may not always be feasible in realistic scenarios due to potential memory errors or other communication bottlenecks.

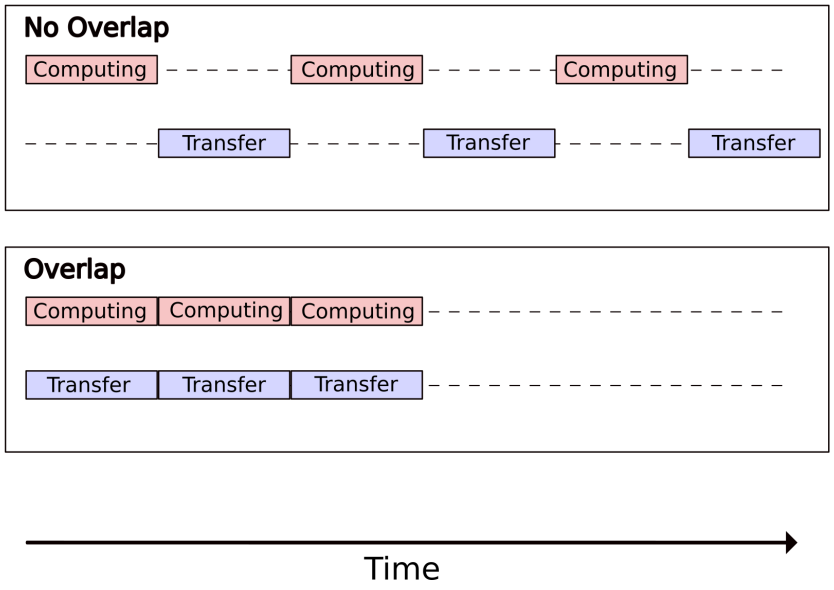

Figure 8: Total overlap between communication and computation illustrating maximum theoretical improvement.

4.6 CPU-free execution path

Recent research introduces a CPU-free execution model for multi-GPU applications, significantly enhancing data transfer efficiency between GPUs. By eliminating the CPU from both data and control paths, GPUs gain full autonomy, reducing latency and increasing throughput. This approach leverages techniques such as persistent kernels, thread-block specialization, GPU-initiated data transfers, and device-side synchronization[22].

The CPU-free model enables direct GPU-to-GPU communication, bypassing the CPU, which traditionally acts as a bottleneck. This results in significantly reduced communication overheads and better overlap between computation and communication tasks. For example, in iterative solvers like the 3D Jacobi stencil and Conjugate Gradient methods, the model has demonstrated up to a 96.2% reduction in communication latency and over 1.6x speedup in overall performance[22].

Overall, the CPU-free execution model offers a promising approach to improving performance in multi-GPU systems, making it an essential advancement for high-performance computing applications[22].

5 Compute Unified Device Architecture(CUDA)

CUDA is an NVIDIA developed programming language which is an extension of C and C++ and it allows for a very direct control over communication between devices and computation on CPUs and GPUs. By implementing it one has access to an more user friendly library which is used for controlling NVIDIA's GPUs directly, an example is through its usage of CUDA threads and its control of data transfers[16].

To begin with we should introduce what CUDA actually is and why it is still being developed and used since its beginning in 2006 by NVIDIA. CUDA serves as a versatile general-purpose parallel computing platform and programming model, offering users access to NVIDIA GPUs through familiar programming languages like C, C++, and FORTRAN. Beyond mere syntax familiarity, CUDA gives developers with a comprehensive framework of concepts and functions essential for using the full potential of GPU acceleration[16].

In today's computing world, characterized by the increasing prevalence of heterogeneous architectures, programming languages like CUDA's relevance continues to grow. By leveraging the power of such languages within heterogeneous architectures, developers can efficiently distribute computing tasks across multiple intra-node devices, optimizing performance and scalability. The combination of architecture and user friendly syntax provides a robust yet adaptable approach to simultaneous computation across diverse devices, promising significant advancements in computational efficiency and speed.

5.1 Programming Model

So what distinguishes CUDA and its programming model from C or C++ syntax? At its core the difference is in the introduced concept of threads, blocks and grids. These constructs allows the developers to harness the power of GPUs by breaking down the computational task into thousands of parallel threads, each executed by individual GPU cores.

It also introduces kernel functions, which represents the code executed in parallel by GPU threads. These functions are defined with the "__global__" qualifier and are started by the host CPU to execute on the GPU[23].

5.1.1 Threads

A CUDA thread operates as the unit of parallel distribution within a GPU's processing architecture, just like a CPU thread works within a CPU. The thread is controlled by the programmer which uses it to fetch instructions and data from the memory to any available CUDA core, the core will then compute and allow the thread to return the output back into its memory[23].

It will based on availability and its provided instruction provided from the programmer fetch the required data and then look for an free CUDA core, it will then let the core compute on the data until it is finished, for then the thread to return the data to memory. This allows for several thousands of threads to work in parallel on a GPU processor while at the same time giving the programmer full control over what work they perform[23].

Each CUDA thread functions as an independent worker within the GPU, capable of running its own operations concurrently with other threads. This contrasts with serial programming where a single thread carries out instructions step by step. In CUDA, threads are organized into groups known as warps, which consist of a certain number of threads, usually 32, that execute the same instruction on different pieces of data. This is to decrease the overhead of creating a unique instruction for each thread, but it results in the number of unique instructions to be done at once to be equal to total number of threads divided by warp size. For example if one is to perform 2 unique synchronous instructions, one has to active 2 warps, where each one allocates one instruction for each warp. We call this execution method Single Instruction, Multiple Threads (SIMT) and it is a method that allows for more freedom when it comes to executing thread instructions than Single Instruction, Multiple Data (SIMD). This is because while SIMD is also made around the concept of performing one instruction on a number of data elements, it does not allow for

branching within, it will instead run the same instruction on all elements which can be a problem when programming if-statements. SIMT usage of threads allow for some threads in a warp to run while other wait, this allows for the warp to run through the for-loop for the threads which fulfills the statement for then when those threads are done it will run the rest of the threads if they fulfill a statement. It could be say that the threads which are not active for an instruction which is run for the warp it is a part of is sleeping[23].

5.1.2 Blocks

CUDA blocks serve as the next level grouping above individual CUDA threads, it is in essence a grouping of warp bundled CUDA threads that are grouped together for coordinated execution and resource allocation. This allows for threads within the same block to communicate and synchronize using its shared memory, allowing for efficient data sharing and coordination of parallel tasks within each block. When launching a CUDA kernel, the programmer specifies the number of blocks to create, each containing a multiple of threads. Each block will be assigned to a free collection of CUDA cores called Streaming Multiprocessors(SM)[23].

5.1.3 Grid

The top-level grouping for parallel execution on a GPU with CUDA is called a grid. Grids are a collection of all the blocks used in the CUDA kernel and allows for the potential of even higher level synchronization and communication across blocks. As with both number of threads for the CUDA blocks, one selects the number of blocks within the grid when launching a CUDA kernel. It also allows for programs where all the threads used in the kernel can be assigned an unique id, allowing for easier programming and giving the programmer a better overview than having to specify block id for assigning an id for each thread[23].

Listing 1: Thread groups

```
//CUDA, selfmade code

int main(){
    // Allocating the quantity of threads within each block. 32*32*1 ->
        1024 threads
    dim3 blockDim(32, 32, 1);
    // Allocating the quantity of blocks with the grid 16*1*1 -> 16 blocks
    dim3 gridDim(16, 1, 1);

    // A total of 16384 threads is being launched, split up between 16
        blocks

    // Launched GPU kernel, kernelFunc, with the 2 specified dim3 variables
    kernelFunc<<<gridDim,blockDim>>>();
}
```

The listing refernced as `listing:1` illustrates a relatively straightforward method for allocating the sizes of groups and grids in a CUDA program. By implementing it in this manner, users can easily determine the desired dimensions for their blocks and grids when implementing them as variables in a kernel launch like for "kernelFunc".

5.2 Computing on CUDA

As previously mentioned, CUDA programming closely resembles the syntax used in languages like C, C++ or FORTRAN. CUDA does however offer an extended set of libraries and APIs that expand its capabilities beyond these more traditional programming languages that it is based off.

To illustrate the similarities and distinctions between CUDA and these languages, consider the code snippets provided below, the first written in C, `listing:2`, and the latter in CUDA, `listing:3`. Both examples aim to create an array of length 1024, with each element initialized to its own index value, followed by adding 11 to

each element. But it must be mentioned that CUDA is built upon C syntax, so if one wanted to just write a C code program within CUDA, one would be able to do so, and the code listed underneath for C would therefore also work for CUDA. The CUDA example underneath is specifically designed to show how one has to code to include the be able to utilize a GPU for computing.

Listing 2: C example

```
//C, selfmade code
int main(){
    // Allocates length on the CPU
    int length = 1024;
    // Allocates pointer to the allocated contiguous memory on the CPU
    int *array_cpu = (int*)malloc(length*sizeof(int));
    // Gives each elemenent in the array its index as value pluss 11
    for(int i = 0; i < length; i++){
        array_cpu[i] = i + 11;
    }
    // Should print array with value {11, 12, ..., 1034}
    for(int i = 0; i < length; i++){
        printf("%i, ", *(array_cpu + i));
    }
    free(array_cpu);
}
```

Listing 3: CUDA example

```
//CUDA, selfmade code

__global__ void Add(int *array_device){
    // Each thread gets its own index to separate them
    int index = threadIdx.x + blovkDim.x * blockIdx.x
    // Adds the index + 11 onto each element of the array
    array_device[index] = index + 11;
}

int main(){
    // Allocates variables on the CPU
    int length = 1024;
    // Creates 1024 threads within each block
    dim3 blockDim(32, 32, 1);
    // Create 1 block in the grid
    dim3 gridDim(1, 1, 1);
    // Allocates array on the CPU of size length*bitsize of int and
        creates a pointer on it
    int *array_cpu = (int*)malloc(length*sizeof(int));
    // Allocates variables for the GPU
    int *array_device;
    cudaMalloc(&array_device, length*sizeof(int));
    // Starts the CUDA kernel with 1024 threads and 1 block.
    Add<<<gridDim,blockDim>>>(array_device);
    // Transfers the data from array_device to array_cpu
    cudaMemcpy(array_cpu, array_device, length*sizeof(int),
        cudaMemcpyDeviceToHost);
    // Should print array with value {11, 12, ..., 1034}
    for(int i = 0; i < length; i++){
        printf("%i, ", *(array_cpu + i));
    }
    free(array_cpu);
    cudaFree(array_device);
```

```
32 }
```

Upon initial inspection, it is evident that both code snippets share many similarities as expected. But one can see that the difference lies in the CUDA implementation of a device kernel and its allocation of variables to enable the GPU kernel to be launched on a device. It is worth noting that there are multiple ways to implement CUDA code, and the chosen approach here demonstrates that it is possible to write CUDA coda that closely resembles C while effectively utilizing GPU kernels.

5.3 Cooperative Groups

CUDA Cooperative Groups are a programming feature provided by NVIDIA's CUDA platform that allows for among else more efficient communication and cooperative parallel execution of threads within or between thread blocks within a grid.

In GPU programming synchronization within a thread block can be limited, as threads within a block is scheduled independently by the hardware, potentially leading to efficient synchronization mechanisms. Cooperative Groups address this limitation by providing a high-level abstraction for synchronization and communication within a thread block. This allows threads within a group to work together more effectively and synchronously. Cooperative Groups offer various synchronization primitives, such as barriers and ballot operations, which enable threads to communicate and synchronize their progress more efficiently than with low-level synchronization methods like $'\_\_synchthreads()'$.

The key idea behind Cooperative Groups is to improve the coordination between threads to better utilize the GPUs resources and improve performance. By using these groups the programmers can write more efficient and readable code that takes advantage of the GPUs architecture to a greater extend. It is important to note that Cooperative Groups require a good understanding of GPU architecture and CUDA programming principles to be used effectively, but they offer a powerful toolset for optimizing parallel applications running on NVIDIA GPUs.

A common challenge that CUDA programmers, or other thread based programs, often face is the intricate balance between thread divergence and efficient synchronization in complex algorithms. For instance, consider a parallel reduction operation performed by multiple thread blocs. Traditional approaches may require multiple synchronization points, leading to decreased parallelism and inefficient memory access patterns. Here Cooperative Groups offer a solution by introducing its synchronization mechanisms like thread block tiles. This allows programmers to create streamlined reduction processes. Thread block can collaborate efficiently, minimize thread divergence and reduce the need for frequent synchronization barriers across all threads.

5.3.1 Grouping Types

When programming with just CUDA, one has access to the simple call for synchronizing threads within a thread group, the previously mentioned $\_\_synchthreads$ intrinsic function. Just having this action does however limit the possibilities of flexibility in improving performance and Cooperative Groups introduces several new data types to solve these problems. The following groups are used as a mechanism to allow for easier synchronization across all threads included in it, we will go through the different sizes of thread groups we can use below[24].

With the introduction of CUDA Cooperative Groups we encounter the **Warp Group**, a fundamental unit in the hierarchy of synchronization. Before Cooperative Groups threads within a warp were executed in lockstep, meaning that the same instruction was started at the same time in parallel. But CUDA lacked a mechanism to synchronize threads within the warp which could potentially create problems for a programmer in certain scenarios. With Warp Group we do however have a function that can solve these problems at the lowest level, allowing for extra finesse when programming[24].

Advancing the synchronization hierarchy, we encounter a significant step up from the Warp Group through the well-known CUDA utility, the $\_\_synchthreads()$ function for **Block Group**. This functions plays a pivotal role in synchronizing all

threads within a block. Its recognition within the CUDA programming landscape stems from its ability to ensure coherent execution and data consistency among threads within a block[24].

Going a step further out than individual blocks, a challenge existed in directly synchronizing threads across different blocks within a grid. Cooperative Groups gave a solution to this with the introduction of **Grid Group**. This abstraction removes the need for programmers to struggle with convoluted strategies involving global memory access or difficult and slow atomic operations. The Grid Group encapsulates the intricacies of synchronization, furnishing a streamlined mechanism to foster interconnect interaction amongst disparate threads blocks within a grid[24].

The last stage of Cooperative Groups contributions is the **Multi-Device Group**. This innovation empowers programmers to synchronize threads across multiple CUDA devices, proving invaluable in scenarios where a heterogeneous architecture involving multiple GPUs or multiple processors of identical nature. The Multi-Device Group bridges the synchronization divide that arises when performing parallelism across diverse devices, rendering an indispensable tool for optimizing performance in multifaceted scenarios[24].

To show an example of a scenario where we might consider using Cooperative Groups instead of just CUDA let us look at the following scenario where we want to synchronize the threads for each iteration of a for-loop across the whole grid. We will launch the device kernels with a (32,32,1) dimensional threadblock and (16,1,1) dimensional gridblock[24].

Listing 4: CUDA without Cooperative Groups

```
//CUDA, selfmade code
__global__ void kernelFunc(int *array_device, int length_of_array){
    int global_index_x = blockIdx.x * blockDim.x + threadIdx.x;
    int global_index_y = blockIdx.y * blockDim.y + threadIdx.y;
    // Convert the 2-dimensional global index to a 1-dimensional index
    int thread_index = global_index_x + global_index_y * gridDim.x *
        blockDim.x;

    // If thread index is between (1, length_of_array-1)
    if(thread_index != 0 && thread_index < length_of_array-1){
        for(int i = 0; i < 10; i++){
            // Perform any computation
            array_device[thread_index] = array_device[thread_index]/2 + i;
            // Synchronize all threads within a block
            __syncthreads();
            // Use an atomic variable to ensure all blocks have completed
                their work
            if (thread_index == 0) {
                atomicAdd(array_device, 0);
            }
        }
    }
}
```

Listing 5: CUDA with Cooperative Groups

```
//CUDA with cooperative groups, selfmade code
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

__global__ void kernelFunc(int *array_device, int length_of_array){
    // Groups all threads within the grid together into 1 grid group
    cg::grid_group grid_g = cg::this_grid();
    // Gives each thread within the group their own index
```

```
9       int thread_index = grid_g.thread_rank();
10
11      // Assigns each element in array_device its own index as value if
            there is enough threads
12      if(thread_index != 0 && thread_index < length_of_array-1){
13          for(int i = 0; i < 10; i++){
14              // Perform any computation
15              array_device[thread_index] = array_device[thread_index]/2 + i;
16
17              // Synchronizes all thread within the grid
18              grid_g.sync();
19          }
20      }
21  }
```

When comparing CUDA without cooperative groups `listing:4` and CUDA with cooperative groups `listing:5`, differences become visible in how cooperative groups streamline programming. With cooperative groups, programmers can simply utilize the "grid_g.thread_rank()" function to obtain the unique thread rank for each thread across the entire grid, resulting in cleaner code. More importantly, eliminating the need for atomic operations. While atomic operations are relatively quick, their frequent usage can significantly increase computational overhead.

5.4 CUDA Stream

When it comes to providing mechanisms for managing parallelism and overlapping computation with data transfer on GPUs, CUDA Streams are a concept within NVIDIA's CUDA programming model which is made to provide a solution. Streams enable developers to control asynchronous execution of multiple operations, allowing for efficient use of GPU resources and improving performance[25].

A CUDA Stream can be looked at as a sequence of operations that are executed on the GPU in the order they are submitted to the stream. These operations

can include kernel calls, memory transfers between CPU and GPU and other GPU related tasks. The key feature of streams is the operations within the same stream are executed sequentially, while different streams can potentially run asynchronously.

So compared to a CPU stream which runs linearly through your code and waits for the previous command to finish before going onto the next, CUDA stream allows for several GPU commands to be queued up in different streams where they all run parallel until they reach a point where you want them to synchronize up again.

Several CUDA functions can be executed with specified CUDA streams, enabling asynchronous operations. In the following example `listing:6`, we will run one CUDA Stream for each GPU. This setup enables independent execution of tasks on each GPU while incorporating synchronization mechanisms. Each stream will run its own device kernel, wait for kernel completion and the copy its computed array back to the CPU.

Listing 6: CUDA Stream

```
1  //CUDA, selfmade code
2  int streamFunc(int *array_host, int *array_device, int length_of_array,
       int number_of_gpus, dim3 blockDim, dim3 gridDim){
3      // Allocating unique id CUDA streams and CUDA events for each GPU
4      cudaStream_t streams[number_of_gpus];
5      cudaEvent_t events[number_of_gpus]
6      for (int g = 0; g < number_of_gpus; g++) {
7          cudaSetDevice(g);
8          cudaStreamCreate(&streams[g]);
9          cudaEventCreate(&events[g]);
10     }
11     // Launches a kernel on each device
12     for(int g = 0; g < number_of_gpus; g++){
13         cudaSetDevice(g);
14         kernelLaunch<<<gridDim, blockDim, 0, stream[g]>>>(array_device,
               length_of_array);
15         // Records the asynchronous launch so we know when it is finished
16         cudaEventRecord(events[g], streams[g]);
```

```
    }
    for(int g = 0; g < number_of_gpus; g++){
        // Waits for the kernel to finish
        cudaStreamWaitEvent(streams[g], events[g]);
        // Starts transfering data from device to host
        cudaMemcpyAsync(array_host + g*length_of_array, array_device,
            length_of_array, cudaMemcpyDeviceToHost, streams[g]);
    }
}
```

5.5 Asynchronous Operations

CUDA allows for a much deeper control over GPU Asynchronous Operations. These operations play an important role in boosting the performance and efficiency of GPU computing through enabling parallelism and task overlapping. By enabling multiple tasks to execute concurrently, asynchronous operations eliminate the need for a stream to wait for a task to finish before starting the next one, and instead allow for all streams to perform their own asynchronous operations for then to synchronise at specified parts of the program, resulting in potential improved throughput and reduced processor downtime[25].

One of the standout advantages of asynchronous operations is their ability to achieve communication-computation overlap, a strategy that is can be essential for maximizing system efficiency. This approach becomes particularly significant when handling data transfers between the CPU and the GPU or between multiple GPUs. Through the use of functions like **cudaMemcpyAsync** data can be transmitted in the background while the processor simultaneously performs computations. This eliminates the need for processors to idle during data transfers, effectively utilizing both the compute and memory subsystems in parallel.

While asynchronous operations can offer performance benefits for a program, it is essential to understand the importance of correctly using synchronization mech-

anisms effectively. It can be easy to lose a bit of the overview of things happening when one has to program around several streams running concurrently doing different tasks and synchronizing at different steps of the way and through this stumble upon a situation where data integrity is at stake, as example through either a memory race condition or accessing data which has not yet been allocated.

Therefore understanding the different CUDA synchronizations and methods of documenting when an asynchronous call is finished is essential when trying to compute a program around using asynchronous computing, as when one overlaps computing and communication between two devices.

The CPU however can not estimate when asynchronous operations are finished as its job is just queuing up the GPU operations, but this leads to the situation where one might have to know when the asynchronous operations are done before one can continue. This is where events are introduced as a concept to handle these scenarios. Events are a synchronization primitives used to measure and record the execution time of CUDA kernels and other GPU operations. They allow the programmer to time the execution of GPU code, synchronize GPU operations and measure performance metrics.

Here are some of the synchronization functions which can be used to either synchronize all threads, based on event or based on event and stream.

- **cudaEventSynchronize** - Synchronizes all streams to wait for the specified event to provide the event finished signal.
- **cudaStreamSynchronize** - Makes the stream wait for all its asynchronous functions to finish before continuing.
- **cudaStreamWaitEvent** - Makes the specified stream wait for the specified event finished signal.
- **cudaDeviceSynchronize** - Waits for all processors to finish computing before continuing.

Finding a balance between these three CUDA mechanisms, events, streams and

synchronization can therefore be one of the focuses when writing asynchronous code, especially since they can become a bottleneck and either lead to memory errors or even make the performance of the code worse when implemented badly compared to a code without. It is therefore something one should have in mind when trying to convert synchronous code into asynchronous code, if the invested development time is worth the potential benefits.

The adoption of asynchronous computations does introduce a level of complexity to the code as mentioned, but its benefits depend on the specific problem at hand. One has to strike a balance between the time and resources invested in implementation weighted against the potential gains, as optimization efforts should be used on problems which has potential of returning noticeable improvements.

Furthermore, asynchronous operations come to the forefront when optimizing resource utilization within the system is essential. By running tasks concurrently, they prevent various processing units from sitting idle and ensure maximum resource utilization. In contrast to synchronous operations, where tasks run sequentially, asynchronous tasks can overlap, reducing idle time and enhancing overall system throughput. Moreover, asynchronous operations can be used in scenarios where tasks exhibit overlapping periods, making it inefficient to wait for sequential execution. In cases where one task can begin before another has completed, asynchronous operations prove highly beneficial. They enable ready-to-execute tasks to proceed without waiting for prior tasks to conclude, resulting in reduced overall execution time and more efficient resource utilization.

6 Implementation

To understand the mechanisms of overlapping computation and communication in parallel computing, it is important to construct well-defined scenarios that realistically represent real-world challenges and allow for easy observation of performance changes and challenges introduced by implementing overlapping code. We will primarily explore three communication strategies: communication between CPUs on different nodes, communication between GPUs on the same node, and communication between GPUs on different nodes. By exploring these strategies through code snippets, illustrations, and descriptions, we can further understand what differentiates the three, why we might want to observe each one individually, and later compare them to find common factors that can affect the potential benefits of overlap.

- **Initialization:**

$$\mathbf{A}^0 = \mathbf{A}_{\text{initial}}$$

  where $\mathbf{A}_{\text{initial}}$ is the initial 3D grid.

- **Iterative Update:** For $n = 0, 1, \ldots, N-1$ (where N is the total number of iterations):

$$\mathbf{A}^{n+1}_{i,j,k} = \frac{1}{6}\left(\mathbf{A}^n_{i+1,j,k} + \mathbf{A}^n_{i-1,j,k} + \mathbf{A}^n_{i,j+1,k} + \mathbf{A}^n_{i,j-1,k} + \mathbf{A}^n_{i,j,k+1} + \mathbf{A}^n_{i,j,k-1}\right) \quad (5)$$

  for $0 < i < \text{width} - 1$, $0 < j < \text{height} - 1$, and $0 < k < \text{depth} - 1$.

In this thesis, I have chosen a scenario that revolves around 3D grid operations, a common computational task in many scientific domains. Specifically, the expression shown in `expression:`(5). The primary objective is to calculate a new value for each element within the 3D grid, except for the elements on the borders of the grid, which are set to 0. These new values are calculated not only based on the element's own value but also on the values of its neighboring elements in all three dimensions, as illustrated in `fig.` 9. The new value will then be the sum of all its neighboring values divided by 6, which is the number of elements used in the

sum. As the values of the elements on the outer border of the grid is equal to 0 throughout the computation and not changed, the interior elements of the grid will converge towards 0.

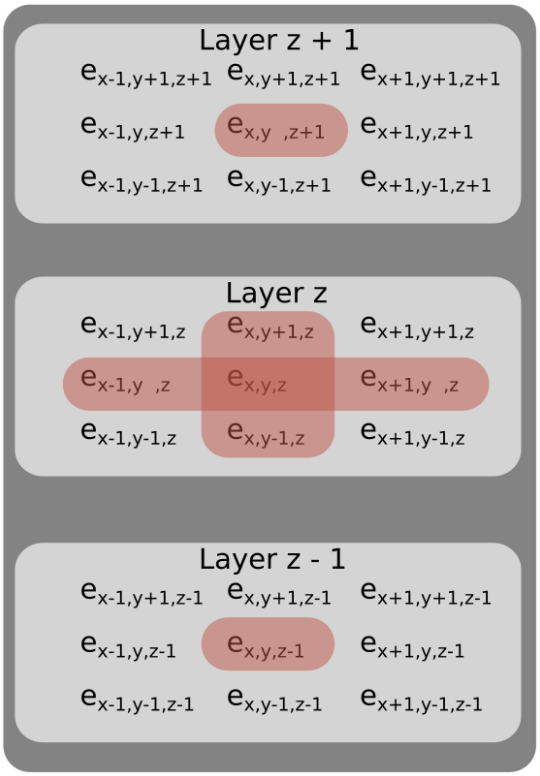

Figure 9: For each computation of an element, we require the neighboring elements from 3 layers

The computation can be found in its simplest implementation when written in C for use on a single CPU core. By writing it for a single CPU core, we remove the necessity of including any specific communication functions, and the code is forced to be run in a serial sequence manner. The CPU only implementation is not useful for finding the potential benefits of overlapping, as there is no communication being performed. Instead, it serves as a blueprint for later implementations that may include GPUs or multiple nodes. To illustrate how this code differs from later implementations, refer to `fig. 10`, which provides a simplistic overview of a system comprising 2 nodes, each containing 1 CPU and 2 GPUs. By only using a single CPU from this system, we as mentioned removes the complication of having to implement communication between processors, but we also exclude the potential computational power benefits one can gain from including other processors or nodes into the computation.

It is worth mentioning that an additional scenario one can use involves a heterogeneous architecture with one CPU and one GPU. This configuration is frequently used in personal desktop computers and other system, making it relevant when considering general computations. However, it quickly became apparent that the only communication in such an architecture for the specific problem we are to work on occurs at the start and the end of the program. From the CPU to the GPU, the GPU then computes the entire grid independently and transfers the results back to the CPU when done with all the iterations. It is therefore no significant opportunity for overlap, except during the initial and final phases. As a result, it was decided not to implement this specific combination of processors, as it would merely demonstrate the benefits of running computations on a GPU instead of a CPU, rather than showcasing the potential advantages of overlapping communication and computation.

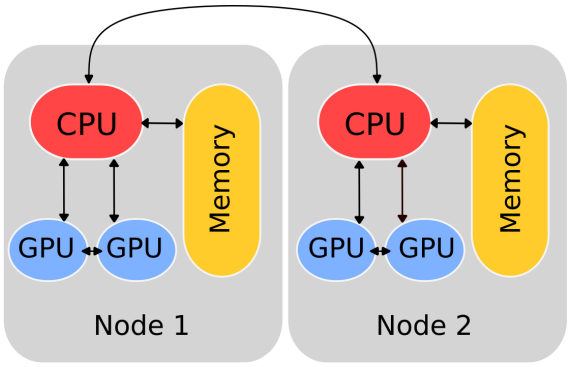

Figure 10: An simplistic illustration of some components which can potentially be used within a heterogeneous architecture

As a result of running the code on a single CPU core `listing:7`, without any inter-processor communication, the computation being performed can represents the program in its simplest form. This code operates similarly to the others but lacks the additional functions required for inter-processor communication and for harnessing the strengths of diverse processors. As the CPU code is notably shorted compared to its counterparts, it will be the only almost fully implemented code included in this thesis. Other codes will be given in sections, mostly the part of the code which delegate computation and performs communication, highlighting their most important differences from the CPU-only implementation.

Listing 7: Single Core CPU implementation

```
#include <math.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int width = atoi(argv[1]);
    int height = atoi(argv[2]);
    int depth = atoi(argv[3]);
    int iter = atoi(argv[4]);
    double division = 1/6.0;
    double *data;
    double *data_next;
    data = (double*)malloc(width*height*depth*sizeof(double));
    data_tmp = (double*)malloc(width*height*depth*sizeof(double));
    double dx = 2.0 / (width - 1);
    double dy = 2.0 / (height - 1);
    double dz = 2.0 / (depth - 1);
    // Use function to fill the grid with its initial values
    while (iter > 0) {
        for(int i = 1; i < depth - 1; i++){
            for(int j = 1; j < height - 1; j++){
                for(int k = 1; k < width - 1; k++) {
                    int index = k + j * width + i * width * height;
                    data_next[index] = division * (
                    data[index + 1]          + data[index - 1] +
                    data[index + width]      + data[index - width] +
                    data[index + width*height] + data[index - width*height]);
                }
            }
        }
        double *data_swap = data_next;
        data_next = data;
        data = data_swap;
        iter--;
```

```
34      }
35      free(data);
36      free(data_next);
37      return 0;
38  }
```

The main part to examine in this CPU-centered C code is the while loop which encompasses the lines from 20 to 38 in `listing`:7. The while loop is the part of the code where the most important part of the computation occurs, and it is also where the future communication implementation will be included. Currently the while loop only has one part we will focus on, the triple for-loop which performs the computation. In the future another part will be included in this while loop, the inclusion of communication.

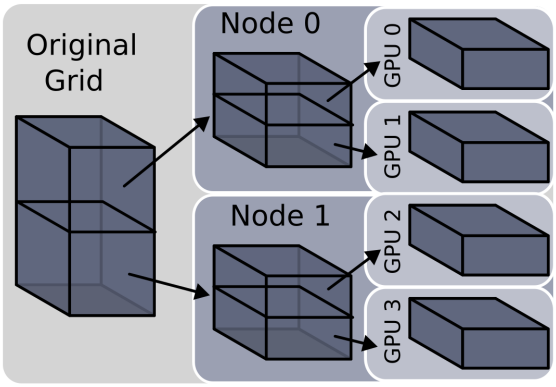

Figure 11: We divide the grid into small segments, one for each GPU. The GPUs are here ranked from 0-3, but would within each node normally be ranked 0-1.

Implementing CPU-centered C code is fortunate in that all the data it requires is most likely stored in memory and does not require any pre-allocation to specific memory locations. This is, however, not the case when working on multiple processors. When distributing computations across multiple processors, data movement can potentially become a significant challenge. Therefore before we can proceed with implementing communication between devices, we need to partition the 3D grid among them so that each device has data to compute on and to communicate with.

The method for partitioning the 3D grid is illustrated in a simplified manner

in fig. 11 and may consist of up to two steps, depending on the computation scenario. Initially, the division of the grid among processors implies partitioning along one dimension rather than distributing data elements with exact precision. In this context, we focus on division along the z-axis, referred to as `depth` throughout the thesis. We define each increment along the z-axis as a `layer`. As a consequence each subgrid will retain the same values along the x-axis and y-axis but may differ in the number of layers they have.

It is important to note that computations are not performed on the boundary elements of the grid, therefore the outermost layers at the top and bottom are excluded from the partition. This strategy is employed to strive for an equitable distribution of the computational workload.

So to follow the pattern of division shown in fig. 11 it depends on if we have more than 1 nodes and if a node has 1 or more GPUs. For any scenario being performed on more than 1 node we first divide the grid up into as many parts as there are nodes, this is illustrated through the code in listing:8. Remember that depth is the number of layers a grid has, so when we start this we first remove 2 layers and then divide the layers evenly among the available nodes. After giving each node its number of layers it it going to compute, we add 2 new layers whose purpose it is to contain the receiving data from neighboring nodes, or to be the border layers for node 0 and the last node. Often the number of total layers cannot be divided equally between the nodes, som nodes might require to contain one more layer than the rest for all the layers to be included, this is calculated through the if statement where all nodes with rank lower than leftover layers calculated from is given an additional layer.

As shown with the division pattern presented in fig. 11, the distribution process varies depending on the system's content. It depends on whether there is more than one node and upon the number of GPUs each node possesses. For tasks running across multiple nodes, the initial step is to partition the grid into as many subgrids as there are nodes, as demonstrated in listing:8.

Recall that 'depth' denotes the number of layers within a grid. Before we begin

dividing two boundary layers are removed from the total depth before evenly dividing the remaining layers among the nodes. Once the number of layers assigned to each node for computation is established, two additional layers are appended to each subgrid. These additional layers serve as boundaries who hold data received from adjacent nodes or define the boundaries for the first and last nodes in the sequence.

It is common that the total count of layers cannot be exactly divided among the nodes. Consequently, some nodes may be required to hold one extra layer to ensure all layers are assigned. This unequal distribution is determined through a conditional statement, in which nodes with an node_rank less than the number of leftover layers derived from the equation on line 5 in `listing:8`, are allocated one more layer to incorporate.

Listing 8: Dividing 3d Grid For Nodes

```
1  //C, selfmade code
2      // Split total depth of grid into parts depending on number of nodes.
           Removes upper and lower layer to easier divide them up
3      int depth_node = (depth-2)/number_of_nodes;
4      // If there are any layers left it is stored
5      int depth_leftover = (depth-2)%number_of_nodes;
6      // We then add the upper and lower layers back to each node grid
7      // If there are any layers stored, it gets added as an extra layer to
           the first nodes
8      if(depth_leftover > node_rank){
9          depth_node += 3;
10     }
11     else{
12         depth_node += 2;
13     }
```

After assigning subgrid values to each node, or proceeding directly to this step if there is only a single node, we continue to compute the values for the elements within each subgrid. This computation is independently carried out by the CPUs within their own nodes, utilizing a specific function where the values assigned are

contingent upon which node is performing the computation.

After we are done with the assignment and computation of values for the subgrids, we subdivide them further according to the number of GPUs available within each node. This is only done if we are to compute with GPUs instead of CPUs. The division process for the GPUs is illustrated in `listing:9` and is similar to the method previously applied to nodes. However, the key distinction between these two divisions lies in the allocation. Instead of just finding the number of layers assigned to each node, the implementation instead allocates data on the GPU depending on the values it computes.

Listing 9: Dividing Subgrids For GPUs

```
1  // Use function to fill the grid with its initial values
2
3  // We then divide the node grid into equal sized grids for each GPU
       the node has
4  int layers_per_device = (depth_node-2)/number_of_gpus;
5  // We store the leftover layers
6  int layers_leftover = (depth_node-2)%number_of_gpus;
7
8  // Allocating number of layers per GPU in an array on the CPU
9  int *layers_device;
10 cudaMallocHost(&layers_device, number_of_gpus*sizeof(int*));
11 cudaMallocHost(&layers_starting_index, number_of_gpus*sizeof(int*));
12 // Calculate the number of layers for each device
13 for (int g = 0; g < number_of_gpus; g++) {
14     int extra_slice = (g < layers_leftover) ? 1 : 0;
15     layers_device[g] = layers_per_device + extra_slice + 2;
16     layers_starting_index[g] = g * layers_per_device + min(g,
           layers_leftover);
17 }
```

Having completed the process of determining the number of layers to allocate to each GPU on each node through the procedures in the previous listings, we must

now address the actual storage of data on each individual GPU. This is accomplished using a combination of cudaMalloc, cudaMallocHost, and cudaMemcpy functions, as illustrated in `listing:10`. Here, the cudaMalloc and cudaMallocHost functions allocate memory on the GPU and host, respectively, utilizing C pointers for each specific GPU. Subsequently, cudaMemcpy facilitates the transfer of data from the CPU memory, denoted by the variable `data`, to the corresponding GPU memory, which is indicated by `data_gpu[g]`.

Listing 10: Allocating GPUs Subgrid

```
1  double **data_gpu, **data_gpu_tmp;
2  cudaErrorHandle(cudaMallocHost(&data_gpu, gpus*sizeof(double*)));
3  cudaErrorHandle(cudaMallocHost(&data_gpu_next, gpus*sizeof(double*)));
4  for(int g = 0; g < gpus; g++){
5      cudaErrorHandle(cudaSetDevice(g));
6      cudaErrorHandle(cudaMalloc(&data_gpu[g],
           width*height*slices_device[g]*sizeof(double)));
7      cudaErrorHandle(cudaMalloc(&data_gpu_next[g],
           width*height*slices_device[g]*sizeof(double)));
8  }
9  for(int g = 0; g < gpus; g++){
10     cudaErrorHandle(cudaSetDevice(g));
11     cudaErrorHandle(cudaMemcpy(data_gpu[g],
           data+layers_starting_index[g]*width*height,
           layers_device[g]*width*height*sizeof(double),
           cudaMemcpyHostToDevice));
12 }
```

We have now reached the phase where computation and communication with GPUs will begin. The computational aspect of this program is similar to that of the original CPU-only version, but a crucial difference lies in how it runs the computation. In contrast to a single CPU thread processing $width * height * depth$ elements, each GPU thread now computes a subset of elements within the 3D grid concurrently. This parallelism leads to a reduction in the computational load per thread, which becomes $width * height * depth/number\_of\_CUDA\_threads$. The

number of CUDA threads should be specified by the developer to match the problem size and hardware capabilities. In this thesis, I have chosen to set it at 16,384, configuring thread blocks of size (32, 32, 1) and grid blocks of size (16, 1, 1).

Changes have also been made to the iterative for-loop computations within the while loop, as depicted in `listing:11`. These modifications are made to enable the increased thread count to work in parallel, ensuring that each thread performs computations in a way that benefits from the warp-level memory coalescing features inherent to CUDA threads. Threads ranging from 0 to thread_size will tackle the initial thread_size elements of the grid. Subsequently, they will advance to the next set of thread_size elements, continuing this pattern all the way to the last element is computed.

Listing 11: GPU Computation Implementation

```
1  __device__ void calc(double *data_gpu, double *data_gpu_tmp, int
       elementsPerThread, int index_start, int width, int height, int thread,
       int thread_size){
2      double division = 1.0/6;
3      for(int i = 0; i < elementsPerThread; i++){
4          int index = index_start + i*thread_size;
5          int x = index % (width - 2) + 1;
6          int y = (index / (width - 2)) % (height - 2) + 1;
7          int z = index / ((width - 2) * (height - 2)) + 1;
8          index = x + y*width + z*width*height;
9          data_gpu_tmp[index] = division * (
10             data_gpu[index + 1]          + data_gpu[index - 1] +
11             data_gpu[index + width]      + data_gpu[index - width] +
12             data_gpu[index + width*height] + data_gpu[index -
                   width*height]);
13     }
14 }
15
16 __global__ void jacobiMid(double *data_gpu, double *data_gpu_tmp, int
       width, int height,
```

```
17                      int slices_Leftover, int device_nr, int
                            slices_compute, int elementsPerThread, int
                            elementsLeftover,
18                      int elementsPerThreadExtra, int
                            elementsLeftoverExtra, int overlap_calc){
19      // Using Cooperative Groups
20      cg::grid_group grid_g = cg::this_grid();
21      int thread = grid_g.thread_rank();
22      int thread_size = grid_g.size();
23
24      // Runs the computation
25      calc(data_gpu, data_gpu_tmp, elementsPerThread, thread+overlap_calc,
            width, height, thread, thread_size);
26  }
```

With the necessary updates for grid allocation and computation on multiple processors in place, we must now consider the methods for inter-processor communication. The strategies we have implemented include MPI, for communication between nodes, as detailed in `listing:12`, and GPUDirect, for communication among GPUs, outlined in `listing:13`.

Focusing first on MPI communication, the implementation showcased in `listing:12` demonstrates how a node with two neighboring nodes can engage in data exchange asynchronously, utilizing `MPI_Isend` for sending data and `MPI_Irecv` for receiving data. A node executing this code would initiate four MPI calls, or only two if the node is connected to a single neighboring node, and would subsequently wait for the completion of these asynchronous operations using `MPI_Waitall`. After ensuring that all MPI operations are finalized, the node can proceed to perform new computation commands, using the newly received values from its neighbors.

Listing 12: MPI Communication example

```
// MPI transfers, selfmade code
    // Receives and transfers data asynchronously from node with rank
        "node_rank-1" to node with rank "node_rank"
    MPI_Irecv(&data_tmp[0], width*height, MPI_DOUBLE, node_rank-1, 0,
        MPI_COMM_WORLD, &myRequest[0]);
    MPI_Isend(&data_tmp[width*height], width*height, MPI_DOUBLE,
        node_rank-1, 0, MPI_COMM_WORLD, &myRequest[1]);
    // Transfers and recieves data asynchronously from node with rank
        "node_rank+1" to node with rank "node_rank"
    MPI_Isend(&data_tmp[width*height*(depth_node-2)], width*height,
        MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[2]);
    MPI_Irecv(&data_tmp[width*height*(depth_node-1)], width*height,
        MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[3]);
    // Waits for all the asynchronous transfers to finish
    MPI_Waitall(4, myRequest, myStatus);
```

Inter-GPU communication is facilitated predominantly via CUDA's GPUDirect technology when working on NVIDIA GPUs, granting easier control over exchanges between the involved GPUs. When implemented with CUDA Streams, it enables the concurrent execution of multiple asynchronous operations, each within its own stream, as previously written in `section:5.4`.

The code snippet provided in `listing:13` details how intra-node GPU communication is done through asynchronous transfers using `cudaMemcpyPeerAsync`. It shows that by leveraging threads, it is possible to start several of these operations in parallel, each thread handling a distinct data transfer, allowing them to operate independently of one another. Subsequent synchronization ensures that all the data transfers complete before the next stage of computation begins, maintaining the integrity and consistency of the computations across the multiple GPUs.

Listing 13: GPUDirect Communication

```
// CUDA Transfers, selfmade code
    // CUDA transfer between devices
    for(int g = 1; g < gpus; g++){
        // Transfers data asynchronously between device g and g-1
        cudaSetDevice(g);
        cudaMemcpyPeerAsync(data_gpu_tmp[g-1] +
            (slices_device[g-1]-1)*width*height, g-1, data_gpu_tmp[g] +
            width*height, g, width*height*sizeof(double), streams[g][1]);
        cudaEventRecord(events[g][1], streams[g][1]);
    }
    for(int g = 0; g < gpus-1; g++){
        // Transfers data asynchronously between device g and g+1
        cudaSetDevice(g);
        cudaMemcpyPeerAsync(data_gpu_tmp[g+1], g+1, data_gpu_tmp[g] +
            (slices_device[g]-2)*width*height, g,
            width*height*sizeof(double), streams[g][1]);
        cudaEventRecord(events[g][2], streams[g][1]);
    }
    for(int g = 0; g < gpus; g++){
        // Waits for the transfers to finish
        cudaSetDevice(g);
        cudaEventSynchronize(events[g][1]);
        cudaEventSynchronize(events[g][2]);
    }
```

The integration of communication between GPUs situated on different nodes is achievable through the use of CUDA-aware MPI in combination with GPUDirect RDMA. This hybrid technique is used for scenarios where GPUs within one node need to exchange data with GPUs on another node. Notably, hardware capabilities can constrain this method, for it requires specific features to execute the commands successfully. If compatible hardware is available, the method can streamline the process, reducing the complexity, and reducing the intermediary steps required for

data transfer between the source and destination memory.

The implementation illustrated in `listing:14` shows this technique. At first glance it may appear similar to the code depicted in the preceding `listing:12`; however, it is important to see the small but significant difference in the whereabouts of the data being transmitted. While the previous implementation worked with data allocated on the CPU, the current method transfers direct data transmission between the GPUs, bypassing the CPU and thereby optimizing the data path.

Listing 14: inter-node GPU Communication

```
1  MPI_Isend(data_gpu_tmp[gpus-1] + width*height*(depth_node-2),
       width*height, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, &myRequest[0]);
2  MPI_Irecv(data_gpu_tmp[gpus-1] + width*height*(depth_node-1),
       width*height, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, &myRequest[1]);
3  MPI_Isend(data_gpu_tmp[0] + width*height, width*height, MPI_DOUBLE,
       rank-1, 0, MPI_COMM_WORLD, &myRequest[2]);
4  MPI_Irecv(data_gpu_tmp[0]           , width*height, MPI_DOUBLE,
       rank-1, 0, MPI_COMM_WORLD, &myRequest[3]);
5  MPI_Waitall(4, myRequest, myStatus);
```

6.1 Partitioning Computation

Adopting an overlapping strategy for computation and communication in scenarios involving various types of processors necessitates some modifications in contrast to the CPU-only code approach. One of the key adaptations pertains to the partitioning of computational tasks.

In the current CPU-only version of the code, all computations are executed within a single, triple for-loop structure. However, for the purpose of achieving overlap between computation and communication, this structure requires alteration to enable the start of data transfers before the completion of all computational elements.

The solution to enable such overlap is relatively straightforward: we partition the

computational loops into two segments. In the first segment, we carry out computations for the layers of data that need to be communicated first. Upon completing the computation of these initial layers, we concurrently initiate the transfer of these layers and proceed with the computation of the remaining elements. This method allows for both computation and communication to be performed at the same time.

To implement this is different depending on if one is computing on the CPU or GPU. On the CPU the method that is chosen is to just split up the nested triple for-loop into 3 nested loops, 1 triple and 2 double. The 2 double loops will each compute their own layer, each starting transfer when it is finished.

The implementation of the previously mentioned overlap technique varies depending on whether the computation is occurring on the CPU or GPU. For CPU computations, the approach involves re-coding the original nested triple for-loop into one triple for-loop and two separate double for-loops. Each of these double loops is responsible for executing computations on a the layer which is to be transferred and initiates the transfer of its data upon completion `listing:15`.

Listing 15: Inter-node CPU Computation Partition

```
// C and MPI code, no overlap, selfmade code
    // Before partitoning of computation
    // Computes the the whole grid
    for(int i = 1; i < depth_node - 1; i++){
        for(int j = 1; j < height - 1; j++){
            for(int k = 1; k < width - 1; k++) {
                int index = k + j * width + i * width * height;
                data_tmp[index] = division * (
                data[index + 1]          + data[index - 1] +
                data[index + width]      + data[index - width] +
                data[index + width*height] + data[index - width*height]);
            }
        }
    }
    // After partitoning the computation
    // Computes the second lowest layer of the grid
```

```
17      for(int j = 1; j < height - 1; j++){
18          for(int k = 1; k < width - 1; k++) {
19              int index = k + j * width + (depth_node-2) * width * height;
20              data_tmp[index] = division * (
21              data[index + 1]          + data[index - 1] +
22              data[index + width]      + data[index - width] +
23              data[index + width*height] + data[index - width*height]);
24          }
25      }
26      // Computes the second layer of the grid
27      for(int j = 1; j < height - 1; j++){
28          for(int k = 1; k < width - 1; k++) {
29              int index = k + j * width + width * height;
30              data_tmp[index] = division * (
31              data[index + 1]          + data[index - 1] +
32              data[index + width]      + data[index - width] +
33              data[index + width*height] + data[index - width*height]);
34          }
35      }
36      // Computes the rest of the grid
37      for(int i = 2; i < depth_node - 2; i++){
38          for(int j = 1; j < height - 1; j++){
39              for(int k = 1; k < width - 1; k++) {
40                  int index = k + j * width + i * width * height;
41                  data_tmp[index] = division * (
42                  data[index + 1]          + data[index - 1] +
43                  data[index + width]      + data[index - width] +
44                  data[index + width*height] + data[index - width*height]);
45              }
46          }
47      }
```

When performing computations on a GPU, the process of partitioning differs due to the nature of GPU kernel execution. Originally, GPU computations are being per-

formed inside a single GPU kernel, named `jacobiMid`, as referenced in `listing:11`. In order to facilitate overlapping computation with communication, it may be necessary to partition the original kernel into three separate kernel functions—one for each outer layer that requires communication and a third for the remaining central layers.

However, in our implementation, we opted to consolidate the computations of the two layers required finished to start communication into a singular kernel function. As a result, we have two primary kernel functions: `jacobiMid` for the central layers and `jacobiEdge` for the outer layers. The alterations to the code, which accommodate these kernel functions and enable the overlapping of computation with communication on a GPU, are detailed in `listing:16`.

Listing 16: GPU Computation Partition

```
1  __global__ void jacobiMid(double *data_gpu, double *data_gpu_tmp, int
       width, int height,
2                        int slices_Leftover, int device_nr, int
                             slices_compute, int elementsPerThread, int
                             elementsLeftover,
3                        int elementsPerThreadExtra, int
                             elementsLeftoverExtra, int overlap_calc){
4      // Using Cooperative Groups
5      cg::grid_group grid_g = cg::this_grid();
6      int thread = grid_g.thread_rank();
7      int thread_size = grid_g.size();
8
9      // Runs the computation
10     calc(data_gpu, data_gpu_tmp, elementsPerThread, thread+overlap_calc,
           width, height, thread, thread_size);
11 }
12
13 __global__ void jacobiEdge(double *data_gpu, double *data_gpu_tmp, int
       width, int height,
14                       int slices_compute, int elementsPerThread, int
```

```
                        elementsLeftover){
15      cg::grid_group grid_g = cg::this_grid();
16      int thread = grid_g.thread_rank();
17      int thread_size = grid_g.size();
18      // There are more threads than elements
19      if(elementsPerThread > 0){
20          if(thread < elementsLeftover){
21              elementsPerThread++;
22          }
23          calc(data_gpu, data_gpu_tmp, elementsPerThread, thread, width,
                height, thread, thread_size);
24          calc(data_gpu, data_gpu_tmp, elementsPerThread, thread +
                (slices_compute+1)*(width-2)*(height-2), width, height, thread,
                thread_size);
25      }
26      // There are less threads than elements in 1 slice
27      else{
28          // There are more threads than half of the elements
29          if(thread_size >= elementsLeftover*2){
30              elementsPerThread++;
31              // Selects all threads with index less than width
32              if(thread < elementsLeftover){
33                  calc(data_gpu, data_gpu_tmp, elementsPerThread, thread,
                        width, height, thread, thread_size);
34              }
35              // Selects all threads with index between width and width*2
36              else if(thread < elementsLeftover+elementsLeftover){
37                  calc(data_gpu, data_gpu_tmp, elementsPerThread, thread +
                        (slices_compute+1)*(width-2)*(height-2), width, height,
                        thread, thread_size);
38              }
39          }
40          else{
41              elementsPerThread++;
```

```
42            if(thread < elementsLeftover){
43                // The same threads will compute both slices
44                calc(data_gpu, data_gpu_tmp, elementsPerThread, thread,
                      width, height, thread, thread_size);
45                calc(data_gpu, data_gpu_tmp, elementsPerThread, thread +
                      (slices_compute+1)*(width-2)*(height-2), width, height,
                      thread, thread_size);
46            }
47        }
48    }
49 }
```

The partitioning of the computation, as described, sets the groundwork for subsequent steps, yet the implementation of communication functions remains to be addressed. These functions are integral to the overall operation of the system and ensure that data is transferred efficiently between the processors.

6.2 CPU to CPU on Different Nodes

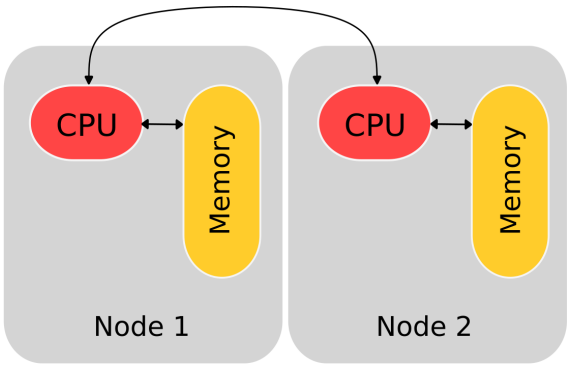

Figure 12: 2 connected nodes, each with their own CPU

The initial scenario we implemented aimed to facilitate the overlapping of communication between CPUs stationed in distinct nodes. To achieve this, we incorporated MPI functions. The setup for this scenario is similar to the basic configuration illustrated in `fig. 12`. Nonetheless, it is important to recognize that the simplistic illustration of this concept to actual hardware configurations is quite different when

it comes to complexity. Real-world implementations often involve additional considerations and other elements that must be carefully managed to ensure successful inter-node communication and computational efficiency.

Listing 17: Inter-node CPU Communication

```
// C and MPI code, no overlap, selfmade code
    // If node_rank is 0 it is only mean to communicate with node_rank 1
    while(iter > 0){
        // <- Computes the grid here
        // Communication
        if(node_rank == 0){
            MPI_Isend(&data_tmp[width*height*(depth_node-2)], width*height,
                MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[0]);
            MPI_Irecv(&data_tmp[width*height*(depth_node-1)], width*height,
                MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[1]);
        }
        else if(node_rank == size-1){
            MPI_Irecv(&data_tmp[0],                      width*height,
                MPI_DOUBLE, node_rank-1, 0, MPI_COMM_WORLD, &myRequest[0]);
            MPI_Isend(&data_tmp[width*height],           width*height,
                MPI_DOUBLE, node_rank-1, 0, MPI_COMM_WORLD, &myRequest[1]);
        }
        else{
            MPI_Irecv(&data_tmp[0],                      width*height,
                MPI_DOUBLE, node_rank-1, 0, MPI_COMM_WORLD, &myRequest[0]);
            MPI_Isend(&data_tmp[width*height],           width*height,
                MPI_DOUBLE, node_rank-1, 0, MPI_COMM_WORLD, &myRequest[1]);
            MPI_Isend(&data_tmp[width*height*(depth_node-2)], width*height,
                MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[2]);
            MPI_Irecv(&data_tmp[width*height*(depth_node-1)], width*height,
                MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[3]);
        }
        MPI_Waitall((node_rank == 0 || node_rank == size - 1 ? 2 : 4),
            myRequest, myStatus);
        // <- Iteration reduction and pointer swapping
```

```
22      }
```

`listing:17` shows the implementation of a non-overlapping program structure that distinguishes computing and data transfer into distinct sequential steps. It first completes the entire computation phase, then proceeds to data transfer, and finally synchronizes all processes before starting on a new iteration. To transition from this sequential approach to one where CPU-to-CPU communication is overlapped with computation, some modifications are necessary.

The principal change is to divide the computational tasks into two discrete segments, as indicated in `listing:15`. The initial segment specifically addresses the calculation of the upper and lower layers of the grid—these are the areas of data that will be involved in the first communication. Once this partial computation is finished, communication can begin immediately, capitalizing on the ability to transfer these data while the remaining computation continues. Meanwhile, the second segment of the computation undertakes the processing of the interior elements of the grid. The finished implementation will in the end look like `listing:18`.

Listing 18: Inter-node CPU Communication With Overlap

```
1  // C og MPI code, overlap, selfwritten code
2      while(iter > 0){
3          // <- Computes second highest layer here
4          if(node_rank != size-1){
5              // Starts sending and recieving data asynchronously to
                   neighboring node with rank size+1
6              MPI_Isend(&data_tmp[width*height*(depth_node-2)], width*height,
                   MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[0]);
7              MPI_Irecv(&data_tmp[width*height*(depth_node-1)], width*height,
                   MPI_DOUBLE, node_rank+1, 0, MPI_COMM_WORLD, &myRequest[1]);
8          }
9          // <- Computes the second lowest layer here
10         if(node_rank != 0){
11             // Starts sending and recieving data asynchronously to
                   neighboring node with rank size-1
```

```
12          MPI_Irecv(&data_tmp[0],                      width*height,
                MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &myRequest[2]);
13          MPI_Isend(&data_tmp[width*height],           width*height,
                MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &myRequest[3]);
14      }
15      // <- Computes the rest of the elements here
16      MPI_Waitall(node_rank == 0 || node_rank == size - 1 ? 2 : 4,
            myRequest, myStatus);
17       // <- Iteration reduction and pointer swapping
18  }
```

6.3 GPU to GPU Within a Node

The next scenario to consider focuses on overlapping communication between GPUs within the same node, which also requires more change to the code compared to CPU-to-CPU communication. In this GPU focused environment, new elements come into play, such as the implementation of GPU kernels and the management of memory allocation and initialization on the GPUs. Like for the CPU to CPU communication you can view the simplistic illustration in `figure 13` which gives a brief overview of which components are in action.

While the fundamental principles of computation and communication overlap persist, the execution is now distributed to the GPUs, exploiting their capability for parallel processing. This handover of computing duties to GPUs promises an improvement in performance, thanks to the high throughput of these specialized processors.

Additionally, the adoption of a heterogeneous computing architecture, consisting of both CPUs and GPUs, introduces the challenge of managing inter-processor communication. This component adds a layer of complexity compared to a code written for a purely CPU-driven setup. Such complexity is more noticeable during memory operations, as effective data transfer between CPU and GPU memories—and indeed between different GPUs is important to maintaining the efficiency and speed of the

combined system. Attention must be paid to ensure that data is correctly synchronized across different computation units while mitigating the overhead incurred by these data movements.

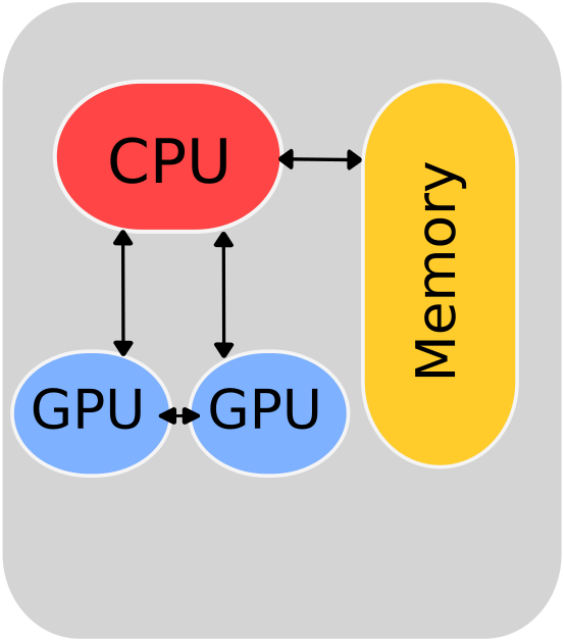

Figure 13: A single node with both a CPU and 2 GPUs

To evaluate any potential performance gains from overlapping communication, we constructed two separate versions of the code. One version executes computations followed by communications in a serial manner, while the other initiates communication concurrently with computation, as portrayed in `fig. 7`.

Presented in `listing:19` is the non-overlapping code variant, which adheres to a straightforward four-step procedure. The first step involves computing on the grid utilizing the `cudaLaunchCooperativeKernel` to invoke the `jacobiMid` device kernel, which implements the Cooperative Groups framework. Upon completion of these calculations, the next phase—data communication—commences. This includes a single layer of communication for the first and last GPUs, and two layers for any intermediate GPUs. These communications are conducted asynchronously, meaning they are initiated and run concurrently. In the third step, we ensure that all computations and data transfers have concluded before proceeding further. The final step involves toggling the device memory pointers to swap between grids—a necessary step to prepare for the next iteration.

Listing 19: Intra-node GPU Communication

```
// No overlapping of communication and computation, selfmade code
    while(iter > 0){
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                gridDim, blockDim, kernelCollMid[g], 0, streams[g][0]));
            cudaErrorHandle(cudaEventRecord(events[g][0], streams[g][0]));
        }
        // Transfers the second layer of the matrix
        for(int g = 1; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaStreamWaitEvent(streams[g][1],
                events[g][0]));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g-1] +
                (slices_device[g-1]-1)*width*height, g-1, data_gpu_tmp[g] +
                width*height, g, width*height*sizeof(double),
                streams[g][1]));
            cudaErrorHandle(cudaEventRecord(events[g][1], streams[g][1]));
        }
        // Transfers the second last layer of the matrix
        for(int g = 0; g < gpus-1; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaStreamWaitEvent(streams[g][1],
                events[g][0]));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g+1], g+1,
                data_gpu_tmp[g] + (slices_device[g]-2)*width*height, g,
                width*height*sizeof(double), streams[g][1]));
            cudaErrorHandle(cudaEventRecord(events[g][2], streams[g][1]));
        }
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaEventSynchronize(events[g][0]));
            cudaErrorHandle(cudaEventSynchronize(events[g][1]));
```

```
26            cudaErrorHandle(cudaEventSynchronize(events[g][2]));
27        }
28        // <- Iteration reduction and pointer swapping
29    }
```

In the non-overlapping code implementation, as seen in `listing:20`, the program follows a linear sequence: compute, communicate and then synchronize. However, the introduction of overlapping requires modifications in the code to allow compute and communication operations to occur concurrently. In the overlapped scenario, computation is split into two distinct parts as explained in `listing:16`.

Here, it is worth highlighting that in the version of the `jacobiEdge` function, both the top and bottom layers of the matrix are computed in one go before any communication begins. While this method is functional, it is not the most optimized. A more sophisticated implementation would split these computations further into two distinct functions, assigning a unique CUDA event handle to each. This would allow for starting the transfer of a single layer as soon as its computation completes, rather than waiting for both to be ready. Despite the potential for improved efficiency offered by this dual-function strategy, the `jacobiEdge` function was designed for simplicity, computing both layers prior to initiating the transfer.

Listing 20: Intra-node GPU Communication With Overlap

```
1  // Overlapping communication and computation, selfmade code
2      while(iter > 0){
3          for(int g = 0; g < gpus; g++){
4              cudaErrorHandle(cudaSetDevice(g));
5              // Computes the upper and lower layer
6              cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiEdge,
                   gridDim, blockDim, kernelCollEdge[g], 0, streams[g][0]));
7              cudaErrorHandle(cudaEventRecord(events[g][0], streams[g][0]));
8              // Computes the rest of the layers
9              cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                   gridDim, blockDim, kernelCollMid[g], 0, streams[g][1]));
10             cudaErrorHandle(cudaEventRecord(events[g][1], streams[g][1]));
```

```
11      }
12      // The rest of the code is similar to the Listing 15: Intra-node
            GPU Communication
13  }
```

The modifications required to introduce overlapping in the code are relatively minimal it seems. The primary difference in the overlapping implementation arises from the point at which communications are initiated. Communications now commence upon the completion of the `jacobiEdge` kernel, as signaled by its associated CUDA event, rather than after the completion of the `jacobiMid` kernel. Additionally, the advent of the `jacobiEdge` kernel represents the most significant change to the codebase, necessitating an implementation to handle the computation of the edge layers, which facilitates the overlap of communication and computation.

6.4 GPU to GPU on Different Nodes

As we progress to complex scenarios, we integrate all previously discussed implementations to utilize computation on GPUs while facilitating communication both between nodes and within GPUs. It is important to recognize that the computational workload is entirely managed by GPUs after the data has been allocated; the CPUs are no longer responsible for direct computation but serve as potential helpers for data movement.

However before starting it is worth noting that the communication portion can be implemented via different methods, each with its distinctive advantages and suitable use cases. In this thesis, we will explore two such methods, examining how they operate within our system and analyzing their impact on performance and efficiency. The figure 14 gives an overview of the potential components which can be used.

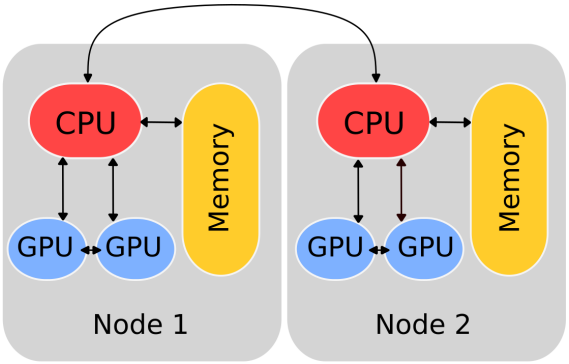

Figure 14: 2 connected nodes, each with their own CPU and GPU

6.4.1 Communicating Through CPU

The first approach combines the communication strategies we have already developed, enabling GPUs within the same node to communicate directly. However, if a GPU needs to communicate with another GPU on a different node, the data must be routed through their respective CPUs. This method may be viewed as the most intuitive step forward from previous implementations, especially given that we have individually implemented intra-node and inter-node communications. In this context, we merely need to integrate these established methodologies, as outlined in the sample code presented under `listing:21`.

This communication pathway does however require the introduction of an additional communication pathway between the CPUs and GPUs, an pathway that was not accounted for in previous iterations where communication was contained to either between CPUs on different nodes or between GPUs on the same node. To facilitate the necessary data movement between CPUs and GPUs, a simple implementation of the `cudaMemcpyAsync` function is utilized. This versatile function can transfer data in either direction—CPU to GPU or GPU to CPU—based on the specified parameters.

Listing 21: Multi-node Communication

```
//selwritten code
    while(iter > 0){
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                gridDim, blockDim, kernelMid[g], 0, streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][0], streams[g]));
        }
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaEventSynchronize(events[g][0]));
            cudaErrorHandle(cudaStreamWaitEvent(streams[g], events[g][0]));
        }
        // GPU to GPU
        for(int g = 1; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g-1] +
                (slices_device[g-1]-1)*width*height, g-1, data_gpu_tmp[g] +
                width*height, g, width*height*sizeof(double), streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][1], streams[g]));
        }
        for(int g = 0; g < gpus-1; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g+1], g+1,
                data_gpu_tmp[g] + (slices_device[g]-2)*width*height, g,
                width*height*sizeof(double), streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][2], streams[g]));
        }
        // GPU to CPU
        cudaErrorHandle(cudaSetDevice(gpus-1));
        cudaErrorHandle(cudaMemcpyAsync(data_cpu, data_gpu_tmp[gpus-1] +
            (slices_device[gpus-1]-2)*width*height,
            width*height*sizeof(double), cudaMemcpyDeviceToHost,
            streams[gpus-1]));
```

```
26        cudaErrorHandle(cudaEventRecord(events[gpus-1][0],
              streams[gpus-1]));
27        cudaErrorHandle(cudaSetDevice(0));
28        cudaErrorHandle(cudaMemcpy(data_cpu + width*height, data_gpu_tmp[0]
              + width*height, width*height*sizeof(double),
              cudaMemcpyDeviceToHost));
29        for (int g = 0; g < gpus; g++) {
30            cudaErrorHandle(cudaStreamWaitEvent(streams[g],
                  events[gpus-1][0]));
31        }
32        // CPU to CPU
33        if(node_rank == 0){
34            MPI_Isend(&data_cpu[0],           width*height, MPI_DOUBLE,
                  node_rank+1, 0, MPI_COMM_WORLD, &myRequest[0]);
35            MPI_Irecv(&data_cpu[3*width*height], width*height, MPI_DOUBLE,
                  node_rank+1, 0, MPI_COMM_WORLD, &myRequest[1]);
36        }
37        else if(node_rank == size-1){
38            MPI_Isend(&data_cpu[width*height], width*height, MPI_DOUBLE,
                  node_rank-1, 0, MPI_COMM_WORLD, &myRequest[0]);
39            MPI_Irecv(&data_cpu[2*width*height], width*height, MPI_DOUBLE,
                  node_rank-1, 0, MPI_COMM_WORLD, &myRequest[1]);
40        }
41        else{
42            MPI_Irecv(&data_cpu[2*width*height], width*height, MPI_DOUBLE,
                  node_rank-1, 0, MPI_COMM_WORLD, &myRequest[0]);
43            MPI_Isend(&data_cpu[width*height], width*height, MPI_DOUBLE,
                  node_rank-1, 0, MPI_COMM_WORLD, &myRequest[1]);
44
45            MPI_Isend(&data_cpu[0],           width*height, MPI_DOUBLE,
                  node_rank+1, 0, MPI_COMM_WORLD, &myRequest[2]);
46            MPI_Irecv(&data_cpu[3*width*height], width*height, MPI_DOUBLE,
                  node_rank+1, 0, MPI_COMM_WORLD, &myRequest[3]);
47        }
```

```
48          MPI_Waitall(number_of_requests, myRequest, myStatus);
49          // CPU to GPU
50          cudaErrorHandle(cudaSetDevice(gpus-1));
51          cudaErrorHandle(cudaMemcpyAsync(data_gpu_tmp[gpus-1] +
                (slices_device[gpus-1]-1)*width*height, data_cpu +
                3*width*height, width*height*sizeof(double),
                cudaMemcpyHostToDevice, streams[gpus-1]));
52          cudaErrorHandle(cudaEventRecord(events[gpus-1][0],
                streams[gpus-1]));
53          cudaErrorHandle(cudaSetDevice(0));
54          cudaErrorHandle(cudaMemcpy(data_gpu_tmp[gpus-1], data_cpu +
                2*width*height, width*height*sizeof(double),
                cudaMemcpyHostToDevice));
55          for (int g = 0; g < gpus; g++) {
56              cudaErrorHandle(cudaStreamWaitEvent(streams[g],
                    events[gpus-1][0]));
57          }
58          for(int g = 0; g < gpus; g++){
59              cudaErrorHandle(cudaSetDevice(g));
60              cudaErrorHandle(cudaEventSynchronize(events[g][0]));
61              cudaErrorHandle(cudaEventSynchronize(events[g][1]));
62              cudaErrorHandle(cudaEventSynchronize(events[g][2]));
63          }
64          // <- Iteration reduction and pointer swapping
65      }
```

This code implementation does however introduce the necessity of having to perform several communication transfers each iteration, not just one data transfer function call like the previous which has been between either GPUs or nodes. Now, this problem only concerns the GPUs which has to transfer and receive data from another node, but in computing like many other parts of life the slowest part is often the one that matters. The fact that it has to perform 3 different data transfers can imply that data transfers plays a bigger role when it comes to absolute

communication time.

The second implementation that includes overlap would in that case be more relevant as one can expect the communication time to be more similar to computation, and as previously showed, the more equal the execution time of those two actions are, the more one can be rewarded by implementing overlap. To achieve overlapping communication and computation the code below has been developed `listing:22`

Listing 22: Node GPUs overlap Example

```
// If node_rank != 0 && node_rank != size-1
    while(iter > 0){
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            // Computes the upper and lower slice
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiEdge,
                gridDim, blockDim, kernelCollEdge[g], 0, streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][0], streams[g]));
            // Computes the rest of the slices
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                gridDim, blockDim, kernelMid[g], 0, streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][1], streams[g]));
        }
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaEventSynchronize(events[g][0]));
            cudaErrorHandle(cudaStreamWaitEvent(streams[g], events[g][0]));
        }
        for(int g = 1; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g-1] +
                (slices_device[g-1]-1)*width*height, g-1, data_gpu_tmp[g] +
                width*height, g, width*height*sizeof(double), streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][1], streams[g]));
        }
        // Transfers n-2 slice of the matrix
        for(int g = 0; g < gpus-1; g++){
```

```
23            cudaErrorHandle(cudaSetDevice(g));
24            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g+1], g+1,
                  data_gpu_tmp[g] + (slices_device[g]-2)*width*height, g,
                  width*height*sizeof(double), streams[g]));
25            cudaErrorHandle(cudaEventRecord(events[g][2], streams[g]));
26        }
27        // The rest of the code is similar to the Listing 21: Multi-node
              Communication
28    }
```

When overlap is implemented, the complexity of the code examples increases significantly, potentially leading to confusion and a loss of clarity. However, the fundamental distinction between overlapping and non-overlapping implementations comes down to the segmentation of the main computational phase. In the overlapping scenario, the primary computation is divided to allow for concurrent operations.

6.4.2 Direct GPU Communication

The final scenario presented in the code incorporates CUDA-aware MPI to facilitate Direct Memory Access (DMA) transfers, thereby enabling direct communication between GPU memories across nodes and bypassing the CPU entirely. This approach not only decreases the quantity of data transfer operations for GPUs that communicate with GPUs on different nodes but also liberates the CPU from handling GPU-related communication tasks. As a result, the CPU can remain undisturbed by the overhead typically associated with orchestrating GPU data transfers.

Additionally, utilizing CUDA-aware MPI for DMA transfers simplifies the communication code substantially. With this technique, we are able to bypass the additional step of CPU-mediated communication entirely, resulting in a more streamlined and efficient transfer process as shown in `listing:23`. The ability to directly transfer data from the memory of one GPU to another cam minimize latency and improve throughput, ultimately contributing to improved performance in scenarios with significant inter-node GPU communication.

Listing 23: Node GPUs Cuda-aware MPI Example

```
else{
    while(iter > 0){
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                gridDim, blockDim, kernelMid[g], 0, streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][0], streams[g]));
        }
        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaEventSynchronize(events[g][0]));
            cudaErrorHandle(cudaStreamWaitEvent(streams[g], events[g][0]));
        }
        for(int g = 1; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g-1] +
                (slices_device[g-1]-1)*width*height, g-1, data_gpu_tmp[g] +
                width*height, g, width*height*sizeof(double), streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][1], streams[g]));
        }
        for(int g = 0; g < gpus-1; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaMemcpyPeerAsync(data_gpu_tmp[g+1], g+1,
                data_gpu_tmp[g] + (slices_device[g]-2)*width*height, g,
                width*height*sizeof(double), streams[g]));
            cudaErrorHandle(cudaEventRecord(events[g][2], streams[g]));
        }
        MPI_Isend(data_gpu_tmp[gpus-1] + width*height*(depth_node-2),
            width*height, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD,
            &myRequest[0]);
        MPI_Irecv(data_gpu_tmp[gpus-1] + width*height*(depth_node-1),
            width*height, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD,
            &myRequest[1]);
        MPI_Isend(data_gpu_tmp[0] + width*height, width*height, MPI_DOUBLE,
```

```
            rank-1, 0, MPI_COMM_WORLD, &myRequest[2]);
25        MPI_Irecv(data_gpu_tmp[0]           , width*height, MPI_DOUBLE,
            rank-1, 0, MPI_COMM_WORLD, &myRequest[3]);
26        MPI_Waitall(4, myRequest, myStatus);
27        for(int g = 0; g < gpus; g++){
28            cudaErrorHandle(cudaSetDevice(g));
29            cudaErrorHandle(cudaEventSynchronize(events[g][0]));
30            cudaErrorHandle(cudaEventSynchronize(events[g][1]));
31            cudaErrorHandle(cudaEventSynchronize(events[g][2]));
32        }
33        for(int g = 0; g < gpus; g++){
34            double *data_change = data_gpu[g];
35            data_gpu[g] = data_gpu_tmp[g];
36            data_gpu_tmp[g] = data_change;
37        }
38        iter--;
39    }
40 }
```

Transforming the code to run overlap using DMA transfers with CUDA-aware MPI is conceptually similar to the approach we previously used, which is listed in `listing:17`. The main difference is that in this current scenario, the communication is streamlined to make use of CUDA-aware MPI functions instead of regular MPI functions.

In essence, the overlap in this CUDA-aware MPI context operates under the same principles as in the previous listings: it leverages the inherent non-blocking behavior of MPI calls. However, because these calls are now cognizant of CUDA's memory model, data can be transferred directly between GPU memories without CPU intervention. This optimization leads to an even more efficient overlap, potentially alleviating bottlenecks that could arise from the CPU to GPU memory copy steps.

Listing 24: Node GPUs CUDA-aware MPI overlap Example

```
1   else{
2       while(iter > 0){
3           for(int g = 0; g < gpus; g++){
4               cudaErrorHandle(cudaSetDevice(g));
5               // Computes the upper and lower layer
6               cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiEdge,
                    gridDim, blockDim, kernelCollEdge[g], 0, streams[g]));
7               cudaErrorHandle(cudaEventRecord(events[g][0], streams[g]));
8               // Computes the rest of the layers
9               cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobiMid,
                    gridDim, blockDim, kernelMid[g], 0, streams[g]));
10              cudaErrorHandle(cudaEventRecord(events[g][1], streams[g]));
11          }
12          for(int g = 0; g < gpus; g++){
13              cudaErrorHandle(cudaEventSynchronize(events[g][0]));
14              cudaErrorHandle(cudaStreamWaitEvent(streams[g],
                    events[g][0]));
15          }
16          // The rest of the code is similar to the Listing 23: Node GPUs
                Cuda-aware MPI Example
17      }
18  }
```

7 Results and Discussion

Within this thesis, the experimental programs are executed on three distinct partitions of the eX3 supercomputing infrastructure, made available to me by the Simula Research Laboratory[26]. Two of these partitions, named dgx2q and hgx2q, are designated for intra-node communication tests exclusively; this is because each partition comprises only a single node, excluding the possibility of performing computations that span multiple nodes. Alternatively the third partition, named a100q, encompasses two nodes, thereby allowing the execution of inter-node communication tests. This multi-node setup allows the investigation of communication dynamics and performance across distinct computing units within the eX3 supercomputing environment.

Partitions			
Partition name	CPU per Node	GPUs per Node	Nodes
a100q	AMD EPYC Milan 7763 64-core	2 x NVIDIA Volta A100/40GB	2
dgx2q	Intel Xeon Scalable Platinum 8176	16 x NVIDIA Volta V100	1
hgx2q	AMD EPYC Milan 7763 64-core	8 x NVIDIA Volta A100/80GB	1

Table 1: Types of partitions used for tests

The results will be presented in a way where we first see if the computation and communication of the program is implemented correctly through NVIDIA's Nsight Systems[27], which we will explain in the next subsection.

We will examine the outcomes of our experiments in a sequence that follows the progression of our implementation strategy, ensuring a sound presentation of results. Firstly, we will assess the findings associated with CPU-to-CPU communication across distinct nodes, highlighting the behavior and performance characteristics of this inter-node data transfer paradigm.

Subsequently, our focus will shift to the results gathered from GPU-to-GPU communication within the confines of a single node, an analysis that delves into the dynamics of intra-node communication and how the proximity of these processors

affects the communication efficiency and latency.

Finally, we will view the results computed from both CPU-to-CPU and GPU-to-GPU communications by exploring the combination of these methods. This examination will provide an overview of the system's communication capabilities, showing the performance when all forms of communication paths are employed concurrently.

All of the scenarios are computed on the a100q partition, with the exception of communication between intra-node GPUs, which is conducted on the dgx2q and hgx2q partitions.

7.1 Profiling

The usage of NVIDIA's Nsight Systems[27] in this implementation is motivated by its user-friendly interface and the graphical representation it provides. This visualization facilitates an easy understanding of the parallel activities of the system's various processors.

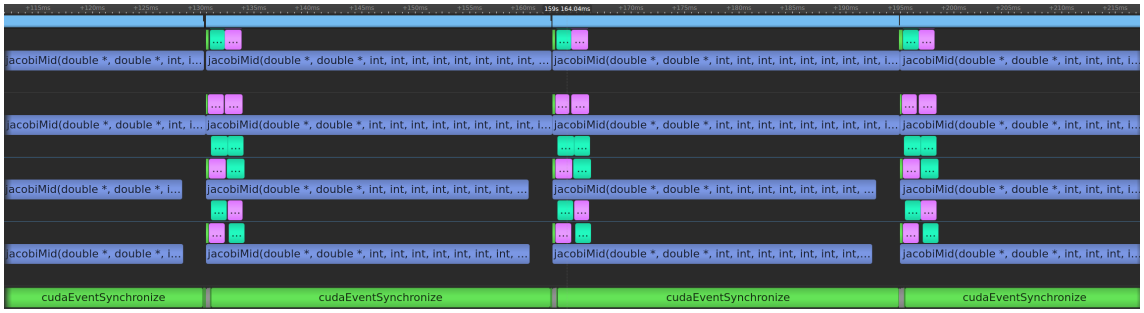

Figure 15: Example of NVIDIA Nsight System output for a program launching asynchronous operations.

The graphic in `Figure 15` displays an output from a CUDA program captured with the `nvprof` tool enabled. The scenario illustrated shows a program executing on four separate devices, all located within a single node. Each device repeatedly runs a kernel named `jacobiMid`, with at least four iterations evident in the given visualization. A synchronization step using `cudaEventSynchronize` is used as a barrier for all devices before they embark on the next iteration of `jacobiMid`. Moreover, the devices actually start their execution sequence with a kernel called `jacobiEdge`, symbolized by the very brief bar intervals above the `jacobiMid` blocks

at the beginning, which are to short to show any indication of what job it performs.

Additional activities carried out by the devices include communication tasks, denoted by the bars marked with ellipses. Normally, these would be labeled with "Memcpy DtoH" or "Memcpy HtoD," signifying the operation `cudaMemcpyPeerAsync`. A difference in communication bars is observed among the devices: the outermost devices, layerd at the top and bottom of the graphic, each conduct two communications, while the intermediate devices engage in four.

Figure 16: Example of NVIDIA Nsight System output for a program launching with non-overlapping operations.

Our objective in examining the profiling results is to verify that for scenarios with non-overlapping computation, all memory transfers occur subsequent to the completion of the computation. This desired pattern is exemplified in `figure 16`, where the sequence of operations aligns with the execution strategy we intended to implement. Compare this to the pattern shown in `figure 15` which has introduced overlap and performs several operations asynchronous.

Each iteration in the non-overlapping profile kicks off with the computation of the `jacobiMid` kernel, which is responsible for the computational workload. Upon the completion of this kernel's operations, we start the process of communication between the various devices located within the same node.

The profiling results should show a clear division between computation and communication phases when not overlapping, with computation consistently occurring first, followed by communication, and concluding with synchronization. In the visualization, the `cudaEventSynchronize` bars mark the points at which the program enforces barriers, pausing until the CUDA threads complete their tasks. The initial `cudaEventSynchronize` bar indicates the waiting period for the `jacobiMid` kernel to finish before communication begins. The subsequent bar reflects the duration

required for communication to conclude and start of a new iteration.

7.2 Heatmaps

Heatmaps serve as a visual tool and can be used to illustrate the impact of implementing overlap into different scenarios. These heatmaps are generated using Python, and in our plots they are designed around that the dimensions of the grid layers are represented on the y-axis of the heatmap in an "axb" notation, where 'a' corresponds to the grid's width and 'b' to its height. The depth of the grid, or the number of layers, is denoted along the x-axis.

The values are calculated from executing the program twice, first without overlap and then with it. The heatmap compares the resulting runtimes and assigns colors to each grid within the heatmap based on the magnitude of the time difference. A greater disparity between the two runtimes results in a higher percentage value, which is reflected in the intensity of the color on the heatmap. Refering to `figure 17`, the brighter the color, the more greater the performance improvement achieved through overlapping percentage wise.

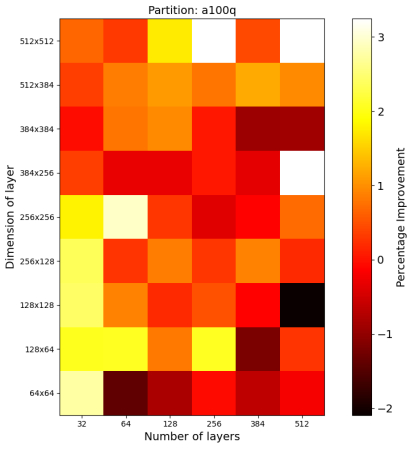

Figure 17: Example of a heatmap used on grids ranging from (64, 64, 64) to (512, 512, 512)

Using heatmaps is very helpful for showing the good effects of overlap. Heatmaps show data with colors, so it's easy to see changes quickly. When we look at groups of grids of almost the same size, a heatmap helps spot which ones work better with overlap. If a group gets a big boost in performance, the heatmap will show this area

with a brighter color. This makes it simple to compare different sizes and see where overlap helps the most. For people reading the research, the bright colors on the heatmap make it easier to understand where the big changes are without complex words or lots of numbers. If the heatmap contains an element which has no data assigned to it, the element will be colored completely white.

7.2.1 Expected results

The design of the grids ensures that computation increases regardless of which axis we expand, whereas communication only increases when we enlarge the grid's width or height. As a result, we can anticipate that communication time will be affected solely by increases in width or height. In contrast, computation time will rise with expansion along any axis. Consequently, when examining the heatmap, it might initially appear that communication escalates with the dimension of layers, while computation only heightens with the number of layers. However, this perception is incorrect. In reality, although computation does indeed increase with the number of layers, it also escalates when the dimensions of the layers are augmented.

Consequently, along the x-axis of the heatmap, we can expect to witness a clear tipping point. To the left of this point, communication tends to be faster, while to the right, computation takes longer. The exact location of this tipping point on the x-axis is influenced by how the relationship between communication and computation evolves as the size of each layer increases. If the rates of increase for both are similar, the tipping point will likely remain at a consistent position on the x-axis. However, if the rate of increase for computation significantly outpaces communication, or vice versa, we will see the tipping point shift and possibly form an angled trajectory on the heatmap, demonstrating how the tipping point moves along the x-axis with changes in the layer size.

7.3 CPU to CPU on Different Nodes Results

The nodes which we are working on are connected through InfiniBand connection which has bandwidth of `table 2` which has been created with `RDMA_Write BW Test` through the commands `ib_write_bw -d mlx5_2 -a -F -report_gbits -q 1 n014.ib.cluster`.

Table 2: Performance metrics based on the number of bytes and iterations.

#bytes	#iterations	BW peak [Gb/sec]	BW average [Gb/sec]	MsgRate [Mpps]
2	5000	0.0378	0.0347	2.1663
4	5000	0.0757	0.0748	2.3377
8	5000	0.1500	0.1500	2.3454
16	5000	0.3000	0.3000	2.3715
32	5000	0.6100	0.6000	2.3469
64	5000	1.2000	1.0700	2.0948
128	5000	2.4200	2.3700	2.3131
256	5000	4.8000	4.7900	2.3388
512	5000	9.5100	9.5100	2.3209
1024	5000	18.500	17.440	2.1294
2048	5000	30.830	30.460	1.8594
4096	5000	30.840	27.980	0.8538
8192	5000	150.34	48.920	0.7465
16384	5000	153.21	70.670	0.5392
32768	5000	194.27	105.87	0.4039
65536	5000	196.05	141.34	0.2696
131072	5000	195.72	152.72	0.1456
262144	5000	195.61	173.70	0.0828
524288	5000	196.06	185.51	0.0442
1048576	5000	195.89	190.49	0.0227
2097152	5000	195.84	194.15	0.0116
4194304	5000	195.78	195.24	0.0058
8388608	5000	196.08	196.08	0.0029

The combination of high bandwidth and using single CPU core in this scenario leads to a very large difference between the time it takes to transfer data and the time it takes to compute the data. To illustrate this gap, we conducted two separate runs of the same code on identical grid sizes: one focusing solely on communication and the other on computation. The results, presented in Table 3, highlight the marked difference between computation and communication times.

Given the substantial difference here, where communication times are orders of

Grid Size	Computation Time	Communication Time
(256, 256, 32)	6.06323	0.00144
(256, 256, 68)	14.36730	0.00073
(256, 256, 128)	32.28652	0.00059

Table 3: Benchmark results for various grid sizes

magnitude lower than computation times, the overall potential for speedup through overlap is severely limited. The execution time is thus predominantly governed by computational workload. As such, the theoretical upper limit for performance gain due to overlapping becomes small, leading to an insignificant impact on the total runtime.

Moreover, the output quality, as evidenced in `figure 18`, is potentially more influenced by variance in computational processing than the benefits derived from overlapping. Consequently, the output `figure 18`, instead of presenting an improved performance landscape due to overlapping, can appear scattered and inconsistent.

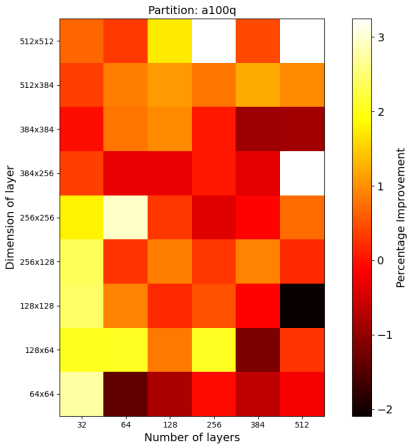

Figure 18: Performance improvement of implementing overlap for CPU to CPU communication on different nodes

7.4 GPU to GPU Within a Node Results

Figure 19: Result for no overlapping computation for an (512, 512, 512) grid with 4 GPUs

Firstly, let us examine whether the program that has implemented a non-overlapping method follows the intended pattern. This is depicted in `figure 19`, where we have profiled the execution of a grid with dimensions (512, 512, 512) distributed across four different GPUs. From the profiling data, we observe that the sequence of operations follows our expectations. The process completes the computation of the entire grid before initiating data transfer. This sequential approach highlights the program's operational pattern, which distinctly separates computation phases from communication tasks.

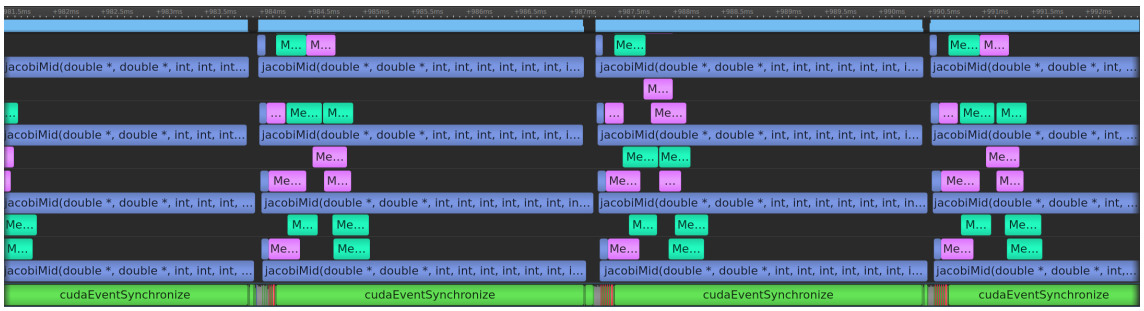

Figure 20: Result for overlapping computation for an 512, 512, 512 grid with 4 GPUs

Next, we turn our inspect the profiler results for an implementation that includes overlapping of computation and communication, as illustrated in `figure 20`. As anticipated, the profiler data reveals that communication is run in parallel to the computation. This indicates that the overlapping implementation is successfully reducing idle time for the processors that would otherwise be occurring during communication phases. As a result, we can confirm that the overlapping approach has been implemented effectively.

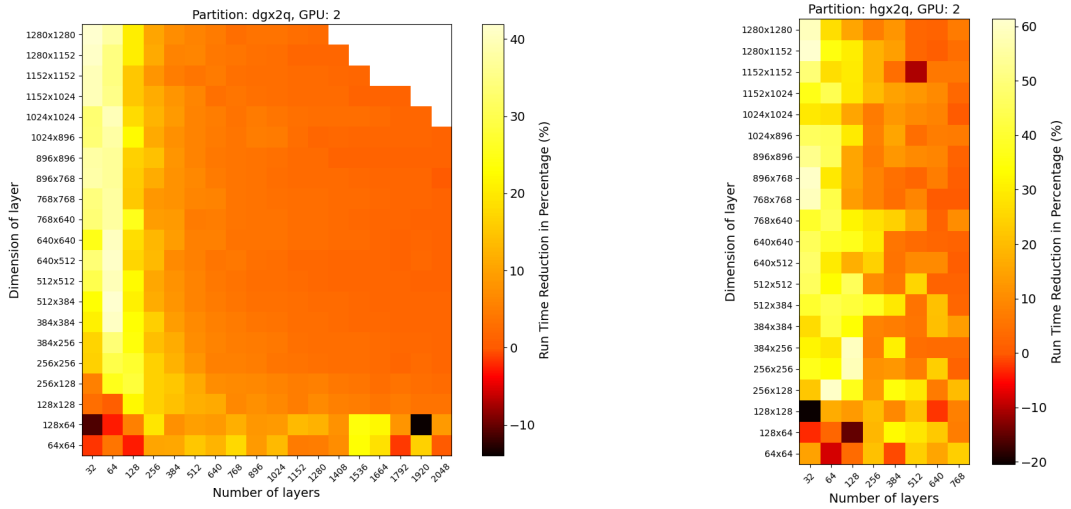

Figure 21: A table over which grid sizes yields the most performance boost when implementing overlap for 2 GPUs

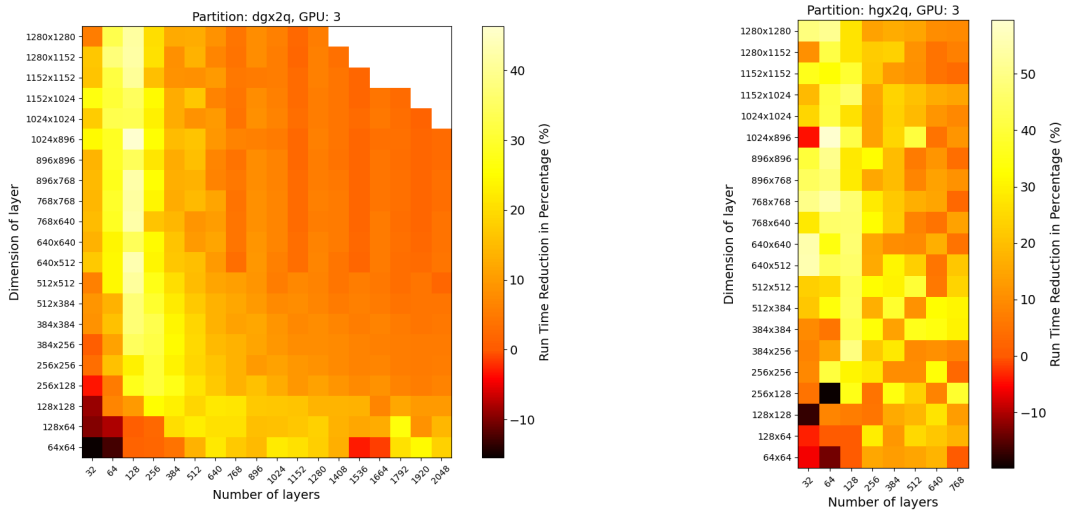

Figure 22: A table over which grid sizes yields the most performance boost when implementing overlap for 3 GPUs

The heatmaps shown `figure 21`, `figure 22`, `figure 23` detail the performance improvements observed for different grid sizes when implementing overlap with 2, 3, or 4 GPUs, respectively. These tables highlight that certain grid sizes benefit more from overlap, suggesting that the effectiveness of implementing overlap may depend on the specific computational workload and data transfer sizes.

Additionally, it can be observed that when the grid size is too small, compared to the processing power or communication bandwidth, implementing overlap can actually degrade performance. This can be attributed to several factors, notably the extra kernel calls and synchronization functions that must be added. These

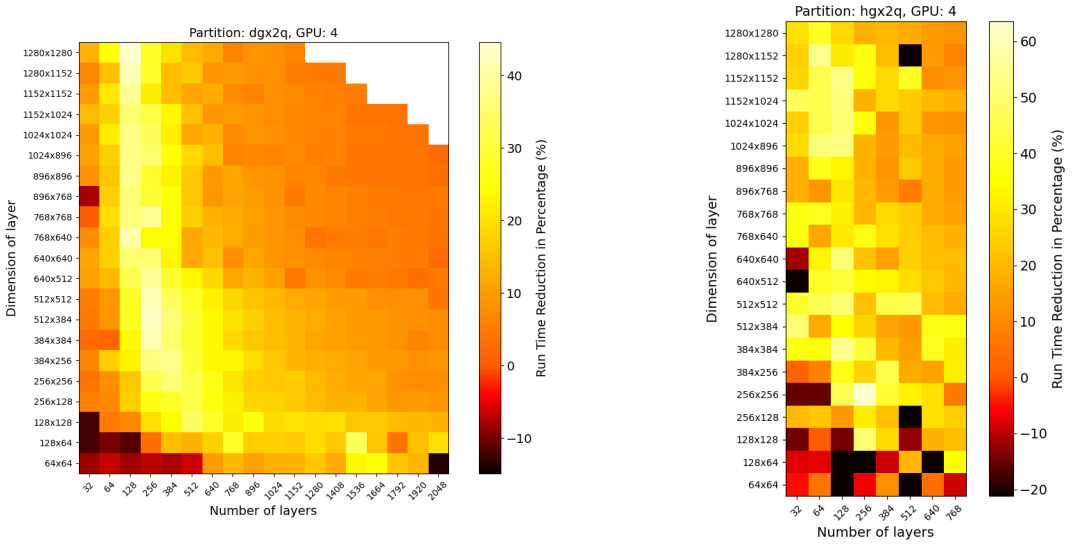

Figure 23: A table over which grid sizes yields the most performance boost when implementing overlap for 4 GPUs

additional operations can introduce overhead, negating the benefits of overlapping computation and communication.

However, most values are primarily found in three distinct clusters. The first cluster appears when the grid is so small that other factors, such as overhead, start to affect the runtime, or when the communication run duration exceeds that of the computation. This cluster is located to the left and towards the bottom of the heatmaps. The second cluster emerges when the communication and computation components of the program take almost equal time to complete. This scenario is where implementing overlap yields the greatest benefits, as indicated by the brightest yellow on the heatmap. The third and largest cluster forms when computation begins to take longer than communication.

It can also be observed that as more GPUs are incorporated into the program, the second cluster shifts further to the right. This shift is due to the increase in computational power. As more processors are added, the computation becomes faster relative to the communication. Consequently, the processors must handle more layers to achieve the same running time as the communication. This results in the overlap benefit cluster shifting rightward on the heatmap, illustrating the change in balance between communication and computation times.

7.5 GPU to GPU on Different Nodes Results

As with implementation we are measuring the results from two different methods of communication with the same hardware, one communicates directly through the CPUs, while the other communicates with the help of CUDA-aware MPI that allows it to bypass going through the memory of the CPUs.

7.5.1 Communicating through CPU memory

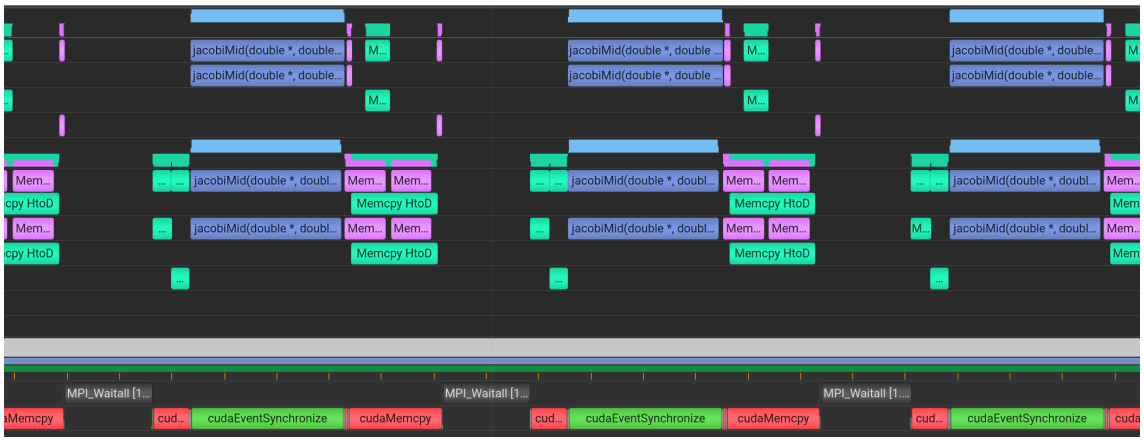

Figure 24: Result for no overlapping computation for an 512, 512, 512 grid with 2 nodes, each with 2 GPUs

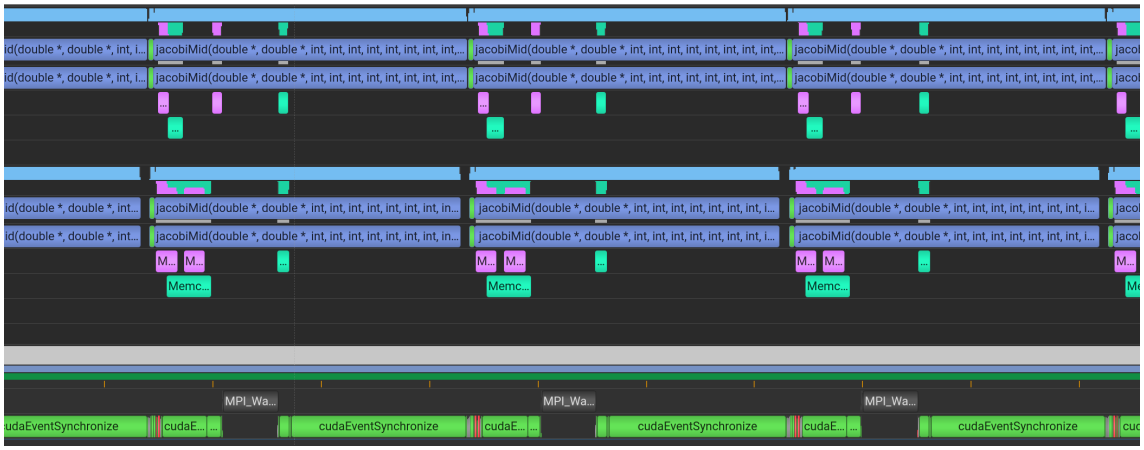

Figure 25: Result for overlapping computation for an 512, 512, 512 grid with 2 nodes, each with 2 GPUs

By comparing the profiler outputs in `figure 24 and figure 25` we can observe the intended changes. Previously, synchronization occurred in three stages: from GPU to CPU, CPU to CPU, and then CPU to GPU. This synchronization process

has now been optimized by launching it during the execution of device kernels instead of after their completion. The three communication stages are illustrated by either a teal bar or a pink bar labeled `Memcpy HtoD` in this case (Device to Device Memory Copy) or through the `MPI_Waitall` synchronization. The `MPI_Waitall` synchronization bar represents the time period during which MPI handles communication, from initiation to completion.

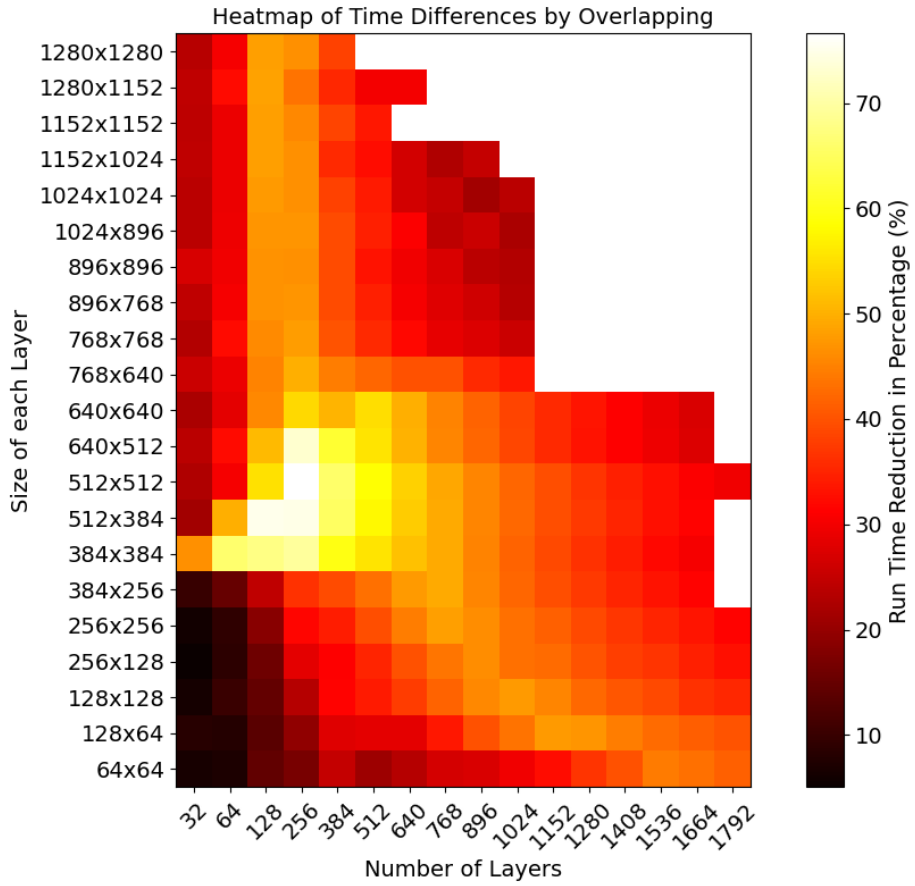

Figure 26: A table over which grid sizes yields the most performance boost when implementing overlap for 4 GPUs spread over 2 nodes

While figure `figure` 26 does follow the same pattern as the previous heatmap 23, it has a different outset. Where the previous pattern started at x-axis value (384,384,_), the new pattern here is a combination of a straigth tipping point from layers sizes (384,384,_) and above. Anything under instead follows the pattern where communication is performed faster than computation compared to above.

Additionally, certain grid sizes around (512,384,256) exhibit improvements that surpass the theoretical 50% limit. The profiler's output, as shown in `Figure` 27,

suggest that this difference is likely due to variations in communication throughput, despite employing the same functions and hardware. The profiling attributes highlighted in the referenced figure explain the exceptional performance observed for these grid sizes.

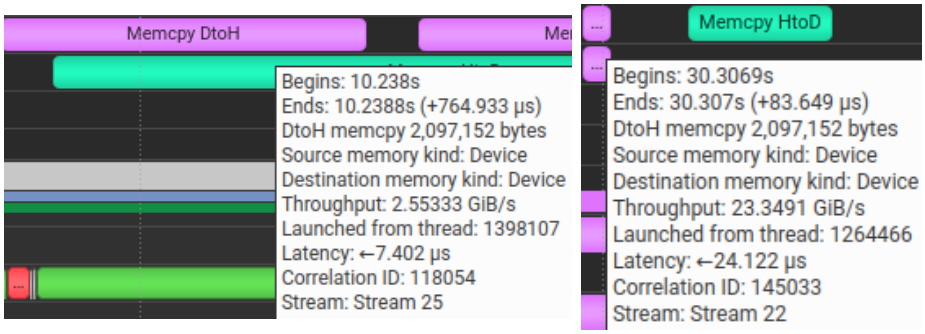

Figure 27: Two different CUDA transfers being performed between GPUs

7.5.2 Through RDMA

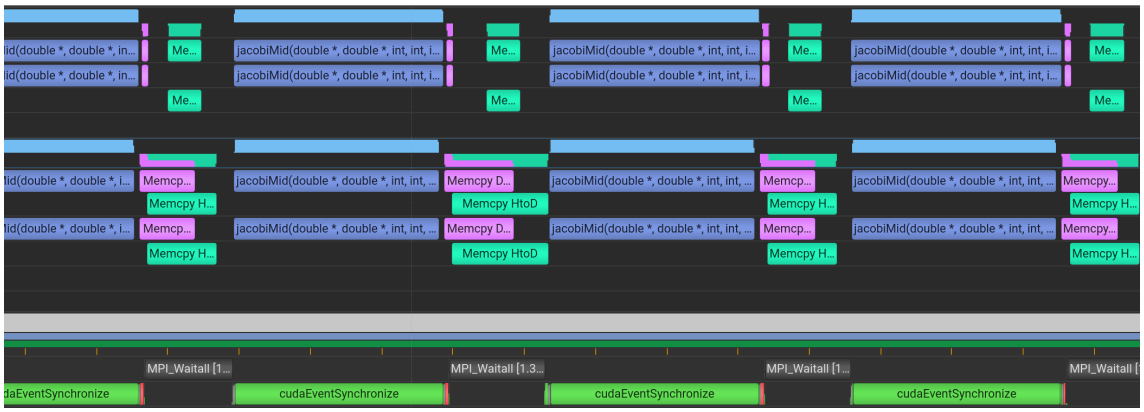

Figure 28: Result for no overlapping computation for an 512, 512, 512 grid with 2 nodes, each with 2 GPUs

The profiling results presented in `figure 28` and `figure 29` illustrate the significant impact that bypassing the CPU has on the communication phase. By eliminating the need to go through the CPU, we have been able to reduce a three-step communication process down to just one step, gathered under the `MPI_Waitall` function call which communicates directly between the GPUs on different nodes. The remaining communication exchanges take place between GPUs within the same node.

The heatmap `figure 30` for the program illustrates similar patterns as the pre-

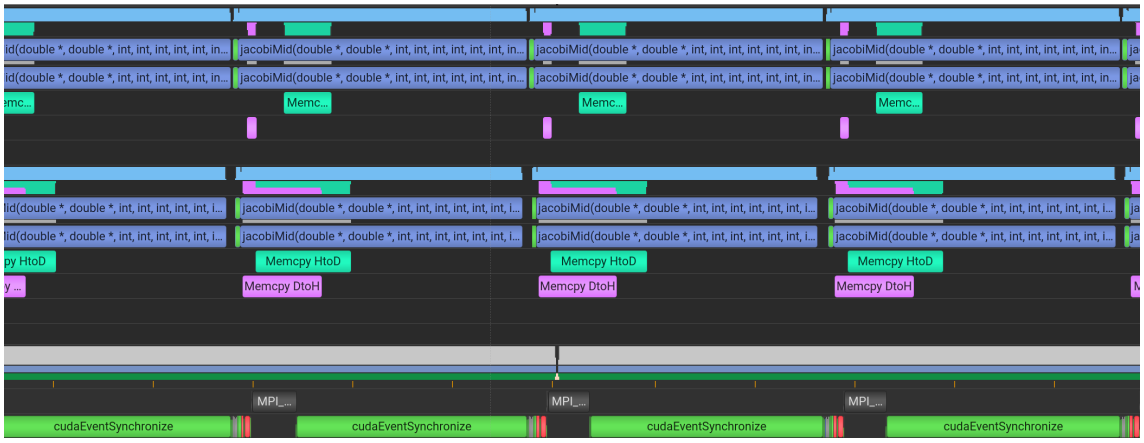

Figure 29: Result for overlapping computation for an 512, 512, 512 grid with 2 nodes, each with 2 GPUs

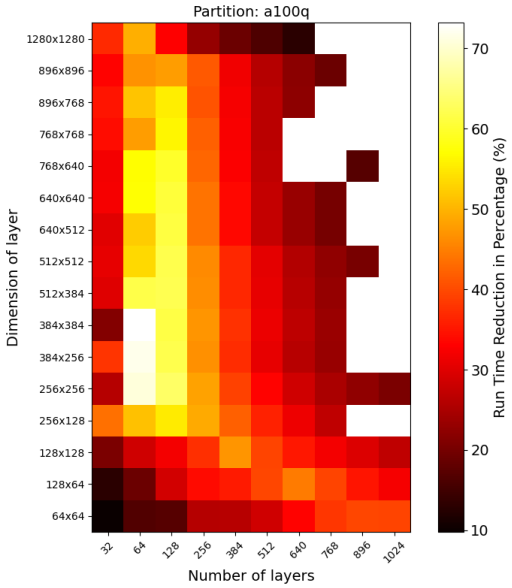

Figure 30: A table over which grid sizes yields the most performance boost when implementing overlap for 4 GPUs spread over 2 nodes

vious scenarios where it reaches improvements of over 50% at certain grids, and all the tipping points follows the same number of layers for bigger layer sizes, but requires more number of layers for smaller layer sizes.

8 Conclusion

This thesis has demonstrated that the implementation of overlapping communication and computation in computing environments yields significant performance benefits. These benefits are predominantly observed when the computation and communication times are balanced. Specifically, the performance improvements can range from 0 to 50 percent, with the optimal overlap occurring when these two times are nearly equal.

The heatmap analyses presented in this work have shown that the highest performance gains are achieved when the grid sizes and computational workloads are configured such that the communication duration matches the computation duration. In these scenarios, the overlapping technique effectively hides the communication overhead, allowing for more efficient use of computational resources. This balance is crucial in heterogeneous computing systems where different processors, such as CPUs and GPUs, must work in tandem.

However, achieving this balance requires careful consideration of the computational workload and data transfer patterns. The findings indicate that as more processors are added to the system, the balance point shifts, necessitating adjustments in the computation-communication strategy to maintain optimal performance. Additionally, while the benefits of overlap are substantial, they are also influenced by the specific hardware and software configurations used, underscoring the need for tailored optimization approaches in different computing environments.

In summary, the research confirms that overlapping communication and computation can significantly enhance performance in heterogeneous parallel computing systems, particularly when the computation and communication times are well-balanced. Future work should focus on refining these techniques to accommodate a broader range of applications and computing architectures, further unlocking the potential of high-performance computing systems.

8.1 Limitations and Problems

The use of multiple partitions was not solely for the purpose of comparing different results, another reason was that partitions are a shared resource among Simula researchers, often leading to queues or scheduled reservations. Consequently, there were periods when partitions were unavailable, and when I obtained reservations, I had to prioritize which programs to run. For this reason, I chose not to run the intra-node communication on the a100q partition. Instead it was run on the dgx2q and hgx2q partitions as the program did not require several nodes.

It can also be observed that there were done several less runs on the hgx2q partition compared to the dgx2q, this is as a result from the previously mentioned problems with partitions being unavailable.

Another limitation was the scope of the scenarios I could undertake. Although I was granted access to many partitions, none offered a combination of several nodes with GPUs within each one. The a100q partition, which I eventually selected, provided both features but was limited to just two nodes, each equipped with two GPUs. This restriction constrained the variations I could explore.

8.2 Future Improvements

There are several improvements to be done for the project, this first is to divide the computations into 3 groups instead of 2 which i have done. This will allow for the communication to start after 1 layers is finished computing instead of 2 layers and would potentially see bigger reward with large grids.

Another task can be implementing the CPU-free execution path for the communication between GPUs without enabling the CPU. This will most likely also shorten the time even further when the communication part is larger than the computation part, which means it often is more effective when working on large clusters of processors.

References

[1] Sally A. McKee and Robert W. Wisniewski. "Memory Wall". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1110–1116. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_234. URL: https://doi.org/10.1007/978-0-387-09766-4_234.

[2] Britannica. *central processing unit*. Oct. 2023. URL: https://www.britannica.com/technology/central-processing-unit.

[3] Jake Frankenfield. *What Is a Graphics Processing Unit (GPU)? Definition and Examples*. Oct. 2021. URL: https://www.investopedia.com/terms/g/graphics-processing-unit-gpu.asp.

[4] *IPU Programmer's Guide*. Aug. 2023. URL: https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/.

[5] George Kyriazis. *Heterogeneous System Architecture: A Technical Review*. 2012. URL: https://web.archive.org/web/20140328140823/http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf.

[6] *What Is Accelerated Computing, and Why Is It Important?* URL: https://www.xilinx.com/applications/adaptive-computing/what-is-accelerated-computing-and-why-is-it-important.html (visited on 08/10/2023).

[7] Changxu Song. "Analysis on Heterogeneous Computing". In: *Journal of Physics: Conference Series* 2031.1 (Sept. 2021), p. 012049. DOI: 10.1088/1742-6596/2031/1/012049. URL: https://dx.doi.org/10.1088/1742-6596/2031/1/012049.

[8] *HPC_Architecture*. Aug. 2023. URL: https://www.intel.com/content/www/us/en/high-performance-computing/hpc-architecture.html.

[9] RishabhJain12. *Memory Hierarchy Design and its Characteristics*. June 2023. URL: https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/.

[10] *What is VRAM?* June 2023. URL: https://www.purestorage.com/knowledge/what-is-vram.html.

[11] Josna. *VRAM or RAM*. June 2023. URL: https://www.electronicshub.org/vram-or-ram/.

[12] *What is Data Communication?* URL: https://onlinedegrees.sandiego.edu/data-communication/ (visited on 12/10/2023).

[13] Stephen J. Bigelow. *Peripheral Componen Interconnect Express(PCIe, PCI-E)*. Apr. 2015. URL: https://www.techtarget.com/searchdatacenter/definition/PCI-Express.

[14] Robert Sheldon. *What is InfiniBand?* Dec. 2021. URL: https://www.techtarget.com/searchstorage/definition/InfiniBand.

[15] *NVLink - Nvidia*. Nov. 2008. URL: https://en.wikichip.org/wiki/nvidia/nvlink.

[16] Fred Oh. *What is CUDA?* Oct. 2012. URL: https://blogs.nvidia.com/blog/what-is-cuda-2/.

[17] *NVIDIA GPUDirect*. URL: https://developer.nvidia.com/gpudirect (visited on 08/10/2023).

[18] Margaret Rouse. *What is Data Transfer?* Oct. 2023. URL: https://www.techopedia.com/definition/18715/data-transfer.

[19] Pejman Lotfi-Kamran and Hamid Sarbazi-Azad. "Chapter One - Introduction to data prefetching". In: *Data Prefetching Techniques in Computer Systems*. Ed. by Pejman Lotfi-Kamran and Hamid Sarbazi-Azad. Vol. 125. Advances in Computers. Elsevier, 2022, pp. 1–17. DOI: https://doi.org/10.1016/bs.adcom.2021.11.003. URL: https://www.sciencedirect.com/science/article/pii/S0065245821000772.

[20] *NVSwitch - Nvidia*. Nov. 2008. URL: https://en.wikichip.org/wiki/nvidia/nvswitch.

[21] *DGX-2 - Nvidia*. Oct. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-2/dgx-2-print-datasheet-738070-nvidia-a4-web-uk.pdf.

[22] Ismayil Ismayilov et al. “Multi-GPU Communication Schemes for Iterative Solvers: When CPUs Are Not in Charge”. In: *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 192–202.

[23] *CUDA Resfresher: The CUDA Programming Model*. June 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[24] *Execution Control*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html#group__CUDART__EXECUTION_1g63685d849da7565b5774f5321a342f05 (visited on 08/10/2023).

[25] *NVIDIA Streams and Concurrency*. URL: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[26] *eX3*. May 2014. URL: https://www.ex3.simula.no/.

[27] *NVIDIA Nsights Systems*. URL: https://developer.nvidia.com/nsight-systems.