Masteroppgave

Vidar Lunde Olaisen

October 19, 2023

Contents

Project title: Aiming at heterogeneous parallel computing for bioinformatics: To overlap or not to overlap?

Keywords: heterogeneous computing, parallel programming, computational bioinformatics

The overall objective of this master project is to to investigate the necessity and potential performance benefits of overlapping communication with computation, in the context of modern parallel computing. A proper handling of this topic is essential for ensuring good performance of many applications in computational science, with computational bioinformatics being a prominent domain of applications.

Introduction

The need for more computing power applies to all subjects of computational science. This is no exception in the case of computational bioinformatics, where typical applications that require supercomputing power include DNA sequence analytics and molecular dynamics simulations. Another timely example is the modeling of spread of virus. One way to answer the urgent need of supercomputing is to adopt heterogeneous computing platforms, that is, a system has at least two types of processor architectures. A typical configuration is a cluster of compute nodes where each node consists of a host CPU and one or several GPUs (graphics processing units). One of the research questions related to parallel programming of such heterogeneous systems is whether to overlap communication with computation for the purpose of "hiding" the communication overhead. In particular, the appropriateness of overlap in the era of heterogeneous computing requires new research. This is both due to the extra programming complexity that will be induced by implementing communication-computation overlap, and due to the impact of, for example, memory bandwidth contention thus incurred, which may lead to a slower computation speed as a whole.

Goal

This master-degree project aims at a detailed and quantifiable understanding of the impact of overlapping communication with computation on heterogeneous parallel computers. The candidate will start with developing simple synthetic benchmark programs that implement various scenarios of communication-computation overlap, where it should be flexible to control the amounts (and designated segments) of node-to-node and CPU-to-GPU data communication, as well as CPU-GPU computation division. These benchmarks will be carefully experimented and time-measured for understanding the potential performance benefits (or disadvantages). An effort will be made to establish performance models related to communication-computation overlap or non-overlap. Another objective of this master-degree project is to summarise some programming guidelines in this regard. The scientific findings will then be tested in applications that fall in the domain of computational bioinformatics, to check the effectiveness of the resulting heterogeneous programming approach (with or without communication-computation overlap), as well as the applicability of the performance modeling methodology.

Learning outcome

The candidate will learn about advanced parallel programming: applicable to both multicore CPUs and at least one specific GPU architecture. The candidate will also become an expert on performance profiling and modelling. The candidate will get the chances to be familiarised with real-world examples of computational bioinformatics. The candidate will also be exposed to cutting-edge hardware for parallel computing. All these skills and experiences are highly sought-after expertise of future workforce for scientific/technical computing.

Qualifications required

The candidate is expected to be skilful in technical programming (experience with parallel programming is not required, but preferred). Very important: The candidate must be hard-working and eager to learn new skills and knowledge (such as about basic mathematical modeling and basic bioinformatics applications).

1 Abstract

2 Introduction

3 Harnessing the Power of Heterogeneous Computing

In modern computing, the concept of heterogeneous architecture has emerged as a formidable strategy, one that allows for the fusion of the realms of communication and computation. It capitalizes on the synergy between various processing units, such as Central Processing Units(CPUs)[1], Graphics Processing Units(GPUs)[2] or Intelligence Processing Units(IPUs)[3], all seamlessly integrated within a single computing system. In stark contrast to the traditional homogeneous architecture, where identical processing units labor in unison, heterogeneous architectures unite different processor types, each meticulously crafted to excel at specific tasks. This combination addresses the problem of not having enough computation power across a spectrum of applications, like artificial intelligence, scientific simulations, multimedia processing and more.[4]

Heterogeneous architectures works best when entrusted with the acceleration of specific computations.[5] They direct a number of tasks by assigning them to 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, a heterogeneous architecture is created, harnessing the strengths of each component to gain peak performance.

This architecture, while promising, does however introduce its fair share of challenges that demand attention and resolution. Some of these challenges include:

Crafting software for heterogeneous architectures can be a complex endeavor. Programmers must adeptly manage an array of processing units, memory hierarchies and intricate the transfers of data between them. This typically necessitates the use of specialized programming languages and libraries capable of harnessing 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 approached which works on a limited number of architectures but is faster.

Moreover, when working with a multiple of processors one must take into account the potential bottlenecks that arises from data transfers, more so than normal. Efficient utilization of heterogeneous architecture hinges on meticulous data transfer management, often entailing minimizing data movements and optimizing data locality.

Another challenge lies in the ability of programmers to decide which tasks can and should be offloaded to specialized units. Optimizing these tasks for their respective units often calls for a fresh approach to computing, one that may normally be more straightforward within the confines of computing within a homogeneous architecture.

So heterogeneous architectures beckon with he promise of superior computational performance compared to their homogeneous counterparts, provided their strengths are harnessed and their challenges overcome. While these challenges persists, advancement in hardware design and software tools hold the potential to ease the programmers journey, unlocking the architectures full capabilities across a broad spectrum of applications

3.1 The Key Building Blocks

A 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 architectures functionally and capabilities. In the contexts of my thesis, i will delve into the central components that play pivotal roles.

3.1.1 Central Processing Units(CPU)

At the core of most of the heterogeneous and homogeneous architectures that exists lies the CPU, a versatile workhorse responsible for a large amount of general-purpose computing tasks. The CPUs architecture is tailored to handle sequential processing, intricate control flow and system-level management. Its adeptness at executing a wide array of instructions and managing tasks distribution makes it indispensable for interface with the broader system.[1]

3.1.2 Graphics Processing Unit(GPU)

The GPU takes center stage in heterogeneous architecture, designed in stark contrast to the CPU to shine in parallel computation. Comprising a multitude of cores, the GPU excels in tasks characterized by massive data parallelism, including graphics rendering and scientific simulations.[2]

3.1.3 Memory Hierarchy and Communication Interconnects

When one really starts trying to optimize and improve computer code to work in an unhindered fashion one really has to pay attention to efficient memory management and seamless communication between processing units. These two dual challenges can be worked through with the help of working with memory hierarchy and communication interconnects.

A crucial part of heterogeneous architecture, the memory hierarchy is tasked with optimizing data accessibility and minimizing bottlenecks. This intricate system governs the speed and accessibility of registers, caches and device memory. By intelligently organizing data storage and access, the memory hierarchy ensures that diverse tasks can efficiently retrieve the data they require, enhancing overall system performance.[6]

Physical links that connect various devices via high-speed pathways like Peripheral Component Interchange Express(PCIe)[7] or NVLink[8] also play an important role in heterogeneous architecture. These communication interconnects facilitate swift data exchange between processing units. They serve as the neural pathways of the architecture, ensuring that data can flow smoothly between components. Effective communication interconnects are vital in eliminating latency and enabling efficient cooperation among the diverse processors.

3.1.4 Collaborative Technologies in Heterogeneous Architecture

Collaborative technologies form a crucial pillar within heterogeneous architecture, fostering efficient cooperation and data sharing among different processing units. These technologies encompass a range of tools and techniques designed to optimize inter-device communication and cooperative processing.

At its core, collaborative technologies enable disparate processing units, such as CPUs and GPUs, to work together to tackle complex tasks. They include software libraries, protocols and interface that facilitates the exchange of data and instructions among these units.

One prominent example is GPUDirect[9], a technology developed by NVIDIA. GPUDirect allows GPUs to communicate directly with eachother bypassing the CPU and reducing data transfer overhead. This not only enhances data transfer speeds but also reduces CPU utilization, making an efficient choice for certain workloads.

Collaborative technologies play a critical role in unlocking the full potential of heterogeneous architecture. They streamline data transfer, reduce bottlenecks and enable processors to work in harmony, ultimately leading to improved overall system performance and responsiveness. As the field of heterogeneous computing continues to evolve, collaborative technologies will remain instrumental in harnessing the power of diverse processing units for a wide range of applications.

4 Data Transfer

Data transfer mechanisms within a heterogeneous architecture plays an important role in determining system performance and efficiency. The ability to move data seamlessly and rapidly between different processing units, each optimized for specific tasks, is crucial for achieving optimal utilization of resources and minimizing bottlenecks.

Modern heterogeneous architectures rely on various technologies to facilitate efficient data movement. One of the primary communication interfaces it the PCIe[7] which allows for high speed communication between components like CPUs and GPUs, allowing data to flow between different devices and memory spaces. There is also a rapid progress in finding new communication interfaces which can allow for smoother data transfer between devices, an example is NVIDIA's NVLink[8] and NVSwitch[10] which is specifically made for communication between NVIDIA GPUs where it offers data share rates significantly faster than traditional PCIe connections.

There also different technologies which adds another layer of efficiency to data transfers. GPUDirect is an example for this, as it is an technology from NVIDIA[9] that enables Direct Memory Access(DMA)[11] between GPUs and other devices like storage devices by bypassing the CPU and system memory and allowing direct access between two devices connected to a CPU.

In essence, data transfer mechanisms are the lifeblood of a heterogeneous architecture. Efficient data movement ensures that processing units are fed with the necessary information in a timely manned, allowing them to perform tasks effectively. Technologies like PCIe, NVLink, NVSwitch and GPUDirect collectively shape the architectures ability to handle data intensive workloads, enabling seamless collaboration between components and unlocking the full potential of heterogeneous computing.

Data transfer is not just about sending raw data, it involves encoding, compression and other techniques to optimize the utilization of available bandwidth. Encoding methods for example play a vital role in how data is packed and transmitted. Efficient encoding can reduce the volume of data transferred, saving saving valuable bandwidth and minimizing latency. This becomes particularly critical when dealing with large datasets and high-speed communication interfaces.

Furthermore, attributes like error checking, correction mechanisms, prioritization of data streams and congestion control are also integral to the data transfer process. These attributes ensure the reliability and integrity of data as it traverses the heterogeneous architecture. Error detection and correction, in particular, help safeguard against data corruption during transmission, which is essential in mission-critical applications.

4.1 Data Buses

Before delving into the various types of data bus configurations, it's essential to gain an understanding of how data buses operate and their fundamental role. Data transmission within a computer is not a mystical process occurring through thin air. Instead, it relies on the utilization of bus lines,hardware pathways etched directly onto the motherboard, enabling direct communication between components. These bus lines serve as essential conduits, enabling the exchange of data between interconnected elements.

4.1.1 Parallel Bus

The concept of a parallel bus involves the simultaneous transmission of multiple data bits across an array of wires. Historically, parallel buses were the preferred communication method due to their perceived efficiency in data transfer. However, as technology evolved, challenges emerged with this approach. As the number of bus lanes increased in parallel configurations, signal integrity became a concern. The interactions between different bus lines could lead to interference and data corruption. But it is still widely included into different systems.

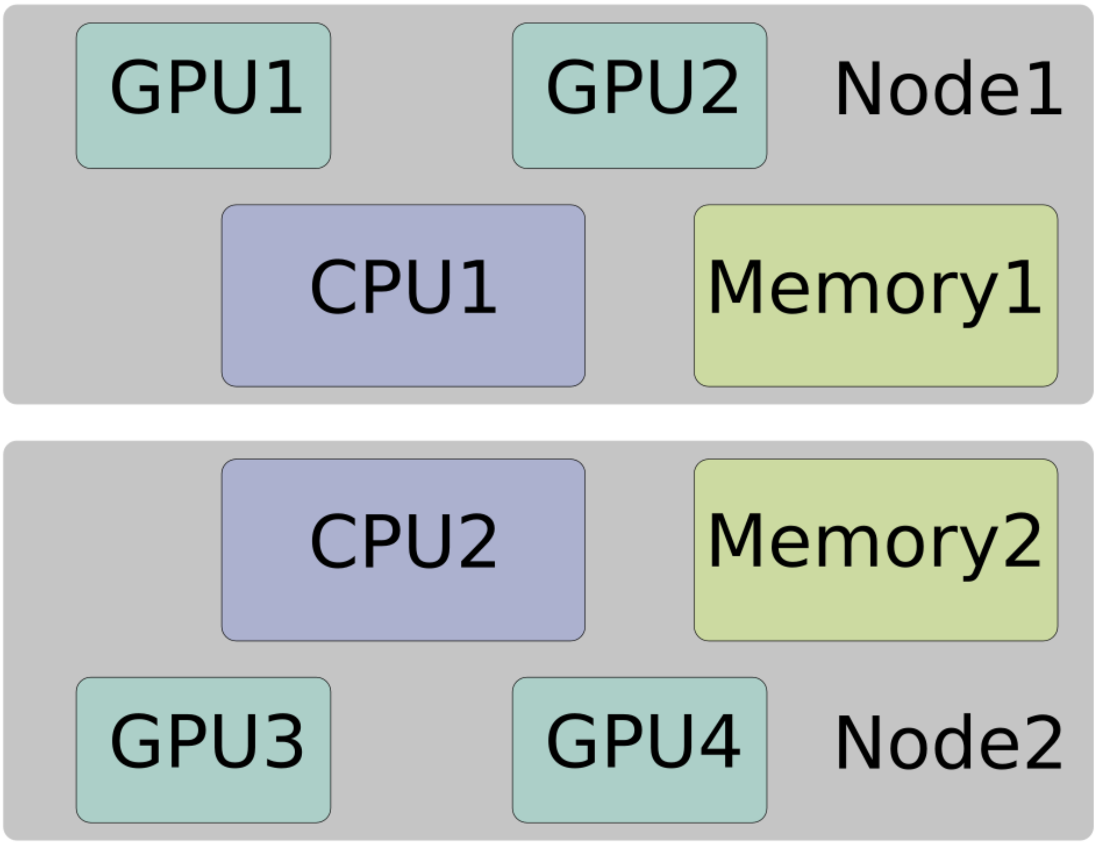

Figure 1: Visualization of 2 systems(Node 1 and Node 2) with heterogeneous architectures where each contain 2 GPUs and 1 CPU

- **Advantages**
  - **High Data Throughput**: Simultaneous transmission of multiple bits allows for high data throughput, making parallel buses efficient for transferring large data volumes.
  - **Simplicity**: Parallel buses can be simpler in terms of protocols and electronics, facilitating implementation
- **Disadvantages**
  - **Signal Integrity**: Maintaining a signal integrity becomes challenging with more parallel lines, leading to issues like cross alk and interference.
  - **Complex with Width**: Wider buses increase complexity, especially for routing and manufacturing, limiting scalability.

4.1.2 Serial Bus

In contrast to parallel buses, serial buses performs a sequential approach to data transmission. This method prioritizes data integrity and interference prevention. A notable step forward in this direction is evident in technologies like Peripheral Component Interconnect Express (PCIe). PCIe employs individual bus lanes to transmit data sequentially. This design choice optimizes performance while concurrently preserving signal integrity, culminating in an architecture that offers enhanced overall system efficiency.

- **Advantages**
  - **Better Signal Integrity**: Sequential transmission reduces the risk of interference, resulting in improved signal integrity.

  - **Scalability**: Serial buses are highly scalable, allowing the addition of more lanes to increase data transfer rates without compromising signal integrity.

- **Disadvantages**
  - **Slower Transfer for Small Data**: For transferring small data amounts, serial buses can be slower due to their sequential nature.
  - **Complex Protocols**: Serial communication often requires complex protocols for encoding, decoding and error checking, increasing overhead.

4.1.3 Hybrid Approaches

There does not exist any rule that says one has to choose between serial and parallel communication, hybrid approaches exist that combine the advantages of both methods. While some systems adhere exclusively to either serial or parallel principles, others leverage a combination of the two. This hybridization allows designers to tailor communication strategies to the specific needs of their applications, optimizing performance and minimizing potential bottlenecks.

- **Advantages**
  - **Optimized Performance**: Hybrid approaches leverages the speed of parallel buses and the reliability of serial communication, optimizing overall system performance.
  - **Versatility**: Designers can customize systems to suit application needs.
- **Disadvantages**
  - **Complexity**: Implementing hybrid approaches can introduce complexity in system design and management, requiring careful consideration.
  - **Potential Cost**: Combining both parallel and serial components may increase overall system cost.

4.2 Bus types

4.2.1 PCIe

Peripheral Component Interconnect Express(PCIe)[7] 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. 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 PCI 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 improves the previous generation by performing at twice the speed. A basic overview can be looked at like underneath. 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, wit each iteration offering increased data transfer rates. PCIe Gen 3 and 4 for example delivers 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 need for tasks like GPU acceleration, AI processing and high-performance computing.

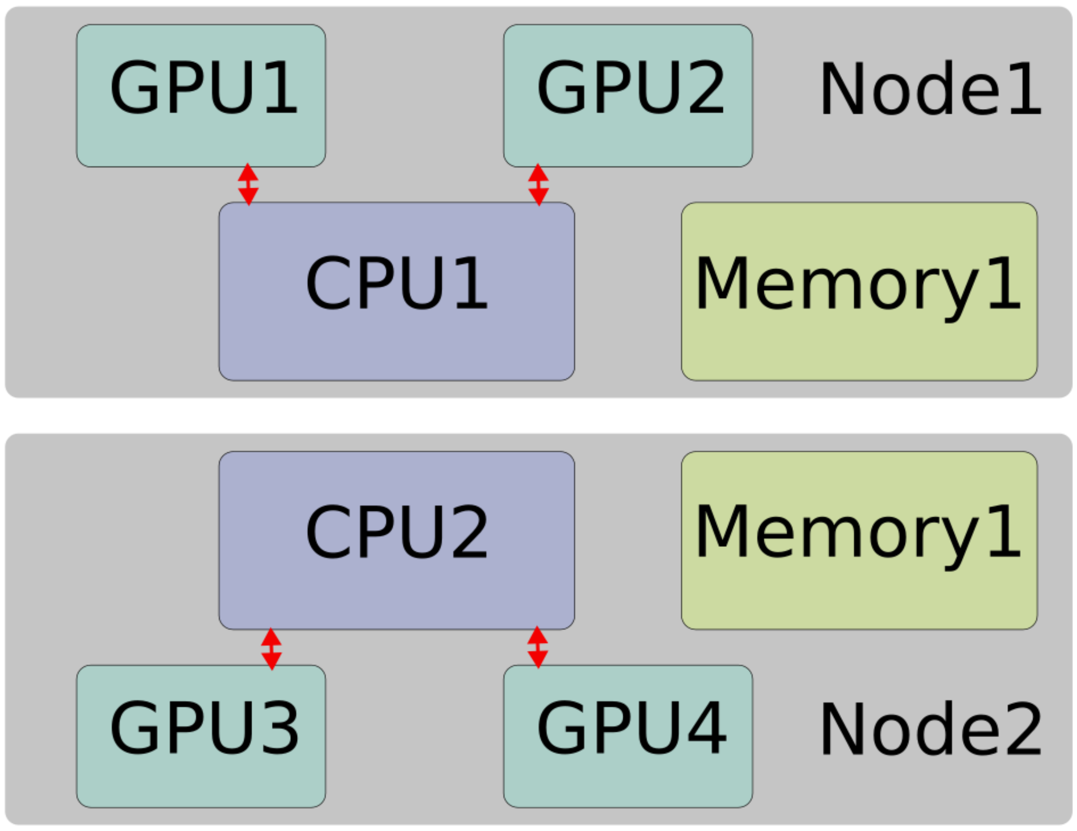

Figure 2: Visual of how PCIe is normally implemented in an heterogeneous architecture. PCIe can also be used when communicating between two systems

4.2.2 NVLink

NVLink is a cutting-edge interconnect technology developed by NVIDIA[8] 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.[8]

NVLink operates as a high-speed, point-to-point, serial interconnect that enables direct serial communication between NVIDIA GPUs. This direct connection minimizes latency and maximizes data bandwidth, making it well suited for applications that demand rapid and seamless data exchanging between GPUs. NVLinks architecture allows GPUs to share data at rates significantly faster than those achievable with traditional PCIe connections.[8]

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.[8]

One of the standout features of NVLink is its ability to scale. The technology support configurations with multiple NVLink links, allowing for increased data transfer bandwidth and enabling large scale parallel processing across multiple GPUs. This scalability essential for tackling workloads that require massive computational power.[8]

NVIDIA has introduced multiple generations 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.[8]

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.[8]

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 need to find the wanted data and organize, format and encode it before starting. The GPU also need to establish communication through the NVLink to see if its even possible to send.

2. It then encodes the serialized data using a NVIDIA encoding program. This is done to help with clock synchronization and other considerations in serial data transmission. It does this by adding additional bit 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 decoded and it checks if the additional bits gives 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.

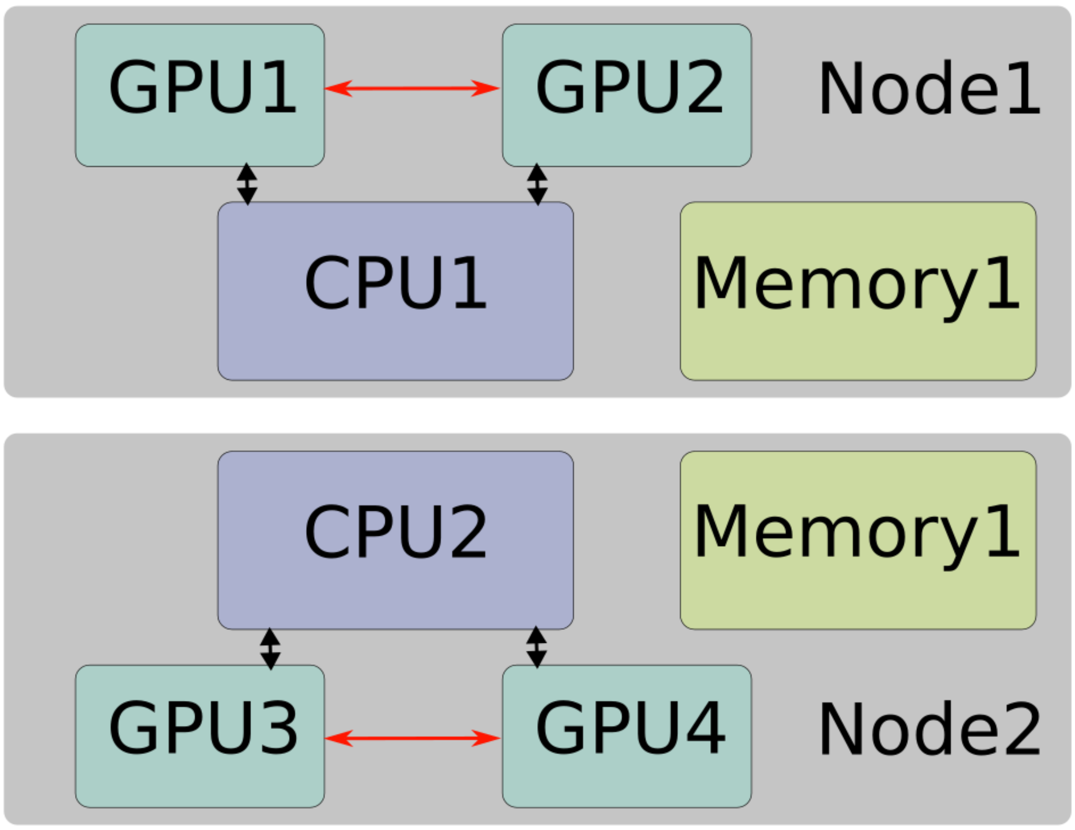

Figure 3: NVLink is especially made for communication between NVIDIA GPUs

4.2.3 NVSwitch

NVSwitch is a relatively recent addition to NVIDIAs hardware portfolio. It plays an important role in architectures that harness multiple GPUs requiring direct communication among themselves. NVSwitch can be viewed as a dedicated hardware integrated alongside GPUs, with each GPU connected to one or more NVSwitch via NVLink connections. The NVSwitch ports can be operate at remarkable speed of 50GB/s each, contributing to the NVSwitches overall bandwidth of 900GB/s. This impressive bandwidth allows up to nine devices to route data directly to any other nine devices within the system that is connected to the same NVSwitch. [10]

4.3 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 storage, 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 transfer of data between locations without having to involve the CPU.

4.3.1 GPUDirect Storage

GPUDirect Storage is a technology that enables direct communication between storage devices and GPUs, bypassing the CPU. This allows for faster data transfer and reduced CPU overhead when reading or writing data from storage to GPU memory. By streamlining the data path, GPUDirect

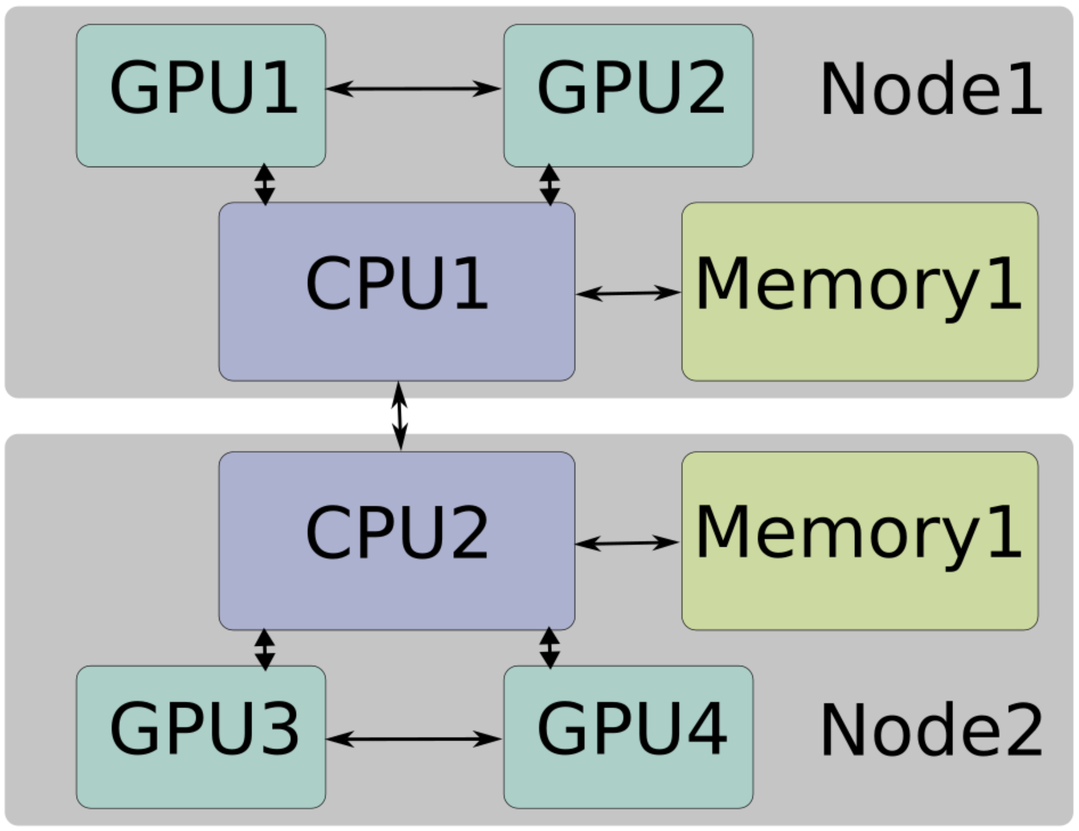

Figure 4: An overview of connections, 3 new connections have been added since figure 3. 1 Between the CPUs which can be an infinityband technology, and 1 bus between each CPU and its memory called memory bus.

Storage can significantly accelerate data-intensive tasks, such as loading large datasets, processing high-resolution videos, and conducting simulations that involve frequent data access. This technology enhances overall system performance by optimizing data movement and minimizing latency between storage and GPU resources.

4.3.2 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.

4.3.3 GPUDirect RDMA

GPU Remote Direct Memory Access(RMDA) 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 collaborative tasks across

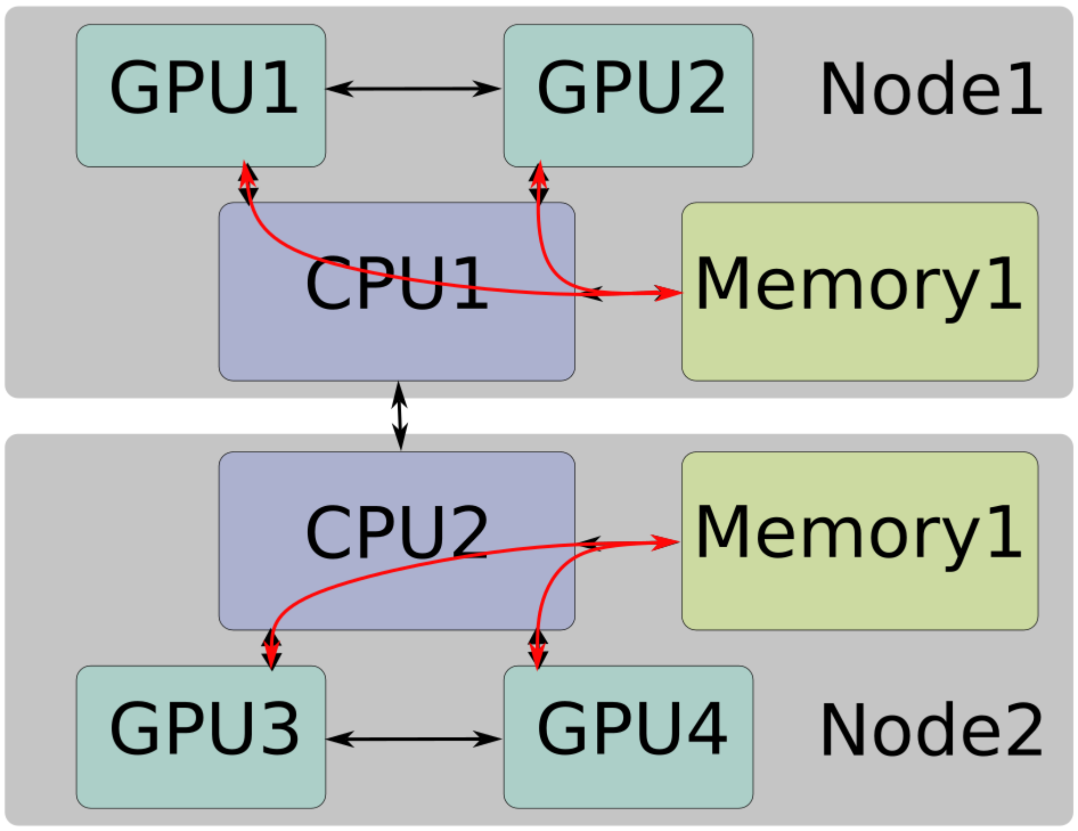

Figure 5: Illustrates how the data does not stop within the CPU, but instead goes directly to the memory.

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.

4.4 Transfer Rate

So how can we estimate transfer rates between two devices? It depends on a number of factors, but it all comes down to the combination of hardware and software during computation. As a hypothetical scenario lets try to find the transfer rate over an PCIe version 3.0 with 8 lanes is as following.

$$\begin{aligned}
\text{Max Data Transfer Rate(GB/s)} &= \text{Gigatransfers per Second(GT/s)} \\
&\quad * \text{ Channel Width(number of lanes * word size)} \\
&\quad * \text{ Encoding Efficiency(Line code)} \\
\text{GB/s for PCIe 3.0, with 8 lanes} &= 8.0 \text{ GT/s} * 8 * 1 \text{ byte} * 128\text{b}/130\text{b} \\
\text{GB/s for PCIe 3.0, with 8 lanes} &= 64 \text{ GT/s} * 128\text{b}/130\text{b} \\
\text{GB/s for PCIe 3.0, with 8 lanes} &\approx 32.0154 \text{ Gb/s} \\
\text{GB/s for PCIe 3.0, with 8 lanes} &\approx 7.8769 \text{ GB/s}
\end{aligned}$$

One would then expect a top transfer potential of 7.8769 GB/s across the PCIe 3.0 from one device to another if we programmed in a perfect environment with perfectly optimized code for data transfers.

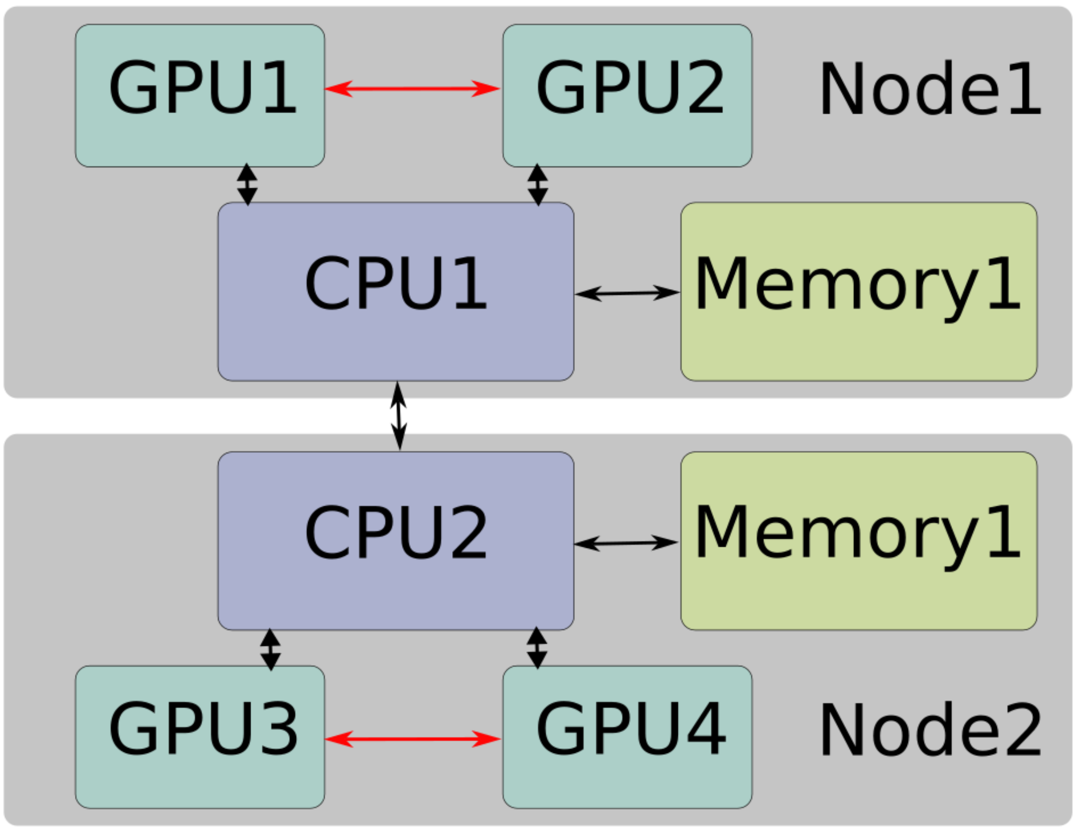

Figure 6: Shows how GPUDirect P2P communicates by avoding the CPU

But how does the rate of data transfer work in real-life scenarios? Having explored various components that impact the efficiency of data transfer, let's delve into a practical example to gain a better understanding. Consider a situation where we aim to transfer data between two GPUs, providing us with an insightful overview of the process.

```c
//C, selfmade code

#define DATA_SIZE 1000000000 // 1 billion bytes to be transferred
#define PCIe_BANDWIDTH 32000000000 // 32 billion, PCIe bandwidth

int main(){
    // Creates an array on the CPU
    int *data = (int*)malloc(DATA_SIZE * sizeof(int));

    // Gives each elemenent in the array its index as value
    for(int i = 0; i < DATA_SIZE; i++){
        array_cpu[i] = i;
    }

    // Estimated minimum time required for data transfer
    double transfer_time = (double)DATA_SIZE * sizeof(int) / PCIe_BANDWIDTH;

    // Time when starting Communication
    clock_t start_time = clock();

    // Code used for transferring data over the PCIe

    // Time when ending Communication
    clock_t end_time = clock();
```

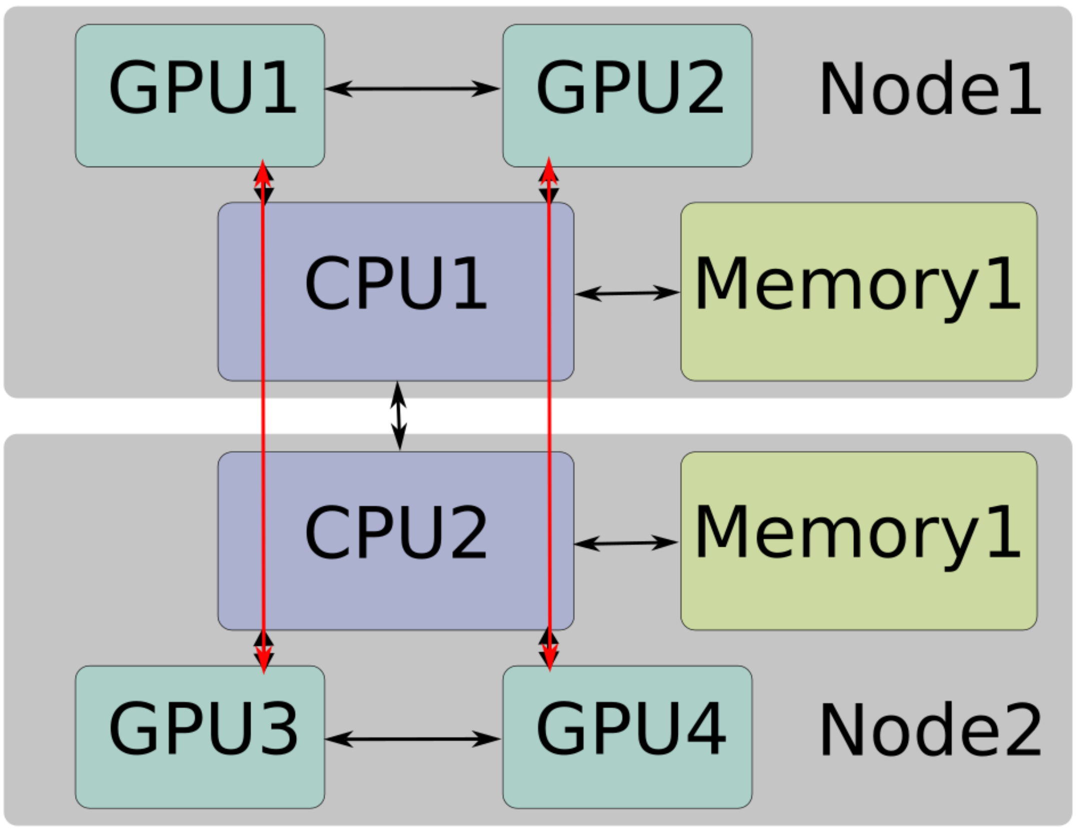

Figure 7: Pictures the way the RDMA goes straight to the device of choice on the other node.

```
    double execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Transfer Time (based on PCIe bandwidth) - %f sendons\n", transfer_time);
    printf("Total Execution Time - %f seconds\n", execution_time);

    free(data);
}
```

While the provided code offers valuable insights, it's essential to acknowledge that real-world scenarios involve additional factors not covered in this simplified version. Complexities related to memory fetching, hardware-specific behavior, and interconnect latency's influence the actual execution time.

The formula that has been employed in the code provides a basic calculation for the expected time to transfer data

$$\text{Time (seconds)} = \frac{\text{Data Size (bytes)}}{\text{Bandwidth (bytes/s)}}$$

An insightful observation can be drawn from this formula: when the calculated time is less than 1 second, it suggests that the bottleneck is unlikely to be attributed solely to the data transfer process. However, this observation should be interpreted cautiously, as the overall system behavior is influenced by a multitude of interconnected factors.

4.4.1 Memory Hierarchy and Access Patterns

Utilizing the memory hierarchy is a crucial factor influencing data transfer efficiency. Modern computing systems feature multiple levels of memory, each with varying access latency's and capacities. Optimizing data access patterns to minimize memory hierarchy fetching time is essential for reducing memory-related bottlenecks. Techniques like data reordering, data alignment and exploiting data locality can help improve data transfer efficiently. By ensuring that data is accessed in a manner that matches the memory hierarchy's characteristics one can significantly enhance the overall throughput of data transfers.

4.4.2 Latency and Throughput

To delve deeper into the other factors affecting data transfers we can start introducing latency and throughput. These are critical considerations when designing efficient data transfer mechanisms. Latency represents the time it takes for a data transfer to initiate after it is requested. Minimizing latency is crucial for reducing communication overhead and ensuring swift response times. On the other hand, throughput denotes the rate at which data is transferred once the transfer is active. High throughput is essential for efficiently moving large datasets. Balancing both latency and throughput is essential for achieving optimal data transfer performance. Techniques such as prefetching, pipelining and overlapping operations can be employed to reduce the idle times and maximize data transfer throughput.

4.4.3 Buffering and Pipelining

Buffering and pipelining strategies are fundamental for smoothing out data transfer irregularities and maintaining a steady for of information. Buffers act as a temporary storage area that hold data before it is transmitted or used. Pipelining involves breaking down data transfer processes into stages and allowing each stage to work concurrently. By using these techniques, data transfers can be more efficiently managed, reducing the impact of variations in data variability or system latency. Effective buffering and pipelining can lead to better resource utilization and improved overall system performance.

5 Compute Unified Device Architecture(CUDA)

To begin with we should introduce what CUDA actually is and why it is still developed and used since its conception in 2006 by NVIDIA.

CUDA is a general-purpose parallel computing platform and programming model. This leads to two conclusions of its meaning, firstly it provides the user of CUDA with a software environment which enables access to the NVIDIA GPUs through a C, C++ and FORTRAN looking syntax. Secondly it also gives the user a blueprint of concepts and abstractions which are to be used when wanting to fully optimize CUDAs functionality.

It has also become more and more relevant, which we will delve into later, in recent times as computing on a combination of different processors at the same time such as a combination of a CPU and several GPUs is becoming more and more the norm for big data clusters. This type of programming, also called heterogeneous programming, is a way of offsetting the downsides of a processor by introducing another type of processor more specialized to handle these downsides and transferring them over. It is a method which most often includes an CPU and one or more GPUs depending on the computer size.

5.1 Programming Model

So what distinguishes the CUDA programming model from casual C or C++? Centrally it is its implementation of threads and blocks. These concepts allows for the program to be designed around parallel processes instead of serial.

Now what are each of these components and how do they work within a CUDA program? We can first explain each of them, from lowest level(threads) to highest(grid). They do however work together so closely that just knowing how 1 works, the understanding of the others can be made much easier.

5.1.1 Threads

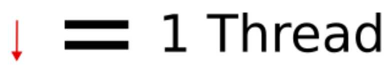

Figure 8: Visualising a thread

A CUDA thread operates as the fundamental unit of parallel execution within a GPUs processing architecture. Unlike the linear execution model of traditional serial programming, where tasks are processed sequentially on a single core, CUDA threads operate on a massively parallel manner across the numerous cores of a GPU. this unique approach allows for a sequential increase in computational power and efficiency, particularly for tasks that can be divided into several parallelizable parts.

Each CUDA thread functions as an independent worker within the GPU, capable of execution 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. One therefore not execute a single thread at a time, but we instead run them in warps. It follows an execution model called Single Instruction, Multiple Threads(SIMT) and it allows for exploiting data parallelism by performing the same operation across multiple data elements in parallel.

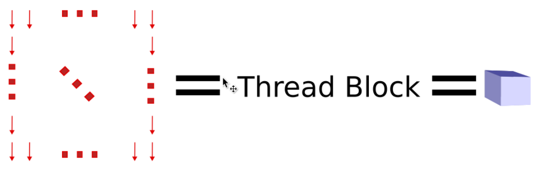

Figure 9: A thread block is made up of several threads, often a multiplication of 32 threads(warp size)

5.1.2 Blocks

A defining attribute of CUDA threads is that they are a part of a hierarchy that includes thread blocks and grids. So a thread block, also just called a block, is essentially an organizational unit for executing parallel tasks on a GPU. The blocks are a group of multiple CUDA threads, and these threads work collaboratively to perform a specific operation. Much like the way individual CUDA threads function as the core worker of parallel execution, thread blocks serve as higher-level units, planning collaborative and efficient parallel computations.

Each block also is designated with its own memory which it shares with all of its threads. This allows each block to communicate and synchronize for all its threads and one can therefore designate each block to its own part of the program. The programmer is however required to allocate the number of threads per block.

5.1.3 Grid

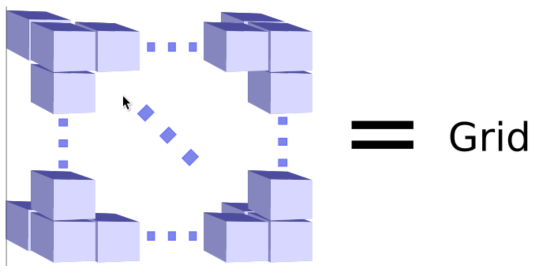

Figure 10: A grid is made up of several thread blocks

A fundamental aspect of CUDA's hierarchical organization involves the concept of a grid, which is a fundamental structure for organizing parallel tasks on a GPU. Just as blocks serve as a collection of threads, a grid can be looked at a collection of blocks. So where a thread block serve as an collection of warps to execute in parallel, a grid is used to execute intricate computational tasks.

Within a grid the synchronization and distribution of parallel work are of the highest importance. The role of each thread block within the grid is to address a specific segment of the overall computation as each block has its own shared memory. This tailored allocation of resources empowers the programmer to assign each block a unique program segment, this foresting a modular and well structured approach to parallel program-solving. Importantly the programmers responsibility lies in accurately allocating the number of threads per block, ensuring an optimal distribution of computation across the grid.

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 traditional languages.

To illustrate the similarities and distrinctions between CUDA and these languages, conside the code snippets provided below, the first written in C and the latter in CUDA. 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.

```c
//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
    for(int i = 0; i < length; i++){
        array_cpu[i] = i;
    }

    // Adding 11 to each element within the array
    for(int i = 0; i < length; i++){
      array_cpu[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);
}
```

```cuda
//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), cudaMemcpyHostToDevice);
    // 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);
}
```

Upon initial inspection, it is evident that both code snippets share many similarities. However the CUDA code is slightly longer, being made up of 19 lines, while the C code consists of only 14 lines. The most significant difference lies in the CUDA codes implementation of a device kernel, which allows computations to be executed on GPU devices. 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.

While the code snippets reveal a significant resemblance between CUDA and C, it is essential to recognize that the CUDA code may demand a deeper understanding of memory management, which assumes a more prominent role in CUDA than in standard C. In CUDA memory allocation and data transfer between the CPU and GPU, as well as among GPU memory spaces, require careful consideration as it if more often needed to specify its location/locations, size and type as it is transported around.

For instance, in the CUDA code, memory allocation for the array on the GPU is achieved using **'cudaMalloc()'**. This distinction highlights the need for the programmer to be well-versed in GPU memory management, ensuring efficient memory allocations and deallocation. Additionally, data transfers between CPU and GPU memory spaces are facilitated using **'cudaMemcpu()'**, which necessitates understanding data movement intricacies.

5.3 Cooperative Groups

CUDA Cooperative Groups are a programming feature provided by NVIDIA's CUDA platform that allows for more efficient and cooperative parallel execution of threads within or between thread blocks. In GPU programming synchronization within a thread block can be limited, as threads within a block are 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. 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 $'\_\_synchthreds()'$.

The key idea behind Cooperative Groups is to enhance 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 me 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.

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.

Advancing the synchronization hierarchy, we encounter a significant step up from the Warp Group through the well-known CUDA utility, the *__synchthreads*() function. 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.

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 collaborative interaction amongst disparate threads blocks within a grid.

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.

5.4 Dynamic Parallelism

Dynamic Parallelism is also an implementation that was later in CUDAs development, more specifically in CUDA 5.0[12], and it is designed to allow for higher levels of flexibility and complexity in parallel computing scenarios. While the standard CUDA model relies on the host CPU to initiate GPU kernel launches, dynamic parallelism introduces a shift of this way of thinking by allowing kernels running on the GPU to launch other kernels called child kernels independently.

This new capability is particularly valuable in scenarios where parallelism demands intricate and more control. Dynamic Parallelism allows a kernel launched from the CPU, known as the "parent kernel" to launch its own kernel called "child kernel" creating 2 layer parallel execution program. This approach enables programmers to dive into the problems with parallelism by further breaking down tasks into smaller more manageable tasks that can be executed in parallel with remarkable efficiency.

The importance of Dynamic Parallelism becomes evident in use cases involving adaptive algorithms. Adaptive algorithms are then problems where the solution can change during runtime depending on different factors. Such algorithms often require runtime decisions about the initiation of parallel processes based on varying conditions or data characteristics. Dynamic Parallelism gives kernels the power to take charge of these decisions and enable a level of adaptability and optimization that was previously challenging to achieve.

Furthermore the concept of parallel recursion which involves algorithms calling themselves with varying parameters is significantly enhanced by dynamic parallelism. Recursive calls can now launch their own kernels, allowing for performance improvements in parallel computation even further. This relatively newfound capability enables the implementation of deeply recursive algorithms that were once limited by hardware limitations.

5.5 CUDA Stream

When it comes to providing mechanisms for managing parallelism and overlapping computation with data transfer on GPUs, CUDA Streams are a fundamental concept within NVIDIAs CUDA programming model which is made to give a solution. Streams enable developers to control concurrent execution of multiple operations, allowing for efficient use of GPU resources and improving performance.

A CUDA Stream can be look at as a sequence of operations that is 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 tha toperations within the same stream are executed sequentially, while different streams can potentially run concurrently.

5.6 Asynchronous Operations

An additional implementation that CUDA introduces is the possibility of Asynchronous operations and its role in providing the programmer with the tools to further increase the performance and efficiency of GPU computing through parallelism and overlapping tasks. In the context of CUDA programming, asynchronous operations enable multiple tasks to be executed concurrently, eliminating the need for on task to way for another to complete before proceeding. This allows for better throughput by enabling the reduction of downtime for the processors.

One of the notable benefits of asynchronous operations lies in achieving communication-computation overlap, a strategy vital for maximizing system efficiency. This technique is particularly relevant when dealing with data transfers between the host(CPU) and device(GPU) or between several devices like GPUs. By employing asynchronous memory transfers using functions like *cudaMemcpyAsync*, data can bet transmitted in the background while the GPU is simultaneously performing computations. This mitigates the need for the GPU to wait idly during data transfers, effectively utilizing both the compute and memory subsystems at the same time.

It is important to note that while asynchronous operations offer significant performance advantages, proper synchronization mechanisms must be employed to ensure that the GPU computations do not operate on incomplete or outdated data. CUDA provides synchronization mechanisms like *cudaStreamSynchronize* or event to manage these dependencies and ensure data integrity.

So while asynchronous computation does add a new level of complexity to the code, its benefits depends very much on what problem one is working on as implementing it on a code which is memory-bound, to much data compared to bandwidth, one should consider leaving it be. Therefore finding a golden balance between using time and resources on implementing it should be weighted against the potential benefits as one should always try to get the most out of ones investment.

5.7 Inter-Device Communication

Inter-device communication is an important aspect of heterogeneous computing that allows multiple processing units, such as CPUs and GPUs to collaborate effectively. In CUDA achieving efficient inter-device communication is essential for harnessing the full potential of heterogeneous architectures. This section explores the complexities and strategies associated with inter-device communication within the broader CUDA framework.

Inter-device communication is a fundamental component of heterogeneous computing, enabling the efficient collaboration of various processing units within a system. In CUDA, understanding the programming model, synchronization mechanisms and the use of cooperative Groups is essential for achieving efficient inter-device communication. By carefully managing data transfers and tasks between devices, programmers can fully exploit the capabilities of heterogeneous architectures, leading to enhanced performance and computational efficiency.

5.7.1 The Role of Inter-Device Communication

In a heterogeneous architecture, various devices with distinct capabilities and functionalities work together to address complex computation tasks. These architectures leverage the strength of each device to offset their individual limitations. Inter-devices communication plays a central role in this collaboration by facilitating the exchange of data and tasks between devices, thus optimizing performance.

Inter-device communication is particularly relevant in scenarios where computational workload exceed the capabilities of a single device. These scenarios often involve:

- **Parallelism**: The simultaneous execution of tasks across multiple devices to accelerate computations.
- **Memory-Intensive Tasks**: Processing tasks that require a large amount of memory, which may exceed the available memory of a single device.
- **Specialized Processing**: Leveraging specialized processors, such as GPUs, for specific tasks that they are well-suited to handle, while offloading other tasks to more general-purpose processors like CPUs.

5.7.2 Considerations for Inter-Device Communication

Whenimplementing inter-device communication in CUDA, several considerations come into play:

1. Architecture Variability CUDA supports a large range of GPU architectures and different architectures may have varying capabilities and communication methods. Which is why programmers must decide whether to develop a solution that is broadly compatible with multiple architectures or one that is specialized for a specific architecture. This decision can impact the portability and performance of the application.

2. Data transfer Methods CUDA provides functions like **'cudaMemcpy()'** for transferring data between devices. The choice of the most suitable data transfer method depends on factors such as data size, device capabilities and available bandwidth. Larger data transfers may benefit from high-bandwidth interconnects like NVLink, while smaller transfers can use PCIe or memory buses.

3. Synchronization Proper synchronization mechanisms are critical to ensure that computations do not operate on incomplete or outdated data. CUDA offers various synchronization mechanisms, including events and memory fences to manage dependencies between tasks running on different devices. Synchronization ensures data integrity and the correct sequencing of operations.

4. Multi-Device Communication In scenarios involving multiple CUDA devices, such as system with multiple GPUs or heterogeneous processors, inter-device communication bridges the gap between devices. The ability to synchronize and exchange data across these devices is essential for optimizing performance. CUDA's Multi-Device Grouping allows programmers to synchronize threads across multiple CUDA devices, providing a powerful tool for complex parallel computing scenarios.

5.7.3 Inter-Device Communication and Cooperative Groups

Inter-Device communication can benefit from the use of CUDA Cooperative Groups. Cooperative Groups offer high.level abstractions for synchronization and communication within and between thread blocks, making it easier to coordinate tasks across devices. These grouping types: Warp group, Block group, Grid Goup and Multi-Device Group, provide programmers with fine-grained control over synchronization, helping to optimize inter-device communication patterns.

5.7.4 Overlapping Communication and Computation

One of the primary goal of inter-device communication is to overlap communication and computation to maximize system efficiency. This approach minimizes the idle time of processing units, allowing devices to work in parallel. Asynchronous operations, discussed earlier, play a crucial role in achieving this overlap. For example, data transfers can occur asynchronously while GPUs perform computations, reducing overall execution time.

6 Scenarios and Clusters

6.1 Scenarios

In the pursuit of understanding the dynamics of overlapping communication with computation in heterogeneous parallel computing it is essential to construct well-defined scenarios and employ suitable computing clusters that can represent the real-world challenges faced in computational science. These scenarios allow us to systematically evaluate the impact of overlapping strategies on performance, as well as the complexities they introduce.

6.1.1 Partial Derivation with Jacobian Iterations

For this study i have chosen a scenario that embodies the essence of the challenges and opportunities in implementing overlapping communication and computation within heterogeneous computing environments. This carefully selection scenario revolves around matrix operations, a common computational task in bioinformatics and other scientific domains. The primary objective is to calculate the value of each element within the matrix, taking into account not only its own attributes but also those of its neighboring elements. This scenario serves as a versatile test bed to explore the potential benefits and complexities associated with overlapping communication and computation

To effectively execute this scenario, i have adopted an approach that involves partitioning the matrix into segments, with each segment assigned to a different GPU within the heterogeneous computing cluster. This partitioning strategy leverages the parallel processing capabilities of modern GPUs, enabling simultaneous computation across multiple devices. However it is also necessitates inter-device communication as elements located on the boundaries of these GPU-assigned segments must exchange data with their neighboring GPUs. This aspect of the scenario closely mirrors the challenges encountered in real-world scientific computing, particularly within fields such as computational bioinformatics.

The advantages of this chosen scenario are manifold. Firstly it allows us to harness the computational power of heterogeneous computing platforms effectively. By distributing the workload among multiple GPUs, we can potentially achieve substantial speedup in computations. Additionally the scenario encourages the optimization of communication-computation overlap to mitigate communication overhead, which is crucial for overall performance improvement.

It is however essential to acknowledge the address the challenges posed by this scenario. These challenges encompass executing data transfers between GPUs, synchronizing computations across different devices and managing potential memory bandwidth contention. All of these complexities require meticulous consideration to realize the full potential of overlapping communication and computation.

The computation process within this scenario can be summarized by the following steps:

1. **Initial Matrix Splitting**: We would start by splitting the matrix up into potentially several parts, one part on each GPU. These parts would be made up of each side of the matrix, in addition an extra column from the other part, called ghost points. We would then end up with ((M/n)+1)xN matrices on both GPUs, where M is elements in height and N is elements in width, n is number of GPUs. This is only required to do once.

2. **Parallel Computing**: Each GPU would then start to compute on all the elements of the matrix except the ghost points

3. **Asynchronous Data Transfer**: At once the elements which use the ghost points for computation are finished, we start the asynchronous operation of sending those elements over to update the ghost points on the other GPU.

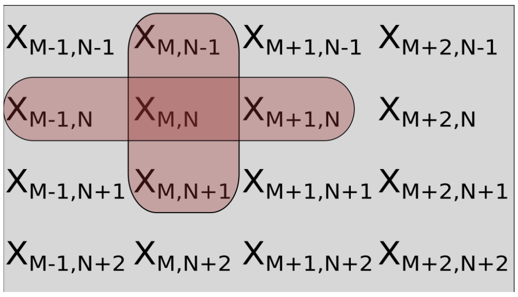
$X_{M-1,N-1}$ $X_{M,N-1}$ $X_{M+1,N-1}$ $X_{M+2,N-1}$
$X_{M-1,N}$ $X_{M,N}$ $X_{M+1,N}$ $X_{M+2,N}$
$X_{M-1,N+1}$ $X_{M,N+1}$ $X_{M+1,N+1}$ $X_{M+2,N+1}$
$X_{M-1,N+2}$ $X_{M,N+2}$ $X_{M+1,N+2}$ $X_{M+2,N+2}$

Figure 11: Required elements for Partial Differential Value for element $X_{M,N}$ in the matrix

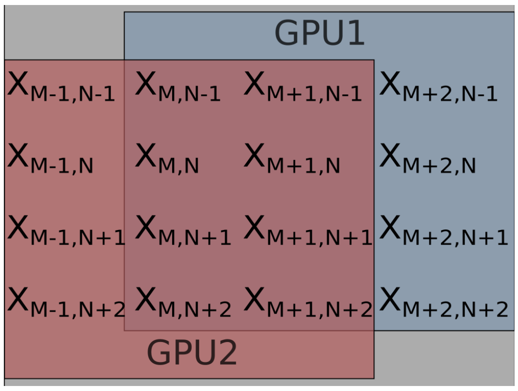
GPU1
$X_{M-1,N-1}$ $X_{M,N-1}$ $X_{M+1,N-1}$ $X_{M+2,N-1}$
$X_{M-1,N}$ $X_{M,N}$ $X_{M+1,N}$ $X_{M+2,N}$
$X_{M-1,N+1}$ $X_{M,N+1}$ $X_{M+1,N+1}$ $X_{M+2,N+1}$
$X_{M-1,N+2}$ $X_{M,N+2}$ $X_{M+1,N+2}$ $X_{M+2,N+2}$
GPU2

Figure 12: Partition of data between 2 GPUs

Different steps in computing Jacobian Matrix between more than 1 GPU

4. **Iterative Computation**: We do this each iteration, starting with elements using ghost points, sends them over as the processor keeps computing on the other points until we are satisfied.

By adhering to this computational strategy we aim to reduce or potentially eliminate the downtime that GPUs would typically encounter when communication and computation are not overlapped. The effectiveness of this approach depends on factors such as the size of the transferred data and the capacity of the interconnect, like NVLink, which should ideally not become a bottleneck.

This scenario serves as a pivotal component of our investigation into the necessity and benefits of overlapping communication with computation in heterogeneous parallel computing, particularly within the domain of computational bioinformatics. Through extensive experimentation and analysis, we intend to draw valuable insights and conclusions regarding the practicality and performance implications of this approach.

6.2 Clusters

I have been allowed to use the eX3 infrastructure which has been provided to me from Simula, and within this infrastructure i have especially utilized 2 Clusters, hgx2q and dgx2q. I have used these as they are Cluster made up of a CPU and several GPUs.

6.2.1 dgx2q

Dgx2q is a system within the eX3 infrastructure which is made up of an Dual Processor Intel Xeon Scalable Platinum 8168 with a cluster of 16 NVidia Volta V100.

To break it down lets first look at the CPU. Being a dual processor implies that the system is equipped with two separate CPUs, each of which is an Intel Xeon Platinum 8168. Intel Xeon is also a server-grade CPU brand from Intel, this means its designed for performance reliability and scalability. Platinum 8168 is a specific model of the Intel Xeon Scalable processor family and it has many attributes.[13]

Now turning our attention to the GPU configuration, the DGX-2 by NVIDIA is also implemented in dgx2q, and it is within this powerhouse that the GPUs are located. These 16 NVIDIA Volta V100 GPUs meant eh system is built to deliver massive computational capabilities. What allows for its high performance is its integration of NVSwitch technology between all the GPUs allowing for intercommunication. [14]

6.2.2 hgx2q

The hgx2q system is made up of an DualProcessor AMD EPYC 7763(Milan) and several NVIDIA Volta A100/80GB GPUs.

CPU	Cores	Frequency	Cache
AMD EPYC 7763(Milan)	64(128 threads)	2.45(3.5 turbo)	4MiB L1, 32MiB L2, 256MiB L3
Dual Intel Xeon Platinum 8168	24(48 threads)	2.7(3.7 turbo)	1.5MiB L1, 24MiB L2, 33MiB L3

Table 1: Testing subject CPUs

GPU	Compute Units/Tensor Cores	Stream Processors/CUDA Cores
AMD MI100	120 Compute Units	7680 Stream Processors
NVIDIA V100	640 Tensor Cores	5120 CUDA Cores
NVIDIA A100	432 Tensor Cores	6912 CUDA Cores

Table 2: Testing subject GPUs

Test System	CPU	GPU
Test System 1	AMD Milan	AMD MI100
Test System 2	Dual Intel Xeon Platinum	NVIDIA V100
hgx2q	AMD Milan	NVIDIA A100

Table 3: Testing Systems Architecture

7 Obstacles

When it comes to writing code for parallel computing in CUDA, one of the initial challenges is designing the code architecture. How can we structure our code to effectively balance computation and communication among different computing devices in a computer?

In traditional C programming, we often write code in a sequential fashion, following a top-down approach, where each function call is executed one after the other in a linear manner. However this method need to evolve when developing code intended to run om multiple devices concurrently. We must intoduce the concept of parallelism which is crucial for optimizing the performance and efficiency of our code, especially when dealing with communication between devices.

With the Matrix Calculation mentioned above in mind we first implemented the basic formula C

```
// C code, selfmade code.

/* Performing Jacobian Matrix Calculation */
// Performing a number of iterations while statement is not satisfied
while (maxdelta > eps && iter > 0) {
    // Maxdelta is the highest delta value found in the matrix
    maxdelta = 0.0;
    // Loops through the matrix from element 1 to -2
    for(int i = 1; i < height - 1; i++){
        // Calculates the element value from row
        int i_nr = i*width;
        // Loops through the matrix from element 1 to -2
        for(int j = 1; j < width - 1; j++) {
            // Calculates each element in the matrix from itself and neightbor values.
            mat_new[i_nr + j] = 0.25 * (
            mat[i_nr + j + 1]   + mat[i_nr + j - 1] +
            mat[i_nr + j + width] + mat[i_nr + j - width]);

            // Finds the highest difference for an element over two iterations.
            maxdelta = max(maxdelta, abs(*(mat + j + i*width)
                                    - *(mat_new + j + i*width)));
        }
    }

    iter--;

    /* pointer swapping */
    float *mat_tmp = mat_new;
    mat_new = mat;
    mat = mat_tmp;
}
```

7.1 GPU Code

To harness the power of additional computer devices, such as GPUs, we need to adapt our code to support parallel execution. The first step in this process is introducing GPU kernel calls.

In this section we will delve into the challenges encountered when transitioning from a CPU-centric codebase to one that works efficiently on a GPU while facilitating communication between thread blocks.

7.1.1 GPU Kernel Implementation

To begin with one must consider the unique memory allocation challenges posed by heterogeneous architectures. Unlike single-CPU implementations where memory allocations are straightforward, working with two distinct processors, each with its own memory requires a more nuanced approach to allocate and manage memory efficiently.

In a traditional CPU implementation, memory allocation is as simple as using the **’malloc’** function, as shown below

```
float *mat;
float *mat_tmp;
mat = (float*)malloc(width*height*sizeof(float));
mat_tmp = (float*)malloc(width*height*sizeof(float));
/* initialization */
fillValues(mat, dx, dy, width, height);
```

However this method allocated memory on the CPU, making it inaccessible to the GPU. To allocate memory on the GPU, we turned to CUDA APIs:

```
    float *mat;
    float *mat_gpu;
    float *mat_tmp;
    cudaMallocHost(&mat, total*sizeof(float*));
    cudaMalloc(&mat_gpu, total*sizeof(float*));
    cudaMalloc(&mat_tmp, total*sizeof(float*));

    /* initialization */
    fillValues(mat, dx, dy, width, height);
```

This code achieves the same memory allocation as the CPU-centric approach but extends it to the GPU. The **’cudaMallocHost’** function allocates memory on the CPU, while **’cudaMalloc** is used for GPU memory, However once GPU memory is initiated, it becomes inaccessible for direct modification by non-device kernels, necessitating the use of CUDA APIs or device kernels for data manipulation.

If one had Unified Memory available, which allows multiple GPUs to access a common pool of memory, one could implement this instead as it simplifies memory management in multi-device scenarios by automatically migrating data between the CPU and GPU as needed. Unified memory eliminates the need for explicit memory transfers between devices, making it easier to handle data in multi-GPU setups. Developers can allocate memory with **’cudaMallocManaged’**[15] and CUDA takes care of data movement.

To enable GPU utilization, we need to integrate GPU kernels into our code. GPU kernels are specialized functions that execute on the GPU, enabling parallel processing. Let us discuss the steps we took in this transformation.

Traditional GPU kernel Call: ’«<blockDim, gridDim»>function(features)’

```
    // CUDA code for launching device kernel

    <<<blockDim, gridDim>>>function(features);
```

Initially when doing the transition to GPU-centric codebase we implemented GPU kernels using the traditional syntax shown above. This approach is commonly used for launching GPU kernels in CUDA. However we encountered a significant challenge related to memory management within the GPU device. The **’«<blockDim, gridDim»>’** syntax is designed with intra-block communication in mind. Which means it is optimized for threads within the same block to communicate efficiently. Threads within a block can easily share data and synchronize. However it does not facilitate communicate between threads in different blocks in any efficient manner. To achieve inter-block communication we would need to rely on accessing global memory, which can be less efficient.

Specialized API for Inter-Block Communication: 'cudaLaunchCooperativeKernel(function, gridDim, blockDim, features)'

```
    // CUDA code for launching Cooperative Group device kernel

    cudaLaunchCooperativeKernel(function, gridDim, blockDim, features);
```

To address the challenge of inter-block communication, it was opted for a different approach by using the **'cudaLaunchCooperativeKernel'** API. This API is specifically designed for kernels that require inter-block communication and synchronization. By adopting the API the code were able to overcome the limitations of traditional kernel calls and improve the efficiency of inter-block communication. This change was crucial for enabling parallel processing on the GPU and achieving optimal performance for out code.

Incorporating **'cudaLaunchCooperativeKernel'** into our code marked a bit of a shift. It not only addressed the challenges associated with inter-block communication but also brought with it a range of additional functionalities inherent to this API.

An example of problem which occurred when using the traditional GPU kernel call which was solved when implementing cudaLaunchCooperativeKernel was when we wanted to find the highest change for an element between two iteration, specifically the following part of the C code written for just an CPU.

```
    // C code, selfmade code.

    // Finds the highest difference for an element over two iterations.
    maxdelta = max(maxdelta, abs(*(mat + j + i*width)
                            - *(mat_new + j + i*width)));
```

The traditional approach in CUDA programming encounters limitations due to the architectures restriction of having a maximum of 1024 threads per thread block, as described in CUDAs documentation.[16] This means that when you need more than 1024 threads, you must utilize multiple thread blocks. This will lead to the challenge where the threadblocks do not naturally support direct inter-block communication. Consequently to share data between different threadblocks, you are forced to transfer data through global memory, which can be less efficient due to slower access times compared to shared memory.

In the initial implementation using the traditional GPU kernel call approach, you need to allocate a global variable named **shared_var**. This variable served as a synchronization point for threads across different thread blocks. To modify such a variable, we had to rely on atomic operations to not achieve racing errors when trying to change it. Here is a simplified representation of how it was first implemented:

```
__device__ int shared_var = 1;

__device__ void calc(float *mat_gpu, float *mat_tmp, int thread, int iter,
    int amountPerThread, int index_start, int jacobiSize, int width, int height,
    float eps, int *maxEps){
   int local_var;
    // Calculating Jacobian Matrix
    while(iter > 0 && shared_var == 1){
        local_var = 0;
        // Calculates each element
        for(int i = 0; i < amountPerThread; i++){
            int index = index_start + i;
            if(index < jacobiSize){
                int x = index % (width - 2) + 1;
                int y = index / (height - 2) + 1;
                int ind = x + y * width;
                mat_tmp[ind] = 0.25 * (
                    mat_gpu[ind + 1]   + mat_gpu[ind - 1] +
                    mat_gpu[ind + width] + mat_gpu[ind - width]);
                if(abs(mat_gpu[ind] - mat_tmp[ind]) > eps){
                     local_var++;
                }
            }
        }
        // If any element in the matrix is 1, the jacobian matrix is not finished, and we
            therefore continue
        maxEps[thread] = local_var;
        __syncthreads();
        // Try to utilize the threads, combining the value of two and two threads, and
            ending up with 1 value at element 0 which
        // is 0 if all elements in are 0, or 1 if one or more elements are 1.
        for(int i = 2; i <= grid_g.num_threads(); i*=2){
            if(thread < grid_g.num_threads()/i){
                maxEps[thread] = maxEps[thread] + maxEps[thread + grid_g.num_threads()/i];
            }
        }
        // If the combined value is larger than 0, it means that there is at least one
            element which could be reduced further.
        if(thread == 0){
            atomicExch(&shared_var, (maxEps[0] == 0 ? 0 : 1));
        }
        // Changes pointers
        float *mat_arr = mat_tmp;
        mat_tmp = mat_gpu;
        mat_gpu = mat_arr;
        iter--;
        __syncthreads();
    }
    maxEps[0] = iter;
    grid_g.sync();
}
```

The primary issue with this code is the use of **atomicExch**, which can be a performance bottleneck when utilized extensively. This bottleneck is especially evident when you have a large number of threads, as each thread block incrementally adds to the queue of threads competing to perform atomic exchanges on **shared_var**. Consequently on systems with a substantial number of threads, the cumulative time spent on atomic operations can significantly impact computation efficiency, especially when performing many Jacobi iterations in a precise manner.

The traditional **'__syncthreads()'** function operates within a single threadblock, meaning it cannot syncrhonize threads across different blocks. As a result, achieving grid-wide syncrhonization becomes a more demanding task. Without a bilt-in mechanism for inter-block synchronization, CUDA programmers often resort to a combination of techniques, such as shared memory, atomic operations and global memory flags. These workaround can introduce additional complexity and potential perofmrance bottlenecks, as threads form different blocks may have varying execution speeds, leading to syncrhonization issues. Therefore careful design and consideration are essential when implementing frid-wide synchronization to ensure that threads across multiple block can effectively coordinate their tasks without introducing race conditions or compromising performance.

To overcome the challenges posed by traditional inter-block synchronization and atomic operations, we turned to the power of Cooperative Groups. Cooperative Groups provide a solution by allowing us to organize threads into groups that extend beyond the boundaries of a single thread block. This approach alleviated the issues we encountered when striving for efficient inter-block communication as mentioned above. By importing Cooperative Groups, we gained the ability to synchronize and coordinate threads more effectively across different blocks. This change not only resolved the synchronization challenges but also helped with its other functionalities such as improved memory management and data sharing between threads. As a result, our code transitioned from a CPU-centric design to a highly efficient GPU-enabled solution, capable of taking full advantage of parallel processing on modern GPUs.

The following code was the utilized when using Cooperative Groups, not to big of a difference, but it would decrease the computational time as it allowed for removing the atomic operation and we managed to remove the shared varaiable between the devices.

```
__device__ void calc(float *mat_gpu, float *mat_tmp, int thread, int iter,
    int amountPerThread, int index_start, int jacobiSize, int width, int height,
    float eps, grid_group grid_g, int *maxEps){

   int local_var;
    int shared_var = 1;
    // Calculating Jacobian Matrix
    while(iter > 0 && shared_var == 1){
        grid_g.sync();
        local_var = 0;
        // Calculates each element
        for(int i = 0; i < amountPerThread; i++){
            int index = index_start + i;
            if(index < jacobiSize){
                int x = index % (width - 2) + 1;
                int y = index / (height - 2) + 1;
                int ind = x + y * width;
                mat_tmp[ind] = 0.25 * (
                    mat_gpu[ind + 1]   + mat_gpu[ind - 1] +
                    mat_gpu[ind + width] + mat_gpu[ind - width]);
                if(abs(mat_gpu[ind] - mat_tmp[ind]) > eps){
                     local_var++;
                }
            }
        }
        // If any element in the matrix is 1, the jacobian matrix is not finished, and we
            therefore continue
        maxEps[thread] = local_var
        grid_g.sync();
        // Try to utilize the threads, combining the value of two and two threads, and
            ending up with 1 value at element 0 which
        // is 0 if all elements in are 0, or 1 if one or more elements are 1.
        for(int i = 2; i <= grid_g.num_threads(); i*=2){
            if(thread < grid_g.num_threads()/i){
                maxEps[thread] = maxEps[thread] + maxEps[thread + grid_g.num_threads()/i];
            }
        }
        // If the combined value is larger than 0, it means that there is at least one
            element which could be reduced further.
        if(thread == 0){
            shared_var = (maxEps[0] == 0) ? 0 : 1;
        }
        // Changes pointers
        float *mat_arr = mat_tmp;
        mat_tmp = mat_gpu;
        mat_gpu = mat_arr;
        grid_g.sync();
        iter--;
    }
    maxEps[0] = iter;
    grid_g.sync();
}
```

7.1.2 Multi-Device Implementation

When delving into the world of multi-device implementation within a heterogeneous computing environment, several unique challenges emerge compared to single-device implementations.

One of the most important challenges when computing in a system which is multi-device within a heterogeneous computing environment is memory management. Unlike single-device implementations where data typically just requires to be located within a single GPU when computing and then sent back to the CPU when finished, multi-device configurations present a unique set of hurdles. In such setups, each GPU device possesses its distinct memory space, necessitating coordination to ensure data availability where it is required.

Crucially, memory allocation strategies must also adapt to this multi-device paradigm. It is no longer sufficient to allocate memory on a single CPU and a single GPU, instead, multiple GPUs must be considered. Consequently the way memory is allocated requires a rethink to ensure the efficient utilization of resources.

To address these challenges, the new approach that has been used when allocating memory on several devices is shown below:

```
float *mat;
float **mat_gpu, **mat_gpu_tmp;

cudaMallocHost(&mat, total*sizeof(float*));
cudaMallocHost(&mat_gpu, gpus*sizeof(float*));
cudaMallocHost(&mat_gpu_tmp, gpus*sizeof(float*));

for(int g = 0; g < gpus; g++){
    cudaSetDevice(g);
    cudaMalloc(&mat_gpu[g], total*sizeof(float));
    cudaMalloc(&mat_gpu_tmp[g], total*sizeof(float));
}
```

In this adapted approach, we initiate memory allocation on the CPU as before, recognizing that we continue to operate from a single CPU. However with the inclusion of several devices, out memory allocation strategy evolves. We allcoate an array, whose length corresponds to the number of available GPU devices. Each element within this array points to memory allocated on a specific device, thus ensuring that each device has its own dedicated memory.

This transition in memory allocation is pivotal, as it enables our code to function efficiently in a multi-device environment. It lays the groundwork for optimized data sharing, synchronization and parallelism, essential for computing effectively with multiple GPUs.

Coding for multi-device environments necessitates a shift in mindset when computing on data. Traditional CUDA programming often focuses on optimizing a single devices performance, whereas multi-device coding requires coordination and synchronization across multiple devices. Handling dependencies and ensuring data consistency between devices can be intricate tasks. Additionally the choice of parallel algorithms and load balancing strategies becomes more critical as different GPUs may have varying computational capabilities. The intricacies of inter-device communication, synchronization and error handling add layers of complexity that must be addressed during the coding process.

Coders usually use techniques like explicit threads synchronization with **'cudaDeviceSynchronice'** and inter-device communication through peer memory access **'cudaMemcpyPeerAsync'** to coordinate multi-device executions. Advanced strategies such as overlapping data transfers and communication can also help mitigate synchronization.

CUDA developers have also sought efficient solutions for multi-device parallelism. While **'cudaLaunchCooperativeKernelMultiDevice'**[17] and the **'multi_grid'**[18] cooperative group were designed to tackle challenges with instructing multiple GPUs effectively, they have been deprecated since CUDA 11.3. Developers therefore still need to explore alternative approaches to address these challenges, like the one mentioned in the section above. It therefore required us to keep using **’cudaLaunchCooperativeKernel’** where we run it for each device available.

The choice of memory handling and synchronization in a multi-device setup introduces its own set of challenges. In a multi-GPU scenario, the communication between devices becomes a significant issuer. CUDA communication APIs cannot be invoked directly within a device kernel, requiring the CPU kernel to manage kernel communications. This implies that the CPU kernel must contain the while loop, which can lead to performance challenges as it necessitates restating the device for each iteration.

As

The choice of having to handle memory differently now starts to bring some problems, firstly it is how we share the elements and transfer them between eachother. Now that we split up memory, we have several options which has its own advantages and disadvantages. Do we send the whole matrix to all the matrixes or do we split the matrix up so each device only requires a potion of the memory. When we transfer data, do we send the whole line as a continuous memory array or do we split it up so we can remove the 2 non required elements which is the border elements. How do we decide when the iteration is done, and how do we communicate this?

This lead to one of the largest problems, the communication between devices. One cannot write CUDA communication APIs within a device kernel, and as such we are required to make the CPU kernel declare the kernel communications. This implies we have to put the while loop in the CPU, which creates problems when it comes to performance as we then for each iteration have to restart the device

Implementing the code in a multi-device therefore required us to make some choices. Firstly it was to implement CUDA Stream as it allows for the devices to be able to run in parallel without to much overhead from the CPU. The stream itself is not to difficult as most of the CUDA APIs has it an a feature in their calls, we therefore only have to create the streams to begin with, which we did with the followin code, one stream for each device.

```
cudaStream_t streams[gpus];
for(int g = 0; g < gpus; g++){
    cudaErrorHandle(cudaSetDevice(g));
    cudaErrorHandle(cudaStreamCreate(&streams[g]));
}
```

After introducing streams it was decided that the while iteration for the Jacobian Computation had to be moved out from the device kernel into the CPU kernel, as having each device run a while loop while synchronizing would be inefficient.

We therefore implemented

```
while(iter > 0){
    for(int g = 0; g < gpus; g++){
        cudaSetDevice(g);
        cudaLaunchCooperativeKernel((void*)jacobi, gridDim, blockDim, kernelColl[g],
            0, streams[g]);
        udaMemcpyAsync(&maxEps_print[g], maxEps[g], sizeof(int),
            cudaMemcpyDeviceToHost, streams[g]);
    }}

    for(int g = 0; g < gpus; g++){
        cudaErrorHandle(cudaStreamSynchronize(streams[g]));
    }

    for(int g = 0; g < gpus; g++){
        float *mat_change = mat_gpu[g];
        mat_gpu[g] = mat_gpu_tmp[g];
        mat_gpu_tmp[g] = mat_change;
    }

    iter--;
}
```

where all the computation would be performed within the device-kernel launched by **'cudaLaunchCooperativeKernel'**.

Thirdly it came to how to even run the kernels, as mentioned above we had to use the **'cudaLaunchCooperativeKernel'** call but it requires some input to be able to run, specifically the features input. As we wanted each kernel to run with its own features, we needed to create an newobject each time.

```
    while(iter > 0){
        for (int g = 0; g < gpus; g++) {
            void **kernelArgs = new void*[9];
            kernelArgs[0] = &mat_gpu[g];
            kernelArgs[1] = &mat_gpu_tmp[g];
            kernelArgs[2] = &maxEps[g];
            kernelArgs[3] = &device_nr[g];
            kernelArgs[4] = &dataPerGpu;
            kernelArgs[5] = &eps;
            kernelArgs[6] = &width;
            kernelArgs[7] = &height;
            kernelArgs[8] = &iter;

            kernelColl[g] = kernelArgs;
        }

        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaSetDevice(g));
            cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobi, gridDim, blockDim,
                kernelColl[g], 0, streams[g]));
            cudaErrorHandle(cudaMemcpyAsync(&maxEps_print[g], maxEps[g], sizeof(int),
                cudaMemcpyDeviceToHost, streams[g]));
        }

        for(int g = 0; g < gpus; g++){
            cudaErrorHandle(cudaStreamSynchronize(streams[g]));
        }


        for(int g = 0; g < gpus; g++){
            float *mat_change = mat_gpu[g];
            mat_gpu[g] = mat_gpu_tmp[g];
            mat_gpu_tmp[g] = mat_change;
        }

        iter--;
    }
```

Now this is an poor implementation of implementing the features, but it was the way i managed to make it work is we are required to update the iter variable each iteration. Implementing a way where we give kernelColl its value outside the iterations and where we manage to update only the 9 element in it each iteration would be a much better implementation, but i have not been able to do it.

Implementing it like the following and making it work would be a more preferred method.

```
for (int g = 0; g < gpus; g++) {
    void **kernelArgs = new void*[9];
    kernelArgs[0] = &mat_gpu[g];
    kernelArgs[1] = &mat_gpu_tmp[g];
    kernelArgs[2] = &maxEps[g];
    kernelArgs[3] = &device_nr[g];
    kernelArgs[4] = &dataPerGpu;
    kernelArgs[5] = &eps;
    kernelArgs[6] = &width;
    kernelArgs[7] = &height;
    kernelArgs[8] = &iter;

    kernelColl[g] = kernelArgs;
}

while(iter > 0){


    for(int g = 0; g < gpus; g++){
        cudaErrorHandle(cudaSetDevice(g));
        cudaErrorHandle(cudaLaunchCooperativeKernel((void*)jacobi, gridDim, blockDim,
            kernelColl[g], 0, streams[g]));
        cudaErrorHandle(cudaMemcpyAsync(&maxEps_print[g], maxEps[g], sizeof(int),
            cudaMemcpyDeviceToHost, streams[g]));
    }

    for(int g = 0; g < gpus; g++){
        cudaErrorHandle(cudaStreamSynchronize(streams[g]));
    }


    for(int g = 0; g < gpus; g++){
        float *mat_change = mat_gpu[g];
        mat_gpu[g] = mat_gpu_tmp[g];
        mat_gpu_tmp[g] = mat_change;
    }

    for(int g = 0; g < gpus; g++){
        kernelArgs[g][8] = iter;
    }
    iter--;
}
```

And lastly it was how to make all of actually work in parallel. As mentioned before we introduced streams for each stream to be computed in parallel, but the streams were still required to be synchronized as the devices are required to communicate with eachother to give the other devices the data they require. We therefore introduced **’cudaMemcpyPeerAsync’** which is a CUDA API that allows streams to communicate directly in parallel.

```
while(iter > 0){
    for (int g = 0; g < gpus; g++) {
        void **kernelArgs = new void*[9];
        kernelArgs[0] = &mat_gpu[g];
        kernelArgs[1] = &mat_gpu_tmp[g];
        kernelArgs[2] = &maxEps[g];
        kernelArgs[3] = &device_nr[g];
        kernelArgs[4] = &dataPerGpu;
        kernelArgs[5] = &eps;
        kernelArgs[6] = &width;
        kernelArgs[7] = &height;
        kernelArgs[8] = &iter;
```

```
            kernelColl[g] = kernelArgs;
        }

        for(int g = 0; g < gpus; g++){
            cudaSetDevice(g);
            cudaLaunchCooperativeKernel((void*)jacobi, gridDim, blockDim, kernelColl[g],
                0, streams[g]);
            cudaMemcpyAsync(&maxEps_print[g], maxEps[g], sizeof(int),
                cudaMemcpyDeviceToHost, streams[g]);
        }

        for(int g = 0; g < gpus; g++){
            if(g == 0){
                // Transfers data to the second from the first device
                cudaMemcpyPeerAsync(mat_gpu[1] + dataPerGpu, 1, mat_gpu_tmp[0] +
                    (dataPerGpu-width), 0, width*sizeof(float), streams[g]);
            }
            else if(g < gpus-1){
                // Transfers data to device g-1 from g
                cudaMemcpyPeerAsync(mat_gpu[g-1] + dataPerGpu*g , g-1, mat_gpu_tmp[g] +
                    dataPerGpu*g, g, width*sizeof(float), streams[g]);
                // Transfers data to device g+1 from g
                cudaMemcpyPeerAsync(mat_gpu[g+1] + dataPerGpu*g-width, g+1, mat_gpu_tmp[g]
                    + dataPerGpu*g-width, g, width*sizeof(float), streams[g]);

            }
            else{
                // Transfers data to second last device from the last device
                cudaMemcpyPeerAsync(mat_gpu[g-1] + dataPerGpu*g-width, g-1, mat_gpu_tmp[g]
                    + dataPerGpu*g-width, g, width*sizeof(float), streams[g]);
            }
        }

        for(int g = 0; g < gpus; g++){
            cudaStreamSynchronize(streams[g]);
        }


        for(int g = 0; g < gpus; g++){
            float *mat_change = mat_gpu[g];
            mat_gpu[g] = mat_gpu_tmp[g];
            mat_gpu_tmp[g] = mat_change;
        }

        iter--;
    }
```

When all of these implementations was done we stumbled upon another problem, mainly how do we send a signal to the CPU that all the GPUs have achieved acceptable states? As this would require each device to send data to the CPU to let it know that the while loop is fulfilled. We therefore implemented a variable called **maxEps** which would be an array consisting of pointers to one element within each device, for which the CPU can check afterwards and decide if the GPUs are finished.

7.1.3 Multi-System Implementation

When delving into the realm of multi-system implementation within heterogeneous computing environment, a host of unique challenges arises compared to a single-system or single-device implementations.

One of the central challenges is the instruction of communication and coordination between disparate computing nodes. Unlike single-device or multi-GPU setups, where communication primarily occurs within a single machine, multi-system setups require robust networking and communication protocols to enable data exchange between different computing nodes. Efficient strategies for data distribution, synchronization and fault tolerance become paramount to be able to truly utilize the collective power of multiple systems.

Fault tolerance mechanisms play an important role in scenarios implemented on multi-system computation. Their job is to ensure that the overall system can continue to operate correctly and reliably, even in the face of hardware failures, software errors or network issues. In a multi-system environment where the failure of a single node can disrupt the entire operation, fault tolerance mechanisms become not just beneficial but critical for system stability and data integrity. They provide the necessary safety net to mitigate the impact of failures, maintain system availability and uphold data consistency across heterogeneous systems.

Memory management in multi-system environments presents a significant hurdle. Each system comes with its own memory hierarchy and address space, necessitating careful planning for data allocation, sharing and synchronization. Strategies such as distributed memory management and memory replication may be some of the strategies employed to ensure data consistency and availability across multiple systems.

Coding for multi-system scenarios demands a holistic approach to parallelism and scalability. Developers must consider load balancing across systems, partitioning of tasks and fault tolerance mechanisms. Ensuring that the computational workload is evenly distributed across multiple systems is crucial for achieving optimal performance. Additionally, dealing with failures and network-related issues requires error handling and recovery strategies to maintain system stability.

Inter-system communication introduces complexities that are absent in single-system implementations. Network latency, bandwidth limitations and communication protocols all play a significant role in system-to-system interactions. Techniques like message passing and distributing computing frameworks are often employed to facilitate communication and collaboration among multiple systems.

Moreover deploying applications across heterogeneous systems involves selecting the right hardware and software components that align with the specific requirements of the task at hand. The choice of system architectures, interconnect technologies and parallel programming frameworks greatly impacts the overall performance and scalability of the application.

8 Results

```
//C, selfmade improvised 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), cudaMemcpyHostToDevice);

    // 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);
}
```

8.1 Expected Results

Estimating the expected runtime presents a complex puzzle, vastly different from the traditional CPU-centric scenarios. At its core, this challenge stems from the multifaceted nature of modern hardware configuration. Algorithm complexity, always a crucial consideration, now intertwines with the intricacies of parallelism introduced by GPUs. Unlike CPUs, which predominantly follows a linear execution path, GPUs excel in parallel processing, opening up the potential for substantial speedsups. However this parallelism also adds layers of complexity when attempting to estimate runtime. Te effectiveness of distributing workloads across multiple GPU cores becomes a pivotal factor and optimizing an algorithm for these scenarios require an intimate understanding of the hardware's capabilities.

Memory bandwidth emerges as another critical facet of the estimation puzzle. The abundant memory capacity and bandwidth of GPUs offer great promise, but efficient memory access patterns and minimizing data transfers between CPU and GPU memory become paramount for realizing these benefits. Every aspect of the data pipeline, from data retrieval and transfer to computation and result storage must be meticulously programmed. This intricate dance between computation and memory introduces nuances that affect runtime unpredictably.

To further complicate matters, calculating the expected runtime involves navigating the unique challenges of overlapping communication and computations. In many scenarios, including scientific computation and computational bioinformatics, computations are distributed across multiple GPUs, each responsible for a distinct section of data. While this harnesses the parallel processing capabilities of GPUs, it also necessitates inter-device communication. Elements on the boundaries of these GPU-assigned sections must exchange data with their neighboring GPUs, a process that introduces latency and synchronization complexities. This interplay between computations and communication showcases the real-world challenges inherent in heterogeneous computing environments.

In essence, attempting to estimate the expected runtime in the presence of GPUs and heterogeneous system is akin to deciphering a multidimensional puzzle. It involves harmonizing algorithm intricacies, understanding the nuances of parallelism, optimizing memory access, and grappling with communication challenges. It is a testament to the evolving landscape of computing, where harnessing the full potential of GPUs and heterogeneous systems offer both unprecedented opportunities and intricate challenges that researchers and practitioners must navigate to unlock new frontiers in computational performance.

8.2 Only CPU

To begin with we implemented a solution for the partial derivation through only solving it on the CPU in hgx2q cluster. We did this to get an comparison when delving deeper into developing codes which utilize other devices in parallel. So what does the truly

```c
#include <stdio.h>
#include <math.h>
#include <time.h>


void fillValues(float *mat, float dx, float dy, int width, int height){
    float x, y;

    memset(mat, 0, height*width*sizeof(float));

    for(int i = 1; i < height - 1; i++) {
        y = i * dy; // y coordinate
        for(int j = 1; j < width - 1; j++) {
            x = j * dx; // x coordinate
            mat[j + i*width] = sin(M_PI*y)*sin(M_PI*x);
        }
    }
}

int main() {
    /*
    width       | int   | The width of the matrix
    height      | int   | The height of the matrix
    iter        | int   | Number of max iterations for the jacobian algorithm

    eps         | float | The limit for accepting the state of the matrix during jacobian
        algorithm
    maxdelta    | float | The largest difference in the matrix between an iteration
    dx          | float | Distance between each element in the matrix in x direction
    dy          | float | Distance between each element in the matrix in y direction

    mat         |*float | Pointer to the matrix
    mat_tmp     |*float | Pointer to the matrix
```

```
*/

int width = 512;
int height = 512;
int iter = 100000;
int print_iter = iter;

float dx = 2.0 / (width - 1);
float dy = 2.0 / (height - 1);
float maxdelta = 1.0;
float eps = 1.0e-14;

float *mat;
float *mat_tmp;

clock_t start, end;

/* cudaMallocHost(&mat,  width*height*sizeof(float*));
cudaMallocHost(&mat_tmp, width*height*sizeof(float*)); */

mat = (float*)malloc(width*height*sizeof(float));
mat_tmp = (float*)malloc(width*height*sizeof(float));

/* initialization */
fillValues(mat, dx, dy, width, height);

start = clock();

/* Performing Jacobian Matrix Calculation */
// Performing a number of iterations while statement is not satisfied
while (iter > 0 && maxdelta > eps) {
    // Maxdelta is the highest delta value found in the matrix
    maxdelta = 0.0;
    // Loops through the matrix from element 1 to -2
    for(int i = 1; i < height - 1; i++){
        // Calculates the element value from row
        int i_nr = i*width;
        // Loops through the matrix from element 1 to -2
        for(int j = 1; j < width - 1; j++) {
            // Calculates each element in the matrix from itself and neightbor values.
            mat_tmp[i_nr + j] = 0.25 * (
                mat[i_nr + j + 1]   + mat[i_nr + j - 1] +
                mat[i_nr + j + width] + mat[i_nr + j - width]);

            // Finds the highest difference for an element over two iterations.
            maxdelta = max(maxdelta, abs(*(mat + j + i*width)
                                    - *(mat_tmp + j + i*width)));
        }
    }

    iter--;

    /* pointer swapping */
    float *mat_tmp_cha = mat_tmp;
    mat_tmp = mat;
    mat = mat_tmp_cha;
}

end = clock();

for(int i = 0; i < 20; i++){
    for(int j = 0; j < 20; j++){
        printf("%.5f ", mat[i*width+j]);
        }
        printf("\n");
    }


    free(mat);
    free(mat_tmp);

    if(maxdelta <= eps){
        printf("The computation found a solution. It computed it within %i iterations(%i -
             %i) and %.3f seconds.\nWidth = %i, Height = %i\n", print_iter - iter,
             print_iter, iter, ((double) (end - start)) / CLOCKS_PER_SEC, width, height);
    }
    else{
        printf("The computation did not find a solution. It computed through the whole %i
             iteration in %.3f seconds \nWidth = %i, Height = %i\n", print_iter, ((double)
             (end - start)) / CLOCKS_PER_SEC, width, height);
    }

    return 0;
}
```

When running the code shown over we got the following result when computing it on a single AMD EPYC 7763 CPU.

```
make[1]: Entering directory '/global/D1/homes/vidaro/Master/Mas/CPU'
nvcc -c main.cu -o main.o
nvcc main.o -o out -L/../../usr/lib/x86_64-linux-gnu/libnccl.so
./out
The computation found a solution. It computed it within 409020 iterations(1000000 -
    590980) and 1763.874 seconds.
Width = 512, Height = 512
make[1]: Leaving directory '/global/D1/homes/vidaro/Master/Mas/CPU'
make[1]: Entering directory '/global/D1/homes/vidaro/Master/Mas/CPU'
rm -f main.o out
make[1]: Leaving directory '/global/D1/homes/vidaro/Master/Mas/CPU'
```

Within the context of the whole project and its target of seeking to find information about the relation and importance of heterogeneous computing, parallel computing and computation-communication overlap, the reported result of running the code on the AMD EPYC 7763 CPU holds large relevance. This benchmark result serves as a foundational reference point for investigating the necessity and potential performance benefits of overlapping communication with computation in modern parallel computing, a topic of paramount importance for various applications in computational science, particularly computational bioinformatics.

9 Thoughts

References

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

[2] 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.

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

[4] 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.

[5] *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).

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

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

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

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

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

[11] Nauman Ijaz. *What is direct memory access(DMA)?* URL: https://www.educative.io/answers/what-is-direct-memory-access-dma (visited on 09/22/2023).

[12] *Adaptive Parallel Computation with CUDA Dynamic Parallelism*. May 2014. URL: https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/.

[13] *Xeon 8168 - Nvidia*. URL: https://en.wikichip.org/wiki/intel/xeon_platinum/8168 (visited on 09/22/2023).

[14] *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.

[15] *CUDA Runtime API*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1gd228014f19cc0975ebe3e0dd2af6dd1b (visited on 08/10/2023).

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

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

[18] *CUDA C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multi-grid-group (visited on 08/10/2023).