Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Tobias Meyer Andersen

Performance Modeling of a Load-Balanced FDM Wave Equation Solver on Heterogeneous Clusters

Master's thesis in MIDT
Supervisor: Jan Christian Meyer
June 2023

NTNU
Norwegian University of
Science and Technology

Tobias Meyer Andersen

Performance Modeling of a Load-Balanced FDM Wave Equation Solver on Heterogeneous Clusters

Master's thesis in MIDT
Supervisor: Jan Christian Meyer
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Dedication

I dedicate this thesis to my supervisor Jan Christian Meyer for his excellent feedback and motivating words throughout the last academic year. I would also like to thank all the members of the HPC lab and Anne Cathrine Elster for letting me be a part of it.

Problem Description

This study aims to develop performance models and load-balancing methods for an effective hybrid HPC implementation of the 2D wave equation.

Abstract

This thesis develops a scalable implementation of the finite difference method to solve the 2D wave equation. We develop algorithms that utilize load balancing at multiple levels to ensure that the computational resources are utilized effectively, even in heterogeneous clusters. The implementation uses GPUs and CPUs to execute stencil operations concurrently and overlaps the communication with computation. We also develop a performance model to predict the runtime of the algorithms accurately.

The most important computational kernels are validated to be close to optimum using roofline analysis. Strong scaling is measured on a single node, achieving a 97% efficiency when using ten GPUs, and across nodes, with 96% efficiency when using one GPU from ten different nodes. Weak scaling is also tested, and on large grids, we measure a 93% efficiency when using ten nodes to compute a set of discretized grids containing almost ten billion points. The implementation uses semi-static load balancing to distribute the workload among heterogeneous nodes and between the computational devices present on each node. We demonstrate the benefits of load balancing by running benchmarks with and without it to measure the difference.

We model inter-process communication and memory transfers between devices using the Hockney model. For computation, we fit simple analytic functions to measurements of how many stencil computations each device can achieve for grids of varying sizes. This will implicitly account for caching and memory-level parallelism. Our performance model then needs to measure the latency and bandwidth between the processes and estimate the computational throughput of each node to predict the runtime.

The performance model is validated by comparing it to the results of the strong and weak scaling experiments, as well as the heterogeneous benchmark showcasing the effects of the load balancing. The results show that the model is particularly reliable for large problem instances, with a mean percentage difference of 5.9%. When modeling a large grid being computed in a heterogeneous environment employing the load balancing scheme, the runtime is modeled with a percentage difference of 3.6%.

Sammendrag

I denne avhandlingen utvikles en skalerbar implementasjon av endelig differansemetode for å løse den todimensjonale bølgelikningen. Vi utvikler algoritmer som benytter lastbalansering i flere lag for å sikre at beregningsressursene tilgjengelig blir utnyttet effektivt, selv i heterogene miljøer. Implementasjonen bruker både GPUer og CPUer for å beregne stensilene is samtid, og overlapper kommunikasjon med beregninger. Vi lager også en ytelsesmodell for å forutsi kjøretiden nøyaktig.

Den mest avgjørende beregningskjernen er validert til å være nær optimal ved bruk av rooflineanalyse. Sterk skalering er målt både på en enkelt node, hvor effektiveten er 97% når 10 grafikkort er i bruk, og på tvers av noder er effektiviteten 96% når ti noder med et grafikkort hver brukes. Svak skalering er også undersøkt, og på store gittere oppnås 93% effektivitet på ti noder, som beregner et diskretisert gitter på nesten 10 milliarder punkter. Implementasjons anvender semistatisk lastbalansering for å fordele last på heterogene noder, men også mellom beregningsenhetene tilstede på en enkelt node. Vi demonstrerer fordelen av lastbalasering ved å kjøre en benchmark både med og uten lastbalanseringa aktivert.

Vi modellerer kommunikasjonen mellom prosesser og minneoverføringer mellom enheter ved bruk av Hockneymodellen. For beregninger tilpasser vi enkle analytiske funksjoner til målinger av beregningshastighet ved ulike gitterstørrelser for å modellere ytelsen. Denne metoden vil implisitt ta høyde for minnehierarkiet og minneparallelismer. Ytelsesmodellen trenger altså å måle latens og båndbredde mellom prosessene, samt estimere beregningskapasitet for å predikere kjøretid.

Ytelsesmodellen er validert ved å sammenligne resultatene fra stark og svakskalingstestene, samt den heterogene benchmarken som tydeliggjør effektene av lastbalansering. Resultatene viser at modeller er særlig nøyaktig for store problemstørrelser, med en gjennomsnittlig prosent differanse på 5.9%. Når store gittere modelleres i et heterogens miljø som bruker lastbalanseringen er kjøretiden modellert med en prosentdifferanse på 3.6%.

Table of Contents

List of Tables

List of Figures

Abbreviations

ASIC	=	Application-specific integrated circuit
CPU	=	Central processing unit
CV	=	Coefficient of variance
DSL	=	Domain-specific language
FDM	=	Finite difference method
FLOPS	=	Floating point operations per second
FPGA	=	Field programmable gate array
GPGPU	=	General purpose GPU
GPU	=	Graphics processing unit
GSTOPS	=	Giga stencil operations per second
HPC	=	High performance computing
MIMD	=	Multiple instruction multiple data
MPI	=	Message passing interface
PD	=	Percentage difference
PTX	=	Parallel thread execution
SIMD	=	Single instruction multiple data
SM	=	Streaming multiprocessor
STOPS	=	Stencil operations per second

Chapter 1

Introduction

The finite difference method is broadly used to solve differential equations numerically. Solving them efficiently using modern accelerators and parallel hardware is a research subject within many engineering fields, such as accoustics[1] and climate modeling[2]. The wave equation, in particular, has applications within geophysics when modeling depth migration, and GPUs are applied to accelerate the computations [3].

Alongside the development of efficient numerical methods, we see a tendency in hardware to grow more heterogeneous. Since the end of Dennard scaling [4], many parallel processing units have been developed to meet the growing computational demand. The development and availability of programmable accelerators have also affected scientific computing. The prominence of parallel processors has reached a point where further speedup is often found in specialized and heterogeneous architectures. For instance, many phones now use a heterogeneous set of cores in the CPU to improve energy efficiency[5], and laptops and stationary computers commonly feature graphics processing units to enhance performance.

This thesis contributes a novel FDM solver implementation that uses load-balancing at two levels to stay efficient in heterogeneous environments. The combination of the usefulness of the finite difference method with the recent growth of heterogeneity in computing motivates this thesis. Load balancing constitutes the primary strategy to utilize the resources in a heterogeneous system. The load balancing occurs both between nodes, but also, between computational units present on each node. We implement a mini-app with a proxy kernel to solve the wave equation. We also create a performance model for the numerical method to argue for the performance's reliability, scalability, and transparency.

1.1 Scope

The proxy kernel we develop solves the 2D wave equation on the Idun cluster[6]. The program estimates the local performance on each node to guide the global domain distribution onto the nodes in a semi-static fashion. The nodes perform a border exchange each iteration using MPI, which is communicated asynchronously to run concurrently with the

computation. The FDM stencil computations are parallelized using CUDA on the GPU and OpenMP on the CPU. Additionally, we use multiple CUDA streams to overlap computation with memory traffic between the CPU and GPU. We run benchmarks on Idun to verify the program's weak and strong scaling properties, the reliability of the calibration phase that performs the load balancing, and to tune parameters for the performance model and validate it.

The performance model applies the Hockney model for communication and memory transfers between the GPU and CPU. We employ simple analytic functions with some restrictions to model the expected performance of the computational FDM kernel that implicitly accounts for memory-level parallelism and caching.

1.2 Thesis Structure

Chapter 2 presents the background information necessary to understand subsequent chapters. This includes introducing the wave equation, the finite difference method and its border exchange, parallel computing on GPUs and CPU, performance modeling, and some statistical metrics relevant to interpreting the results. Chapter 3 presents references to relevant work from the field of stencil computations on accelerators and HPC clusters. Chapter 4 details the load balancing implementation, the kernels, and how the algorithm should adapt to the available resources. Chapter 5 describes the performance model used to break down the performance of the algorithms developed. Chapter 6 explains the experimental setup used when running the benchmark to ensure reproducibility. Chapter 7 contains the results, brief descriptions, and interpretations of their consequences. Chapter 8 concludes the thesis with a summary of the results and their significance and suggests further work.

Chapter 2

Background

In this chapter, we cover the background knowledge needed to understand the subsequent chapters of the thesis. The background contains sections about the finite difference method for the wave equation, parallel computing on CPUs, GPUs, border exchanges, load balancing, performance modeling, and some statistical metrics we apply later.

2.1 The Wave Equation

The wave equation is a partial differential equation that models how waves propagate. Equation (2.1) shows one way to write the equation for 2-dimensional space, which is the focus of this thesis. In this expression, the value of the function u describes how far the wave is from its resting position, and c describes how fast the wave propagates. The equation states that the acceleration of this displacement with respect to time, is proportional to the square of c times the sum of the wave's displacement acceleration with respect to each spatial dimension.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2.1}$$

We must limit ourselves to a domain with a starting condition to utilize the finite difference method. In this thesis, we concern ourselves with square domains for simplicity. The first step to applying the method on a square grid is to discretize the domain uniformly in both spatial directions. Each point contains the displacement of the wave. As the discretization becomes more fine-grained, a linear combination of neighboring points to approximate the derivatives of any order approaches the definition of the derivative. To derive our numerical expression, we first derive how second-order derivatives are approximated with neighboring points, using the concept of limits from calculus.

Equations (2.2) and (2.3) shows the definition of the derivative at a point x and a neighboring $x + \Delta x$.

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x} \tag{2.2}$$

$$\lim_{\Delta x \to 0} f'(x + \Delta x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2.3}$$

Equation (2.4) shows the definition of the derivative, of a function that already is a derivative of another one, this is known as the double derivative.

$$f''(x) = \lim_{\Delta x \to 0} \frac{f'(x + \Delta x) - f'(x)}{\Delta x} \tag{2.4}$$

We want to express the second derivative as a linear combination of neighboring points in the discretization of f. We obtain the expression on that form by inserting Equations (2.2) and (2.3) into Equation (2.4). Equation (2.5) shows the result after simplification.

$$f''(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - 2 \cdot f(x) + f(x - \Delta x)}{\Delta x^2} \tag{2.5}$$

Equation (2.5) shows at point x what is known as the central difference. Inserting our approximations of the derivatives using neighboring points, we can expand Equation (2.1) at time t and position (x,y) with the central difference stencils to end up with Equation (2.6). For brevity, we superscript the timestep and subscript the grid coordinate when referring to the function. The five-point stencil we end up with when using this approximation for the double derivative in two-dimensional axes is visualized in Figure 2.1

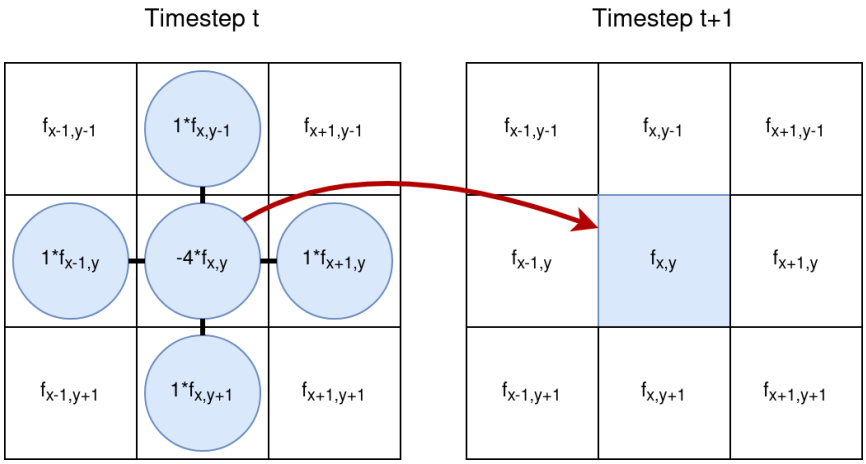


Figure 2.1: Illustration of the 5 point stencil approximating $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ in the point (x, y) at time t

$$\frac{f_{x,y}^{t+1} - 2 \cdot f_{x,y}^{t} + f_{x,y}^{t-1}}{\Delta t^2} = c^2 \left(\frac{f_{x+1,y}^{t} + f_{x,y+1}^{t} - 4 \cdot f_{x,y}^{t} + f_{x-1,y}^{t} + f_{x,y-1}^{t}}{\Delta x^2} \right) \tag{2.6}$$

We solve Equation (2.6) with respect to the value of a point at the next timestep, which results in Equation (2.7).

$$f_{x,y}^{t+1} = 2 \cdot f_{x,y}^{t} - f_{x,y}^{t-1} + \frac{c^2 \cdot \Delta t^2}{\Delta x^2} \left(f_{x+1,y}^{t} + f_{x,y+1}^{t} - 4 \cdot f_{x,y}^{t} + f_{x-1,y}^{t} + f_{x,y-1}^{t} \right) \tag{2.7}$$

Equation (2.7) is our final equation that will be implemented in the code. By having two timesteps as the starting condition, we can estimate the system's state in the next time step. This can be done iteratively to explore how a system behaves over time. We have yet to define what happens when a stencil requires neighboring points that do not exist, which happens when computing the wave equation's stencil for a point at the border of a domain. Many boundary conditions exist, and we use the Neumann boundary condition. It effectively works as a mirror that sends the wave back perfectly when it reaches the edge of the domain. This is implemented by using the neighbor in the opposite direction instead of the missing value outside of the domain.

2.2 Parallel Programming on CPUs

Dennard scaling is the observation that the energy use from a given area on a chip would remain constant, even when shrinking the transistors and filling the area with more microarchitecture [4]. A key property of smaller transistors is that they can work reliably on a higher clock frequency. Dennard scaling worked for several decades, meaning that a lot of the improvement in CPU performance came from the clock frequency being pushed higher as semiconductors were made smaller and smaller. Dennard scaling stopped around 2004, meaning "the necessary transition from complex single core architectures with high operating frequencies to multicore processors with moderate frequencies was caused by the exponentially growing thermal design power (TDP)" [7]. Since the early 2000s, multiple cores working in tandem have substantially improved CPU performance, while the improvement of single-core performance is slowing down.

As of 2023, multicore architectures dominate the CPU market despite the microarchitectural complexity of solving problems like cache coherency and programming difficulties such as avoiding unwanted race conditions. The prominence of multicore systems indicates that the complexities yield results. One of the reasons for their prominence is their ability to implement Multiple Instruction Streams Multiple Data (MIMD), a form of parallelism that is a part of Flynn's taxonomy [8]. MIMD means that the different cores can execute independent streams of instructions simultaneously. The individual cores can also execute their instruction streams on multiple pieces of data simultaneously, for instance, via vector instructions. Programs intended to run on a MIMD architecture are often divided into two categories, shared memory and distributed memory. The two variants are illustrated in Figure 2.2 and are explained in the next paragraphs.

2.2.1 Shared Memory

Shared memory programs have multiple threads of execution that share their view of the memory space. This means that within the same process, multiple threads can read and

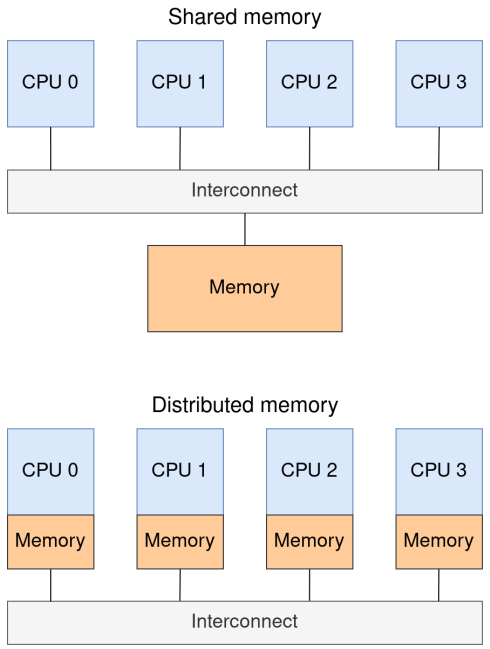


Figure 2.2: Visualization of shared and distributed memory.

write values that are simultaneously visible to the other threads. For this reason, communication between the threads can be done through variables all the threads see. Multithreaded programming comes with pitfalls both in terms of correctness and in terms of performance. It can be difficult to assure correctness in a program where the order in which the threads complete their actions dictates the result, which is known as a race condition. Performance pitfalls can arise for many reasons, often due to inefficient memory usage or synchronization. False sharing of cache lines and excessive use of barriers to make the program easier to understand can both ruin the potential gain of using multiple threads. The benefit of using multiple threads, especially on multiple cores, is that we can multiply our computational throughput, and different threads can hide each others' memory latency as well. We use OpenMP in this thesis as our shared memory programming model.

OpenMP

OpenMP is a specification that provides a high-level interface to multithreading constructs via compiler directives [9]. These directives can, for instance, automatically parallelize for-loops with a known amount of iterations with no jump statements within in using the `#pragma omp parallel for` statement. Since OpenMP aims to abstract away complicated synchronization mechanisms and implementation details, some control over the program is lost, and performance will generally not be quite as efficient as highly optimized implementations with lower levels libraries, such as pthreads.

2.2.2 Distributed Memory

Distributed memory programs work with separate processes that can communicate with each other, but they do not share the view of any memory regions. For these processes to communicate and collaborate, they must send messages to each other. Distributed memory programs then have an additional overhead when communicating that is not present in shared memory programs. In return, we have more isolated parallel execution, and problems like false sharing are avoided entirely. Programming correct behavior and avoiding deadlocks can still be difficult for the programmer. The different processes might be run on different nodes within an HPC cluster, meaning that each process has access to its own hardware. Distributed memory programs can therefore be instrumental when working on a system with multiple CPUs or an HPC cluster. We use MPI to implement distributed memory programs.

MPI

The Message Passing Interface (MPI) is a standard for multiprocess communication [10]. Writing a program within this programming model usually entails using the Single Program Multiple Data (MPMD) paradigm, a subcategory of MIMD. Programs in this class are written in a way that contains the execution of all the different processes in the same source code. This way, the source code will be run in multiple processes simultaneously that use the message-passing interface to cooperate. When using MPI, the process can fetch its own MPI rank, and based on that, we can express that certain computations should be carried out depending on the rank. We also get access to send and receive functions that allow us to send messages between MPI ranks. Two important types of messages for this thesis are blocking and non-blocking messages. Blocking messages wait for messages to be sent before the CPU continues program execution; this serializes the communication with program execution. Non-blocking communication lets the CPU immediately continue running the rest of the program. At some later time, we can wait for the result if we should only continue when certain that the message has been completed. This parallelizes communication with program execution and can play a large role in optimizing programs. Since the maximal overlap of communication and computation is that they overlap entirely, a program can be no more than twice as fast using this parallelization. MPI messages are also sent using different protocols, depending on the message's size. Small messages are usually sent with the eager protocol, which sends the entire message without any network handshake in advance between the ranks. This can be more effective for small messages but causes problems when it is too large to be immediately buffered on the receiving end. When messages are sufficiently large, MPI will use the rendezvous protocol, which initiates communication safely with a three-way handshake. Rendezvous will work effectively for arbitrarily large messages as the size of the messages is communicated at the start. It is possible to set the threshold for when the ranks should choose a particular protocol with environment variables when running an MPI application.

2.2.3 Modes of Scaling

When our problems are divided among ever larger numbers of parallel processors, we need a way to describe their scaling characteristics. Scaling within HPC refers to how much performance we can expect from a program as we vary the problem size, the number of parallel processors, or both. To understand scaling, we must first define speedup and efficiency. Speedup is how much faster one program is than another one. A common way to express it is by comparing the runtime of a parallelized implementation against a serialized one. Speedup would then be written as $\frac{t_1}{t_N}$, where t is the runtime, and in subscript, we write the number of parallel processors. The efficiency is the speedup normalized with respect to the number of processing units N. This is then effectively the speedup per parallel processor, written as $\frac{speedup_N}{N}$.

Two modes of scaling most commonly studied are strong and weak scaling. Strong scaling, first described by Amdahl[11], now referred to as Amdahl's law, considers the case where we scale the number of parallel processors, but the problem size remains constant. To create expressions for speedup and efficiency, we divide the program's runtime into s and p, where s is the proportion of the runtime that is strictly serial, while p is the proportion of the runtime that can be parallelized. See Table 2.1 for the expressions describing both speedup and efficiency of strong scaling. Weak scaling is attributed to Gustafson[12] and considers the case where the problem size per processor remains constant. That is, if we use 4 cores, we also compute a problem instance that is 4 times larger. Table 2.1 also contains the speedup and efficiency when using Gustafson's law.

Table 2.1: Speedup and Efficiency according to Amdahl's and Gustafson's laws

Law	**Speedup**	**Efficiency**
Amdahl's	$\frac{1}{s+\frac{p}{N}}$	$\frac{1}{N \cdot s + p}$
Gustafson's	$s + p \cdot N$	$\frac{s}{N} + p$

These equations imply that for strong scaling, the speedup cannot surpass $\frac{1}{s}$, and the efficiency must go to zero when N increases. This describes diminishing returns, as the potential speedup is limited. With weak scaling, however, we can have arbitrarily large speedups, and the efficiency nears p as the number of processors grows.

It should also be mentioned that these laws are simplifications that, for instance, do not account for the memory hierarchy. This is noted because super-linear speedup does exist. The super-linear speedup can be produced, for instance, by having two cores cooperate on a problem, where when the problem is divided in two, the local sub-problem fits in a higher level of cache.

2.3 GPUs and GPU Programming

Graphics Processing Units (GPUs) have existed since the 1970s to improve the performance of graphics applications. Their computational power was much later used for general computation by taking control over certain parts of the graphics pipeline. With the release of CUDA, general-purpose GPU computing (GPGPU) has become much more accessible [13]. They are some of the most common accelerators in modern computers.

2.3.1 GPU architecture

GPUs differ substantially from CPUs in multiple ways. Primarily they are built for parallel execution at a couple of orders of magnitude larger than CPUs. Instead of having tens of cores, we can run computations on thousands of threads. Some differences at an architectural level are presented in Figure 2.3. Using Nvidia's terminology, their GPUs have a large number of streaming multiprocessors, each of which can execute the same instruction on many pieces of data simultaneously on computational units called CUDA Cores. This thesis will stick to the GPU nomenclature used by Nvidia, as the experiments we conduct only employ Nvidia GPUs.

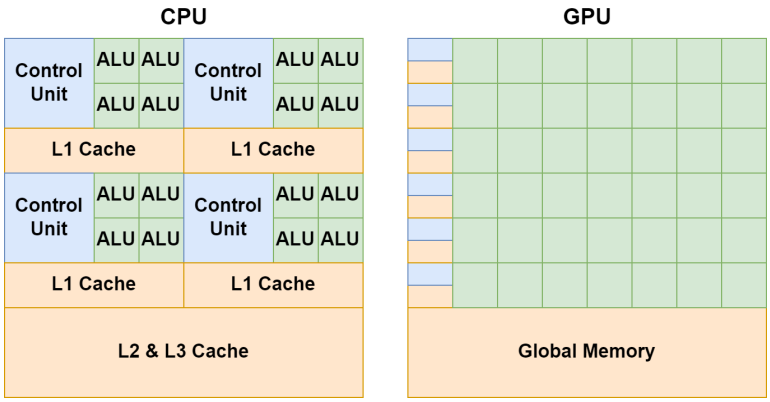


Figure 2.3: Graphic illustrating the difference between CPU and GPU architectures. In the GPU visualization, the rows can, to some extent, be thought of as SMs. The main idea is a larger amount of cores and many computational units that allow for parallel computation on each of those

Note that although we have many more cores available on GPUs, they are individually weaker than CPU cores. GPU cores have slower clock frequency and fewer microarchitectural optimizations. The reason for this is to rather optimize for the number of cores on the same chip, which will also maximize the largest possible amount of concurrent computations.

As Hennessy and Patterson noted in Computer Architecture [14], a modern architectural trend is specialization. The performance of a single, generalist core is improving at a slower and slower rate. Performance gains are more likely now to be found in a more specialized core tailored to a specific problem. Modern GPUs have regular integer computational units, floating point units, tensor cores, and ray tracing cores, among others.

2.3.2 CUDA

Computer Unified Device Architecture (CUDA) was released in 2007 by Nvidia and provided a C-like syntax for programming a GPU freely [15]. The code is separated into functions that will run on the CUDA GPU and the CPU. In CUDA terms, the GPU is the device, and the CPU is the host. We now cover the basics of the CUDA programming model.

Streams, Kernels, and Events

We typically refer to functions that are run on the GPU as kernels. When the CPU calls a GPU kernel, it executes it asynchronously as it simply adds the kernel call to an instruction queue that keeps track of the kernels and runs them when the hardware is ready. This instruction queue runs in order, meaning a previous element must be completed before the next can begin. This is limiting as running multiple kernels at the same time that utilize different parts of the hardware could be run at the same time. For this reason, the programming model includes what is known as streams.

Streams are separate instruction queues that asynchronously with respect to each other, note that all streams still run their own kernels in order. To utilize streams to solve a wide range of problems, it must be possible to synchronize a stream to another to ensure correctness when needed and allow for completely asynchronous computation when it suffices. This mechanism is implemented in CUDA Events. These Events are markers that can signal when a stream has reached a point in its instruction queue that other streams can be aware of. Events are also CUDAs interface to programmatically timing the performance of the code.

Thread Hierarchy

When we call a kernel, we specify how it should be run by describing a wanted thread hierarchy. The hierarchy is illustrated in Figure 2.4. At the lowest level, we have individual threads. A very important caveat to running threads on the GPU is that the number of threads you run will always be a multiple of 32, as this smallest group of threads executes in lock-step. This group of threads is, in CUDA terms, known as a warp. The most important consequence of the lock-step execution is that thread divergence will hurt performance as each possible path through the code will be serialized if threads in the same warp go through different paths. Threads can communicate with each other through various warp-level primitives by sharing information from their registers. This provides the fastest possible communication between CUDA threads, and can, for instance, play a large role in the performance of reduction operations.

At the next level, we have thread blocks. Thread blocks roughly correspond to a streaming multiprocessor (SM). Each SM has its own cache and memory type, known as shared memory. Threads within the same thread block can communicate and cooperate through the use of shared memory. Different thread blocks have no means of synchronization. A thread block may be no bigger than 1024 threads, meaning it contains up to 32 warps.

At the top level of the hierarchy, we have grids. Grids are a description of how many thread blocks are needed. One can think of placing the thread blocks one needs in an actual grid, as the programmer chooses the dimensions of the grid, and CUDA will handle the indexing.

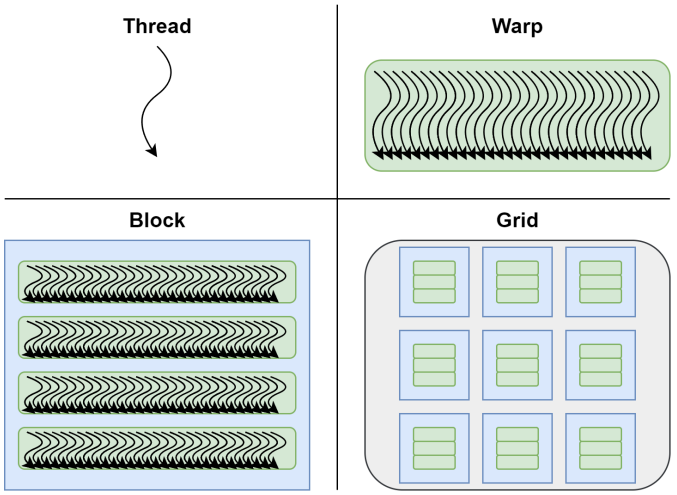


Figure 2.4: The quadrants each illustrate one tier in the most common layers of the CUDA thread hierarchy.

Note that this is not an exhaustive description of the thread hierarchy.

Pinned and Unpinned Memory

CUDA provides functions to transfer memory regions between CPUs and GPUs. The speed of these transfers can depend on how the memory was allocated. When allocating memory on the CPU, one can either use the regular `malloc()` function or the CUDA function `cudaMallocHost()`. The path when transferring data between the two devices is illustrated in Figure 2.5. The crucial difference is that memory allocated with normal C code will end up in the regular pageable memory. If we want to transfer something from the CPU to the GPU, it must first be sent to the pinned memory before transferring it to the device. When using `cudaMallocHost()`, however, the memory is allocated in the pinned memory buffer, and only a single transfer is necessary to reach its destination. Using pinned memory usually yields a speedup of two as we halve the number of memory transfers. The drawback of pinned memory is that it has a fixed size, and if we allocate so much of it that other buffers we want to send can not reside in the pinned memory left, it will slow down the performance of those memory transfers.

Memory Hierarchy

This section explains the parts of the memory hierarchy that are relevant to this thesis. This is not an exhaustive list of the memory hierarchy on Nvidia GPUs.

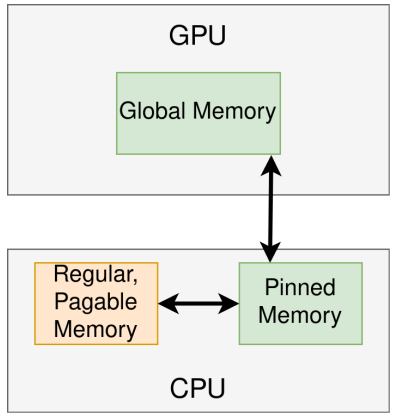


Figure 2.5: Visualization of pinned memory and regular pageable memory (unpinned).

Like CPU programming, maximizing the usage of the memory hierarchy on GPUs can be essential for high performance. In GPU terms, global memory is the lowest level in the memory hierarchy. Global memory can be several GB large but has a high latency.

One level higher, we usually have an L2 cache, but on GPUs, it tends to be only some MB large. Several types of memory are local to the SM, including an L1 cache, shared memory, and registers. In CUDA, the programmer decides how much of the local memory should be shared memory and how much of it should be the L1 cache. There are also instruction caches on the SMs.

The fastest memory level present on the GPU is the local registers to each warp. As stated earlier, threads can use warp-level primitives to access registers of neighboring threads.

Memory coalescing

When a thread issues a fetch from global memory, it is handled through a transaction. Getting as much useful data as possible via a single transaction is essential to avoid the expensive cost of working with global memory. Coalesced memory accesses occur when threads in warps issue a fetch to a contiguous region of memory. When the 32 threads in a warp access a 4-byte value each that is contiguous in memory, a single memory transaction providing a single cache line of 128 bytes will fetch the data needed by all the threads in the warp. Coalesced memory accesses can then reduce the number of memory transactions by a factor of 32 when working with single-precision floats. Considering the cache pollution that would occur when completing 32 transactions that mostly contain unrequested data, speedups of over 32 are possible when perfectly utilizing coalesced memory transactions.

Memory Level Parallelism and Latency Hiding

In addition to packing together as much useful data in each transaction, the high latency of reading from global memory can be hidden by parallel warps executing on the GPU. When one warp is waiting for the result of a read in global memory, the same SM can schedule

new warps that can execute instructions during the wait. If the entire duration of the wait is overlapped with useful computation, we consider the latency of that memory operation to be hidden. It is also possible to wait for the result of many reads simultaneously. This also benefits performance, as the overlap of waiting saves time. The amount of time that can be saved via this form of memory parallelism is formalized via Little's law, which states that the slower the memory accesses, the more we can service in parallel.

2.4 Heterogeneous Computing

Heterogeneous computing is a growing field, with new levels of heterogeneity adding either energy efficiency or performance to modern computing devices. This can be seen in phones using heterogeneous cores, household desktop computers using GPUs, or large cloud server centers using FPGAs or ASICs.

Heterogeneous computing generally refers to making computations on a system that consists of non-identical computation units. This includes using a CPU that controls an accelerator's computations. In this thesis, we use a slightly narrower definition, where we expect a heterogeneous computation to make meaningful computations on different computational units, not simply offloading all computations from one to another. In our case of the finite difference method, this would entail dividing the grid among the CPU and the GPU such that they can carry out computations simultaneously.

2.5 Border Exchanges and Process Topologies

We now cover border exchanges, a vital part of the parallelization of the finite difference method. Suppose we have multiple processes working on the same grid of finite difference computations. To reduce memory usage, each process only stores its local subgrid. Since the stencil computations need the values of the neighboring point, this means at the edge of the local subgrid, we are missing some values, because the neighboring value is stored and computed in another process. This is overcome by having processes with bordering subgrids communicate their border with each other in each iteration. How we divide the grid is of great importance to its performance and how the performance should be modeled.

2.5.1 Process Topology

How we divide a grid among the processes defines the grid's topology. We consider a linear array topology and a cartesian topology. In the linear array topology, each process receives a slice of the global grid where it should execute the FDM computations. This means that every border between neighbors has a side length equal to the global side length and that no process has more than two neighbors. With an increasing number of processes, the amount of communication per process will be a constant proportional to two sidelengths in the grid in the worst case.

The cartesian topology divides the grid along two dimensions, meaning that each process will not necessarily get a full slice of the global grid but ideally is left with a square

portion of the grid. MPIs implementation of the cartesian topology tries to divide the processes among two dimensions, such that the dimensions are as similar in size as possible. This can be done perfectly for square numbers, whereas for prime numbers, this will result in a linear array topology, as there are no alternative factorizations of the number to create the grid. More formally, given N processes, the cartesian topology will constitute a grid of processes of size $A \cdot B$, where A is the smallest divisor greater than or equal to the square root of N, and B is the greatest divisor smaller than or equal to the square root of N. For examples of both topologies, see Figure 2.6

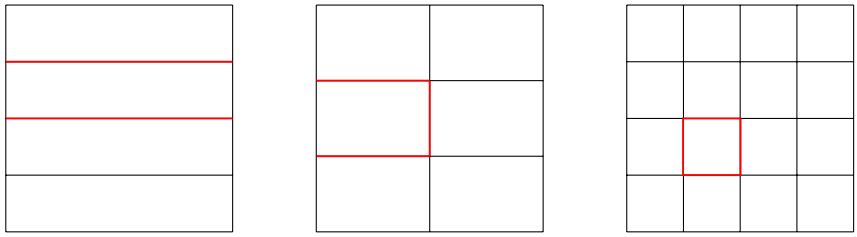


Figure 2.6: Three examples of different grid subdivisions. The leftmost one is a linear array topology with four processes. The middle grid is a cartesian grid topology with six processors, which is not a square number, and hence we get different numbers of columns and rows. The last grid is also a cartesian grid topology, this time with the square number 16, giving a partition of the grid where each process computes the stencils that fall in the area of their square. The red lines highlight the amount of communication needed for an arbitrary process among those that need to perform the most amount of communication.

Both topologies will provide little overhead when the grid has sidelengths that do not divide the number of processors. This still works well because we can extend the domain size until we have no remainder when dividing the grid. Then only the processes that overlap the extra regions created will have fewer columns or rows to compute. The difference in the number of rows and columns must also be lower than the number of processes, so the number of processes must be much larger than the number of nodes we have at our disposition to matter.

There are multiple things to consider when choosing a topology. The most relevant considerations to this thesis are summarized in Table 2.2. The border computation is more efficient when using a linear array topology, as we only need to compute stencils on a contiguous memory region. This will particularly be true on the GPU, where computing a column of stencils will cause significantly more memory transactions. However, the amount of border communication will often be less when using a cartesian topology, but not always, as a prime number of processes will cause a linear topology. Both topologies are appropriate for concise implementations because they have predictable and stable neighborships. The most important difference, however, is that adjusting the size of the local subgrids in a cartesian topology will not preserve neighborships or subgrids being contiguous in memory. This means that load balancing is more feasible when using a linear topology than a cartesian one. For these reasons, the thesis will mainly focus on a linear array topology.

Aspect	Better topology (Linear or cartesian)
Border computation	Linear
Border communication	Cartesian
Implementable neighborships	Both
Load balancing potential	Linear

Table 2.2: Table showing which of linear or cartesian topology is favorable regarding relevant considerations for parallelized FDM on a 2D grid.

2.6 Load Balancing

Load balancing is the act of distributing the workload among parallel processors, usually in a way that will optimize some definition of performance. In our HPC context of differential equations, we want to distribute the workload to yield the lowest runtime possible. We could also optimize for energy usage, but that is outside the scope of this thesis.

We state in Section 2.5.1 that the linear array topology has many advantages over the cartesian topology for the finite difference method. The most important advantage is changing the load balance without adjusting how the code handles changing neighborships. This view of the problem is supported in the survey article on load balancing on heterogeneous systems by Wang *et. al*[16], which states that "distributing independent chunks of work to unidimensional (linear) arrays of heterogeneous processors is easy; distributing independent chunks of work to two-dimensional processor grids is very difficult".

Load balancing is usually a more complicated problem on heterogeneous architectures since an even distribution of the work rarely will be effective. Not all problems can be divided efficiently while preserving correctness, but we can divide the rows into arbitrary subsets when working with FDM grids. When dividing among the GPU and CPU, Figure 2.7 illustrates how splitting the grid in two affects runtime. The main idea is that there is some specific share of the work that should be done on each device to minimize the runtime, which coincides with both units spending the same amount of time in each iteration doing useful work. In addition to balancing the loads between two computational units, we also balance the size of the local grids according to the computational capacity of each node.

2.6.1 Static Load Balancing

Static load balancing splits the work among processes in a planned way, which we often describe at compile time. This includes always dividing it equally or dividing the number of stencil computations proportionally to the computational throughput on a node if that is known at compile time. Static load balancing can be very cheap and effective in cases where the runtime is highly predictable. FDM grids are predictable because the same computation will be done at each point in the grid, and we know exactly how many points exist in each subgrid. Static load balancing does fail to account for situations occurring in runtime. An example of a situation in runtime that might affect the load balance is the GPU context switching to run in another process.

In Figure 2.8, we visualize how the FDM grid can be partitioned between nodes, and

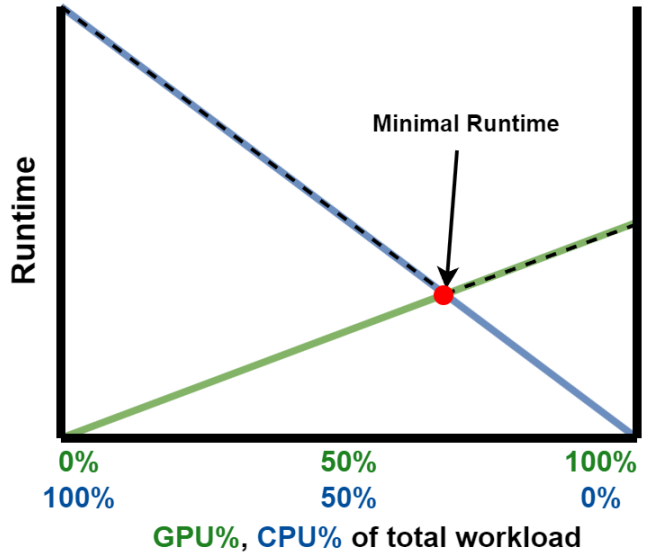


Figure 2.7: Informal graph where the dashed line shows the runtime of a program that can be divided between a CPU and a GPU. The illustration shows that there will be an optimal runtime when the two computational units spend the same amount of time on execution, as none of them will idle.

the computational units on each node, to minimize idle time. Notice that even when we split this local subgrid on a node into two, neighborships are preserved, and the communication patterns remain predictable, as each node implements a linear array topology between its two computational units.

2.6.2 Semi-static Load Balancing

Semi-static load balancing usually entails some form of calibration phase at the start of the runtime that will decide how the workload should be distributed. It differs from static load balancing in that it depends on the performance of the processes at the start of execution and from dynamic schedulers in that it does not continually adjust the balance.

Calibration

When working with multiple nodes cooperating during FDM iterations, we want to give each node a grid of a size that is proportional to the computational capacity of the node. This can be done at the start of the execution by having the first few iterations be used to estimate how many stencil computations each node can execute per second. If we only consider computations when modeling the runtime, this calibration phase will theoretically provide the optimal load balance and runtime.

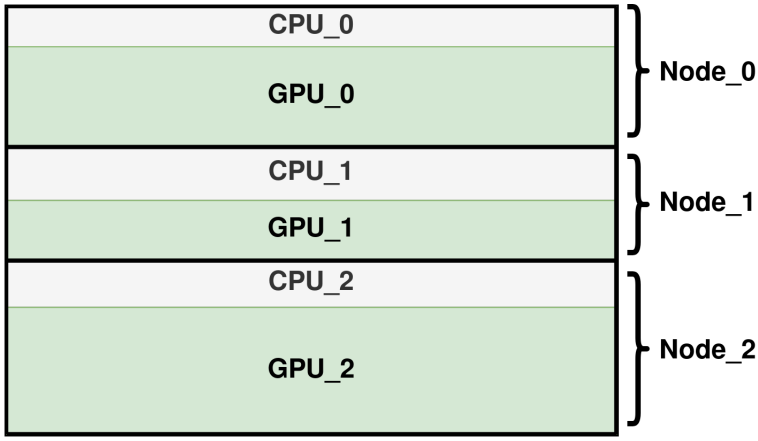


Figure 2.8: Illustration of load balancing at two levels using a linear array topology.

Binary Search

Binary search is a technique to find an element or a value in a sorted array. It works iteratively by reading a value in the middle of the array and reasoning which half of the array desired value must be in. It can also work with implicit arrays. For instance, one could binary search over all float values and find the one closest to the desired number. Binary searches are guaranteed to find the value within a time limit that is proportional to the logarithm of the size of the array we are searching in.

Binary search can be used to pick parameters for heuristics to improve load balancing. We use the heuristic that both computational units should spend the same amount of time running the main computational kernel. For this reason, we can time both units during the calibration phase and binary search over what share of rows on a process should be computed by each unit. This is a valid binary search as we effectively binary search over the differences in runtime between the two kernels and find the percentage where it is closest to zero. In practice, the process that finished first increases its amount of rows and thus will converge to the share that yields the least difference. This heuristic can improve performance by minimizing idle time. By definition, if the two computational units spend a different amount of time computing, one must be idle some of the time. This is still only a heuristic because there are more things to consider when optimizing for runtime than only the main computational kernel.

Ternary Search

Ternary search is a similar technique to binary search, but it is more powerful because it can find the extremum values of a unimodal function. We define a unimodal function as non-decreasing before being non-increasing or vice versa. The ternary search works by sampling two values from an array. Normally we sample the value that is one and two-thirds into the range. The array can be implicit, as when one would ternary search over the real numbers. The ternary search will discard the third of the array that is furthest from

the wanted extremum. Ternary search also has a logarithmic complexity with respect to the number of elements we ternary search over.

For this thesis, the ternary search is relevant as it can be used to calibrate the load balance between computational devices. We can ternary search over the percentage of work on the GPU and find the one that minimizes the runtime of an entire iteration. By timing two iterations with different workload shares between the CPU and the GPU, we can discard a third of the valid workload shares left by reasoning that the minimum runtime for an iteration can not be in the third further from the best workload share we measured. This way, we can very quickly find the workload share that gives the optimal performance of the entire algorithm. This works assuming that the runtimes from each valid workload share form a unimodal function. If this is the case, some workload share is bound to be optimal, and by either increasing or decreasing the number of rows to do on one unit from there, the runtime is bound to go up as we are only increasing how skewed the balance between the units is. In practice, it must work if the CPU and GPU always take longer to compute a larger area.

The main advantage of ternary search over binary search for this thesis is that the ternary search would encompass everything that happens in an iteration, not just the main computational kernel. The binary search is based on a heuristic that the main kernels take the longest time, so it should take the same time on both devices. The ternary search would find the best configuration, regardless of whether that means having those two kernels run in the same amount of time.

2.6.3 Dynamic Load Balancing

Dynamic load balancing can take many forms, but the main idea is that the work share of each process is fluid throughout the runtime. Wang *et al.*[16] concluded in their survey article of load balancing schemes that ”there is no load balancing strategy that can be universally used to solve all problems. Usually, strategies are designed with certain assumptions and favor some type of biases”. Despite this, we will explain one common way of implementing a dynamic load balancer. We can divide all the work to be done into many small tasks, and each process can opportunistically fetch more tasks to execute when they are running out of things to compute. This technique can effectively utilize computational power in highly heterogeneous environments, but it does come with the overhead of synchronization and communication when dealing with tasks. Another way to implement this to avoid a central bottleneck that manages tasks is to have a work-stealing scheme where tasks are divided in the beginning, and neighboring processes may steal a task if they have completed all their tasks. Work-stealing could be implemented with a linear array topology for FDM, where one task could take a row from a neighbor if it spent less time that iteration.

2.7 Performance Modeling

Performance modeling is the field concerned with creating mathematical descriptions of the behavior and performance of a program. This field is particularly prominent within HPC as large systems are purchased to be performant for a long time, and both investors

and programmers must be sure that the system is actually going to scale as expected. A dependable approach to this is to identify machine characteristics that accurately predict the runtime of a certain type of program. This has the advantage of abstracting away hardware details, meaning predictions can be made for hardware that is not unavailable or might not even exist.

We now cover a handful of simple performance models that will be used to characterize and describe the runtime of the wave equation solver this thesis implements.

2.7.1 The Hockney Model

The Hockney Model describes the runtime of a message being sent [17]. The model uses two parameters to predict the runtime of sending a message consisting of N bytes. Firstly, α, describes the latency of sending a message. Secondly, β is the bandwidth between the two communicators. The formula is shown in Equation (2.8). The Hockney model only uses two parameters, which can be measured with a benchmark that sends messages back and forth, often known as a ping pong test.

$$T_{message}(N) = \alpha + \frac{N}{\beta} \tag{2.8}$$

2.7.2 The Fundamental Equation of Modeling

The fundamental equation of modeling was introduced by Barker *et al.*[18] and is shown in Equation (2.9).

$$T = T_{comp} + T_{comm} - T_{overlap} \tag{2.9}$$

The expression says that the runtime of a program can be written as the runtime of the computation, plus the runtime of the communication, minus the overlap of these two. This equation is meant to be used as a starting point for deriving a more detailed model. If the model is insufficient, we could keep expanding the terms depending on what terms are suspected of causing the inaccuracies.

2.7.3 The Roofline Model

The Roofline Model is a powerful tool that can give definitive answers to what parts of a program have the potential for further improvement [19]. The model takes into account the operational intensity of the code and the FLOPS measured at runtime. The operational intensity is the number of logic or arithmetic operations performed per byte of memory traffic. These two values can be used as coordinates to plot a computational kernel in a plane. In this plane, we have floating point operations per unit of time (FLOPS) on the Y-axis. The X-axis is the aforementioned operational intensity. The key property of this diagram is that we can plot a horizontal line which is the absolute peak performance that is possible on the system. This reveals that we can see how close to peak computational performance the kernel is. Additionally, we can draw a diagonal line showing how the computational performance has absolute hardware limits from the memory bandwidth. This captures the memory bottleneck because to achieve a certain amount of FLOPS, the

system must be able to feed the computational units with the necessary data at a fast enough rate. When the program is memory bound, it means that memory is fully saturated, while the computational units must spend time idly, as they complete their job faster than the memory can do its job.

In Figure 2.9 we have drawn a roofline diagram and labeled the memory-bound and compute-bound regions. The point where the memory bounding line and the computationally bounding line meets is called the ridge point.

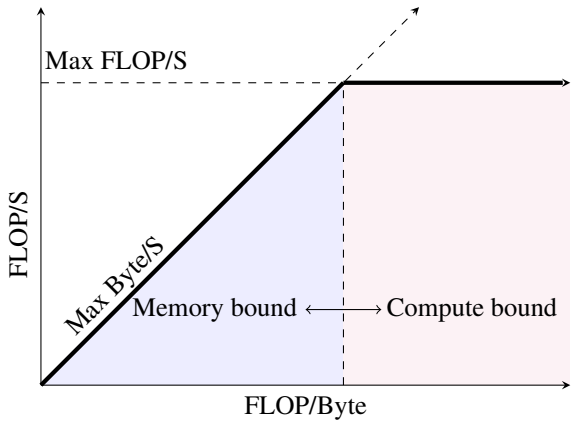


Figure 2.9: Roofline model showing the memory-bound and compute-bound regions. A program will be plotted in one of these regions.

The roofline model captures the dichotomy that every program is limited by memory or computation. The computationally bound programs would benefit from increased processing power, for instance, through faster clock speeds. The memory-bound problems would not benefit from a higher potential computational throughput, but they would benefit from higher memory bandwidths and lower latency through better caching.

2.8 Statistical Metrics

Percentage difference and the coefficient of variation will be applied to interpret the performance model's accuracy and the runtime results' stability. We now quickly review what these metrics represent.

2.8.1 Percentage Difference

The percentage difference metric describes how far away two values are from each other. It works by seeing what percentage of the difference between two values is from their average. The definition is the percentage difference shown in Equation (2.10). The percentage difference is more useful than regular percentages if there is no clear order of the numbers and unclear what number should be a percentage of the other.

$$\text{PD} = \frac{|A - B|}{\frac{A+B}{2}} \cdot 100 = \frac{|A - B|}{A + B} \cdot 200 \tag{2.10}$$

The percentage difference is 0 if the numbers are equal and non-zero. As the relative difference between the values grows, the percentage difference will approach 200.

2.8.2 Coefficient of Variation

The coefficient of variation is also known as the relative standard deviation. it is a metric of variation that adjusts for the absolute sizes of the observations. This makes the variation of measurements at different scales comparable. The formula is the standard deviation (σ) divided by the mean (μ). This is useful for discussing the runtime variation across programs that do not have the same expected runtime.

Chapter 3

Related Work

Applying GPUs to solve numerical problems has been studied since the start of GPGPU computing. ”GPU Computing” by Owens *et al.* [13] was an early argument in favor of GPUs for general-purpose computing in 2008. Since that time, GPUs have been used in countless scientific contexts. Additionally, when the performance of the GPU reaches its peak, hybrid implementations have been explored, which simultaneously use the CPU and GPU to make computations.

This chapter covers relevant articles that research the potential for speeding up the stencil computations that are essential to the finite difference method. Many of the articles attempt to use both CPUs and GPUs for computation, and some of them focus on the scalability of running them on large clusters.

3.1 Development of HPC Hybrid Stencil Computations

In a 2010 article by Xudong *et al.* [20], the authors observed the overhead of moving the data necessary for computation to the GPU before computation could start to be so expensive as not to make it worthwhile on small problem instances. For this reason, the CPU would run the entire computation if it ran faster than the GPU, given the overhead. This way of having either CPU or GPU run the entire computation avoids coding the border exchanges and synchronization between the devices in the implementation. They ran test runs on their machines to measure where the tipping point was, and hardcoded that value into the program. In the same year, the CPU was made to handle some of the boundary conditions of a domain, while the GPU computed the inner points, which ensured some parallel computation that gives speedup[21]. Since the area of a 2D grid scales much faster than the circumference, this does not yield significant speedups for large grids.

To improve upon only either using CPU or GPU, given what would give the best performance, we can split the domain into regions for both and run them simultaneously. A slightly more advanced version of this was done by Vialle *et al.* in 2016 [22]. In addition to a CPU, the machine had two types of accelerators present, both an Nvidia GPU and an Intel Xeon-Phi. They then divided the grid into three parts using a linear array topology

to use all three computational units. This entails two different border exchanges between the different computational units. Still, static load balancing was applied, meaning that the relative computational throughput of each computational unit was used to decide what fraction of the grid it should compute.

In 2018 Siklosi *et al.* [23] contributed to the field with a stencil solver using a CPU and a GPU that implemented a dynamic load balancer. To avoid spending too much time moving memory between the devices when resizing the local grids, the grid of a device can only change every few iterations, taking into account the mean of the last few iterations' runtimes, among other things. The complexity of the implementation increases quickly with this addition, as the program must keep track of dirty and clean regions that must be transferred between computational units whenever the division of the grid is changed. The authors concluded that the implementation was probably too complicated for many scientific applications and that "there is a definite need for higher-level approaches to implementing science code".

Over time simpler interfaces to these efficient FDM solvers have become available. Many DSLs have been developed to abstract away the complexity of writing stencil kernels for parallel systems and, in the last few years, the complexity of writing them for heterogeneous systems. We provide a brief explanation of a few of the articles introducing them.

Sourouri *et al.* [24] created a C-based compiler framework that lets the user describe 3D-stencil operations with high-level directives, similar to that of OpenMP. These directives assume a static work distribution between the GPU and the CPU, but its implementation accounts for clusters with many nodes available. Panda is restricted to 3D stencils for structured grids, making it close to various DSLs for stencil computation. The results were shown to be scalable on a large number of nodes. The paper also bases its GPU kernel on the work of Schäfer *et al.* [25], which implements the *pipelined wavefront algorithm*. We will show through roofline analysis that it is not needed to maximize the usage of the memory bandwidth in the 2D stencil case. Note that this framework tries to split the 3D domain into smaller cuboids present on each node and does not account for heterogeneity in the cluster.

Similar approaches can hide the HPC implementation behind a library interface based on algorithmic skeletons, an idea from Cole in 1989[26]. Hermann *et al.*[27] implemented a library that allowed for using CPUs and GPUs in tandem across many nodes. A linear array topology was used, and the load balancing was a static, even distribution.

Chapter 4

Implementation

This chapter covers the relevant implementation details for the source code. We discuss the algorithms, kernels, and strategies of load balancing. We implement a FDM solver that runs distributed processes that compute a local subgrid each iteration. It also communicates its border points with neighboring processes in the FDM grid. The process topology is a linear array. The user decides at compile-time whether or not the FDM solver will offload computations on the CPU and utilize load balancing at either cluster or node level.

4.1 Load Balancing

When parallelizing a workload, we must decide how to divide a workload among the different computers, threads, or computational units. The rectangular grids we use in our FDM implementation allow for divisions of the global grid into new rectangular grids that preserve MPI rank neighborships. We have the additional advantage of only dealing with contiguous regions of memory when splitting up the global grid in slices. The load balancing in our program aims to distribute the grid to minimize the idle time of computational units.

We first assume that every node will end up with an even chunk of the grid. Then we do internal load balancing on that node. After that, we measure the performance we got on that node and perform external load balancing between the nodes.

4.1.1 Internal Node Workload Balance

Internal workload balance on a single node refers to how we distribute the local subgrid among the computational devices present on a node. In this thesis, we limit ourselves to one CPU and one GPU per MPI rank. We partition the subgrid into two sets of rows, giving the first rows to the GPU set and the remaining rows to the set of the CPU. We define a balanced partition as a partition that gives the best performance, which is closely related to dividing so that the CPU and GPU spend an equal amount of time computing their portion of the grid. This must be true, as any other configuration must produce idle

time for one of the devices, which wastes computational resources. Note that our heuristic is that the computational time spent on the main kernel, the one containing all the interior points of the local domain, should take the same amount of time on the GPU and the CPU.

We implement a program where the user can set the dividing line between the computational units by either providing a percentage of rows that the GPU should compute or by finding it at runtime using a binary search. When the user provides a percentage of the computations we want to run on the GPU, the number could, for instance, be picked by running a STREAM benchmark on the CPU and the GPU and then giving $GPU/(GPU+CPU)$ of the grid the GPU. Finding the right balance using a binary search can be done if our metric of balance is that the CPU and GPU spend the same amount of time on computations. If we time both devices, we can binary search over the number of rows we can give to the GPU. If the GPU spends the least amount of time in one iteration, it increases its size in the next iteration of the search. Otherwise, it decreases. Using a binary search at runtime is bound not to be deterministic, contrary to supplying a predefined balance rate at compile time.

4.1.2 External Node Workload Balance

External node workload balance refers to having a cluster where the workload is partitioned among the nodes to minimize the runtime of an iteration. This is almost the same as ensuring each node spends the same amount of time computing the iteration on its local grid. This division follows that if some node were to spend more time than any of the others, it would waste all the others' computational capacity during that time. For this reason, an ideal distribution gives all the nodes a grid share that is proportional to their computational power. The expression for this is shown in Equation (4.1), where S is the set of nodes each MPI rank has available, and c is the computation capacity, for instance, in the form of stencil operations per second. S_i is the ideal share of the global workload that should be done on node i.

$$S_i = \frac{c_i}{\sum_{i \in S} c_i} \tag{4.1}$$

To estimate S_i, each MPI rank runs ten iterations to measure how many stencil computations per second the MPI rank is capable of. We use `MPI_Gather` to collect the results on the main process. The main process then computes the number of rows for each rank and sends both the number of rows and the local origin of each subgrid using `MPI_Scatter()`.

4.2 Kernels

The kernels carrying out the stencil computation are crucial to high performance. The number of bytes to communicate scales with the circumference of the local subgrid, while its computation scales with the area. For this reason, we expect the kernels to become a performance hotspot when the dimensions are sufficiently large. We now cover how we implement the kernels and later provide arguments for their effectiveness.

The kernel we implement for the FDM solver of the wave equation will carry out the stencil computations for some parts of the grid. The kernel will therefore load values from the previous two timesteps and write the result to the current timestep. Since we read from two timesteps and write to the third, our program allocates memory for three full discretized FDM grids.

We reference two types of kernels; Inner kernels, which compute the interior of a domain, and outer/border kernels, which compute the points on the perimeter of the domain. In the linear array topology, however, the outer kernel only needs to compute the first and last row in the domain.

4.2.1 CUDA kernel

The CUDA kernel uses the large degree of parallelism offered by the numerous SMs on modern GPUs to compute a vast number of stencils simultaneously. We primarily utilize that degree of parallelism by spawning a thread for every point in the grid. The pseudocode for the inner kernel is provided in Algorithm 1. The implementation makes each warp fetch consecutive values in memory, meaning that all the memory transactions are coalesced. Maximizing the amount of coalesced memory accesses is necessary to optimize its performance. We use the roofline model later to argue that the implementation's runtime is very close to being limited by the hardware performance.

This implementation is relatively simple, and since our stencils are in 2D, the wavefront algorithm[25] is not needed to utilize the memory bandwidth efficiently. In that algorithm, a single warp computes multiple rows of stencils, optimizing for L1 cache usage.

Algorithm 1 FDM inner kernel for the wave equation in CUDA

$X \leftarrow threadIdx.x + 32 \cdot blockIdx.x$
$Y \leftarrow threadIdx.y + 32 \cdot blockIdx.y$
if X and Y within local subgrid **then**
    Fetch value of the point above
    Fetch value of the point below
    Fetch value of the point to the left
    Fetch value of the point to the right
    Fetch value of the point at this position
    $U_{X,Y} \leftarrow$ result of stencil computation
end if

4.2.2 CPU kernel

The CPU kernel is parallelized using OpenMP. We use OpenMP directives to abstract away the management of the individual threads. OpenMP will divide the grid into equal portions for each thread, where each thread computes an equal-sized set of consecutive rows. We store the grid row-major, so dividing along the rows will ensure a thread works with a contiguous memory region. We use the standard parallel-for directive as the number of

iterations is known at runtime, and there are no break statements, no threads, or different writes to the same location. All of this means that we need not complicate anything to prevent race conditions. We ensure we have reproducible results on Idun by accounting for thread affinity. Thread affinity defines how different software threads are bound to cores on the underlying hardware. We bind threads to individual cores and make sure the binding is close. A close binding on cores is illustrated in Figure 4.1, meaning that consecutive threads fill up one core before placing a thread on another. This is controlled through the environment variables for OpenMP, *OMP_PLACES*, and *OMP_PROC_BIND*. We also use static scheduling of the rows, which means that each thread computes the same number of rows, and those rows are allocated statically. A roofline chart will be used to examine the performance of the CPU kernel.

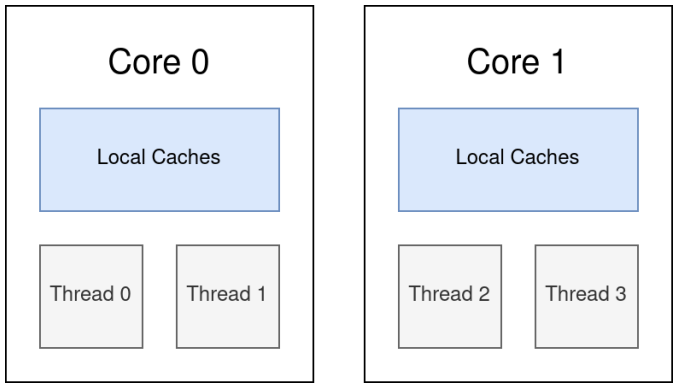


Figure 4.1: Illustration of OMP_PLACES=cores with OMP_PROC_BIND=close

Algorithm 2 FDM kernel for the wave equation on CPU with OpenMP

```
#pragma omp parallel for
for each row Y to be computed on CPU do
    for X less than the width of subgrid do
        Fetch all stencil points (Neuman boundary condition accounted for)
        U_{X,Y} ← result of stencil computation
    end for
end for
```

Note that on the CPU, we have many fewer threads available than rows, which is why for-loops are present in this kernel. In the CUDA kernel, creating a thread that only computes a single point in the grid is performant.

4.3 FDM Iteration Algorithms

Using the FDM to solve the wave equation, we allocate buffers representing three grids, one for each consecutive timestep. To save memory when solving the wave equation

specifically, it is possible to implement using only two buffers. Doing that by writing the result to the oldest of the two buffers would lead to race conditions when using stencils that read more than one value from the oldest time step. Allocating space for a timestep we write to will always work. Since the optimization of reusing the oldest grid in memory does not generalize across stencils, the optimization is omitted. We instead read from the two oldest buffers and write the result of the stencil computation into a third buffer. When working with GPUs, we allocate the space for the grids using `cudaMalloc()`. We use MPI to perform the border exchange between processes, which we do every iteration. We, therefore, also allocate a send-and-receive buffer on the host CPU. We do this using pinned memory with `cudaMallocHost()` to increase the speed of memory transfers between the device and the host, as most communication between the two will be through this small buffer. The OpenMPI version used in this thesis can not use buffers on the GPU, so before sending data, we must move it from the GPU into the send buffer, and after receiving data in the buffer, we must transfer it onto the GPU. We implement this with a packing and unpacking function on the GPU. These packing functions are asynchronous CUDA memory copies that utilize the pinned memory transfer. The packing is only a copy because the data we are interested in is the first and last row, which are already contiguous memory regions. The copying also happens in a separate CUDA stream to that of computation to allow for CPU-GPU memory traffic while the GPU SMs are carrying out stencil computations. When the border being communicated is the border of a CPU domain, we use an extra trick. The receiving and sending MPI buffers that are being sent to and from CPU domains are pointers to the actual FDM grid, meaning that no packing or unpacking is needed for the CPU.

We visualize the main algorithms implemented in Figure 4.2. It shows dependencies between different steps in the algorithm in black arrows and also what the CPU and the two GPU streams can do simultaneously. The green arrows represent a `cudaEventSynchronize()` statement, which we use to ensure correct execution order when using multiple streams. We first cover the algorithm using CPU and GPU on a single node.

4.3.1 Single Process Computation with GPU and CPU

Algorithm 3 shows pseudocode where we use both the GPU and CPU on a single node to compute the stencils. The work of Playne and Hawick inspires the algorithm [28]. Unlike Playne *et al.*, we do not use a stream for inner and outer points but rather a stream for memory operations and one for computations. This way, we know they are not contesting the same resources on the GPU whenever they do things in parallel, ensuring no slowdown from resource contention.

Initially, the compute stream will asynchronously call both the outer and inner compute kernels, with an event recorded between them. Recall that even though both of these kernels are asynchronous to the CPU, they will be executed in order with respect to the GPU stream they run in. Concurrently with their execution, the CPU will run its border kernel. When the CPU border is computed, we synchronize the CPU to the event created in the computation stream after the border kernel. When the synchronization is complete, we can safely start exchanging the borders between the devices. This border exchange is implemented as asynchronous CUDA memory copies that run in the memory stream. The memory transfer is unpinned since the borders reside in the grid buffers allocated on the

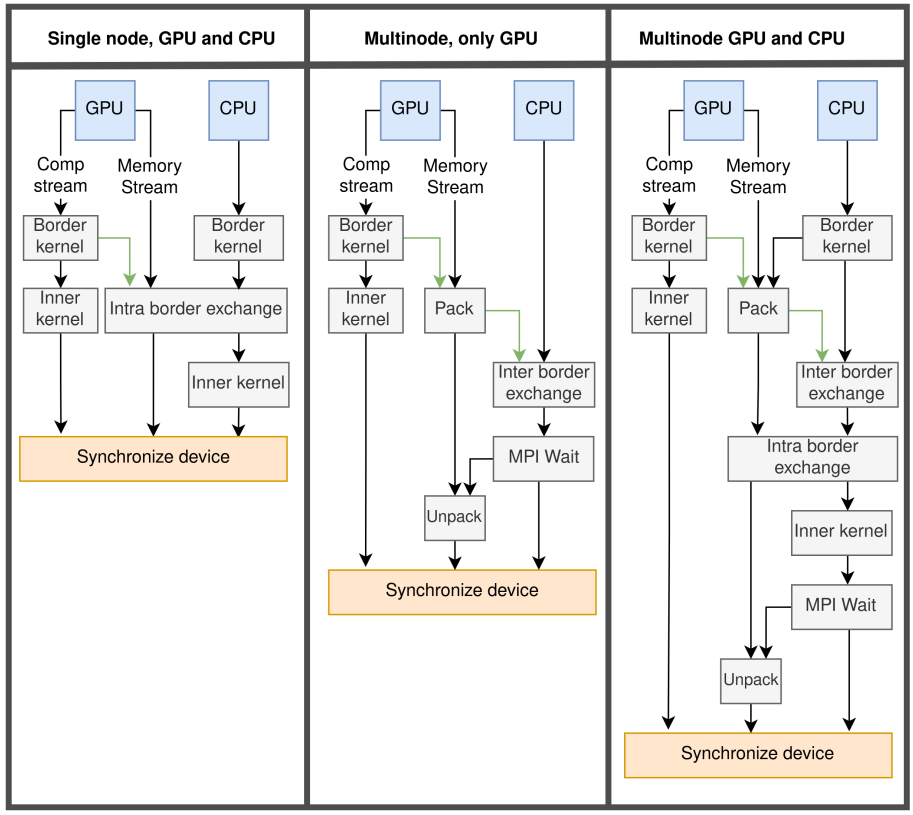


Figure 4.2: The diagram visualizes what is done by the computation and memory stream on the GPU and the CPU. The black lines indicate the order in which the functions are called. Green lines indicate a cudaEventSynchronize, which means the kernel it points from must be completed before the function it points to is called.

CPU with a regular `malloc()`. The purpose of the streams is clear here, the memory copies and the computation of the inner points can run completely in parallel, as they use a disjoint set of resources. After the border exchange is initiated on the CPU, it starts computing the inner points of its domain. When all computations are done, we terminate the iteration by synchronizing the device and rotating the grid buffers.

Algorithm 3 FDM with GPU and CPU on a single node

1. Border kernel (GPU, Computation stream)
2. Inner kernel (GPU, Computation stream)
3. Border kernel (CPU)
4. Synchronize to the end of the GPU border kernel
5. internal border exchange (GPU, memory stream)
6. Inner kernel (CPU)

4.3.2 Multi Process Computation with only GPUs

Algorithm 4 covers the case when we only use the GPU but across multiple MPI ranks. An essential difference from the previous algorithm is that all computations are run on the GPU. Thus, we do not need to perform the internal border exchange on the node. When the border results are ready, we pack them and send them to the MPI memory buffer allocated as pinned memory. The packing happens in the memory stream. After the packing is initiated, we synchronize the CPU with an appropriate CUDA Event to ensure all the values are ready on the CPU before initiating the asynchronous inter-node border exchange with MPI. After that, the CPU waits for all the borders to be received and then calls the CUDA unpacking kernel. The iteration is then complete, and we can synchronize before calling another iteration. Since the CPU does not compute anything while waiting for the asynchronous MPI messages, it could have been written using blocking communication. To keep the code concise, we choose to reuse the implementation of message passing between processes with non-blocking communication.

Algorithm 4 FDM with GPU on multiple nodes

1. Border kernel (Computation stream)
2. Inner kernel (Computation stream)
3. Synchronize to end-of-border computation
4. Pack the borders (Memory stream)
5. Synchronize to end of packing
6. Inter-process border exchange (CPU)
7. Wait for borders to be received (CPU)
8. Unpack the borders (Memory stream)

4.3.3 Multi-process computation with GPUs and CPUs

Algorithm 5 shows the most complicated implementation, where we account for multiple MPI ranks, each of which utilizes GPUs and CPUs to perform computations. This algorithm is a result of merging the two previous algorithms as before we first compute the borders to communicate them as fast as possible. In parallel with said communication, the computational kernels run on both the CPU and the GPU. We make sure to perform both types of border exchange and employ CUDA events where synchronization between GPU kernels and CPU function calls is needed. After the CPU is done with its inner kernel, it waits for the communication to be complete so that it can unpack the values back into the GPU. As mentioned in this chapter, we use the subdomain computed by the CPU as buffers such that we do not need to pack or unpack MPI messages relevant to CPU domains.

Algorithm 5 FDM with GPU and CPU on multiple nodes

1. Border kernel (Computation stream)
2. Inner kernel (Computation stream)
3. Border kernel (CPU)
4. Synchronize with the end of the GPU border kernel
5. Pack the borders (Memory stream)
6. Synchronize with the end of GPU packing
7. Internal border exchange (Memory stream)
8. Inter-process border exchange
9. Inner kernel (CPU)
10. Wait for inter borders to be received (CPU)
11. Unpack the borders (Memory stream)

Chapter 5

Performance model

In this chapter, we provide a high-level, analytic performance model that predicts the runtime of the FDM solver.

5.1 Communication

We model communication using the Hockney model. The formula has two parameters that must be measured on the machine, shown in Equation (5.1)

$$T_{comp}(n) = \alpha + \frac{n}{\beta} \tag{5.1}$$

We model three different forms of communication. Firstly, we must model the runtime of MPI messages that are sent between the processes during the border exchange. Secondly, we model the runtime of the CUDA memory copy which constitutes packing and unpacking. Lastly, we must model the communication between CPU and GPU that occurs during internal border exchange when using a CPU and a GPU. The two last cases are separate cases to model, as the packing and unpacking are made up of memory transfers involving pinned memory, while the internal border exchange uses the regular, unpinned memory. The final model must therefore end up with three sets of Hockney parameters.

5.2 Computation

We model the performance of each GPU and CPU we want to predict the runtime of.

To model the runtime of the computation, we create an expression that describes how many stencil operations a given computational unit can execute per unit of time. More specifically, we fit an expression to find the number of billions of stencil operations that can be completed per second (GSTOPS). The model should also account for the size of the grid. The larger the grid, the more potential for memory-level parallelism. We want

to create an expression with two key properties. The first is that the model for the computational performance should approach an asymptote. When the grids grow large, we expect the performance to level off at a stable value. Functions such as $2 - \frac{1}{x}$ satisfy the asymptotic property but predicts negative runtimes for small grids. The second property we want the model to have is always to predict a positive number of GSTOPS, which goes to 0 when the grid size goes to zero. The function we use to model the performance should then run through the point (0, 0), have an asymptote which it gradually gets closer to, and never be a negative value. We develop separate models for GPUs and CPUs. The CPU can keep the values from the previous iterations in its cache, and for that reason, we expect its performance to rise toward a level of optimal cache usage. After that, the performance will plummet once the grid can not be stored in memory, and finally, we expect the performance to be stable for large grids.

5.2.1 GPU Computation

We choose the formula from Equation (5.2) to model our GPU performance. The function depends only on x, which is the sidelength of a square grid. a dictates the value of the asymptote, and b decides how quickly that asymptote should be approached. To find the values of a and b, we benchmark the computational unit and adjust the values of a and b to fit the function as close to the measurements as possible. We show in the result section that this expression provides reasonable estimations of computational capacity.

$$g(x) = a(1 - e^{-x \cdot b}) \tag{5.2}$$

5.2.2 CPU Computation

Our CPU computation model accounts for cache usage implicitly by modeling a spike in performance for subgrids that fit in the cache. This spike is modeled with a sine function. Since we expect the function to level off once the cache can not store the entire grid, we use a piece-wise function that, after the spike, will model the performance with a flat line. The function takes in the sidelength of a square grid and returns the number of billion stencil operations that we expect to compute per second. The model is shown in Equation (5.3). To fit the function to our results, we thus have to tune the four parameters a, b, c, and d. In the results section, we see that this model does manage to capture the large trends of the CPU's computational power.

$$g(x) = \begin{cases} a \cdot sin(\frac{x}{b}) & \text{if } x \leq c \\ d & \text{otherwise} \end{cases} \tag{5.3}$$

To find the parameters for the two computational models, we use non-linear least squares to fit the function as closely as possible to real measurements. The least squares method will adjust the parameters to approximate the measurements.

5.3 Overlapping Execution

We use the fundamental equation of modeling [18] as a starting point as it is written out in Equation (5.4). In our algorithm, we have a lot of overlapping execution. In several cases, asynchronous functions will be called just after each other, making their execution time practically overlap fully.

$$T = T_{comp} + T_{comm} - T_{overlap} \tag{5.4}$$

Equation (5.5) shows that fully overlapping runtimes can be modeled by only their longest-running component. This observation simplifies the expressions needed to express the runtime of our FDM algorithms.

$$T = T_A + T_B - T_{overlap} = T_A + T_B - min(T_A, T_B) = max(T_A, T_B) \tag{5.5}$$

5.4 The performance models

Which Process to Model

Since we implement a calibration phase that intends to spread the computational load according to computational throughput on each node, the goal is to make every node spend the same amount of time on each iteration. For this reason, it suffices to model only the runtime of a single node when predicting the runtime of the entire iteration. Assuming that each node spends the same amount of time on computations, we should model the one spending the most time on communication to describe the runtime of an iteration. Using a linear topology, it suffices to always model rank 1 when we have multiple MPI ranks if the communication cost between each pair is identical.

Modeling the Calibration

Our calibration phase first finds out what share of work on a node should be done on each device, assuming the local grid on every node is equally large. Afterward, we run the inter-node load balancing by measuring the computational throughput on each node after the local load balance.

We optimistically model the result of the local calibration by comparing the computational throughput both devices have if they had computed the entire local grid. These two values give us the share of the grid that each device will compute. We then know the number of points a device will compute, and use the computation estimation again to approximate its performance, had the points we compute been in a square. Adding the performance of the GPU and CPU together, we will get the value on which we base the external load balance. To model the exact share a node will receive, we must estimate the computational throughput on each node.

The Final Model

We now have an expression for how long it takes to communicate, how fast we can perform the necessary computations, and how to express the overlapping of the two. We also know how many rows to expect on a given rank and how these are distributed among the two devices. We introduce a compact notation for runtimes of different devices in Equation (5.6), and (5.7). The runtime using only a GPU, T_G is dependent on the runtime of the border, $T_{G,b}$, and the runtime of computing the inner points, $T_{G,i}$. The first letter in the subscript describes the device type, where G stands for GPU, and C stands for CPU. The second letter in the subscript describes whether we refer to the inner points, i, or the border points, b.

$$T_G = T_{G,b} + T_{G,i} \tag{5.6}$$

$$T_C = T_{C,b} + T_{C,i} \tag{5.7}$$

We now have everything we need to model the runtime of the FDM algorithm on a homogenous cluster. When using only a single node, with both a CPU and a GPU, we model the runtime with Equation (5.8). We model the runtime using only GPUs but across multiple processes using Equation (5.9). Finally, when using both types of devices across multiple nodes, we model the performance using Equation (5.10). These equations can be verified by inspecting Figure 4.2. The equation's exact form can be obtained by substituting the functions for communication and computation while ensuring that the correct Hockney parameters are inserted for the correct type of communication or transfer.

$$T = max(T_{G,b}, T_{C,b}) + max(T_{G,i}, T_{comm} + T_{C,i}) \tag{5.8}$$

$$T = T_{G,b} + max(T_{G,i}, T_{comm} + T_{packing}) \tag{5.9}$$

$$T = max(T_{G,b}, T_{C,b}) + max(T_{G,i}, T_{C,i} + T_{comm} + T_{packing}) \tag{5.10}$$

Chapter 6

Experimental Setup

This chapter covers the experiments in the thesis. We cover how the experiments are set up and what software and hardware they rely on.

6.1 Hardware Setup

We primarily run the test on hardware offered by the Idun research cluster at NTNU. We now briefly describe the cluster itself, then the key hardware units relevant to the thesis.

6.1.1 The Idun Cluster

The Idun Cluster is an HPC cluster with multiple use cases, including research on energy-efficient computing and prototyping HPC software [6]. The range of shareholders that invest in its expansion explains the heterogeneity present in this cluster. The cluster has 73 nodes connected with InfiniBand. Idun has eight different types of CPUs and five different types of GPUs. Idun thus provides a suitable platform for developing HPC software that thrives in heterogeneity.

The tests we run on a single node are primarily run on idun-04-02, which features an Intel Xeon Gold 6148 CPU and ten V100 32GB GPUs. All the other nodes used also have Intel Xeon CPUs and Nvidia GPUs, which we will cover in more detail in the next section.

The tests we run on multiple nodes are run from Idun-06-01 to Idun Idun-06-16, referred to as Idun-06-XX. These nodes feature the P100 GPU with Intel Xeon E5-2650 v4 CPUs.

The thesis references the specs for each GPU [29, 30, 31] and CPU [32, 33, 34].

6.1.2 Computational Units

Table 6.1, and Table 6.2 contain information about all the computational units used in the experiments.

	Intel Xeon Gold 6148	Intel Xeon Gold 6248R	Intel Xeon E5-2650 v4
Release year	2017	2020	2016
Cores	20	24	12
Base clock rate	2.40 Ghz	3.00 Ghz	2.20 Ghz
L3 Cache size	28MB	36MB	30MB

Table 6.1: CPU specs

	P100	V100	A100
Release year	2016	2018	2020
Architecture	Pascal	Volta	Ampere
Memory size	16 GB	32 GB	80 GB
Memory bandwidth	732.2 GB/s	897 GB/s	1555 GB/s
SM count	56	80	108
Base clock frequency	1.19 Ghz	1.245 Ghz	0.765 Ghz

Table 6.2: GPU specs

6.2 Software Setup

In this section, we provide the versions used of different libraries, compilation flags, and the software used in order to run the benchmarks.

6.2.1 Compilation and Software

Compilation

We use GCC 10.2.0 to compile the C files that constitute most of the source code. This version of GCC implements OpenMP 5.0. Note that this has to be done with a compiler that can compile MPI. We use the OpenMPI 4.0.5 implementation of MPI. We, therefore, use MPICC to compile our C code which uses MPI functionality. Additionally, we have some CUDA files that must be compiled separately using the Nvidia CUDA Compiler (nvcc). We use CUDA version 11.1.1 when compiling the CUDA source code files. When all the appropriate object files have been compiled, we link them using MPICC. Table 6.3 shows the compilation flags we use when compiling the source code.

Compiler	**Flags and Libraries**
MPICC	-lm -fopenmp -O3 -Wall
nvcc	-lstdc++ -lcudart

Table 6.3: Compiler settings

Additionally, we use some environment variables to ensure predictable behavior in runtime when using MPI. We use OMP_NUMBER_OF_THREADS, OMP_PLACES, and

Benchmark name	Benchmark type	measurement
Ping pong test	Proxy kernel	Hockney parameters
STREAM	Proxy kernel	memory bandwidth
Stencil Kernels	Proxy kernel	runtime of computational kernels
Serialized FDM	Miniapp	performance of runtime stages
Strong scaling	Miniapp	strong scaling with processes
Weak scaling	Miniapp	weak scaling with processes
Load balancing	Miniapp	consistency of load balancing
Heterogenous Environment	Miniapp	heterogeneous performance

Table 6.4: Table categorizing properties of the different benchmarks

OMP_BIND_PROC. We explain these variables in the implementation chapter when addressing the CPU kernel.

Slurm

Slurm is used to queue jobs on Idun [35]. It provides an interface that lets us specify the hardware needed and will automatically run the job when the requested resources are available. Slurm sets up the runtime environment according to the users' specifications, providing a reliable experimental setup.

Nsight Compute

Nvidia *Nsight Compute* has also been used to analyze some of the implemented GPU kernels. The result section includes screenshots from the tool. *Nsight Compute* allows for very detailed profiling of individual CUDA kernel calls. We use the tool's autogenerated roofline model when evaluating our main GPU kernel. The memory bandwidth, which produces the diagonal line in the roofline model, has been verified to be accurate with our own STREAM benchmarks.

6.3 Benchmarks

This section explains how the different benchmarks are configured and run. The results are summarized in Table 6.4, which follows the taxonomy for benchmarks described by Donsanjh *et. al* [36]. Note that all the miniapp benchmarks are of the same full-scale FDM program we develop but with different limitations or configurations to isolate certain program properties. The proxy kernels are very small programs that aim only to measure one property of the machine or the performance of a simple action, such as computing the stencil for the wave equation.

6.3.1 Ping Pong Test

The ping pong test measures the time it takes to send and receive a message and is used to measure the parameters of the Hockney model. We measure the runtime of messages of

different sizes and use linear regression to model the communication time. We send messages with sizes ranging from one kilobyte to 100 kilobytes, testing every whole number of kilobytes in between. This interval of message sizes is sufficient to cover all the lengths of messages communicated during the border exchange, as the largest grid we use has a sidelength of 51811, resulting in roughly 100kb of data during a border exchange with two neighbors. We send each message back and forth 100000 times and measure the time using `gettimeofday()` placing MPI barrier to ensure proper timing. We use the arithmetic mean of the runtimes of the same message size to estimate its runtime. This runtime is then used as a data point in the linear regression. The linear regression will provide us with both parameters for the Hockney model. Its slope defines the bandwidth, and where its value when the message size is 0 defines the overhead of sending the message.

6.3.2 STREAM

The STREAM[37] benchmark measures the memory bandwidth of a GPU or a CPU. By implementing simple functions that read from and write to memory without many arithmetic operations, we can calculate at which rate the memory must have been read. The purpose of including a STREAM benchmark is partly to create and verify roofline models. It is also useful since we are dealing with a memory-bound application. The STREAM benchmark can provide an upper boundary to how fast our computations can be made, and we can estimate how close we are to the hardware limit. The implemented STREAM benchmark runs the copy, scale, add, and triad kernels on both CPU and GPU. The benchmark measures the time spent on the kernels when operating on an array of one billion values. We run the kernel 100 times. Because we want to use the result as an upper bound of how fast we can read memory when dealing with stencils, we only keep track of the fastest kernel run of each type across all the runs. On the CPU, we use `gettimeofday()` to measure the spent, while the GPU code uses events and `CudaEventElapsedTime()`.

6.3.3 Stencil Kernels

We run computational kernels to measure how many stencils operations we expect to achieve per second, both on the GPU and the CPU. We are dealing with a memory-bound algorithm, and the larger the grid, the more potential we have for memory-level parallelism. We verify this by running the computational kernel multiple grid sizes. We measure the runtime of a kernel on every grid size with sidelengths $25 \cdot L$, where L is an integer, and the side length is less than 16384. Each grid configuration is run 100 times, and we use the arithmetic mean to aggregate the results into a single runtime estimate. On the CPU we time the results using `gettimeofday()`, whereas on the GPU we use events coupled with `cudaEventElapsedTime()`

6.3.4 Serialized FDM

In order to argue in more detail about the scaling properties of the program, we run a serialized version of the FDM algorithms we create. Serializing the program allows us to measure what portions of the program take what amount of time. We divide the program into its computational kernels, communication, and packing functions. The benchmark

runs 500 iterations of the FDM on different grid sizes, and we store the sum of time spent in each part of the algorithm. The benchmark was run 20 times for each grid size. The arithmetic mean of time spent on each run on each segment is shown in the results. We run this benchmark on an Idun-06-XX node, using P100 GPUs and Intel Xeon E5-2650 v4 CPUs.

6.3.5 Strong Scaling

Strong scaling is measured by running the FDM algorithm for 500 iterations on different grid sizes. The grid sizes are square grids with sidelengths that are powers of two, starting at 128 and ending with 16384. Additionally, we use up to 10 processes that divide the grid among them. This test is run with the processes being on the same and different nodes. Each configuration is run 20 times. Given that we already must test all sizes, with ten different numbers of processes 20 times, the total amount of tests grows very fast when adding more parameters. For this reason, the tests are only run 20 times as we are limited by the time it takes to get our programs through the queue of programs on the Idun research cluster.

6.3.6 Weak Scaling

The weak scaling benchmark is very similar to the strong scaling benchmark. We use the same grid sizes but adjust them when increasing the number of processes such that the local grid size remains the same. We use up to 10 processes on this benchmark as well. We still run each configuration, placing the processes on a single node and across multiple nodes. For simplicity, we implement it on a square grid, meaning that the work of a single process will be a rectangular slice of the total grid. This means that we can accurately keep the number of stencil computations per process constant, but the amount of communication will actually scale as we increase the size of the square. Additionally, the shape of the grid being different when scaling could affect the results.

The weak scaling allows us to test the algorithms on problem instances much larger than what can be present on a single node. For this reason, when measuring the efficiency of the weak scaling, we compare the runtime to the runtime achieved when the local problem instance of the same size was only run on one node. We then obtain perfect efficiency if a given local size runs in the same amount of time regardless of how many processes have that local size.

6.3.7 Load Balancing

We have two benchmarks that test where the load balancing converges. The first version we consider is the one concerning load balancing between processes. In that test, we run the calibration phase of a square grid with sidelength 16384 one hundred times with two processes using identical GPUs. The expected result of this is that the grid is split completely evenly, giving 8192 rows to each process. This benchmark is included to demonstrate the reliability of the calibration phase between GPUs on different nodes. The benchmark is only run 100 times because of limited access to the cluster.

The developed programs are also concerned with load balancing between the CPU and the GPU in a given process. A few small benchmarks are also developed to examine the properties of the binary search load balancing scheme. Firstly, a test measures where the search converges across 500 runs. An additional test runs the program on a single node that uses both a GPU and a CPU, where we manually set the share of work to be done by the GPU. The runtime of each GPU work share rate is then measured such that we can tell what configuration is optimal. We run each configuration 25 times.

6.3.8 Highly Heterogenous Environments

The last benchmark implemented is one meant to test how well the algorithm scales when working on heterogeneous clusters. This test requires queueing up for multiple highly contested resources on the Idun cluster while writing the thesis, which is why the scale of the experiment is much smaller than ideal.

The benchmark works by running 500 iterations of the FDM algorithm on different setups and comparing the average runtimes across five runs. The first two setups use one and two processes using a P100 GPU. The third has one node with a P100 GPU and another with an A100 80GB GPU. The fourth benchmark test also uses a P100 and A100 GPU but enables the inter-node load balance. The fifth setup has the same GPUs as the previous one but uses its local processors: an Intel Xeon E5-2650 v4 and an Intel Xeon Gold 6248R. This benchmark tests how well inter-process load balancing works in practice. The results can be compared to theoretical limits to indicate how far the load balancing is from an idealized scheme.

Naming Convention

We create a systemic naming convention to reference the individual runs from the heterogeneity test. The 3 GPUs will be referenced by their first letter, meaning 'P' stands for P100, 'A' stands for A100, and 'V' stands for V100. If we have multiple GPUs, we concatenate the letters such that we can tell from the string how many GPUs we use. We can also add 'C' after the GPUs to indicate that we also use the local CPUs for computing. We separate the computation units from the load-balancing part of the name with an underscore. We have two types of load balance from the perspective of a node, namely internal and external. Internal refers to the local load balance between computational units on a node and is represented with a capital 'I'. The external load balancer is between nodes in the cluster and is represented by the letter 'E'. Listed below are a couple of examples.

- AA means that we only use A P100 GPUs
- AP_E means we use a node with A100, another one with a P100 card, and the E indicates that we perform the load balancing calibration between the nodes.
- APC_EI means that we use an A100 card, and a P100 card, the CPUs present on the nodes, and we perform load balancing internally on the nodes and externally between the nodes.

Chapter 7

Results and Discussion

In this chapter, we argue for the validity of the FDM algorithm, show the results for all the different benchmarks run, and interpret the results according to the background theory.

7.1 FDM Validity

Numerical solutions to differential solutions implemented incorrectly will usually diverge, as all important physical properties are unlikely to be preserved. Figure 7.1 shows the result of the FDM algorithm visualized after 4600 and 40000 iterations. The results look very reasonable and correspond with a minimal implementation provided by the author's supervisor at the start of the pre-project. Interestingly, the results of our different algorithms are not identical. In a single iteration, the largest absolute difference in the value of a single point computed on a CPU and a GPU was 0.0000031. We attribute this minimal value to the difference in instruction sets on the two devices. This difference is most likely related to the FADD instruction in the PTX instruction set for CUDA GPUs. FADD is the mnemonic for "fused multiply and add," which features a different rounding scheme than executing a MUL instruction and then an ADD instruction on a CPU. With these results, we assume the implementation to be correct, and the results are the same when parallelizing the computation across multiple processes.

7.2 Inner Kernel Profiling

This section shares the results of profiling the inner kernel on the CPU and GPU. The reason for only profiling this particular kernel is that it dominates the runtime on larger instances, so its performance is more important than the others.

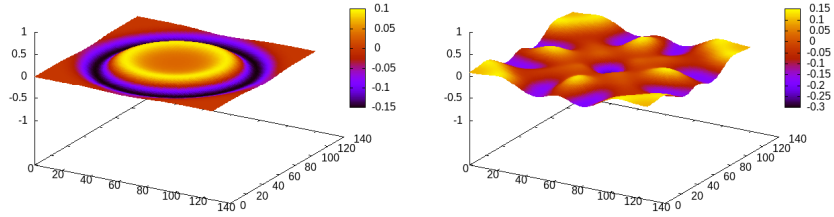


Figure 7.1: Example of FDM one a 128 by 128 grid, after 4600 and 40000 iterations.

GPU

We now analyze the roofline plot inner GPU kernel. Using *Nsight Compute* with an Nvidia Titan RTX, we get the roofline plot shown in Figure 7.2. The figure results from Nvidia's software analyzing a single kernel function call on a square grid with sidelengths 16384. It features a circular point which is where our inner kernel is on the diagram, and a diagonal line indicating programs limited by global memory access bandwidth. Additionally, there are two horizontal lines showing the maximal achievable FLOPS when using double and single precision floats. The position of the kernel indicates that the kernel is memory bound and that its current performance is very close to optimal concerning hardware limits. This indicates that the implementation is close to its limits, not that other implementations with different operational densities cannot surpass it. To trust the results of the CUDA tool, the memory bandwidth is verified with the STREAM benchmarks, which suggests that the roofline drawn is based on some experimental value, not theoretical limits. The generated PTX instructions are also inspected to verify the arithmetic intensity.

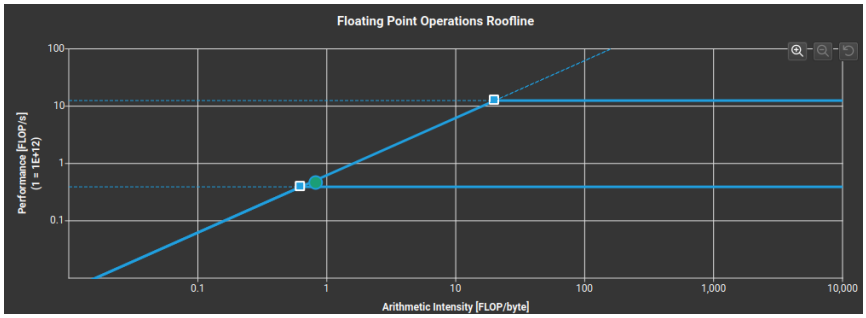


Figure 7.2: Generated roofline plot showing performance of the inner kernel on GPU.

Nsight Compute also provides a bar chart with similar information as the roofline chart. The Speed Of Light plot (SOL) in Figure 7.3 shows the percentage of max theoretical performance the kernel obtains regarding computation and memory bandwidth. This chart indicates that we are not close to using all the computational resources available, which is expected when dealing with a memory-bound proxy kernel. It also shows that our memory throughput is at 87% of theoretical max, which leaves little room for significant

improvement of memory bandwidth.

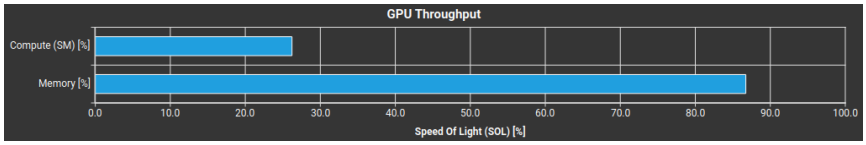


Figure 7.3: Generated bar chart showing how much of theoretical hardware resources are being utilized.

Note that these generated diagrams are created by running a 16384 by 16384 grid on the Titan RTX card. The values may vary between the cards. Given that all the cards are from the same vendor and their structure is very similar regarding memory hierarchy, we assume the results from the Titan RTX sufficiently represent the others in the claim that the kernel is close to its memory-bound limit.

CPU

Figures 7.4, 7.5, and 7.6 show the roofline charts for Intel Xeon Gold 6248R, 6148, and E5 6250 V4 respectively. Using a high-level approach, we create a roofline chart by studying the source code, not the generated machine instructions. Since each stencil operation reads six floats and writes one float, each stencil operation will use $7 \cdot 4 = 28$ bytes of memory. We must, however, account for the fact that most of these values will be cached, as consecutive stencil computations will use many of the same values. The cached values will not affect the memory traffic we care about when creating a roofline chart. Assuming a near-optimal cache, each value will only be read once; this would induce only three reads or writes per stencil, as every point in each timestep must be read once. With this lower bound, we get 12 bytes of memory traffic per stencil operation. We must also compute the result of 10 floating point operations, which leaves us with an arithmetic intensity of 0.833 FLOPS per byte of memory traffic. We use the results from Section 7.3.3 to plot the achieved FLOPS for large grids. We create the roofline diagrams by combining the achieved FLOPS with theoretical limits on CPU memory bandwidth. Note that Figure 7.4 has an additional diagonal line, which is the max bandwidth achieved in the STREAM benchmark, which the kernel outperforms by 4%. This extra line is absent in the other figures because the corresponding STREAM benchmarks were run on two sockets instead of one, making the comparison invalid. Although the kernels utilize most of the bandwidth available, a 1.25 theoretical speedup is still available. Generally, optimizing for results close to the theoretical max is more difficult on CPUs than on GPUs due to the amount of microarchitecture that has to work perfectly in tandem to reach the limit.

These roofline diagrams for the CPU show that the kernels are probably close to their limit in terms of performance because most of the memory bandwidth is utilized in this memory-bound problem.

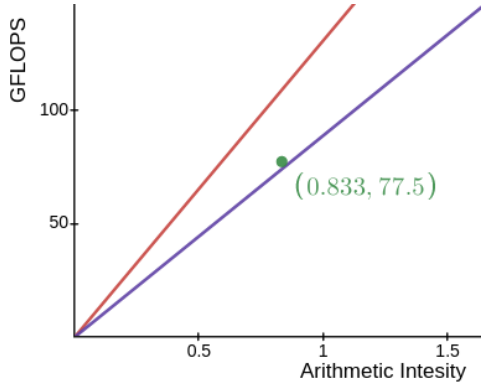


Figure 7.4: Roofline diagram showing the performance of the inner kernel on an Intel Xeon Gold 6248R using only 24 cores. The purple line is the memory limit given by the STREAM results, while the red line is the theoretical max.

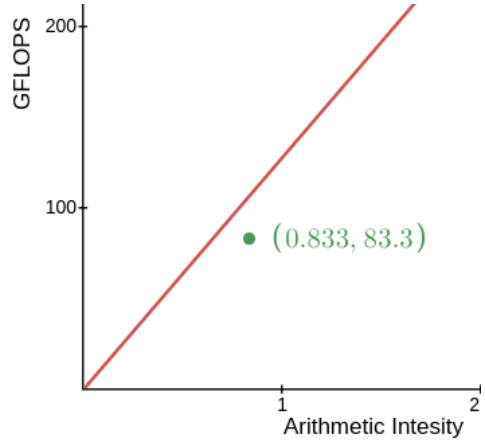


Figure 7.5: Roofline diagram showing the performance of the inner kernel on an Intel Xeon Gold 6148 using all its 40 cores. The red line is the theoretical max.

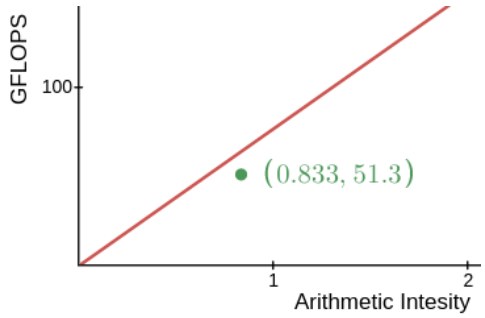


Figure 7.6: Roofline diagram showing the performance of the inner kernel on an Intel Xeon Gold E5 6250 V4 using all its 24 cores. The red line is the theoretical max.

7.3 Benchmark Results

In this section, we show the results from each of the benchmarks described in the experimental setup and interpret the results.

7.3.1 Ping Pong Test

CPU

The ping pong test measures the time it takes for 100000 messages of different sizes to be sent back and forth. Figure 7.7 shows the results from running our MPI benchmark with two processes on Idun-04-02. The linear approximation of the runtime of a message is displayed in Equation (7.1). The function takes in the number of kilobytes MPI will send and returns the runtime in microseconds. The Hockney model then estimates that the bandwidth is 8.54 GB/s, and the overhead of sending a message is 2.92 microseconds. The bandwidth is low compared to the theoretical max because the messages we send are small.

$$f(n) = 0.117n + 2.920 \tag{7.1}$$

From the scatter plot, we can see that the overhead estimate is greatly exaggerated and is, in reality, closer to 0.6 microseconds. These small values are poorly modeled due to the discontinuity of the message passing time. This discontinuity arises from MPI switching from the eager to the rendezvous protocol. The linear model can not capture this critical detail, which also causes the rendezvous messages to be modeled with some inaccuracy.

We also run the benchmark on Idun-06-XX nodes to test scalability across nodes. We run the message passing benchmark for these 14 Idun nodes that are located in the same cabinet physically. These are the nodes Idun-06-01 to Idun-06-16, without nodes Idun-06-9 and Idun-06-10, which are unavailable at the time of running benchmarks. The results are summarized in Figure 7.8 and Figure 7.9. These two figures show the two parameters that make up the Hockney model.

7.8 shows the bandwidth between each pair of nodes in GB/s under the assumption that communication costs both ways are symmetric. The bandwidth is around 4.1GB/s, which is low for the given hardware. The messages we send are not larger than 100 kilobytes and will not fully saturate the memory communication hardware. See Section 6.3.1 for explanations of why only small message sizes are considered.

Figure 7.9 shows the latency between each node in microseconds. The median latency is 2.9 microseconds. The latency is stable among all pairs of the nodes measured. The median latency of 2.9 microseconds of the Hockney model is almost the same as with communication between processes on Idun-04-02. The real latency within the same machine should be significantly lower than that of communication between different ones, even though the Idun-06-XX machines are in the same physical cabinet. This can be explained by looking in more detail at the linear regression for the communication between the nodes. Figure 7.10 visualizes the linear regression and the measurements from benchmarking communication between nodes. The figure shows that since we do not observe a discontinuity in the runtime when MPI switches protocol, the linear regression is a better fit when modeling the communication between two nodes, particularly the latency is

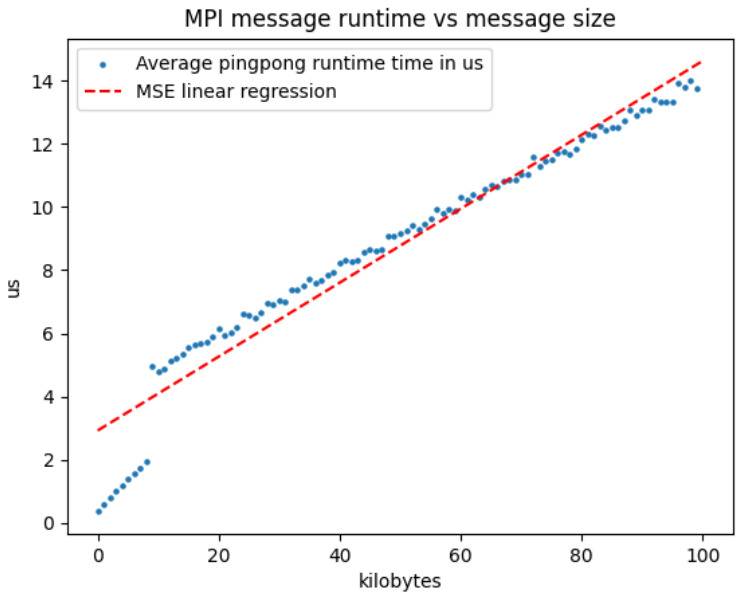


Figure 7.7: This diagram shows the Hockney model fit to the ping pong benchmark results on Idun-04-02.

modeled more accurately. It means that we model the latency to be the same between machines, and inside one, the latency inside a node is overestimated. It is overestimated because the pronounced protocol switch can not be modeled with a straight line. It is by chance that the latency between and in nodes turns out to be the same when modeled with linear regression.

Finally, we show an example of communication between nodes with the P100 GPU and A100 GPU, as these will be used together in the benchmarks testing performance on highly heterogeneous systems. The two nodes communicating are Idun-04-07 and Idun-06-14. The results are shown in Figure 7.11. The latency is 4.6 microseconds. The latency is higher since the two nodes are now localized in different parts of the physical cluster. Since we are still working with node-to-node communication, the bandwidth remains around the same, at 4.1GB/s. Note that there is a discontinuity at around 40kb, indicating a change in MPI protocol.

Combining the results in the section, we see that the linear model struggles to capture the details of the communication costs. The protocol switch causes a discontinuity that the linear regression can not model. Because of this, the latency of communication between processes on the same node and different nodes in the same cabinet is estimated by the model to be the same. On closer inspection, we still see what we expect, namely that the latency between processes on the same node is lower than the latency between nodes. We also verify that the latency between nodes in different cabinets is still higher than between nodes in the same cabinet.

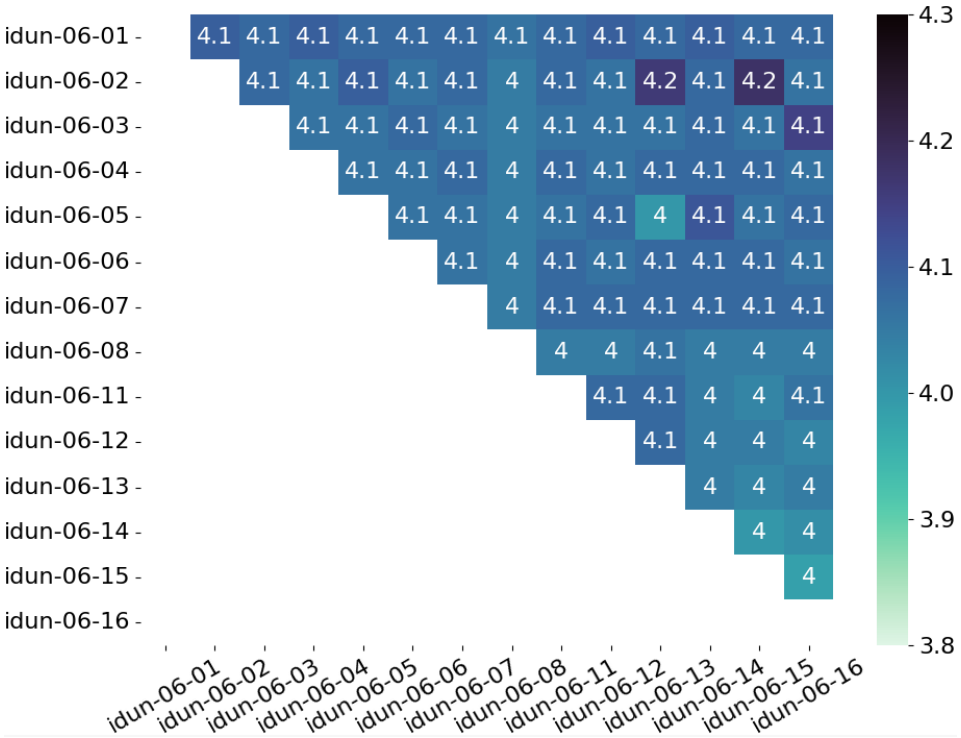


Figure 7.8: This matrix shows the bandwidth between pairs of Idun-06 nodes. The bandwidth is shown in GB/s.

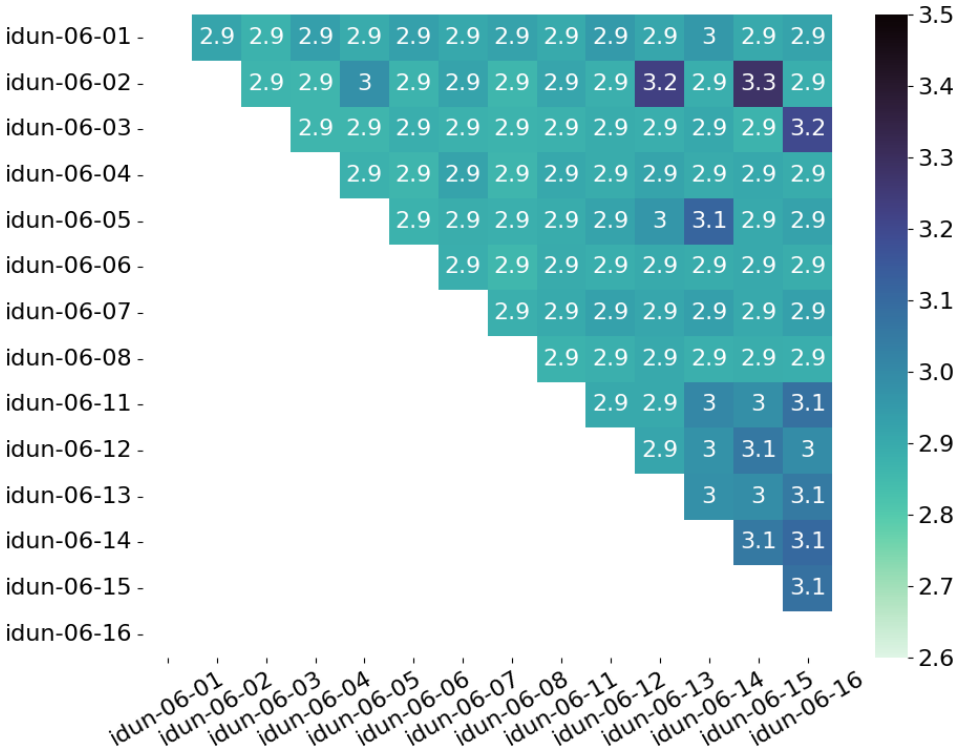


Figure 7.9: This matrix shows the latency between pairs of Idun-06-XX nodes. The latency is shown in microseconds.

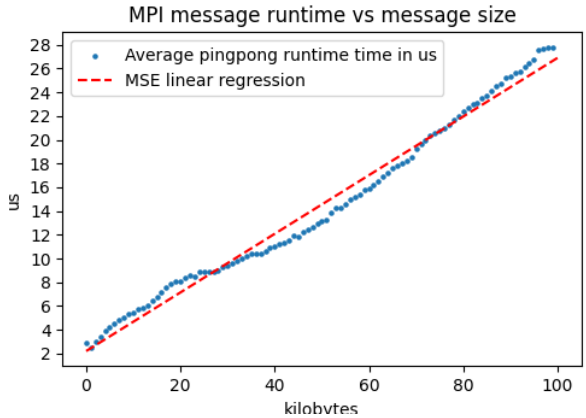


Figure 7.10: Linear regression of message passing benchmark results on two Idun-06-XX nodes.

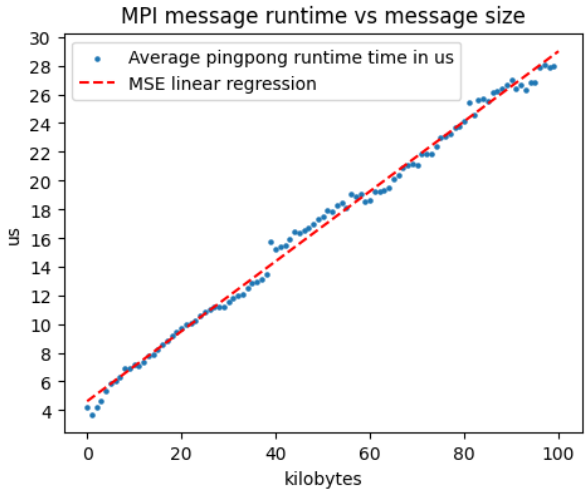


Figure 7.11: Linear regression of message passing benchmark results between Idun-06-14 and Idun-04-07

GPU	unpinned latency	unpinned bandwidth	pinned latency	pinned bandwidth
V100	15.74μs	1.86GB/s	14.90μs	7.52GB/s
P100	15.74μs	2.14GB/s	13.75μs	7.57GB/s
A100	25.50μs	1.95GB/s	21.31μs	10.53GB/s

Table 7.1: Table summarizing results from message passing benchmark for GPU memory transfers.

GPU

The message-passing benchmark was also run on GPUs. We measure both pinned and unpinned memory transfers to determine the Hockney parameters. We test the V100, P100, and A100 cards and summarize the results in Table 7.1. We find that the overhead of a GPU-CPU transfer is higher than the CPU sending an MPI message. We also see that the bandwidth is very low compared to what is theoretically possible. We attribute this to the small message sizes not fully saturating the communication pipelines. Another observation is that for large messages, we expect pinned transfers to be up to twice as fast as unpinned, but for the small messages we use, the speedup appears to be greater.

From Figure 7.12, 7.13, and 7.14, we see how the linear regression models the pinned memory transfers. It does overestimate the latency, but the main characteristics of the dataset are preserved. The approximation of the unpinned memory transfers appears to be less accurate. We see a similar discontinuity as when MPI changes protocols on the measurements of the pinned memory transfers. Interestingly, this gap exists on at least the three different Nvidia architectures we test. We attribute this to a change in protocol or buffers used during the transfer.

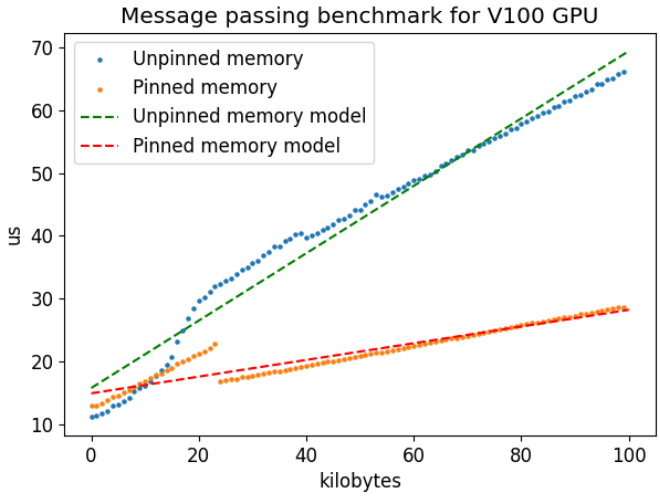


Figure 7.12: We plot the runtime of unpinned memory transfers in blue and approximate the performance with linear regression. The Orange points are the measurements from pinned memory transfers that are approximated by the dashed red line.

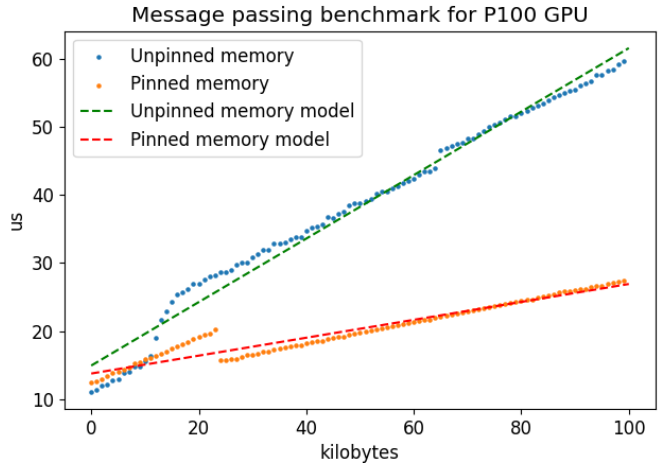


Figure 7.13: We plot the runtime of unpinned memory transfers in blue and approximate the performance with linear regression. The Orange points are the measurements from pinned memory transfers that are approximated by the dashed red line.

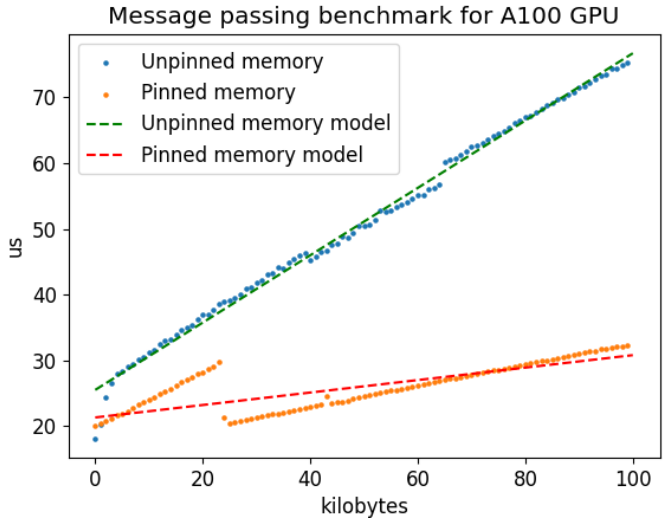


Figure 7.14: We plot the runtime of unpinned memory transfers in blue and approximate the performance with linear regression. The Orange points are the measurements from pinned memory transfers that are approximated by the dashed red line.

7.3.2 STREAM

GPU STREAM

The results from the STREAM benchmarks on our GPUs are shown as a bar chart in Figure 7.15. The leftmost bar is the theoretical max obtained by inspecting the vendor's specifications. This benchmark tries to measure the maximum achievable memory bandwidth. We show that we can expect to see a bandwidth of up to 1584GB/s on the A100 80GB card, 801GB/s on V100, 544 GB/s on P100, and 570 on Titan RTX. The results also show that the P100 was only able to achieve 75% of its theoretical peak bandwidth, whereas the other GPUs managed from 82% to 89%. The results from the Titan RTX are included as we use it to verify the roofline chart provided by Nvidia Nsight Compute.

CPU STREAM

The results for the STREAM benchmark on GPU are visualized as a bar chart in Figure 7.16. The Intel Xeon E5-2640 V4 produced a 49GB/s memory bandwidth, which is 64% of the theoretical maximum. The Intel Xeon Gold 6248R, limited to only using 24 of its cores, had a maximal bandwidth of 89 GB/s, which is 68% of the theoretical maximum. The benchmark measured the memory bandwidth of the Intel Xeon Gold 6148 to be 154 GB/s. The latter is 20% above the theoretical maximum, suggesting some part of the configuration were invalid. Specifying how threads should be bound to sockets was omitted in the slurm script. We, therefore, attribute this result to the threads being split among the two sockets on the system. This would mean that the proper stream measurement is half

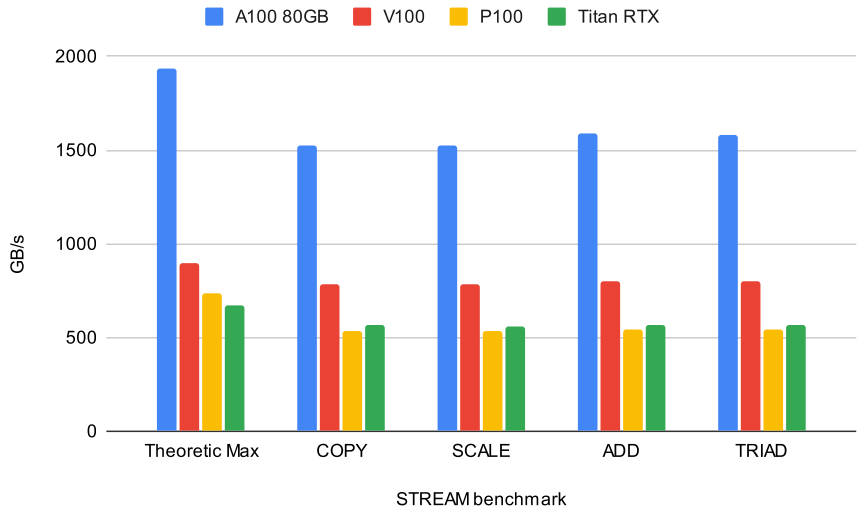


Figure 7.15: STREAM results for different GPUs

of that, which would give a 60% of theoretical max, which is in line with the other results. For this reason, we plot for the Xeon 6148 the theoretical speedup of two CPUs.

The results from the stream benchmark can be used as an upper limit to how much bandwidth we can expect to achieve. We do this in the next section, where we compare the bandwidth the programs obtain during the kernel execution to the upper limit of the STREAM results.

7.3.3 Stencil Kernels

In this section, we cover the performance of the CPU and GPU kernels on grids of different sizes. We also see how well the model fits the computation.

GPU

The computations on GPUs are approximated with the function shown in Equation (7.2), where we adjust a and b to fit the observations reducing the mean squared error.

$$g(x) = a(1 - e^{-x \cdot b}) \tag{7.2}$$

The resulting parameters for the models in Figures 7.17, 7.18, and 7.19 are summarized in Table 7.2.

The model of the V100 card manages to capture the most important characteristics of the card's performance. The asymptote at 60 billion stencil operations per second is accurate, and the curve up to that point is also close to the real values. The largest relative error occurs for the smallest 25 by 25 grids, but such small grids are of little concern

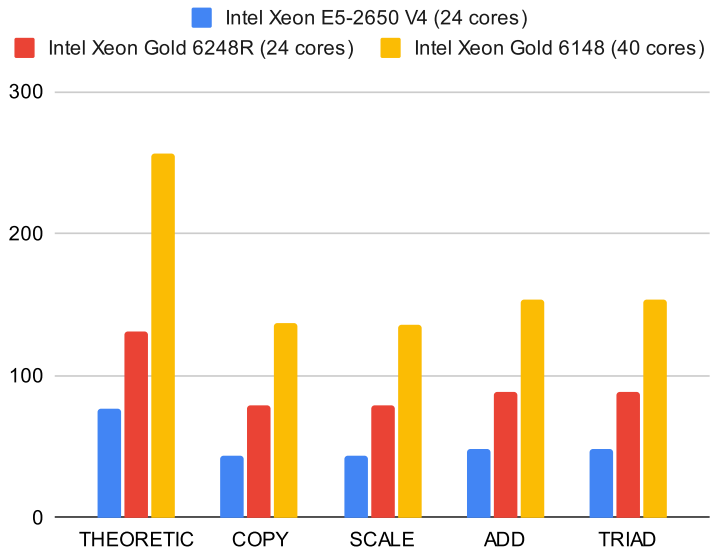


Figure 7.16: STREAM results for different CPUs, using twice the theoretical bandwidth of the Xeon 6148.

GPU	a	b
V100	60.90	0.00101
P100	32.29	0.00123
A100	113.4	0.00079

Table 7.2: Table of parameters for approximating stencils computer per second on different GPUs

within the field of HPC, to begin with. There are small deviations from the smooth curve for grids that have a sidelength of around 1000, but this has a minor impact on the model. The model also slightly overestimates the performance on grids of size 4000 by 4000, just as the model flattens out. We also notice the sawtooth pattern that repeats throughout the entire graph. According to Zhang *et al.* [38], this depends on how the number of thread blocks is divided by the number of memory pipelines in the clusters of SMs (GPCs). The peaks are present when we perfectly utilize all the pipelines, and the small dips occur when we do so imperfectly.

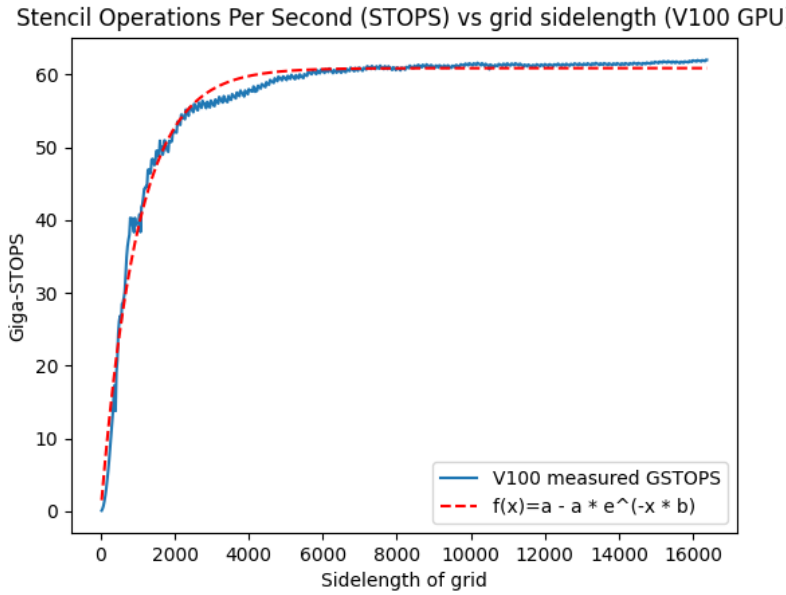


Figure 7.17: GSTOPS vs. grid size for the V100 GPU

The model for the P100 card is compared to the measurements in Figure 7.18. The asymptote is found to be at 32.3 GSTOPS and matches the measurements very well. During the rise of the performance, as the scheduling on the GPU utilizes more and more memory parallelism, the model follows the measurements closely. In part, because the derivative of the modeling function is the largest in the beginning, the performance for small grids is overestimated. Because of this, the largest relative errors occur for very small girds.

The approximation of the A100 card performance on stencil computations is shown in Figure 7.19. The model captures the asymptote well, predicting the performance to level off at 113 billion stencil operations per second. There is a small peak in performance for grids with sidelength 1000 to 1500. We explore explanations by inspecting warp scheduling statistics, memory utilization, and base clock frequency in *Nvidia Nsight Compute* without finding any deviation from the statistics of the other grid sizes. We also investigate if its computational speed is related to the fact that the grids in question fit in the L2

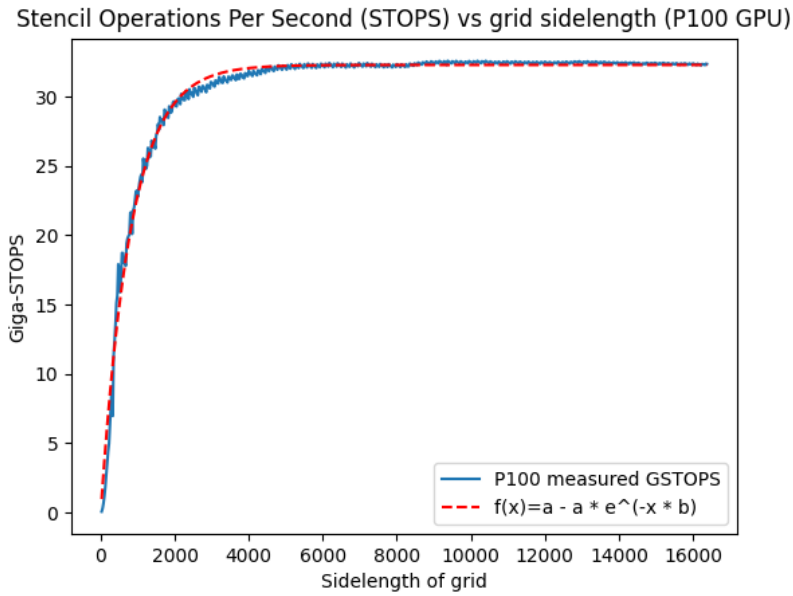


Figure 7.18: GSTOPS vs. grid size for the P100 GPU

cache. The results do not suggest that these grid sizes use the L2 cache any better. Despite this region of anomalous behavior, the model broadly captures the main characteristics of the measured behavior.

We discuss how the achieved GSTOPS induces memory bandwidth usage on each card. Due to caching on the SMs between values fetched by different threads or warps, each stencil computation essentially only forces three reads or writes from global memory. One way to see this is that if local caching is efficient, we still need to read or write from all values in the three timestep grids. If we multiply these three reads of four bytes with the model's max number of GSTOPS, we get the amount of bandwidth needed to support that number of stencils computations per second. To see how close this is to what we can expect from a card, we can see how large of a percentage that bandwidth is from the maximal STREAM measurement of the card. The results are summarized in Table 7.3. The results show that on 2 of the three cards, we can effectively use almost all the bandwidth experimentally shown to be available via STREAM. The P100 does seem to underperform using this metric. The kernel performs extra computations compared to STREAM and relies on highly efficient caching to assume that we only perform three memory reads or writes per stencil. These assumptions and complicating factors limit the performance on the P100 to only 71.29% of the memory bandwidth achieved in the STREAM benchmark.

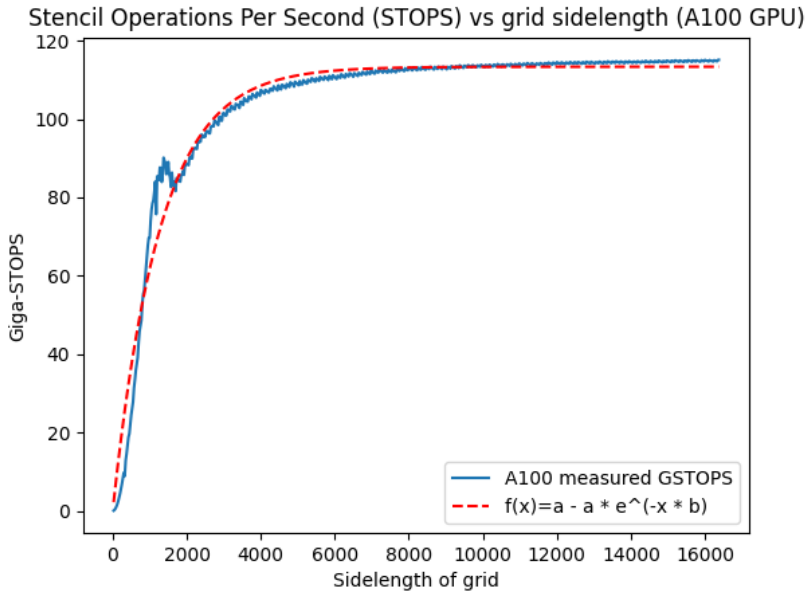


Figure 7.19: GSTOPS vs. grid size for the A100 GPU

GPU	% of STREAM
V100	91.29%
P100	71.29%
A100	99.51%

Table 7.3: Caption

CPU	a	b	c	d
Intel Xeon E5-2650 V4	20.21	1026.86	2400.1	5.13
Intel Xeon Gold 6148	32.03	1169.96	2991.19	8.33
Intel Xeon Gold 6248R	43.31	793.95	2272.06	7.75

Table 7.4: Table of parameters for approximating stencils computer per second on different CPUs

CPU

We model the performance on CPUs according to Equation (7.3).

$$g(x) = \begin{cases} a \cdot sin(\frac{x}{b}) & \text{if } x \leq c \\ d & \text{otherwise} \end{cases} \tag{7.3}$$

Table 7.4 summarizes the resulting parameters for each CPU. Note that given what we will use our performance model for, we only use 24 cores on the Intel Xeon Gold 6248R. We now briefly comment on the results for each CPU's computational performance.

We see that for all tested CPUs in Figures 7.20, 7.21, 7.22, the performance of the CPUs rises quickly with the sidelength of the grid until a peak at around a 2000x2000 point grid. The performance then falls again as the grid becomes too large to contain in the L3 cache. We verify this by inspecting the miss-rates in Valgrind [39]. When the grid is too large for the cache, the performance stabilizes. For all three CPUs, we see that the sine wave in the model manages to model the rise and fall of the GSTOPS when the grid resides in the CPU cache quite well. There is, however, a clear tendency that the model does not manage to model the highest peak of the performance accurately. Since the performance is stable after the sine wave, the constant in the model works well. Note that there is a significant drop in performance for each CPU when the grid has a sidelength of 1025. We now discuss this drop in performance.

Even though the performance dip is significant, these exceptions still achieve higher GSTOPS than on large grids where the grids are much too large to fit in the cache. The performance of the 1025x1025 grid is significantly lower than for grids of sizes 1000x1000 or 1050x1050. This is caused by the grid size being slightly above a power of 2. Upon closer inspection, we find that the performance is more in line with the expectations for a grid that is 1024 wide and 1025 gird points high and worse when the width is 1025 instead. This is because we use row-major indexing in the matrix. We draw the link to, for instance, cache lines, which have a power of two sizes on the tested CPUs and memory pages. When fetching a piece of memory that is just larger than a cache line or a memory page, we can, in practice, be required to fetch double the amount of data, despite only needing the first byte of the next chunk. This is because we must also fetch the next cache line, or load in the next memory page, despite only using a tiny portion of it. This also suggests that there actually should be more dips in our plots. Since we only tested square grids with sidelengths that are multiples of 25, we do not always land on the values just above a power of two. Another reason we do not see more significant dips is because the 1024 by 1024 grid is the power of two grids with the best performance. This means that the other power of twos is either too small, to the point where the expected dips can't go too low, or that the grids are so small that the grid is not stored in the cache anyways.

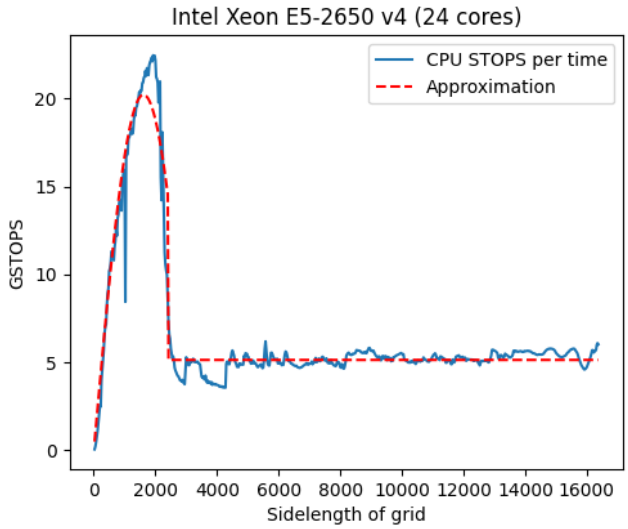


Figure 7.20: Stencil operations per unit of time on varying grid sizes using Intel Xeon E5-2650 V4

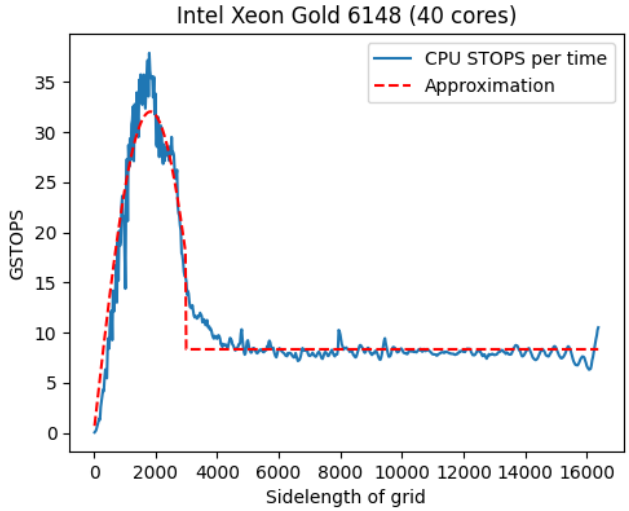


Figure 7.21: Stencil operations per unit of time on varying grid sizes using Intel Xeon Gold 6148

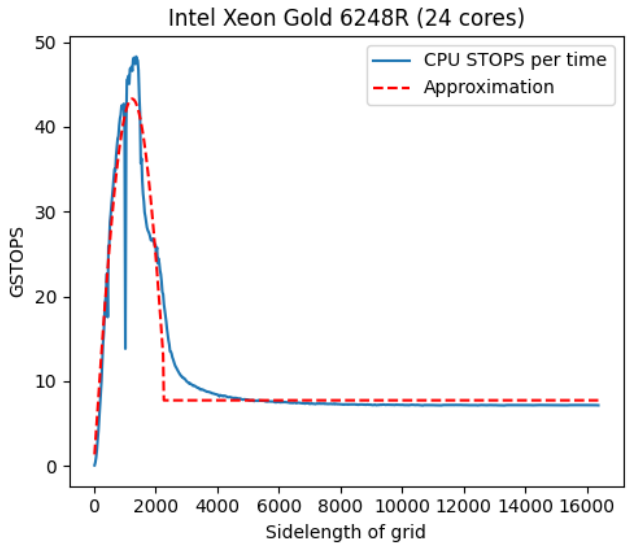


Figure 7.22: Stencil operations per unit of time on varying grid sizes using Intel Xeon Gold 6248R

CPU	percentage of theoretical max
Intel Xeon E5-2650 V4 (24 cores)	79.95%
Intel Xeon Gold 6248R (24 cores)	70.99%
Intel Xeon Gold 6148 (40 cores)	78.09%

Table 7.5: The bandwidth induced by the computational kernel as a percentage of the theoretical max.

The results of the stable CPU performance on large grids are summarized in Table 7.5. The results indicate that the implementation uses most of the bandwidth available. We compare the induced memory bandwidth usage from stencil computations with the theoretical max of the CPU.

7.3.4 Serialized FDM

We run the serialized FDM benchmark to study the runtimes of the individual parts of the FDM algorithms. We create Figure 7.23 by measuring the behavior of the FDM algorithm with two processes on Idun-04-02. We can see that all parts of the program take longer times when the dimensions of the grid increase. More importantly, we can clearly see the importance of the fact that inner computation scales with the area of the grid, while the other runtime components scale with the circumference.

The second stacked bar chart is Figure 7.24, and illustrates the identical case of the previous figure, except that we now measure the time spent for the process in the middle of the linear topology when using three processes. The main difference is that the largest

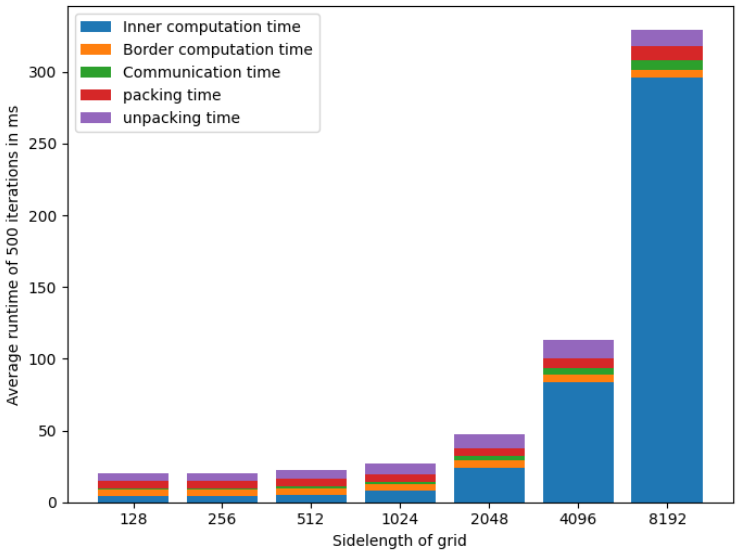


Figure 7.23: Stacked bar chart showing runtime of different runtime stages of the algorithm on a single node with 2 processes, using only GPU.

runtimes are lower than when using two processes. Additionally, communication takes up a significantly more significant portion of program execution as it needs to communicate twice as much.

The last bar chart considers a different algorithm. Figure 7.25 shows the result of serializing the algorithm using both CPU and GPU on a single node. The GPU is set to compute precisely 90% of the rows. As we see on the other bar charts, the inner computation constitutes a growing portion of the runtime when the grid size increases. This plot also reveals that the CPU outer compute takes longer than the GPUs border kernel. This is because the border kernel is the same size always for the GPU and the CPU, and the GPU is faster than the CPU at this task. Note that the CPU takes longer to compute the inner grid than the GPU on the small to medium-sized problems, mainly the 512 by 512 grid and 1024 by 1024 grid. Given how the 512 by 512 grid's inner kernel on the CPU takes longer than on the subsequent problem instance, where the number of stencils to calculate by the CPU is quadrupled, we reason that the CPU measurements are prone to significant noise levels. We deem the larger grid sizes more reliable for this reason and see there that the runtime of the CPUs points consistently takes a shorter time to compute than the GPUs when the workload share between the devices is 90%.

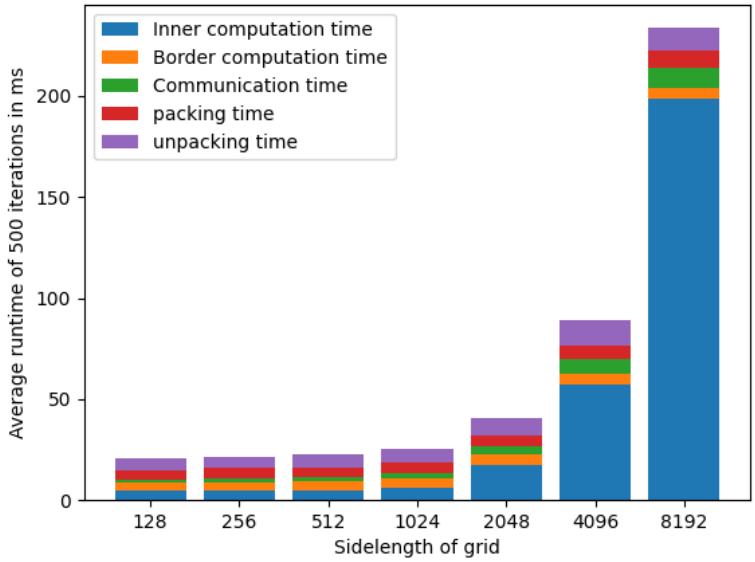


Figure 7.24: Stacked bar chart showing runtime of different runtime stages of the algorithm on a single node with three processes, using only GPU.

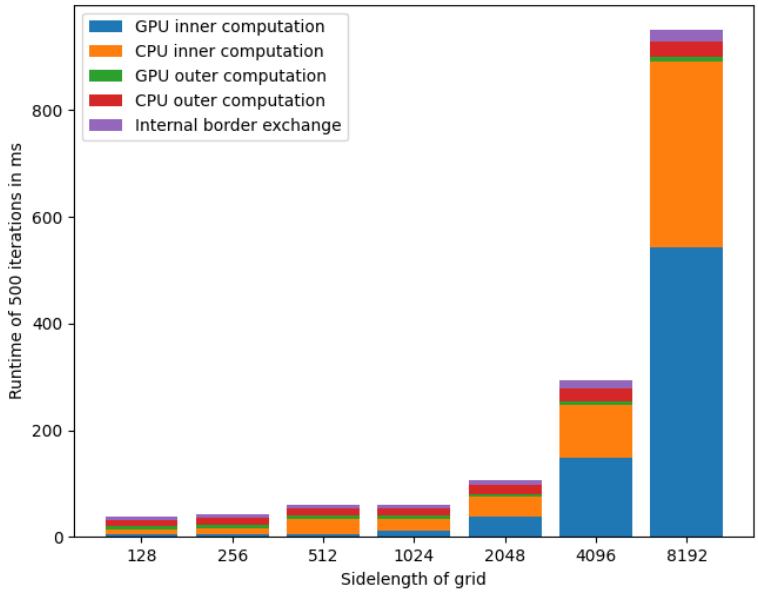


Figure 7.25: Stacked bar chart showing the results from using both the CPU and the GPU for the FDM computations.

7.3.5 Strong Scaling

We now cover the experiments on strong scaling, where the problem size is held constant, and the number of processes increases. The chapter first covers scaling on a single node and then scaling across nodes.

Single Node

Figure 7.26 shows on the left the speedup achieved on combinations of processes and grid sizes on a single node. It was run on Idun-04-02 with up to all 10 V100 GPUs present. When using ten processes to compute the largest grid, we achieve a 9.68 speedup, which is very close to 10, which is the theoretical max. As expected, small instances do not allow for speedup as the communication and border kernels become expensive relative to computation and the total runtime. The runtimes measured in this benchmark are very stable. When using all ten GPUs, which maximizes noise and variation, the coefficient of variation comes out to 0.2% for the largest problem instances and 2.8% for the smallest one.

A corresponding speedup landscape is shown to the right in 7.26. The difference is that the speedup is normalized with respect to the number of processes, which gives us the efficiency. In these plots, good strong scaling entails values very close to one. The efficiency of small grids is low because the overhead of communication and packing is significant compared to the computation it could overlap with. For large problem sizes, the efficiency stays high. It is 0.97 on the largest grid when using all 10 GPUs. From this, we can tell that for large problem instances, the program will effectively use the computational power available.

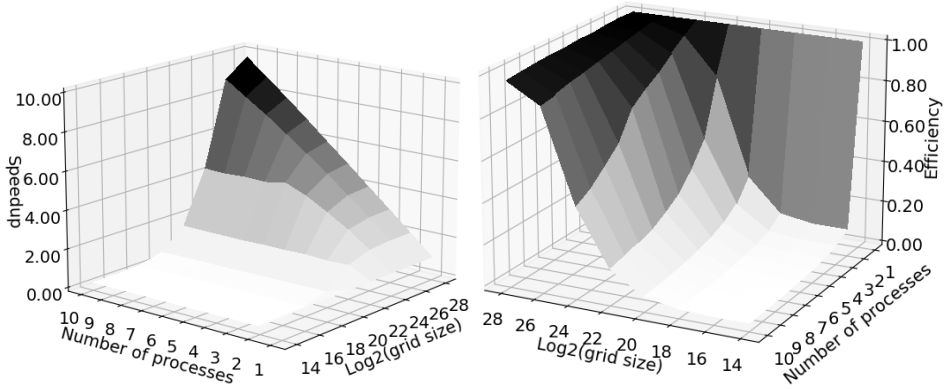


Figure 7.26: Speedup and efficiency landscape measured when using up to 10 GPUs on Idun-04-02 with varying grid sizes

Multiple Nodes

In Figure 7.27, the speedup and the efficiency are shown when using 1 GPU per node, with up to ten Idun-06-XX nodes. The resulting speedup landscape contains deep valleys

contrasting the clear linear pattern for large problem instances on a single node. The values in the plots are arithmetic means of runtimes, which means that, typically, the implemented algorithm does not yield the expected steady speed up when adding more hardware. The coefficient of variation on the largest problem instance with 10 GPUs is 0.33, and some individual runs are twice as fast as others. This suggests that the arithmetic means that are plotted in the landscape are greatly affected by noise. On the smallest problem instance with 10 GPUs, the coefficient of variation is 1.1. The runtime is very unreliable at this scale, with some runs being 30 times faster than others. Interestingly, although the runtimes have a very high variation for all problem instances, the arithmetic mean seems to suffer, particularly when using 7 and 10 processes and on some problem sizes when using 4 or 6 processes. Giving the processes exclusive access to each node does not affect the result. Implementing a benchmark implementing more timers could reveal what part of the iteration is causing the noise. We do not seek to model noise, and thus we explore other ways to present the data.

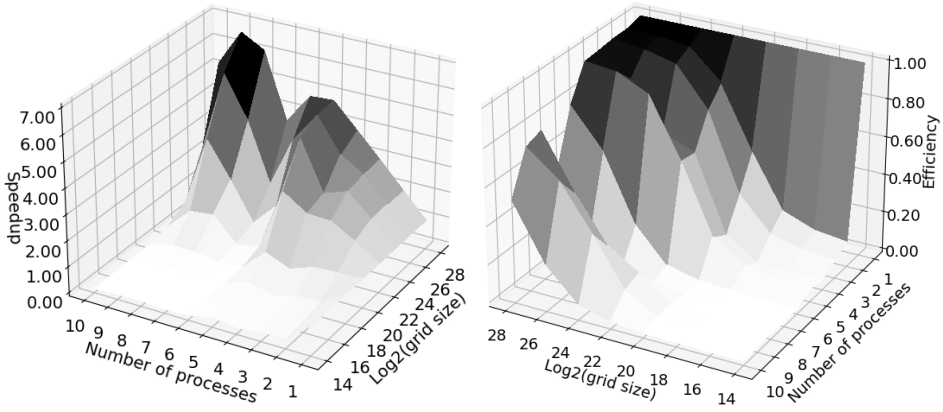


Figure 7.27: Speedup and efficiency landscape measured when using up to 10 P100 GPUs on Idun-06-XX nodes with varying grid sizes

Having unstable results, we also create Figure 7.28, which contains the speedup and efficiency landscapes using only the best runtime measured on each configuration. Here we see that the results are very similar to what we see on a single node and what we would expect. The speedup of using 10 GPUs is up to 9.57, which is very close to a perfect linear speedup. This reveals an important point, the poor results from the strong scaling are caused by noise, and the runs that are not affected perform just as well as expected with a near-perfect scaling. Achieving such efficient scaling is anticipated as the price of computation at some point must outweigh the overhead of communication. By that point, all the computational resources should be productive almost always.

Only our algorithm uses the memory bandwidth when working on a single node, making communication reliable. When using multiple nodes in a cluster shared with many others, our runtime is slowed if one process waits for the communication lines that another uses. Given that the algorithm on a single node performs as expected with the same source code, we assume that a bug is not causing the slowdown. Additionally, we run the same

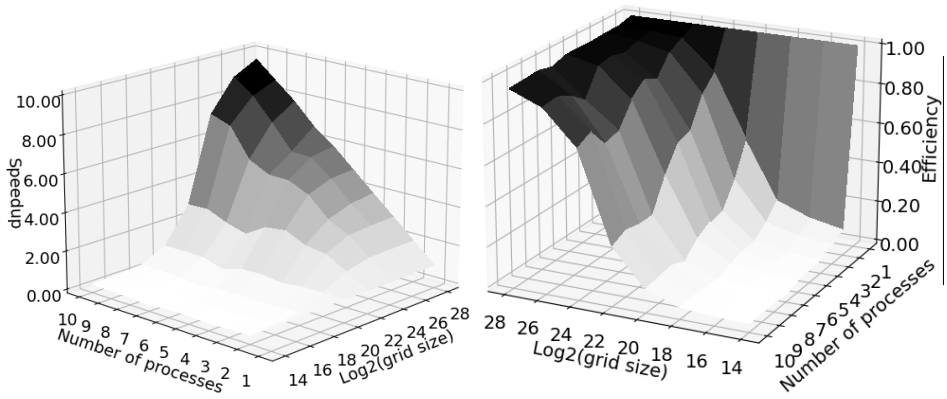


Figure 7.28: Speedup and efficiency landscape measured when using up to 10 P100 GPUs on Idun-06-XX nodes with varying grid sizes, plotting only the best run on each configuration

program with all parameters decided at compile time, so there is little room in the software to explain why the runtime should vary this much from run to run, only when using resources on the cluster shared with other users. The results are also stable across different subsets of the available nodes.

7.3.6 Weak Scaling

The weak scaling experiment tests the algorithm's efficiency when ensuring that the size of the local problem instance to a node is constant.

Single Node

Figure 7.29 shows the weak scaling results. The landscape to the left shows the results on a single node, namely Idun-04-02. The results show an overall trend that the efficiency is relatively high when the problem size is large enough, but otherwise, the efficiency drops quickly. On the largest problem size using ten GPUs, the efficiency is 0.93. This means that the runtime is 8% slower than the runtime of a single GPU, computing a grid the tenth of its size.

There are some peculiarities in the landscape. The single node weak scaling experiment shows reliably that the efficiency is above 1 when the local problem size consists of 2^{24} stencil computations. The coefficient variation is always below 0.003, suggesting the measurements are so stable that no noise is causing this effect. Additionally, this only occurs on one particular problem size. What might cause it is that the shape of the local problem size changes. Since each process computes a slice of a square grid that is increasing in size, the rectangle of a process will become wider and shorter the more processes are involved. This does mean more communication, but a large amount of computation will hide it. It means that we have more effective memory usage on a process since we have larger regions of contiguous memory that may allow more coalesced memory accesses. It is a weakness of the implementation of the weak-scaling test that the local subgrids change

shapes.

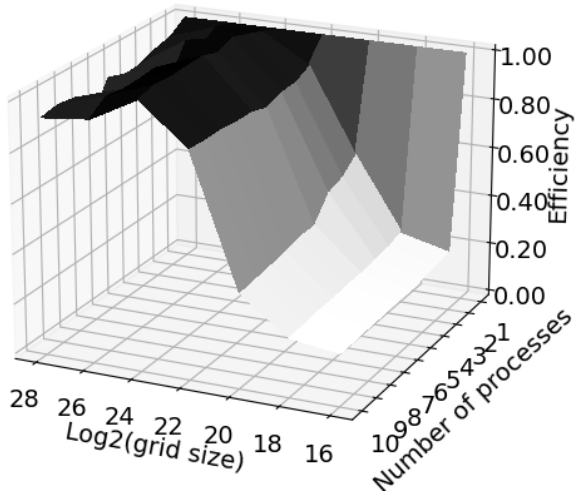


Figure 7.29: Weak scaling results on a single node

Multiple Nodes

The results from the weak scaling experiment on multiple nodes are shown in Figure 7.30. The figure on the left, which shows the mean performance, also captures that larger local problem instances use the local resources much more efficiently. The noise from using multiple nodes for strong scaling is replicated here. Interestingly, the same number of nodes seems to suffer, namely 4, 6, 7, and 10. This happens even though consecutive runs do not run on the same subset of nodes and even with exclusive access to the nodes it runs on. The plot on the right of Figure 7.30 shows the same plot but only using the best individual run on each configuration. This plot, although noisier, resembles the results from a single node much more and aligns more with what we would expect.

7.3.7 Load balancing

In this section, we cover different measured properties of the load balancing that happens on a node, that is, how we distribute the rows of the local grid among the CPU and GPU available.

Internal Load balance

We first demonstrate that finding the right internal load balance between the GPU and the CPU can give speedup. Figure 7.31 is a boxplot showing the runtime per iteration of the algorithm using a single node when varying the share of work that should be done on the GPU. The boxplot displays a horizontal line on the median value and boxes containing one

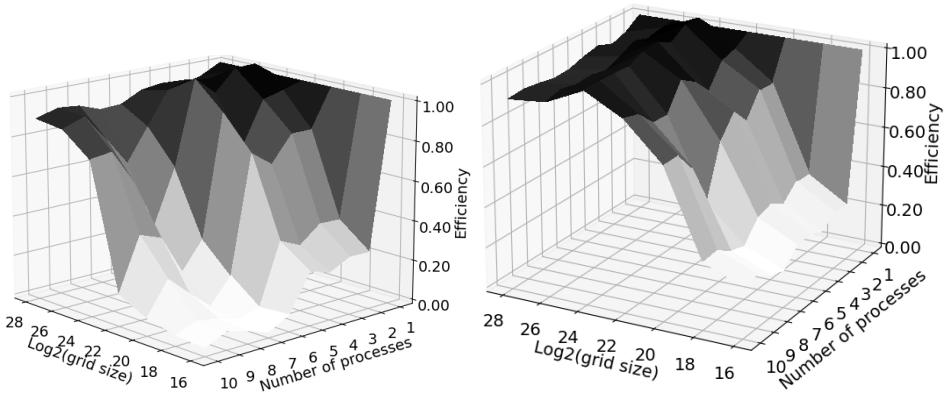


Figure 7.30: Weak scaling results on multiple nodes. The arithmetic mean of runtime on a configuration used on the left, single best run used on the right.

quartile above and below the median. The whiskers extend to the minimal and maximal extrema.

The measurements show that when using a V100 GPU with the Intel Xeon Gold 6148, we can get speedup by having the CPU compute a small share of the grid. More precisely, we expect a speedup of 8.7% when the CPU calculates 9% of the GPU. This demonstrates that there is speedup to be gained by introducing CPUs in implementing the FDM.

This also proves to be a visualization of how the runtime of a kernel on a GPU is more reliable than on a CPU. This is shown through the fact that we see a significant variance increase when the CPU takes longer than the GPU to compute its share. The variance is that of the CPU because when the threshold is passed, the runtime of the CPU determines the program's final runtime, which is more subject to noise.

We now show how well the binary search heuristic works to find the optimal workload balance, which we measure to be 9% of computations on the CPU. Figure 7.32 shows a histogram with the converged workload balance of 500 calibration phases on Idun-04-02, which has a V100 GPU and an Intel Xeon Gold 6148 CPU. We see that the binary search achieves the ideal 91% quite often, and if it misses, it tends to overshoot the work that should be done on the GPU. We see two peaks because if the binary search ever takes a wrong step due to noise in the measurements, it will be impossible to get to the correct value. These run from the peak at 93.5%.

Note that the binary search is just a heuristic that minimizes idle time by making the most considerable portion of the program take the same amount of time on both devices. This heuristic could be improved by instead ternary searching over the lowest possible time for an entire iteration, which would be able to adjust for the fact that the inner kernel is not the only element of consideration. Increasing the number of iterations measured during the search to more than one will also improve the reliability and accuracy of the search.

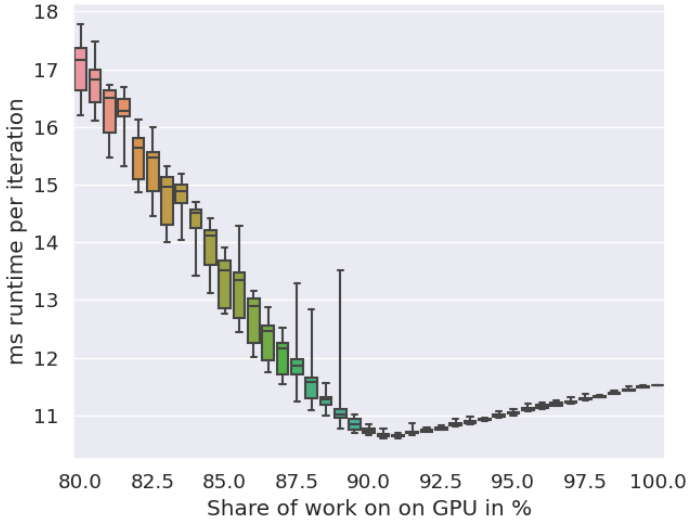


Figure 7.31: Boxplot showing the runtime when varying the share of rows of the grid that should be computed on the GPU

Inter Node Balance

We first demonstrate that the node balance between nodes works reliably when they run the same hardware. One advantage of studying this scenario is that we know to expect that if the load balancing works perfectly, we should see that both processes end up with the same amount of rows to compute. The nodes we use are P100 nodes with an Intel Xeon 6132 CPU. We study a square grid with side lengths of 16384.

The results are shown in a histogram in Figure 7.33. The left histogram shows all the balance points calibrated and how often the value was reached when only giving the algorithm access to the GPUs. It shows that the calibration phase for GPUs is very reliable as the results are mostly less than one percent. Interestingly, we see a bimodal distribution. Most of the time, the calibration is less than 0.2% off the expected value. Quite often still, one of the ranks will run its calibration phase a bit slower than expected and cause a skewed distribution of rows between the ranks. Note that when doing GPU-only inter-node load balance, the calibration only takes a few milliseconds, and running those extra ten iterations takes negligible time when running an FDM physical simulation for real applications.

The right part of the plot shows that calibration of a node's computational capacity with a CPU and a GPU is also very reliable. The results are more unimodal, as opposed to the results of calibrating using only GPUs. The CPU has a slightly less stable runtime. Hence the distributions are more spread out than when only using a GPU. We do not see the instability of the GPUs that caused the bimodal distribution in this benchmark.

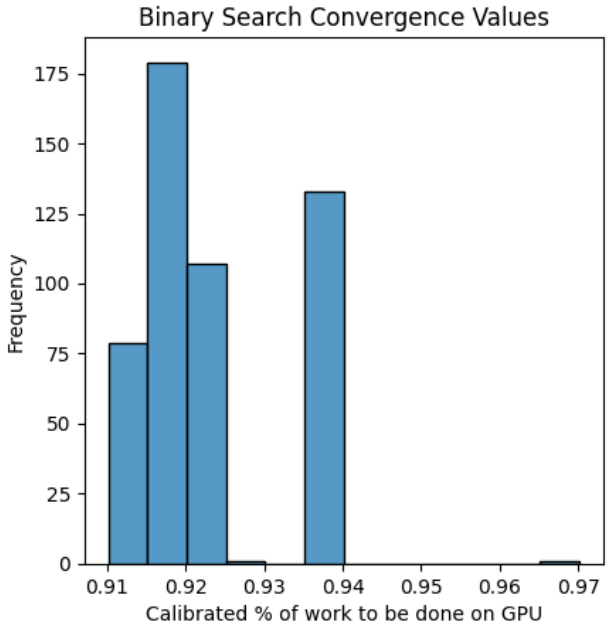


Figure 7.32: Histogram showing where the binary searches for the percentage of work that should be done on the GPU converge.

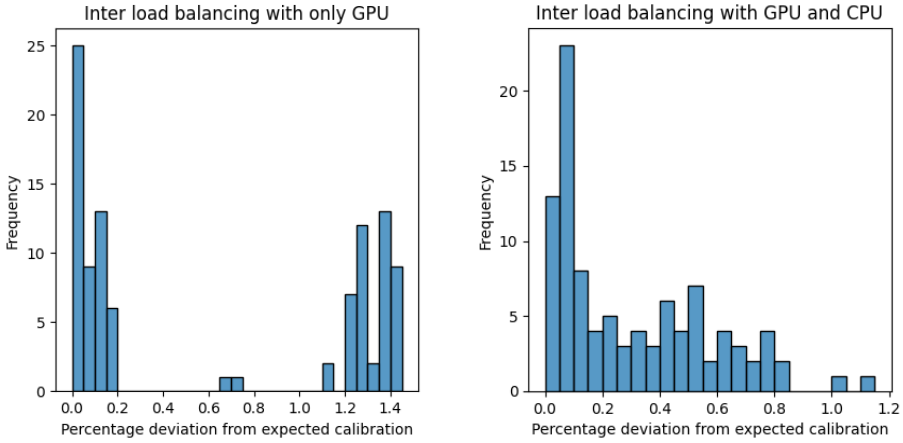


Figure 7.33: Histogram of the accuracy of calibration phase with both CPU and GPU. The accuracy is shown as a percentage deviation away from the expected number of rows

PA_E runtimes
0.941
1.245
0.941
1.245
0.936

Table 7.6: Runtime of the 5 AP_E runs.

7.3.8 Heterogeneous Environment

Our last benchmark covers how the developed algorithms perform when running n a highly heterogeneous environment. We generate the results by running 500 iterations on a 16384 by 16384 grid. The main result is shown in Figure 7.34. Recall that the naming convention we use for these tests is described in Section 6.3.8.

This bar chart shows the runtime for several different hardware configurations. The first configuration only runs on a P100 GPU and takes 4.1 seconds. By adding a second P100 GPU, we get a 1.96 speedup. A speedup close to 2 indicates effective strong scaling on large grids. The third bar uses one P100 GPU and an A100 80GB GPU. We still use the static and completely even workload balance, which is why we do not see any speedup, even when more computational resources are available. We include this to demonstrate that omitting load balancing is bound to be wasteful on heterogeneous clusters. The fourth bar uses the same hardware configuration but uses the inter-node load balancing. We now observe a 3.9 speedup from using just a single P100. According to our computational model, the A100 card can achieve 3.5 times more stencil operations per second than the P100. This gives us a maximal speedup of 4.5 when using both these cards, compared to a single P100. Achieving a speedup of 3.9 instead of 2 shows that the implementation scales with the sum of its parts instead of the weakest link when enabling inter-node load balancing. The measured time of the individual runs is provided in Table 7.6. We here see that there is some instability in the results, with the runtime either being 0.94 or 1.25 seconds. It is most likely the calibration phase that causes this. What is also interesting is that if we only look at the best run, we actually have a speedup of 4.4 over the single P100 card, which is very close to the theoretical optimum of 4.5. Once again, it seems that noise significantly affects the mean runtimes.

The last bar represents the same two nodes in use but with their CPUs available for computation. We then observe a 17% speedup compared to not using processors. The extra speedup from utilizing CPUs demonstrates the potential of using all the computational resources available. When using the two different GPUs, employing both levels of load balancing, we get a speedup of 2.33, compared to no load balance. It demonstrates that the algorithm can use heterogeneity in the runtime environment to gain significant performance over implementations using even load distributions. A couple of details are notable when studying the runtimes. The CPUs compute 5% of the grid on Idun-04-08 with an A100 GPU and 7% on Idun-06-15 with a P100 GPU. This can not yield the 17% we observe. The majority of the speedup actually comes from consistent results. The coefficient of variation is 0.004, which means that the benchmark with heterogeneous GPUs

and CPUs does not have noise dragging the mean value down.

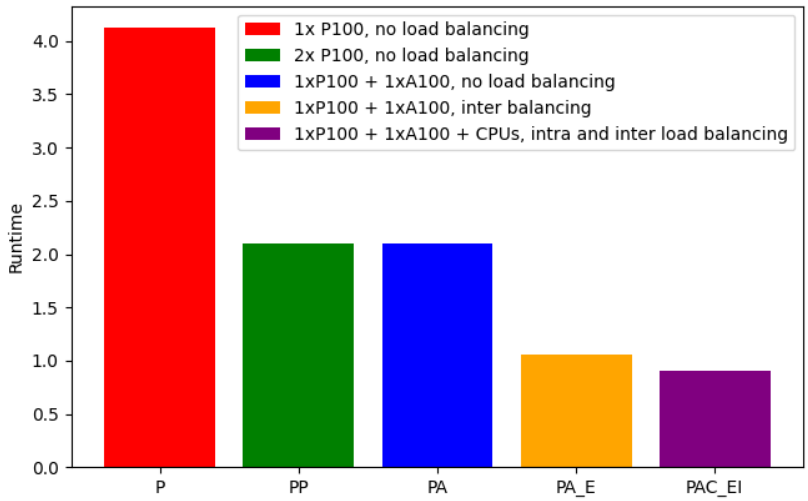


Figure 7.34: The runtime of the FDM algorithm in various heterogeneous environments.

7.4 Performance Model Evaluation

In this section, we compare the theoretical results of the performance model with the experimental results of some of the benchmarks. The section explores how well the model performs when modeling both weak and strong scaling on single and multiple nodes. We use the percentage difference metric introduced in the background to measure its accuracy. All the plots have the same maximum height to make comparison easier by visual inspection. The raw data that produces all of the plots is provided in Appendix A.

7.4.1 Modeling Strong scaling

Figure 7.35 shows the result of modeling strong scaling. Both landscapes show the percent difference between the modeled and the experimental results.

The subfigure on the left shows strong scaling on Idun-04-02, which uses up to ten V100 GPUs. The landscape's most notable feature is the spike when using a single node to compute the 128 by 128 grid. This spike occurs because the model of GPU GSTOPS exaggerates its computational prowess for grids with a sidelength of less than 200. Its relative error is what causes such a spike. The absolute prediction is about 1 GSTOP off the measured mean. The other small grids are modeled better because the communication will bottleneck the runtime. Furthermore, we see that other small grids are modeled with a percentage difference from 9.9% to 16.3%, even when many GPUs are working in tandem.

For the smallest grid size, the most common percent difference is 14%. Large grids are also modeled accurately. The largest grid size has a mean percent difference of 5.2%. Square grids with sidelengths from 1024 up to 4096 form the ridge of inaccuracy that is visible in the plot. The model predicts up to double the performance that we observe experimentally. This is partially caused by the same reason as the aforementioned spike. The ridge represents quite small grids when divided among multiple processes, and so the performance is overestimated. However, the small local grids are just large enough that the communication bottleneck is not severe enough to limit the performance as much as the experimental data suggests. This does not happen on the largest grids because, by then, the model of stencil operations per second on the GPU is very accurate.

The subfigure on the right shows the percent difference between the model and the experimental results when strong scaling on Idun-06-XX nodes with P100 GPUs. Since the strong scaling experiment across nodes had a level of noise that gave inferior results for a select few grid sizes, we attempt to model the best-performing run for each combination of grid size and the number of processes. We, therefore, look away from modeling the large noise of unknown origin on select combinations of grid size and process count. Even though we try to mitigate the noise by modeling the best runs, the landscape is still clearly affected by the noise. We overpredict the runtime on this benchmark, which we attribute to the Hockney model overestimating the latency during the packing and unpacking stage, as the latency for the MPI communication across nodes seems to be a good fit. Since we have greater communication costs, the ridge is flattened out, and we see a clear trend that results are more accurate the larger the grids are. Since these grids are the least affected by communication, it suggests that the performance model for the P100 card has captured its performance asymptote concerning grid size accurately. For the largest grid size, 16384 by 16384, the percent difference varies from 0.1% to 1.2%. We mainly attribute the fact that smaller grids are modeled worse to our communication models, overestimating the latency due to protocol switches that make the runtime non-linear. Still, we also have a percentage difference of up to 54% on small grids with many processes.

7.4.2 Modeling Weak Scaling

The model's accuracy for the weak scaling results is visualized with a landscape of percentage differences in Figure 7.36. The leftmost subfigure shows the model's accuracy when predicting the number of stencils operations per second on Idun-04-02 and uses up to all ten V100 GPUs. We still see a spike on the smallest grid when using a single GPU, as our model overestimates the performance. Note that the larger problem sizes are generally modeled more accurately than the smaller ones. In the weak scaling experiment, the communication actually grows with the number of processes because we are working with a square grid, and hence the dimensions of the sidelengths grow, even when the amount of computational work per processor remains the same. The small sizes have a percentage difference of 33.2% to 45.6%. We also see from the raw data that our model underestimates the performance. This is related to our Hockney model for MPI communication, which on a single node overestimated the overhead of sending a message due to the protocol switch not being properly modeled with a linear function.

The subfigure on the right shows the model's accuracy in percent difference using the weak scaling experiment on the Idun-06-XX nodes. We see the same spike caused by

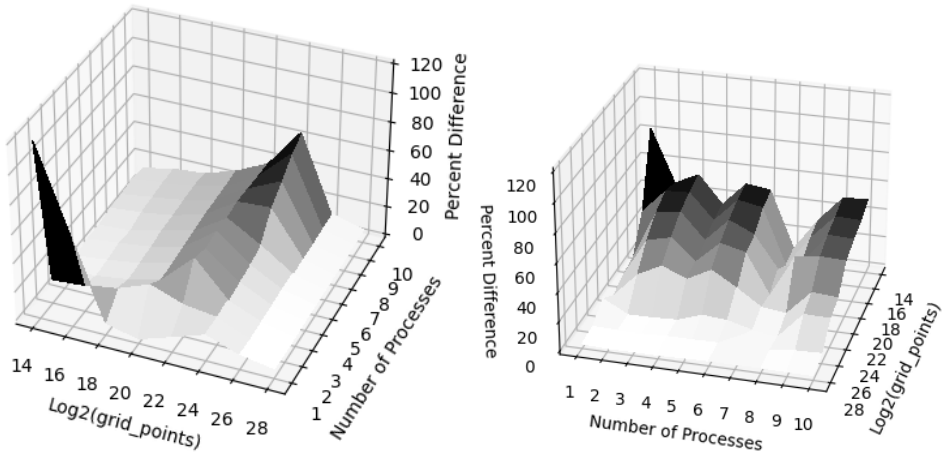


Figure 7.35: The landscape to the left shows the percent difference between the predicted stencils operations per second by the model and the experimental results from the strong scaling experiment on a single node. The landscape on the right does the same, but for the case when the processes are located on different nodes.

overestimating the performance of GPU computation on small grids here. We see a clear tendency that larger grids are modeled better, which again suggests that the performance of the GPU on large grids is modeled very accurately. On large grids, the percent difference ranges between 0.9% and 8.9%. We see a hilly terrain since the weak scaling results across nodes are based on noisy measurements.

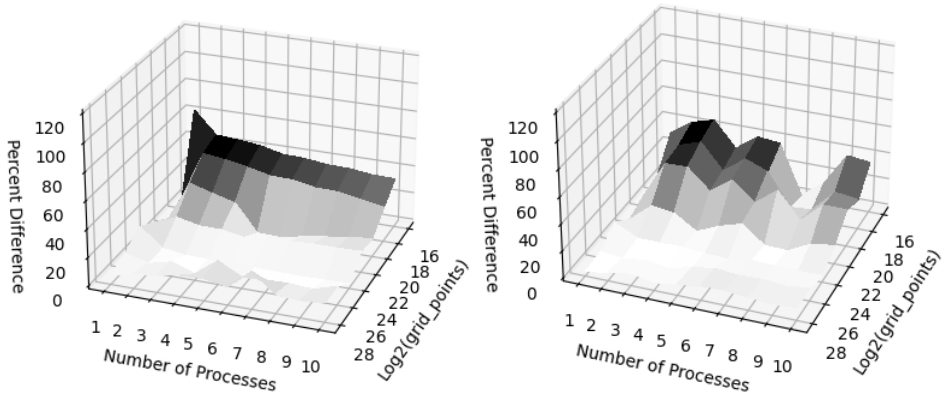


Figure 7.36: The landscape to the left shows the percent difference between the predicted stencils operations per second by the model and the experimental results from the weak scaling experiment on a single node. The landscape on the right does the same, but for the case when the processes are located on different nodes.

7.4.3 Modeling Heterogeneous Performance

Figure 7.37 shows the results of the heterogeneous benchmark compared to the expected runtime the performance model provides. In addition to displaying the mean measured and the predicted runtime as a bar, we offer the single lowest runtime measured on a given benchmark. The high-level results show that for a large grid, the model manages to capture the scaling of adding more hardware, the scaling of introducing load balancing between nodes with different GPUs, and also the runtime when having both internal and external load balancing with two nodes containing both different GPUs and CPUs. The best predictions are within a single percent using the percent difference metric. The worst prediction is PA_E, where we predict a runtime of 0.9216 seconds, but the mean runtime of the program is 1.0616. Note that the model is accurate for the ideal case, as the best runtime measured was 0.9362, which is very close to the predicted value. The most complicated scenario was modeled with a percent difference of 3.6% compared to the mean runtime and 3.0% when compared to the single best run.

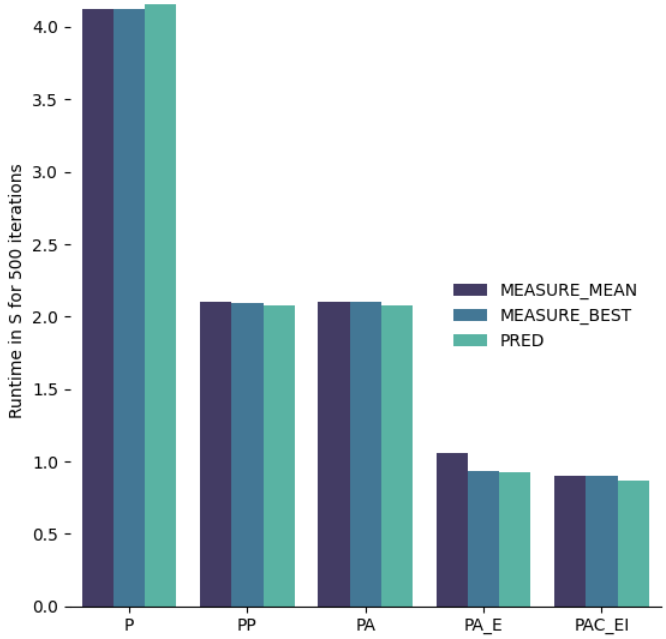


Figure 7.37: Bar chart showing the mean, max, and predicted runtime of the heterogeneity benchmarks.

These results from this benchmark demonstrate the accuracy of the model on large grids. The inaccuracy is more significant on the smaller grids, not only because the relative importance of noise increases but also because the relative error of the model increases.

Limitations of the Model Verification

These tests do not clearly expose what part of the performance model contributes the most to the error. Running a slightly different benchmark that estimates and measures the time spent on each part of the code would target the problematic elements of the model more efficiently than deriving them from runtimes of complicated algorithms on heterogeneous hardware.

The size of the grids tested in the final benchmark are instances where all devices are running fully saturated with work. This leaves the sine-wave modeling of the CPU computational speed for grids that fit in the cache untested.

Chapter 8

Conclusion and Further Work

This chapter summarizes the key observations from the results and highlights potential future spots for improvement.

8.1 Conclusion

In this thesis, we develop an implementation of the finite difference method that is made to run on a cluster of computers, utilizing both the CPU and the GPU. We show that the computation on a single node with multiple processes scales very well, maintaining an efficiency of 0.97 when running on 10 GPUs on a large grid. The results indicate that the large instances completely hide the communication costs, as the performance scales very well when adding more hardware instead of being bottlenecked by increased communication. The results have a significant variance on multiple nodes. Still, when looking at the best runtimes to mitigate noisy measurements, the efficiency was 0.96 when using ten nodes, showing that the implementation can scale just as well across nodes in a cluster. Similar levels of noise and efficiency were demonstrated in the weak scaling benchmark.

The different stages of load-balancing are tested independently and shown to be very reliable. This indicates that when trying to divide the grid between a multitude of nodes or between computational units on a node, the resources can be divided to utilize all computational units available effectively. The load balancing is tested in its totality in the heterogeneity benchmark. That benchmark showed speedup being proportional to the sum of its parts instead of its weakest link. It demonstrated the effectiveness of FDM algorithms in highly heterogeneous environments.

We also create a performance model relying on simple analytic functions to predict stencil operations per second and the Hockney model of inter-process communication and memory transfers. The performance model is validated by comparing it to the strong and weak scaling. We see that in very small instances, the performance is exaggerated due to the relative error of the simple analytic performance models. On the largest problem instances from the weak and strong scaling benchmarks, the mean percentage difference between the model and the measurements is 5.9%. On a large-scale grid with two different

GPUs and CPUs applying both levels of load balancing, the model error in percentage difference is 3.6%.

8.2 Further Work

This section covers three clear improvements to the thesis: implementing binary search and iterative calibration and highlighting a limitation of communication cost models.

Ternary Search

We implement a binary search to find the point where the GPU and the CPU spend the same amount of time on the inner kernel as a heuristic for internal node load balance. This could be improved to be implemented with a ternary search encompassing all execution in a single iteration. This would converge to the point that an iteration spends the least time by definition, avoiding the heuristic entirely.

Iterative Calibration

The current inter-node load balance assumes that the final computational throughput is the sum of CPU and GPU on a grid the size of its even share without load balancing. This is not necessarily a good heuristic since some nodes can be faster than others. Additionally, it might be that when the internal load-balancing is achieved, a different computational throughput is achieved than what was estimated. For this reason, it can be wise to run multiple iterations of the calibration, where we adjust the grids' sizes and rerun the calibration to see if the results are better. This will probably yield more accurate inter-node calibration on highly heterogeneous clusters. This need not be a dynamic load balancer that acts through the entire runtime but can be an extended calibration phase during the first few iterations.

Protocol Aware Hockney Model

As we see multiple examples when measuring the runtime of sending messages, MPI runtimes can jump abruptly when switching protocols. The Hockney model can account for this by extending it to be a piece-wise linear function. The model would be more accurate for both short and long messages by splitting the linear regression into two parts, one for each protocol.

Acknowledgements
This thesis relied on access to the Idun cluster[6] provided by NTNU.

Bibliography

[1] K. Li and W. Liao, "An efficient and high accuracy finite-difference scheme for the acoustic wave equation in 3D heterogeneous media," *Journal of Computational Science*, vol. 40, p. 101063, Feb. 2020.

[2] M. Mineter and N. Hulton, "Parallel processing for finite-difference modelling of ice sheets," *Computers & Geosciences*, vol. 27, pp. 829–838, Aug. 2001.

[3] G.-F. Liu, X.-H. Meng, and H. Liu, "Accelerating finite difference wavefield-continuation depth migration by GPU," *Applied Geophysics*, vol. 9, pp. 41–48, Mar. 2012.

[4] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[5] X. Li, G. Chen, and W. Wen, "Energy-Efficient Execution for Repetitive App Usages on big.LITTLE Architectures," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 1–6, Association for Computing Machinery, June 2017.

[6] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure," Feb. 2022. arXiv:1912.05848 [cs].

[7] C. Märtin, "Post-Dennard Scaling and the final Years of Moore ' s Law Consequences for the Evolution of Multicore-Architectures," 2014.

[8] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, pp. 1901–1909, Dec. 1966. Conference Name: Proceedings of the IEEE.

[9] OpenMP Architecture Review Board, "Openmp standard," 2018. [Online; accessed April 24, 2023].

[10] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 4.0 ." https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Accessed: 2023-04-24.

[11] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, Association for Computing Machinery, Apr. 1967.

[12] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008. Conference Name: Proceedings of the IEEE.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam: Morgan Kaufmann, 5 ed., 2012.

[15] Nvidia, "CUDA Toolkit Documentation 12.1." https://docs.nvidia.com/cuda/. Accessed: 2023-06-02.

[16] W. Wang, W. Liu, and Z. Wang, "Survey of Load Balancing Strategies on Heterogeneous Parallel System," in *Proceedings of the International Conference on Human-centric Computing 2011 and Embedded and Multimedia Computing 2011* (J. J. Park, H. Jin, X. Liao, and R. Zheng, eds.), Lecture Notes in Electrical Engineering, (Dordrecht), pp. 583–589, Springer Netherlands, 2011.

[17] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing*, vol. 20, pp. 389–398, Mar. 1994.

[18] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Using Performance Modeling to Design Large-Scale Systems," *IEEE Computer*, vol. 42, pp. 42–49, Nov. 2009.

[19] "Roofline: an insightful visual performance model for multicore architectures: Communications of the ACM: Vol 52, No 4."

[20] F. Xudong, T. Yuhua, W. Guibin, T. Tao, and Z. Ying, "Optimizing Stencil Application on Multi-thread GPU Architecture Using Stream Programming Model," in *Architecture of Computing Systems - ARCS 2010* (C. Müller-Schloer, W. Karl, and S. Yehia, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 234–245, Springer, 2010.

[21] P. Eller, J.-R. C. Cheng, and D. Albert, "Acceleration of 2-D Finite Difference Time Domain Acoustic Wave Simulation Using GPUs," in *2010 DoD High Performance Computing Modernization Program Users Group Conference*, pp. 350–356, June 2010.

[22] S. Vialle, S. Contassot-Vivier, and P. Mercier, "Generic Algorithmic Scheme for 2D Stencil Applications on Hybrid Machines," in *Architecture of Computing Systems – ARCS 2016* (F. Hannig, J. M. P. Cardoso, T. Pionteck, D. Fey, W. Schröder-Preikschat, and J. Teich, eds.), Lecture Notes in Computer Science, (Cham), pp. 115–129, Springer International Publishing, 2016.

[23] B. Siklosi, I. Z. Reguly, and G. R. Mudalige, "Heterogeneous CPU-GPU Execution of Stencil Applications," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 71–80, Nov. 2018.

[24] M. Sourouri, S. B. Baden, and X. Cai, "Panda: A Compiler Framework for Concurrent CPU + GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers," *International Journal of Parallel Programming*, vol. 45, pp. 711–729, June 2017.

[25] A. Schäfer and D. Fey, "High Performance Stencil Code Algorithms for GPGPUs," *Procedia CS*, vol. 4, pp. 2027–2036, Dec. 2011.

[26] M. Cole, "Algorithmic Skeletons: Structured Management of Parallel Computation," 1989.

[27] N. Herrmann, B. A. de Melo Menezes, and H. Kuchen, "Stencil Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments," *International Journal of Parallel Programming*, vol. 50, pp. 433–453, Dec. 2022.

[28] D. P. Playne and K. A. Hawick, "Asynchronous Communication Schemes for Finite Difference Methods on Multiple GPUs," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, (Melbourne, Australia), pp. 763–768, IEEE, 2010.

[29] Nvidia, "NVIDIA V100 TENSOR CORE GPU." `https://www.nvidia.com/en-us/data-center/v100/`. Accessed: 2023-02-24.

[30] Nvidia, "NVIDIA Tesla P100." `https://www.nvidia.com/en-us/data-center/tesla-p100/`. Accessed: 2023-02-24.

[31] Nvidia, "NVIDIA A100 Tensor Core GPU." `https://www.nvidia.com/en-us/data-center/a100/`. Accessed: 2023-02-24.

[32] Intel, "Intel® Xeon® Processor E5-2650 v4 ." `https://ark.intel.com/content/www/us/en/ark/products/91767/intel-xeon-processor-e52650-v4-30m-cache-2-20-ghz.html`. Accessed: 2023-02-24.

[33] Intel, "Intel® Xeon® Gold 6148 Processor." `https://ark.intel.com/content/www/us/en/ark/products/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz.htmll`. Accessed: 2023-02-24.

[34] Intel, "Intel® Xeon® Gold 6248R Processor." https://ark.intel.com/content/www/us/en/ark/products/199351/intel-xeon-gold-6248r-processor-35-75m-cache-3-00-ghz.html. Accessed: 2023-02-24.

[35] SchedMD, "Slurm manual," 2021. [Online; accessed May 9, 2023].

[36] S. S. Dosanjh, R. F. Barrett, D. W. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. G. Trucano, and J. P. Luitjens, "Exascale design space exploration and co-design," *Future Generation Computer Systems*, vol. 30, pp. 46–58, Jan. 2014.

[37] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[38] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 382–393, Feb. 2011. ISSN: 2378-203X.

[39] Valgrind, "Valgrind user manual." https://valgrind.org/docs/manual/manual.html. Accessed: 2023-05-29.

Appendix A

Appendix A contains the results of the model prediction and measurements for strong and weak scaling in GSTOPS, as well as per cent difference between the two.

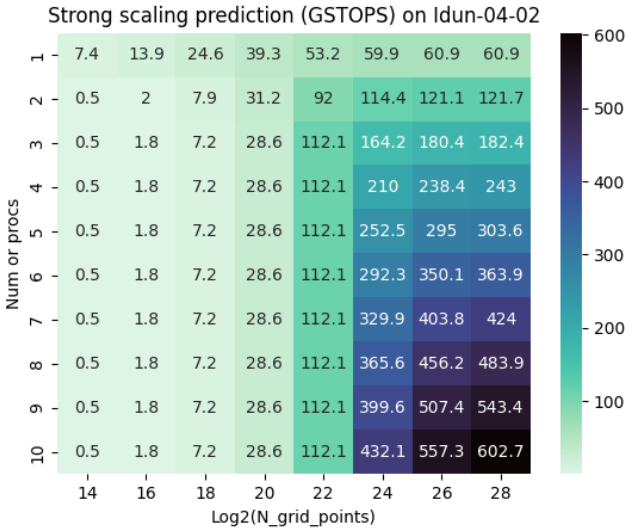


Figure 8.1

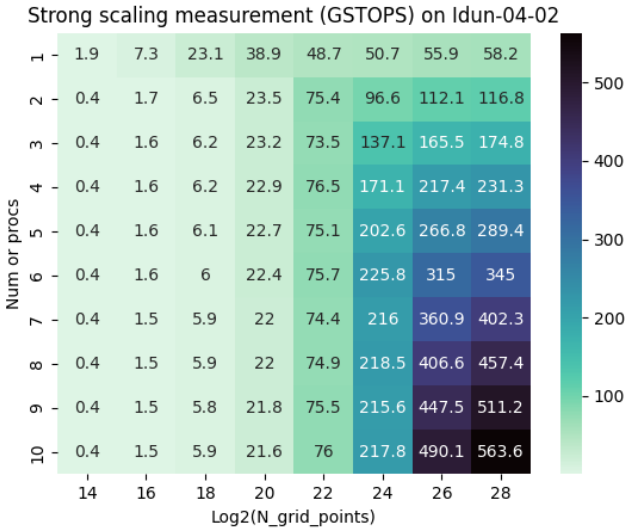


Figure 8.2

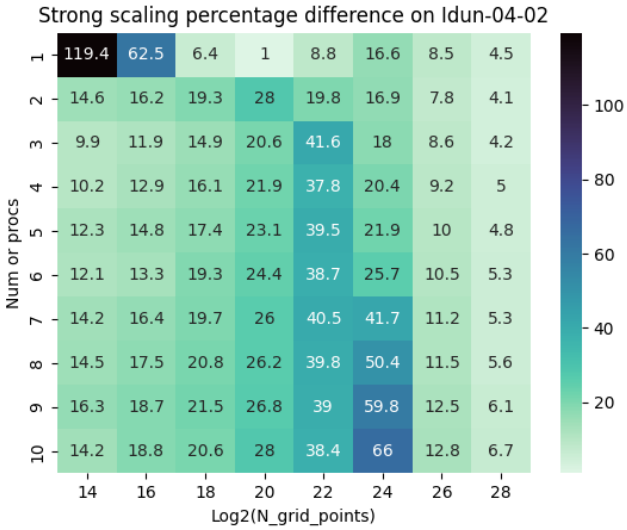


Figure 8.3

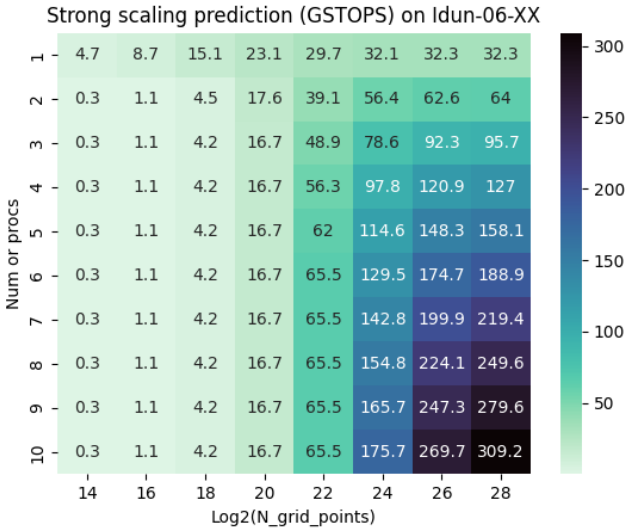


Figure 8.4

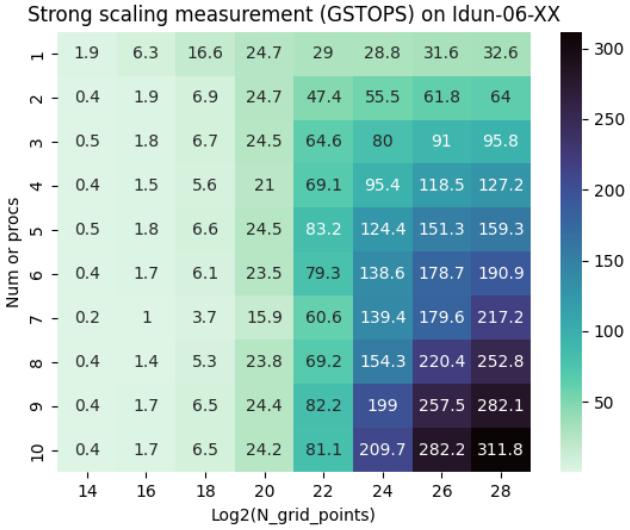


Figure 8.5

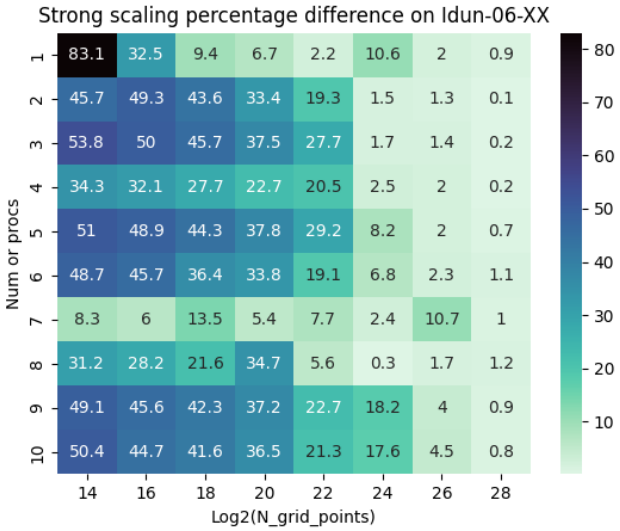


Figure 8.6

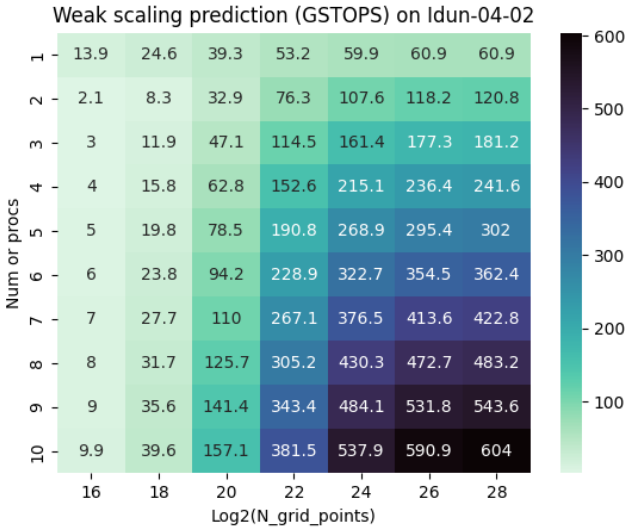


Figure 8.7

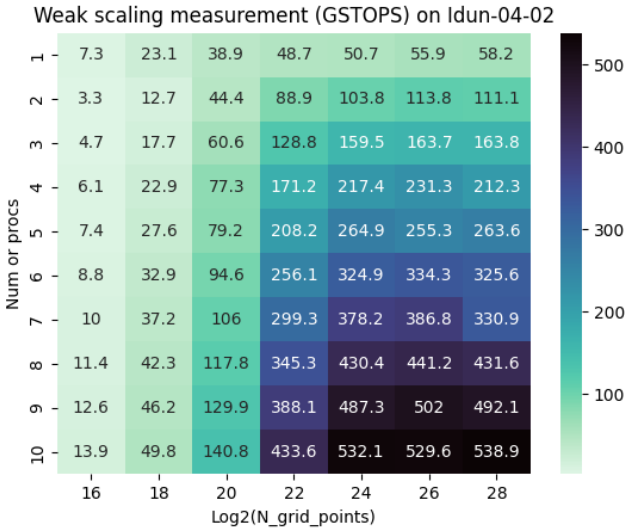


Figure 8.8

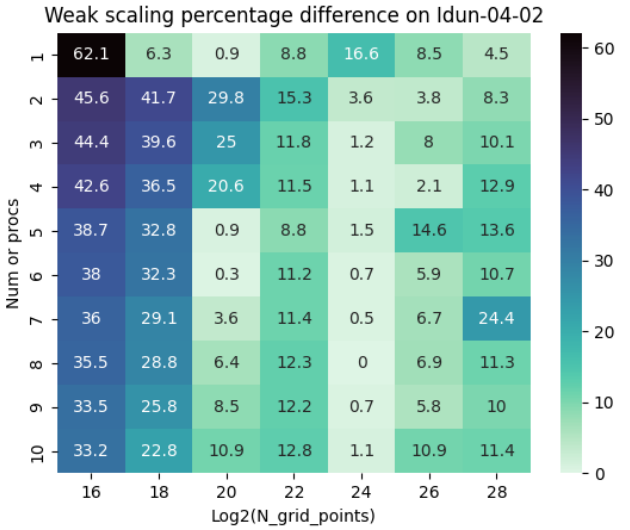


Figure 8.9

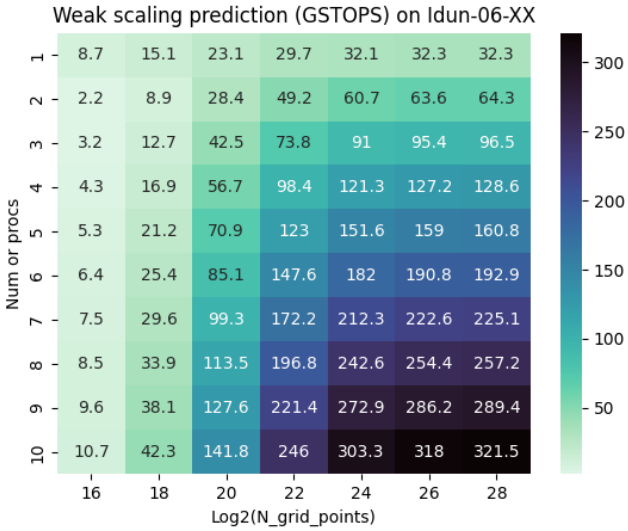


Figure 8.10

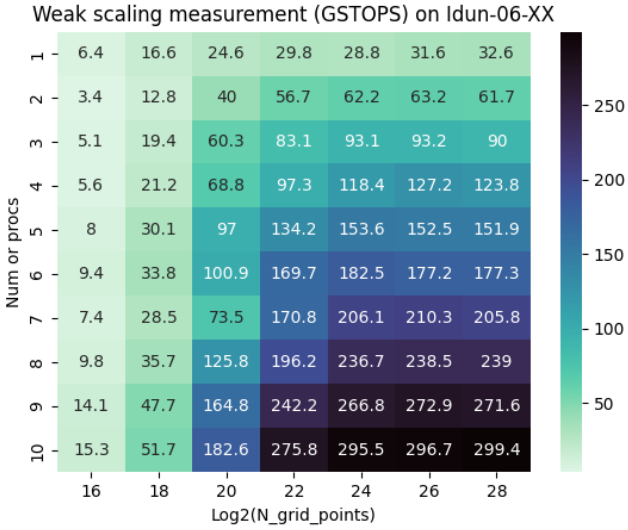


Figure 8.11

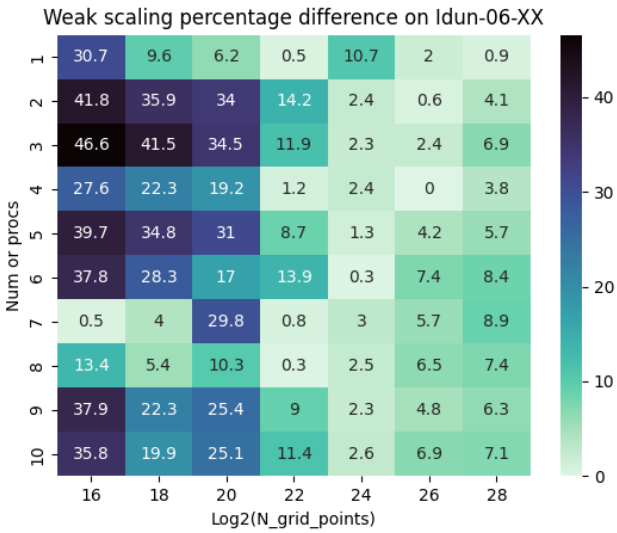
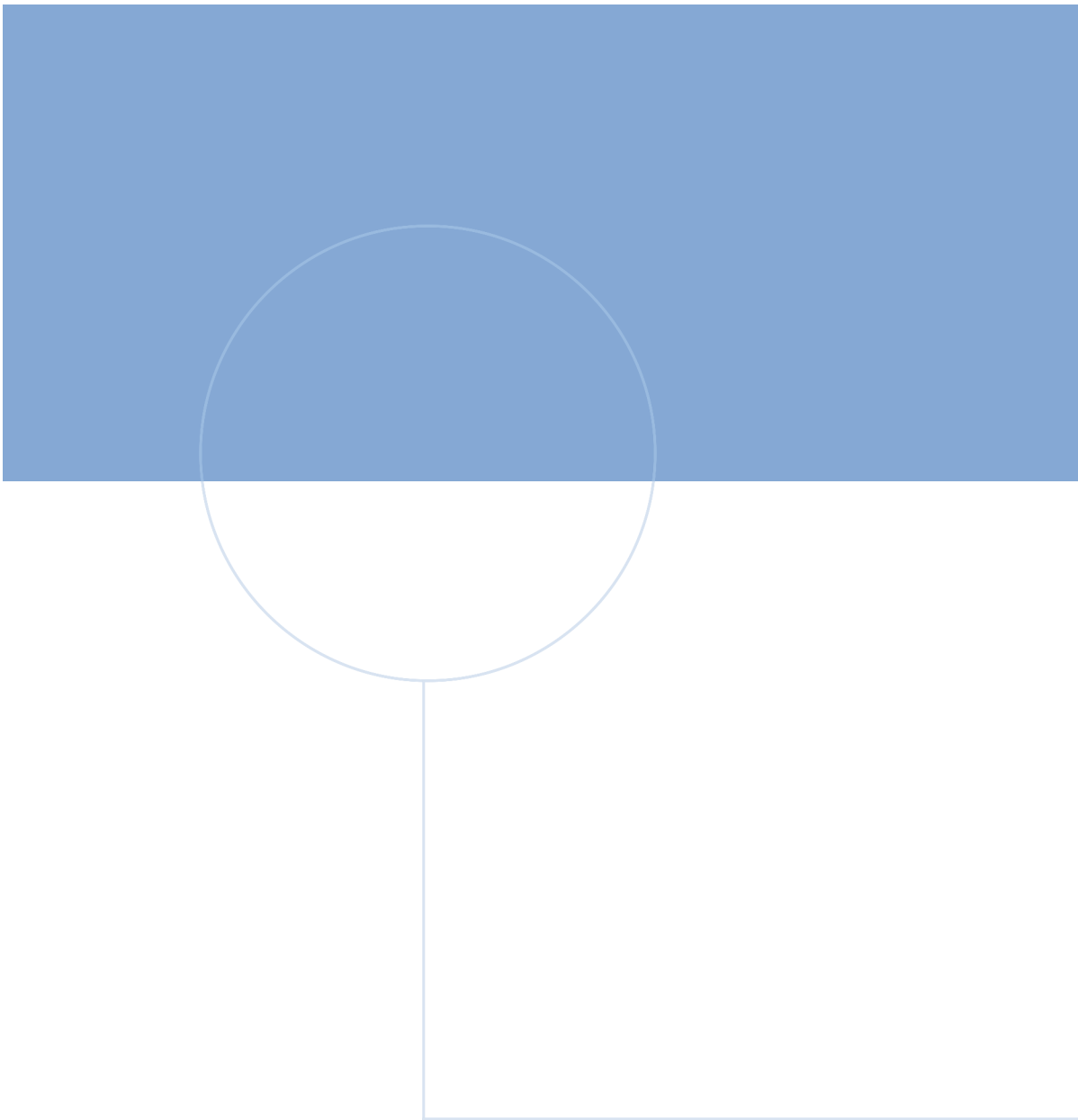

Weak scaling percentage difference on Idun-06-XX
Num or procs
1
2
3
4
5
6
7
8
9
10
30.7 9.6 6.2 0.5 10.7 2 0.9
41.8 35.9 34 14.2 2.4 0.6 4.1
46.6 41.5 34.5 11.9 2.3 2.4 6.9
27.6 22.3 19.2 1.2 2.4 0 3.8
39.7 34.8 31 8.7 1.3 4.2 5.7
37.8 28.3 17 13.9 0.3 7.4 8.4
0.5 4 29.8 0.8 3 5.7 8.9
13.4 5.4 10.3 0.3 2.5 6.5 7.4
37.9 22.3 25.4 9 2.3 4.8 6.3
35.8 19.9 25.1 11.4 2.6 6.9 7.1
16 18 20 22 24 26 28
Log2(N_grid_points)
40
30
20
10
0

Figure 8.12



NTNU

Norwegian University of
Science and Technology