Master's thesis

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

Ruben Solvang Valen

Analysing Deep Halos on Modern GPUs

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
June 2023

NTNU
Norwegian University of
Science and Technology

Ruben Solvang Valen

Analysing Deep Halos on Modern GPUs

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
June 2023

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

Original Problem Description

This master thesis project builds on the student pre-project (a.k.a. "fall" project) “Halo Exchange Performance Using CUDA MPI” which looked at the performance of halo exchanges in a stencil computation using MPI and CUDA to run on multiple GPUs. This thesis and the fall project builds on Andreas’ Hammer’s master’s thesis “Analyzing Halo Computations on Multicore CPUs” which in turn built on Robin Holtet’s 2003 master’s thesis "Communication-reducing Stencil-based Algorithm and Methods". In this thesis we will extend the author’s fall project to 2D variable-depth Halos computations on GPUs, and investigate the performance impact of different optimizations for halo exchanges alone and in combination on modern hardware. Optimizations will include multithreading and asynchronous communications and benchmarks from at least 2 GPU platforms. Other optimizations may also be included.

Abstract

Stencils are a family of algorithms that update points of a multi-dimensional data mesh with the neighbouring values as inputs. They are compute-intensive and used in many domains and thus account for a significant portion of High-Performance Computing workloads.

This thesis presents a benchmarking tool for 2D stencil computations on GPUs. Our work builds on A. Hammer´s work on Multicore 3D stencils. Our implementation uses MPI, and includes three different optimisation techniques: Deep halos, Synchronous halo exchanges, and overlapping communication and computation.

Deep halos group communication together to reduce overhead traded off with increased computation per iteration. This optimisation improves performance as the halo size increases until the decrease in communication time is eclipsed by the increase in computation time.

We show that overlapping communication and computation is a highly impactful optimisation that improves performance over our serial version in almost all cases, particularly for larger problem sizes. On the largest grid sizes tested, overlapping communication and computation hides 83% of the parallel communication.

The synchronous halo exchange optimisation works as expected with or without interacting with the other optimisations. Our asynchronous version highlights the ability to overlap communication and computation optimisation to mitigate the impact of unexpected inter-node communication slowdowns at low halo depths.

Overall, these three parallel optimisations offer significant performance improvement in stencil computations on GPUs with MPI communication.

Note, however, that the performance gains achieved with these optimisations depend highly on the problem size and hardware configuration. On the largest problem size tested, all three optimisations in conjunction gave a 1.09 speedup when using nodes with V100 GPUs and a 1.17 speedup when using nodes with A100 GPUs.

Our results provide insight for decision-makers when considering the implementation of these optimisations given different problem domains. Suggestions for future work on this benchmarking tool is also included.

Sammendrag

Stensilalgoritmer er en familie av algoritmer som oppdaterer punkter i et flerdimensjonalt datanett med naboverdiene som vekter. De er beregningsintensive og brukes i mange domener, og utgjør dermed en betydelig del av arbeidsbelastningen innen høy-ytelses databehandling.

Denne avhandlingen presenterer et verktøy for benchmarking av 2D stensilkalkulasjoner på GPUer. Arbeidet bygger på A. Hammer's arbeid med flerkjernede 3D-stensiler. Vår implementasjon bruker MPI og inkluderer tre forskjellige optimaliseringsteknikker: dype haloer, synkroniserte kantutvekslinger og overlappende kommunikasjon og beregning.

Dype haloer grupperer kommunikasjon sammen for å redusere overhead, men med økt beregning per iterasjon som en avveining. Denne optimaliseringen forbedrer ytelsen når halo-størrelsen øker, inntil reduksjonen i kommunikasjonstid blir overskygget av økningen i beregningstid.

Vi viser at overlappende kommunikasjon og beregning er en svært effektiv optimalisering som forbedrer ytelsen i nesten alle tilfeller, spesielt for større problemstørrelser. På de største rutenettene som ble testet, skjulte overlappende kommunikasjon og beregning 83% av den parallelle kommunikasjonen.

Synkronisert kantutvekslingsoptimaliseringen fungerer som forventet, enten med eller uten samspill med de andre optimaliseringene. Den asynkrone versjonen vår viser evnen til overlapp av kommunikasjon og beregning optimaliseringen til å redusere effekten av uventede forsinkelser i kommunikasjon mellom noder ved lave halo-dybder.

Totalt sett gir disse tre parallelle optimaliseringene betydelig ytelsesforbedring i stensilkalkulasjoner på GPUer med MPI-kommunikasjon. Merk imidlertid at ytelsesgevinstene som oppnås med disse optimaliseringene avhenger i stor grad av problemstørrelsen og maskinvarekonfigurasjonen. På den største problemstørrelsen som ble testet, ga alle tre optimaliseringene i kombinasjon en hastighetsøkning på 1,09 når man brukte noder med V100 GPUer og 1,17 hastighetsøkning når man brukte noder med A100 GPUer.

Våre resultater gir innsikt for beslutningstakere når de vurderer implementeringen av disse optimaliseringene i ulike problemområder. Forslag til videre arbeid med dette benchmark-verktøyet er også inkludert.

Acknowledgements

First and foremost, I would like to thank my advisor Anne C. Elster for her invaluable guidance throughout the project. She kept me grounded when challenges seemed insurmountable. In addition, the community she has fostered in the HPC lab has resulted in a great work environment. And the workstations provided at the lab have been a great boon to the work.

I would also like to thank former HPC-Lab master student Andreas Hammer for a well-written and interesting masters thesis entitled "Analyzing Halo Computations on Multicore CPUs". His solid work was an inspirations for this thesis.

Thanks are also due the Department of Computer Science at NTNU for their continued support of the HPC-Lab and the IDI compute cluster which enabled this work.

I want to thank my fellow HPC-lab members who have provided emotional and academic support throughout the project.

Lastly, I thank my parents, whose encouragement was vital when things were stressful.

Contents

List of Figures

List of Tables

List of Listings

Nomenclature

A100	Nvidia datacenter GPU with Ampere architecture, the successor of the Volta architecture
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture, GPU programming platform
GPU	Graphical Processing Unit
HPC	High-Performance Computing
Idun	Compute cluster at NTNU, see https://www.hpc.ntnu.no/idun/
MPI	Message Passing Interface
Slurm	Job scheduler used on compute clusters
V100	Nvidia datacenter GPU with Volta architecture

Chapter 1

Introduction

In recent years, performance gains in computing have looked less at single-threaded performance on single cores and more at the performance achievable with multi-threaded programs running on powerful multi-core processors. Many compute-heavy problems are excellent candidates for being run massively in parallel. Among these are numerical methods, such as the finite difference method. Often, it is desirable to perform these calculations on problem domains so large that they cannot fit in, or make efficient use of, the memory or computational resources of a single machine. So the computations have to be distributed over a larger cluster of computers. The nodes doing the individual computations will often need to communicate some amount of data about the borders of their domains between each other to get the correct result.

Stencils are a family of algorithms that update points of a multi-dimensional data mesh with the neighbouring values as function inputs. They are used in various domains such as medical imaging, numerical methods like the Jacobi method, and convolution neural networks[1]. Stencil computations account for large amounts of computational resources used in many High-Performance Computing (HPC) centres worldwide, up to 49% of the workload [2].

Since the late aughts, the Graphical Processing Unit (GPU) has become an incredible piece of hardware for massively parallel scientific computations. They offer much slower single-core performance than modern Central Processing Units (CPUs) but, in turn, are outfitted with many times more cores to do computations. They are great candidates for doing stencil computations.

Stencil computations are a well-studied area with many common optimisations to increase performance. It is essential to understand what different optimisations are worth, so we can make informed decisions about whether implementing certain features is worth the human work hours it takes to implement [3].

This thesis looks at a few different optimisations for the communication step of stencil computations running on Graphical Processing Units on cluster machines and communicating using MPI. The optimisations we're looking at are deep halos, overlapping communication and calculation, and synchronised border exchanges.

1.1 Research Questions

These are the questions that we intend to answer in this thesis:

- **RQ1:** What performance impact will deep halos and synchronised halo exchanges have on halo exchanges on modern GPUs?
- **RQ2:** What performance impact can be expected from overlapping communication and calculation with stencil computations on GPUs?
- **RQ3:** How do these different optimisations impact each other?

1.2 Contributions

The primary contributions of this thesis are detailed descriptions of how to implement the optimisations, and the performance increases they provide. On the largest problem size tested, all three optimisations in conjunction gave a 1.09 speedup when using nodes with V100 GPUs and a 1.17 speedup when using nodes with A100 GPUs. In general, it finds that on most of the tested problem sizes, especially the biggest ones, the synchronous halo exchange with overlapping communication and calculations with a halo depth of eight performs best. The results clearly show why the halo exchange's synchronous version is more used than the asynchronous version.

1.3 Outline

The rest of this thesis is outlined as follows:

- **Chapter 2: Background -** A look at the most relevant and necessary theoretical knowledge that lays the groundwork for the rest of the thesis.

- **Chapter 3: Methodology -** Describes the details of how the prototype functions and how the performance of the different parts was measured.

- **Chapter 4: Experimental setup -** Overview of the specific software and hardware the prototype was run on. Also where which parameters were used for each different run are documented.

- **Chapter 5: Results & Discussion -** Shows the results of the benchmarks and talks about what we can deduce from those.

- **Chapter 6: Conclusions & Future Work -** Summarises the thesis and provides a list of future work that may be interesting to study further.

- **Appendix A: Running guide -** Readme from the code repository.

- **Appendix B: Code snippets -** Relevant code from the project that would take up too much space in the text of the thesis.

- **Appendix C: Result tables -** All the benchmarking results in numerical form in tables.

Chapter 2

Background

This chapter will cover the theoretical groundwork for the thesis. Topics will include a short description of General Purpose GPU programming, an introduction to MPI and MPI Datatypes, a short overview of CUDA including Memory transfers and CUDA streams, as well as a general introduction to stencil computations, the 2D Laplacian operator, and 2D grid decomposition and halo exchanges. Finally, an overview of how to overlap communication and computation and packing/unpacking of deep halos is given.

Note that Sections 2.2, 2.4, 2.6 and 2.7.2 are based on equivalent Sections from the Authors pre-project (fall-project) entitled "Halo Exchange Performance Using CUDA MPI".

2.1 General Purpose GPU Programming

General-purpose GPU programming, or GPGPU programming, is a technique that makes use of the massive computational power of GPUs for tasks unrelated to graphics. In the past, GPUs were built only for the purpose of rendering video game graphics, images, or other graphically intensive programs. However, modern GPUs have architectures that have opened the path for new programming techniques, that have made it possible to make use of the parallel processing functionality of GPUs to speed up a host of different compute-intensive tasks. By moving computations from the CPU to the GPU, applications are able to get significant speedups relative to running on the CPU [4]. To make use of the massive parallelism offered by GPUs, GPGPU programming uses specialised programming frameworks or languages, such as CUDA or OpenCL.

2.2 MPI

MPI, or Message-Passing Interface, is a standardised message-passing system designed for parallel computing on various computer architectures [5]. It allows different processes to communicate across different computers, allowing resources to be consolidated to solve a problem that is too large to handle on a single computer.

MPI is a low-level system often used in HPC environments, such as supercomputers and cluster computers. Following the Single-Program Multiple-Data (SPMD) model, MPI executes the same code in different processes. MPI processes still have access to control functions like conditionals, which can make processes behave differently based on their unique world rank despite running the same code.

When MPI processes communicate data between each other, it is often done using the functions `MPI_Isend()` and `MPI_Irecv()`, that respectively send and receive messages sent from one process to another. `MPI_Isend()` and `MPI_Irecv()` are the non-blocking forms of `MPI_Send()` and `MPI_Recv()`, meaning that multiple messages can be sent in succession without waiting for a corresponding receive function to be called on the other end of the communication. MPI also offers collective communication options such as `MPI_Reduce()` that gather values from every MPI process in the root process and applies some reduction operation to it.

When MPI programs read inputs, it is common for each process to read its own part of the input file rather than a single process reading the entire file and distributing the data among the other processes [6]. The processes then communicate the necessary parts between each other, then write their domains back to the output, as shown in Figure 2.1.

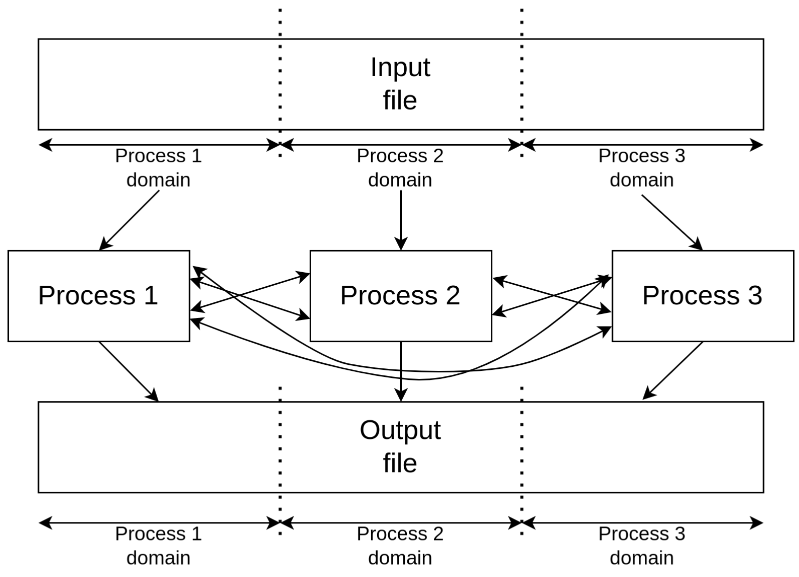

Figure 2.1: IO and communication pattern for MPI programs: Each process reads its own section from the input and writes its own section to the output.

2.2.1 MPI Datatypes

MPI datatypes allow for the coordination of data layout and structure during communication between processes. They describe how the packing and interpretation of data should happen when receiving or sending messages. MPI comes packaged with a lot of MPI datatypes, like `MPI_Float`, `MPI_Int` and `MPI_Byte`.

Often it is however desirable to communicate arbitrary collections of data. MPI allows the programmer to define custom MPI types. A simple such case is strided data, a type for which can be defined using `MPI_Type_vector()`.

2.3 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA [5]. It enables the use of a GPU for General Purpose Computing which can provide a significant increase in computational resources over the CPU. CUDA is supported on a wide range of NVIDIA GPUs which makes it easy to write efficient general-purpose GPU computing programs without specialised hardware or hardware-specific knowledge.

CUDA programs execute code on the GPU through kernels. Kernels are launched with a special syntax where the programmer has to state how many CUDA threads should be started expressly. CUDA kernel launches are inherently asynchronous, meaning that the code on the CPU resumes after the kernel has been launched on the GPU without waiting for it to finish.

2.3.1 Memory transfers

One primary consideration when using CUDA is the split between device (GPU) and host (CPU) memory. By default, a CUDA kernel cannot directly access host memory, and the host cannot directly access the device's memory. Memory on the device needs to be explicitly declared, and data has to be moved between host and device using the function `cudaMemCpy()` or one of its derivatives [7]. There are other methods for managing device and host memory, like Unified Memory, but that is not part of this thesis.

2.3.2 CUDA streams

Even though CUDA kernel launches are asynchronous, CUDA kernels will execute in a serial manner by default. CUDA has functionality for scheduling kernels in different CUDA streams. Such streams allow kernels to be scheduled to run concurrently on the device. An example of this is illustrated in Figure 2.2. Using CUDA streams to run kernels concurrently can often lead to performance increases [8].

When using multiple CUDA streams, the device will wait for all work in all other streams to finish before it begins executing work in the default stream.

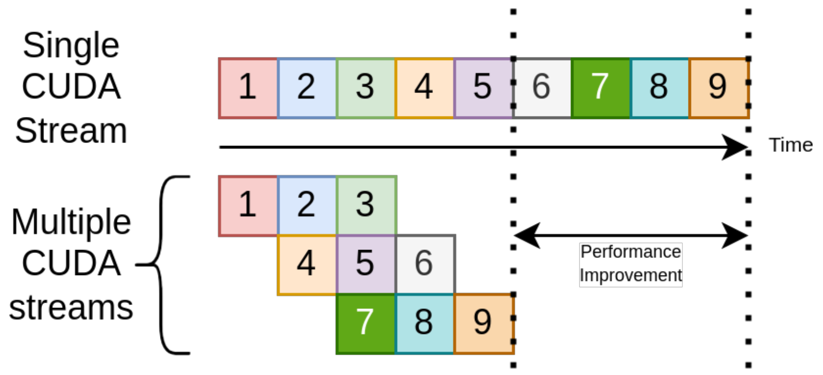

Figure 2.2: Example showing kernels executing with three CUDA streams in parallel to complete faster, based on Figure 3 from [8].

2.4 Stencil computations

Stencil computations are a common type of parallel algorithm used in scientific computing [9]. They are used to solve differential equations using numerical methods like finite difference, finite volume, and finite element. These methods are used in HPC applications like space weather predictions, seismic wave propagation, simulating fluid dynamics and others. In stencil computations, a set of mathematical operations are performed on each element in a grid of data. Each gridpoint in the data grid can have one or multiple values, also referred to as quantities. Each element is updated based on the values of the neighbouring elements in the grid. The stencil defines the magnitude and manner in which the neighbouring elements affect the number being updated.

Stencil computations are easily parallelised, meaning they can be done using many computing cores spread over many different computing units and computers.

Stencil computations are often iterative, for example, with the finite difference method, where each iteration will numerically approximate the grid values for the next time step.

2.5 Laplacian operator

The discrete Laplacian of a function ϕ can be approximated by Equation 2.1. The discrete Laplacian is often used in edge detection for images[10].

$$\Delta_h \phi = \frac{\phi(x+h,y)+\phi(x-h,y)-4\phi(x,y)+\phi(x,y+h)+\phi(x,y-h)}{h^2} \tag{2.1}$$

The discrete Laplacian can be presented as a difference stencil, or difference star, which can be used for stencil computations. The difference stencil for the discrete Laplacian is shown in Figure 2.3[11].

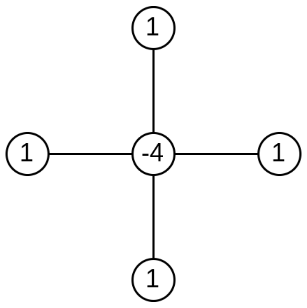

Figure 2.3: Difference stencil for the discrete Laplacian

Since this stencil has five points, it is often referred to as a five-point stencil.

2.6 Domain decomposition

If we want to model phenomena with high resolution, either in the space or time domains, we get a lot of data which leads to gigantic stencil grids. Currently, large-scale simulations running on the CPU use grids of sizes up to 10^{10}, running on 10^5 CPUs [12][13]. Yet they are too small to capture many interesting phenomena in a timely and energy-efficient manner.

When doing large stencil computations on GPUs, the problem grid may be significantly larger than what any single GPU can store in memory. This leads to needing to decompose the grid into smaller subgrids, which are placed in different memories.

For recent large-scale stencil computations, it is typical to have subgrids of size 512^3 when the domain is three-dimensional or 8192^2 for two-dimensional problem domains, with anywhere between one to eight quantities for each grid-point. With a stencil of radius three. With at most 10^{10} total problem domain size [14][15].

When dividing a two-dimensional grid, it can be divided one-dimensionally like in Figure 2.4, or two-dimensionally like in Figure 2.5.

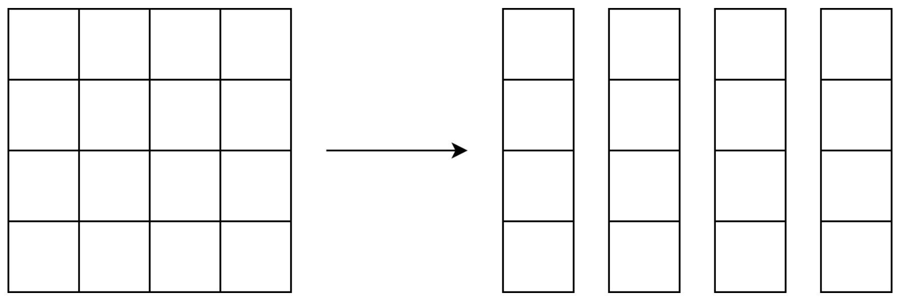
Figure 2.4: Problem domain one-dimensionally divided into smaller subdomains

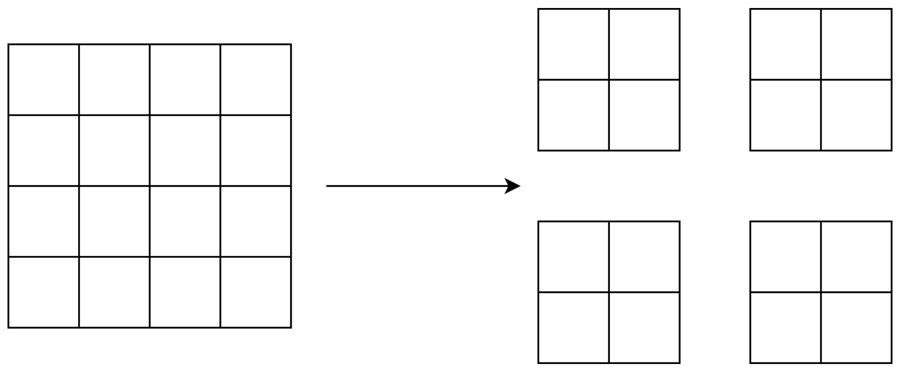
Figure 2.5: Problem domain two-dimensionally divided into smaller subdomains

2.7 Halo Exchanges

Stencil computations are based on calculating the value for a gridpoint based on the neighbouring values. When performing these computations on a distributed computing cluster, the neighbouring values are sometimes not stored on the same machines, which necessitates communication between the computing nodes. This exchange is called a halo exchange due to the ring of data that makes up the perimeter of the subdomain being exchanged [16].

Figure 2.6 shows a subgrid with 6x6 gridpoints. As we can see, the halo requires the extra ghost cells stored along the perimeter of the subgrid. The shape of the subgrid determines the number of ghost cells needed to be stored relative to the size of the subgrid. In Figure 2.4 the domain is split one-dimensionally which leaves us with tall thin subdomains with four data points each. These subgrids will need to store a total of 14 ghost cells each, causing more data to be stored on each node compared to the square subgrids in Figure 2.5 that contain the same amount of data points but need only store 12 ghost cells.

When doing halo exchanges with the domain split in more than one direction, it is common to reduce the number of communication instances by passing data to the corners of the halo by combining them with other communication. This is shown in Figure 2.7 where the lower right corner from the red subgrid, is passed to the blue subgrid alongside the bottom row of the orange subgrid. In a 2D grid,

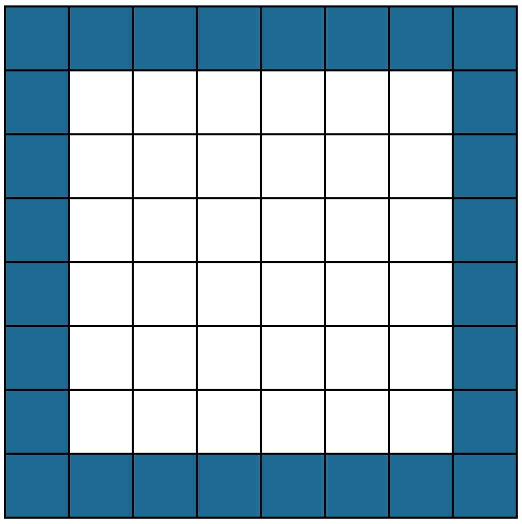

Figure 2.6: A 6x6 subgrid with a halo of size 1

this means that a subgrid that needs data from eight surrounding subgrids only needs to communicate with four of them. Though it introduces some blocking communication like in Figure 2.7, where the left/right exchange must happen before the up/down exchange happens to ensure that the red corner value arrives along with the orange values. For the rest of this thesis, we will refer to this version as a synchronous halo exchange due to it needing to wait for the first step to finish before performing the second step, even if the actual MPI communication is performed using asynchronous functions. The asynchronous version, shown in Figure 2.8 ignores all this coordination and simply sends the data directly to all eight neighbours.

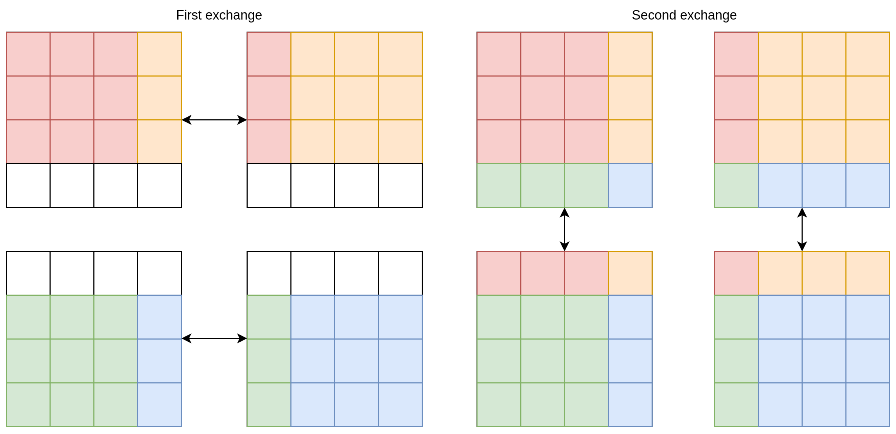

Figure 2.7: Synchronised sharing of border values

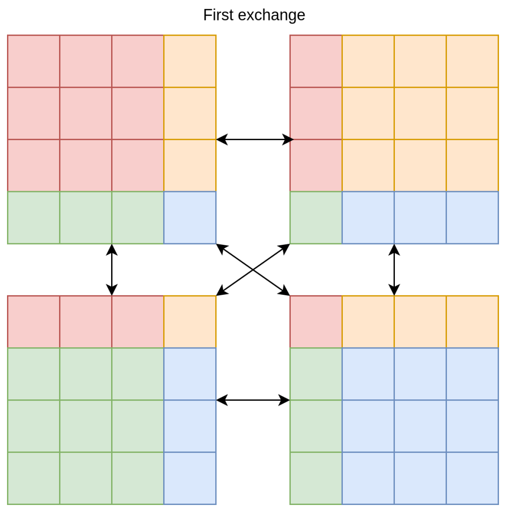

Figure 2.8: Asynchronous sharing of border values

2.7.1 Deep halo

Figure 2.7 shows halo exchanges with a depth of one. That means each subdomain exchanges one layer of data with its neighbours. It is possible to exchange more than one layer of data, which is necessary when using larger stencils but also allows multiple iterations of computations to be done on the subdomain between halo exchanges. Such halo exchanges are called deep halos. Each level of depth allows for one extra iteration of computation to be done without another round of message passing [16].

Deep halos allow each node to calculate what the neighbouring data would be in the next iteration if it was communicated. Each additional layer of halo increases the amount of data needed to be stored in memory for each subdomain, the amount of data being transferred in each communication, and increases the number of calculations. In return, the time spent on communication overhead is reduced due to needing fewer messages between nodes.

Figure 2.9 shows a round of halo exchange and computations on a subgrid of size 4x4, with a halo depth of two. It is important to note that deep halos increase the number of computations needed for each iteration of calculations [16]. In Figure 2.9, the innermost layer of the deep halo needs to have its new values calculated in the first iteration, so that the values will be correct for the second iteration.

2.7.2 Pack/Unpack

Very often, the border data that is to be exchanged with a neighbouring subgrid is not stored contiguously in memory. CUDA is an extension of C++, which is a row-major language. Being row major means that arrays are stored with contiguous rows in memory rather than columns. In Figure 2.7 we see a halo exchange where

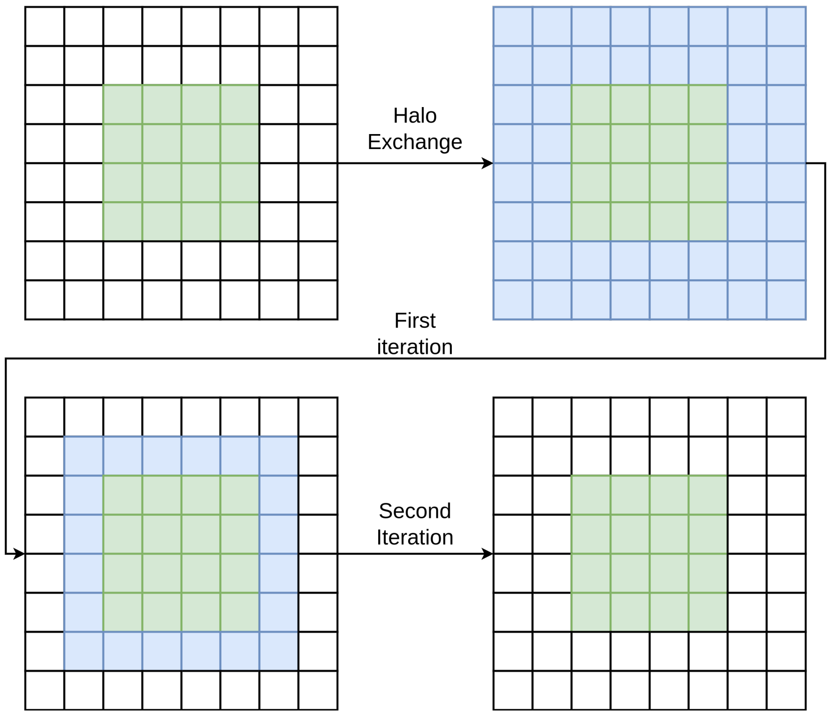

Figure 2.9: Halo exchange with halo depth two and two iterations of calculations, based on Figure 2.4 from Master´s thesis by A. Hammer [10]

columns are being changed between two subgrids. After the border exchange the data stored in one of the subgrids may look something like what is shown in Figure 2.10 which shows how four rows would be stored in memory.

Passing non-contiguous data between computers, or between a GPU and CPU can be costly. Therefore it is often beneficial to pack non-contiguous data into contiguous buffers, and then transfer those buffers between nodes [9]. In Figure 2.10 we need to pack the gridpoints stored to the left, or just before the blue border points in memory, into a contiguous buffer.

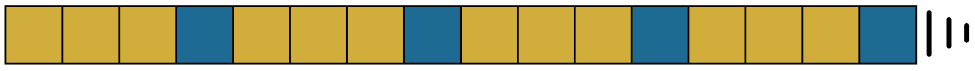

Figure 2.10: Border data is not contiguous in memory

In return, the node working on the yellow subgrid will receive a packed buffer of blue gridpoints, like Figure 2.11 which must then be unpacked into the subgrid before it can be used for calculation.

Figure 2.11: Blue border data packed contiguously

2.8 Overlapping calculation and communication

The subgrid's interior gridpoints can be computed without receiving the border data from neighbouring subgrids as the necessary data needed to perform the next iteration of the computations is already present.

Figure 2.13 shows how the inner and outer data is divided in a subgrid with a halo depth of one, and where the stencil has a length and height of three. The green gridpoints can be computed regardless of whether the blue gridpoints, the border cells, are present. The gridpoints that are dependent on the ghost cells are those coloured in red. The inner and outer compute kernels can be launched separately; this way, a program can run its communication stage concurrently with its calculation stage following the dependency graph shown in Figure 2.12.

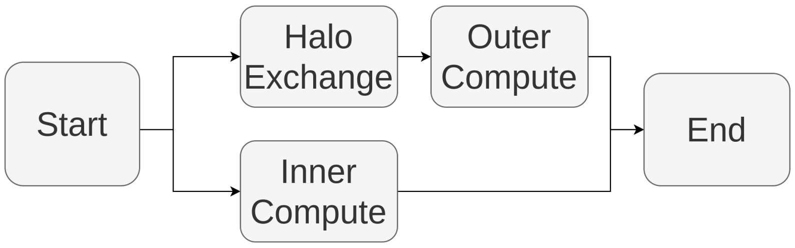

Figure 2.12: Overlapping message passing and calculation.

.

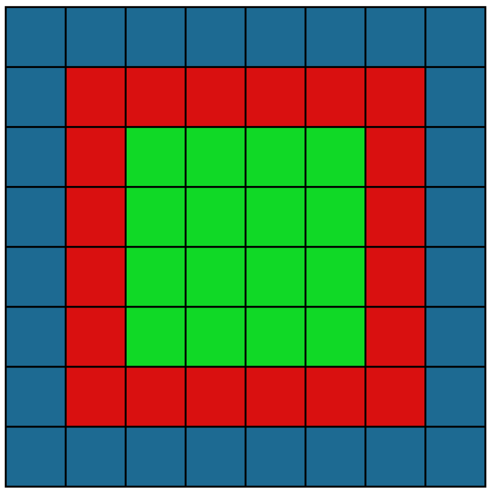

Figure 2.13: The green gridpoints in the mesh can be calculated before the blue border gridpoints are received.

Chapter 3

Creating a 2D Halo Exchange Benchmark for the GPU

This chapter provides details on how our 2D halo exchange benchmark for the GPU was designed and the relevant optimisations we did for the benchmark implementation. The details include initialisation such as input parameters, domain decomposition, defining the MPI topology, domain adjustment and initialisation, and buffer creation on the CPU and the GPU. Also, the CUDA functions we used for inner and outer compute, packing/unpacking of data and time measurements. Section 3.3 provides further communication details, including how to implement synchronous and asynchronous halo exchanges, and how to overlap communications and computations. Finally, we list a set of measuring parameters and highlight some of the challenges with our benchmark implementation.

3.1 Initialisation

Before calculations and measurements can be made, there are many things to initialise. The setup for the implementation is outlined here.

3.1.1 Input parameters

To make testing easier, many of the configurations for running the code were passed through as command line parameters. This allows for many tests to be run with different configurations on the same source code. Table 3.1 outlines the parameters used.

The height and length parameters decide the total size of the problem domain. Depth sets the halo size for each subdomain. Iterations set how many times the stencil will pass over the problem domain. Sparse, when true, changes the print mode so log files with the output of multiple runs are more easily parsed for analysis

Table 3.1: Input parameter table

Parameter name	Command line flag	Default
height	-h	100
length	-l	100
depth	-d	1
iterations	-i	1
sparse	-s	0

3.1.2 MPI domain decomposition

The dimensions the problem domain gets split into are based on how many MPI processes are run. The MPI dimensions are defined by the programmer in the `factorise()` function in the `utils.c` file. The dimensions chosen for the tests try to split the problem domain as even length- and height-wise as possible. The dimensions are shown in Table 3.2.

Table 3.2: Problem domain decomposition dimensions

MPI world_size	Height	Length
1	1	1
2	1	2
4	2	2
6	2	3
8	2	4
9	3	3
10	2	5
12	3	4
16	4	4

The problem domain height and length are divided by the height and length of the MPI dimensions. If the problem domain dimensions are not evenly divisible by the MPI dimensions, the leftover rows and columns are distributed in such a way that if there are *n* leftover rows, the *n* first MPI processes on each column gets an extra value in that direction, the same goes for columns.

3.1.3 Topology

The MPI processes have their neighbours defined in `utils.c` with neighbours in all cardinal and ordinal directions. To benchmark processes having to perform halo exchanges with neighbours in all directions, we decided that the problem domain would be periodical in both north-south and east-western directions, rather than have a separate boundary condition. This means that we only need an MPI `world_size` of four for each process to have neighbours that are not themselves in all eight directions and an MPI `world_size` of nine for every process to have a unique neighbour in all eight directions.

3.1.4 Height and length adjustment

Before the problem domain is initiated the adjusted height and length are calculated for the subdomain. The adjusted height and length are the subdomain height and length plus two times the halo depth.

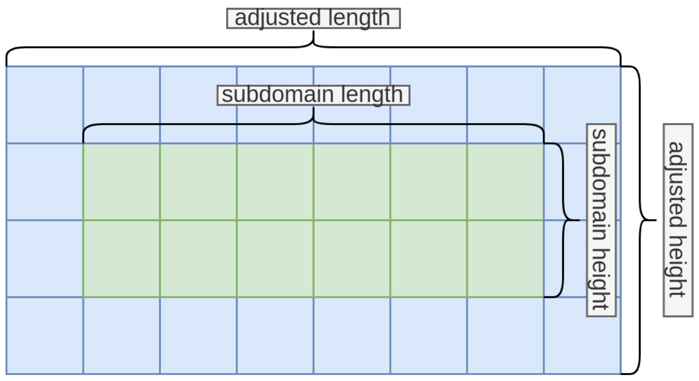

Figure 3.1: Adjusted height and length with subdomain height and length on a 2x6 subdomain with a halo depth of 1

3.1.5 Domain initialisation

As explained in Section 2.2, when reading data for MPI processes it is normal for each process to read its own subdomain from the input, rather than have a single MPI process load the entire domain and scatter it throughout the communicator. Since this implementation uses dummy data it is left to each process to generate its own subdomain. This is done using a single `malloc()` statement, to initialise a one-dimensional array of floats with a size equal to $adjustedHeight * adjustedLength$. The initialisation is shown in code Listing 1. The structure of the subdomain is shown in Figure 3.2. Every value in the subdomain is initialised to one.

```
int area = adjustedHeight * adjustedLength;
float *problemArray = malloc(area*sizeof(float));
for(int i =0;i<adjustedHeight;i++){
    for(int j = 0; j<adjustedLength;j++){
        problemArray[i*adjustedLength+j] = 1;
    }
}
```

Listing 1: Initialising the subdomain

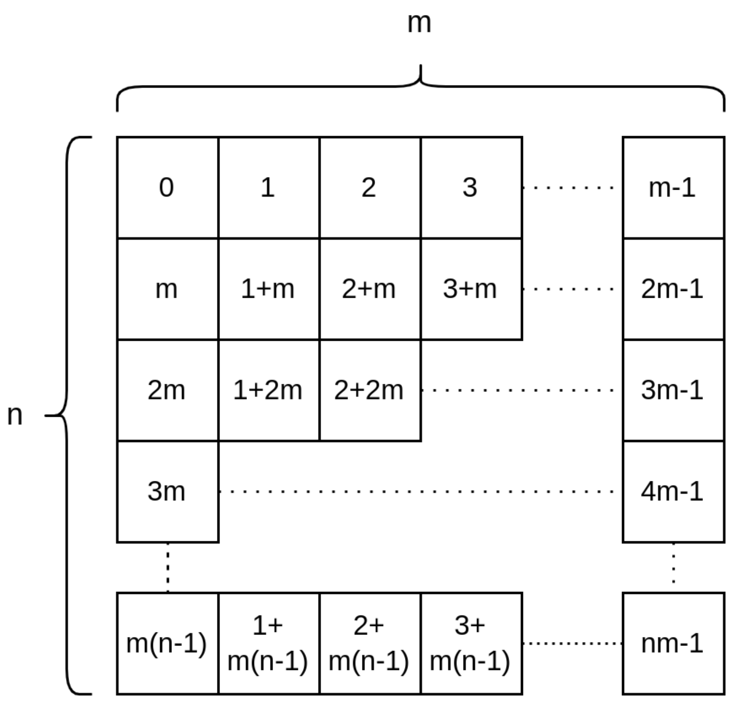

Figure 3.2: The subdomain is stored in a row-major fashion, where m is the length of the subdomain, and n is the height

3.1.6 Border buffers

All the buffers that are used to communicate the borders are initialised in the setup. Each buffer is a one-dimensional array of floats. And there is a buffer for each cardinal direction. Each buffer also has a corresponding receive buffer of the same size. Values from neighbours in ordinal directions are stored in the north and south buffers

Table 3.3: Border buffers and their sizes, there are two copies of each buffer, one for sending and one for receiving messages

Buffer name	Height	Length
left border	original height	depth
right border	original height	depth
north border	depth	adjusted length
south border	depth	adjusted length

3.1.7 Initialising the GPU

The border buffers are initialised on the GPU, with sizes corresponding to those in Table 3.3. This is done using `cudaMalloc()`. Two arrays the size of the subdomain are allocated on the GPU, `deviceinput` and `deviceprocess`. The former is where we will store the subdomain on the GPU, the latter is where the result of each iteration will be stored. A short buffer is also initialised for the stencil. Three CUDA streams, `altStreamOne`, `altStreamTwo`, and `altStreamThree` are initiated here so that they can be used later. The data from the subdomain and the stencil is loaded onto the GPU.

3.2 CUDA functions

To have the MPI processes running a C program call functions from a CUDA file, all the functions in the CUDA file are declared as `extern "C" void` functions. This section describes all the functions that deal with the GPU during the iterative loop.

3.2.1 Primary computations and inner compute

The majority of the computation is performed in the `calculate` CUDA kernel, it is called by the `launchKernel()` and `innerCompute()` functions. The kernel itself is a really simple implementation of stencil computations. It starts by finding its own global coordinates x and y based on the thread and block coordinates. Then it checks to see if the coordinates are in a valid range of the area that is to be computed. Since all blocks in a kernel call are the same size, this check is necessary to avoid segmentation faults, as the kernel will often launch with a few more threads than necessary. Then the new value for the point is calculated by multiplying the value in the `deviceinput` array and its neighbours by the corresponding kernel value and being added together. The kernel can work with any five-point stencil but would need adjustment if, for example, a full 3×3 stencil was to be used. The sum is then stored in `deviceprocess` index with coordinates identical to the one used for the `deviceinput` array. The main CUDA kernel for computations is shown in Listing 2.

When launching the compute kernel, the `deviceinput` pointer should point to the first index to be computed. When launching the kernel on the entire subdomain using `launchKernel()`, the `computeHeight` and `computeLength` values are set to the adjusted height and length of the subdomain minus two, since the outermost values cannot be calculated. When used for the inner compute step during concurrent communication and calculation, the `computeHeight` and `computeLength` is set to be the original height and length of the subdomain minus two.

```
__global__ void calculate(float *deviceinput, float* deviceprocess,
    int *dkernel, int computeHeight, int computeLength,
    int adjustedLength){

    float sum = 0.0;
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;

    if(x< computeLength && y < computeHeight){
      sum+=deviceinput[x+y*adjustedLength-adjustedLength]*dkernel[0];
      sum+=deviceinput[x+y*adjustedLength-1] *            dkernel[1];
      sum+=deviceinput[x+y*adjustedLength] *              dkernel[2];
      sum+=deviceinput[x+y*adjustedLength+1] *            dkernel[3];
      sum+=deviceinput[x+y*adjustedLength+adjustedLength]*dkernel[4];
      deviceprocess[x+y*adjustedLength] = sum;
    }
}
```

Listing 2: Main compute kernel CUDA code. `dkernel` is an abbreviated name for `devicekernel` so the listing fits on the page.

3.2.2 Outer compute

To perform the outer compute stage, such as is described in Section 2.8, two CUDA kernels are used. They are very similar to the normal compute kernel described in Subsection 3.2.1, but are made to take in two z levels of blocks. The vertical kernel has one z level of blocks for the left and right borders of the domain each, and the horizontal kernel has one z level of blocks for the north and south borders of the domain each. The coordinates for the vertical kernel are calculated as shown in Listing 3, and the horizontal coordinates are calculated as shown in Listing 4. `calculationoffset` is an offset calculated on the CPU side, which is equal to the distance between the first elements of the north and south borders for the horizontal kernel, and between the first elements of the left and right borders for the vertical kernel.

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int xd = x + blockIdx.z * calculationoffset;
int y = threadIdx.y + blockDim.y * blockIdx.y;
```

Listing 3: Calculating coordinates for vertical outer compute

x and y are still used for the range check like in Listing 2, but xd and yd are used when accessing elements in `deviceinput` and `deviceprocess`.

These kernels are launched from `main.c` by calling the function `outerCompute()`.

Both of the outer compute kernels are launched in separate CUDA streams to achieve better occupancy on smaller subdomain sizes.

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
int yd = y + blockIdx.z * calculationoffset;
```

Listing 4: Calculating coordinates for horizontal outer compute

3.2.3 Packing and Unpacking

As described in Subsection 2.7.2, it is beneficial to pack data into contiguous buffers before transfer from GPU to CPU, in Listing 5 we see the packing kernel for vertical borders. We don't use a packing kernel for the horizontal borders because the data can be fetched directly from those with a `cudaMemCpyAsync()` call.

```
__global__ void packVert(float *deviceinput, float *border,
    int height, int length, int depth, int ogheight){

    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    int target = x + y * depth;
    int source = x + y * length;

    if(x < depth && y < ogheight){
        border[target] = deviceinput[source];
    }
}
```

Listing 5: Kernel for packing values from subdomain into border buffers

The function `fetchBorders()` launches two instances of the packing kernels, starting them both in different CUDA streams. Then it makes four asynchronous CUDA memory transfers from device to host, one for each border. The memory transfer calls for the left and right border are in the same stream as the corresponding pack kernel was launched in, this way we can ensure that the kernel finishes before the data is read from it without worrying about any further control structures in the code.

In Listing 6 we see the unpacking kernel, which is almost identical to the packing kernel.

```
__global__ void unpackVert(float *deviceinput, float *border,
    int height, int length, int depth, int ogheight){

    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    int target = x + y * length;
    int source = x + y * depth;

    if(x < depth && y < ogheight){
        deviceinput[target] = border[source];
    }
}
```

Listing 6: Kernel for un-packing values from border buffers into the subdomain

The function `placeBorders()` first performs four asynchronous CUDA memory transfers. Similarly to the fetching function, the left and right border memory transfers are launched in different CUDA streams, so that once the unpacking kernels are launched, we can use the serial nature of the streams to ensure that the packed border buffers are fully in place before being packed out.

3.2.4 Measuring time for CUDA Kernels

Timing CUDA kernel launches and memory transfers is done using events. In Listing 7 we see all the calls involved in taking the times of CUDA functions using two different streams to achieve concurrency. This is how time is taken for the pack and unpack functions, and the outer compute function. The inner compute function runs in a third stream, `altStreamThree`, and as such only needs a single set of start and stop events. When launching the compute kernel on the entire subdomain, the calculate kernel is executed and timed in the default stream.

```
extern "C" void someFunction(){
    cudaEvent_t start1, start2, stop1, stop2;
    cudaEventCreate(&start1);
    cudaEventCreate(&stop1);
    cudaEventCreate(&start2);
    cudaEventCreate(&stop2);
    cudaEventRecord(start1, altStreamOne);
    cudaEventRecord(start2, altStreamTwo);

    /* perform kernel launches and cudaMemCpyAsync
       calls in altStreamOne and altStreamTwo */

    cudaEventRecord(stop1, altStreamOne);
    cudaEventRecord(stop2, altStreamTwo);
    cudaEventSynchronize(stop1);
    cudaEventSynchronize(stop2);
    float milliseconds1, milliseconds2;
    cudaEventElapsedTime(&milliseconds1, start1, stop1);
    cudaEventElapsedTime(&milliseconds2, start2, stop2);
    /* store larger of milliseconds 1 and 2
       in global accumulator variable */
}
```

Listing 7: Code using events to take the time of kernel calls and `cudaMemCpyAsync()` calls with two streams

3.3 Communication

For this implementation, we're benchmarking two different patterns for communicating deep halos using MPI. One version is the asynchronous way of exchanging borders shown in Figure 2.8 where each process communicates with all of its neighbours directly. The other is the synchronous method shown in Figure 2.7 where one round of communication is performed along one cardinal axis, and then the corner values are passed through in the second round of communication along the other cardinal axis.

3.3.1 Asynchronous halo exchange

The asynchronous communication function for performing the halo exchange is called `asynchronousMpiBorderExchange()`, which can be found in Appendix B.

`asynchronousMpiBorderExchange()` begins by using eight non-blocking receive operations to receive the border data from its neighbours in all ordinal and cardinal directions. All the receive operations store their request status in a `recvRequests` array.

```
MPI_Datatype MPI_corner;
MPI_Datatype MPI_center_horizontal;
MPI_Type_vector(depth, depth, adjustedLength,
    MPI_FLOAT, &MPI_corner);
MPI_Type_commit(&MPI_corner);
MPI_Type_vector(depth, myLength, adjustedLength,
    MPI_FLOAT, &MPI_center_horizontal);
MPI_Type_commit(&MPI_center_horizontal);
```

Listing 8: Code snippet declaring MPI datatypes with the vector function

Then, the process sends all its border data directly to all eight neighbours using the non-blocking `MPI_Isend` operations.

Lastly, the function waits using `MPI_Waitall` on the `recvRequests` array to ensure all the receives have been completed before the code continues to execute.

We use two custom MPI datatypes to send and receive the correct data out of and into the north and south border buffers. They are declared as in Listing 8. We use `MPI_center_horizontal` when sending and receiving the middle portions of the north and south border. `MPI_corner` is only used when receiving the corner values into the north border. We do not use it when sending because the corner values can be sent directly from the left and right border buffers where the data is already contiguous in memory.

3.3.2 Synchronous halo exchange

The synchronous communication function for performing the halo exchange is called `synchronousMpiBorderExchange()`, and it can be found in Appendix B.

`synchronousMpiBorderExchange` uses non-blocking communication operations to asynchronously receive the border data from a process' left and right neighbours into the corresponding receive buffers (`leftBorderReceive` and `rightBorderReceive`). It then asynchronously sends its left and right border data to the left and right neighbours, respectively (`leftBorder` and `rightBorder`) with non-blocking sending operations.

After posting all the non-blocking send and receives for the horizontal border exchange, the code waits for the non-blocking receive calls to resolve by invoking `MPI_Wait` and stores the received data in the appropriate indices the `northBorder` and `southBorder` arrays. This is done by iterating over all the corners of the borders with **`for`** loops. This operation has a time complexity of $O(n^2)$, scaling with halo depth.

Once the data has been moved between the arrays, the function again uses the non-blocking receive operations to asynchronously receive the border data, this time from the north and south neighbours (`northBorderReceive` and `southBorderReceive`). It also sends the local north and south border data to the north and south neighbours (`northBorder` and `southBorder`) using non-blocking send operations.

Finally, the function uses `MPI_Wait` for the receive operations to complete so that the rest of the code doesn't execute before the `northBorderReceive` and `southBorderReceive` buffers contain the data from the neighbouring processes.

3.4 Overlapping communication and computation

When doing overlapping communication and calculation, we are looking to execute code on the host and the device at the same time. And in the context of running concurrently with halo exchanges, we are really only looking to run the calculations on the interior grid points while performing the communication step.

This is easily achieved since CUDA kernel launches are non-blocking. So after fetching the border data from the device, the host code launches the kernel for the inner computation, the same as the kernel shown in Subsection 3.2.1. But starting at the interior gridpoints and with the compute height and length adjusted so it stays within the interior. The function used to launch the inner compute is shown in Listing 9.

```
extern "C" void* innerCompute2(struct blockDims blockDims){
    dim3 grid(blockDims.gridLength, blockDims.gridHeight, 1);
    dim3 block(blockDims.blockLength, blockDims.blockHeight, 1);
    calculate<<<grid,block,0,altStreamThree>>>
        ( &deviceinput[adjustedLength * (depthVal + 1) + depthVal + 1],
        &deviceprocess[adjustedLength * (depthVal + 1) + depthVal + 1],
        devicekernel, ogheight-2, oglength-2, adjustedHeight,
        adjustedLength, depthVal);
    return NULL;
}
```

Listing 9: Function to launch the calculate kernel on only interior gridpoints

Once this function returns, which happens after the kernel has been queued on the device and not after the kernel finishes, one of the communication methods described in Section 3.3 is used. Once the MPI communication step is complete, the borders are loaded onto the GPU and placed into the subdomain array. Then the outer computation is started.

3.4.1 Iteration loop

In Listing 10 the main iterative loop with overlapping communication and computation is shown. First, the borders are fetched, and then the inner computation is started. While the inner computation is running, the halo exchange happens; in this case, it is the asynchronous version, but the loop is identical to the synchronous version sans the name of the halo exchange function. After the halo exchange is finished, the borders are placed back onto the GPU and moved into the correct positions on the array so the outer compute can begin. The `swapInputProcessPointers()` function synchronises all streams on the device and swaps the pointers to `deviceinput` and `deviceprocess`. Afterwards, the calculation is run on the entire subdomain for the remaining halo depths.

```
for (int i = 0;i<i1;){
    fetchBorders(/* params */);
    innerCompute2(innerDomain);
    threadedSegmentStart = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD);
    threadedSegmentSum += MPI_Wtime() - threadedSegmentStart;
    messageStart = MPI_Wtime();
    asynchronousMpiBorderExchange(/* params */);
    messagePassing += MPI_Wtime() - messageStart;
    placeBorders(/* params */);
    outerCompute(outerBorders);
    swapInputProcessPointers();
    i++;
    for(int j = 1; j<depth && i<i1;j++){
        calcStart = MPI_Wtime();
        launchKernel(problemDomain);
        calcSum += MPI_Wtime() - calcStart;
        i++;
    }
}
```

Listing 10: Iteration loop for a run with concurrent communication and asynchronous border exchange. For the sake of brevity, the parameters of functions with long parameter lists are omitted.

The loop for the non-concurrent version is very similar. The function calls to the inner and outer compute functions are not made, there is no call to the function to swap input buffers, and the loop variable *i* is only incremented inside the second **`for`** loop. The second loop variable *j* is also initialised to zero. The exact code is shown in Appendix B.

3.5 Measurements

Each time our benchmark suite is run, we take measurements of the time spent at different times of the program. Since one cannot time a CUDA kernel reliably with Wall-Time, two different methods are used to take the times of varying program parts: the method using CUDA events described in Subsection 3.2.4, and the method using the function for taking the wall time that comes packaged with the MPI library, `MPI_Wtime`. The program parts timed are:

- **Border packing -** This segment involves everything about moving data from the device to the border buffers in host memory. It takes the time spent on two instances of the packing kernel shown in Listing 5 and the memory transfer of all four border regions with the packed data from the device to the host. This measurement is done using CUDA events.
- **Message passing -** This segment contains only the MPI communication in either of the two functions described in Section 3.3. This segment is timed using `MPI_Wtime`.
- **Border unpacking -** This segment, similar to the packing step, measures four memory transfers, this time from the host to the device, and two kernel launches, specifically the kernel shown in Listing 6. Again, this measurement is done using CUDA events.
- **Calculation time -** This segment refers specifically to the launch and runtime of the calculate kernel shown in Listing 2 when run on the entire subdomain. This measurement is done using CUDA events.
- **Outer compute -** This segment is the time to launch and complete the outer compute kernels discussed in Subsection 3.2.2. The measurements for this segment are taken using CUDA events.
- **Inner compute -** This segment is the time to launch and complete the inner compute. As discussed in Section 3.4, this kernel is the same as the one used to calculate the next iteration of the entire domain, but a smaller section of it. This segment is timed using CUDA events.
- **Desynchronised time -** Before the message passing segment, a `MPI_Barrier` call is made on all processes in the communicator. The purpose of this segment is described in more detail in Subsection 3.6.2. The measurement is done using `MPI_Wtime`.

3.6 Challenges

During development, a few tough challenges arose that took a while to solve. For posterity's sake, they're documented here, along with the approach used to resolve them.

3.6.1 POSIX threads

Early versions of the prototype used POSIX threads (Pthreads) to achieve concurrency while taking time measurements of the inner compute kernel execution. However, Pthreads have an overhead attached to them, and in the end, it turned out to be faster and simpler to launch the inner compute from the main thread since kernel launches are asynchronous already. This means that our implementation is no longer testing CPU-side multithreading.

3.6.2 Desynchornisation issues

Idun doesn't give us complete control over which nodes get assigned to our program. In the early stages of testing, it was discovered that certain runs of the prototype would have atrocious performance in the message-passing stage. This was usually caused by a single or two out of the four or nine nodes the program ran on having slower hardware. For example, the P100 GPUs on the nodes on idun-06 compared to the V100 or A100 on idun-04 and idun-05. Because of this hardware difference, the time measurements for the message passing stage would include the time the fastest nodes spent waiting for the slowest nodes to catch up and send their border data.

The final benchmarks are run on the same hardware for the most part, but demanding specific hardware on Idun will generally increase the time spent in the queue before a job runs. This would increase the time taken between testing code during development significantly. To remedy that, a call to `MPI_Barrier` can be made just before the message passing stage; this helps us get an accurate reading of how the different optimisations impact the message passing stage even when assigned to hardware that performs differently.

In program runs where the segments are individually timed, we also take the time of this `MPI_Barrier` call.

3.6.3 Fetching borders before or after inner compute

An early prototype version showed a slowdown when doing overlapping communication and calculations, especially on large domain sizes. This happened because we started the inner compute before we fetched the border data from the device. Even if everything would run in different streams, the occupancy for the inner compute kernel was so high that there weren't many leftover GPU resources for the border fetching. This meant that the inner compute would often finish entirely before the borders could be fetched from the device. The GPU would then be idle during the halo exchange before running the outer compute. Splitting the compute into an outer and inner step means that calculations can take more time, but it should be worth it due to the time saved on doing the communication in parallel with the calculations. However, by launching the inner compute kernel first, the concurrency could be partially lost, and the iterative loop could take longer due to the increased overhead.

This was solved simply by fetching the border data entirely before starting the inner compute stage.

Chapter 4

Experimental setup

This chapter provides an overview of the hardware and software used to run and test our benchmarking tool for 2D halo exchanges on GPUs. Compilation details and the different parameters used for each run of the benchmarks is also listed in Section 4.2 and Section 4.3.

4.1 The Idun Testbed

Different faculties and departments at NTNU have collaborated and contributed their computational resources to the Idun cluster. It is designed to provide students and researchers with the computational resources they need to rapidly create, test, and prototype HPC software. The university's IT division provides the cluster's backbone, high-speed switches, storage, and servers. The departments collaborating on the cluster are the ones who provide the computing resources. Departments can use the resources they themselves contribute, and any idling resources from other departments [17].

Slurm

The benchmark was run on Idun using Slurm, which is a job scheduler for Linux and Unix-like kernels. A single script ran all the versions of the code with each parameter listed in Section 4.3 so that all the runs for each GPU type would be assigned the same nodes with the same hardware.

Idun Nodes used in our Testbed

The nodes used when running on Idun are shown in Table 4.1.

The V100 nodes were used for all runs on the V100. For the A100, idun-06-[17-19] and idun-04-07 were used for almost all the runs. The exceptions were the 5000 × 5000 runs where Slurm failed to write to the log file, therefore that grid size was repeated separately. The nodes used for that were idun-04-06, idun-04-10 and idun-06-[18-19].

Table 4.1: Idun nodes used in benchmark, all server types are Dell, all Processors are from intel, and all GPUs are from Nvidia

Node	**Type**	**Processor**	**RAM**	**GPU**
idun-04-[01-03]	DSS8440	Xeon Gold 6148	754	V100 8x32Gb, 2x16Gb
idun-05-08	PE740	Xeon Gold 6132	754	Tesla V100 16Gb
idun-06-[17-19]	PE730	Xeon E5-2650 v4	128	A100 40Gb
idun-04-[06-07]	DSS8440	Xeon Gold 6248R	1509	A100 40Gb
idun-04-10	DSS8440	Xeon Gold 6248R	1509	A100 80Gb

4.2 Compilation

The implementation has been built to run MPI calls from the `main.c` file, linked with a CUDA file `mul.cu`. CUDA extends C++ with additional features. The C files using MPI have to be compiled using mpicc, while the CUDA files have to be compiled using nvcc. Finally, the executable is built using mpicc to link all the output files together. Certain libraries have to be included to ensure linking of the CUDA runtime and to supply the library directory for the CUDA runtime. The flags used for compiling are shown in Table 4.2.

Table 4.2: Compiler flags used with mpicc

-lcudart
-lstdc++
-L/usr/local/cuda-11/lib64/
-O3

4.2.1 Modules

When testing the code on idun, these are the modules used to get the runtimes and compilers necessary to perform the tests.

Table 4.3: Modules used to run on idun

foss/2022a
CUDA/11.7.0

Foss is a common compiler toolchain composed entirely of open-source software, hence the name which is short for Free and Open Source Software. Foss is the toolchain that provides the GNU compiler collection, and the OpenMPI library. The CUDA/11.7.0 module provides the CUDA toolkit necessary to run the program on CUDA-compatible GPUs.

4.3 Launch parameters

For all four versions of the code, we ran the tests 35 times with each set of parameters. The results for each different set of parameters when presented in Chapter 5 are the averages of 35 runs for each setting.

- **Problem domain dimensions -** All runs were performed with square problem domains with side heights and lengths of 5000, 10000, 15000, 20000, 25000, 30000, and 35000.
- **Iterations -** All runs were performed with 2048 iterations.
- **Halo depth -** The intervals at which we check the performance of different halo depths is the next power of two up to 64. So runs were done with halo depths of 1, 2, 4, 8, 16, 32, and 64.

These parameters were used on two different sets of nodes on Idun with different hardware. One set was with Nvidia V100 GPUs, and the other was with the Nvidia A100 GPUs.

Chapter 5

Results & Discussion

This chapter presents the results from benchmarking the prototype for 2D deep halos. The process explained in Chapter 4 produced a large amount of data, most of which relay similar information. Therefore, the figures in this chapter are just a selection of the results needed to show the performance impacts of the different optimisations. A complete list of the results with the problem sizes we used with our benchmarking code can be found in numerical form in Appendix C. Those sizes are 5000, 10000, 15000, 20000, 25000, 30000, and 35000.

5.1 Significant outliers

For the V100 benchmark for grid size 25000 × 25000, three runs in the synchronous overlap version at halo depth two in succession were significantly slower than all the other runs for the same input parameters. Those three runs took 5.482, 2.361 and 2.452 seconds passing messages, while the average for that set of runs with the same parameters is 0.193 seconds. We see no other explanation for this than that this is a slowdown caused by the hardware on Idun possibly being busy with another job since we were not able to run on Idun with exclusive access. Those outliers are therefore omitted from the results presented in this chapter.

5.2 Execution time

Here the results of the different execution time measurements for each combination of halo exchange optimisations er shown and discussed.

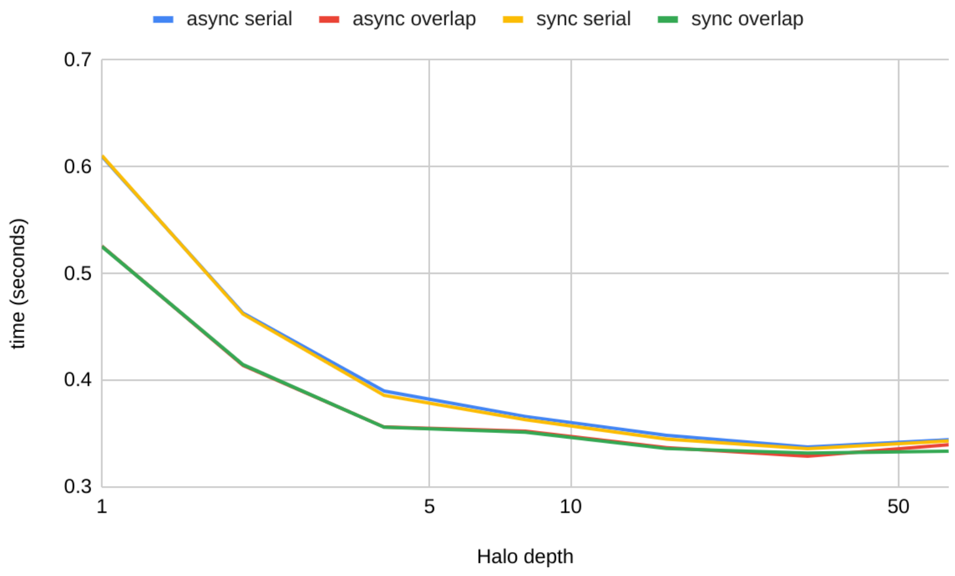

Figure 5.1: Execution times for side lengths of 5000 × 5000 on V100 nodes

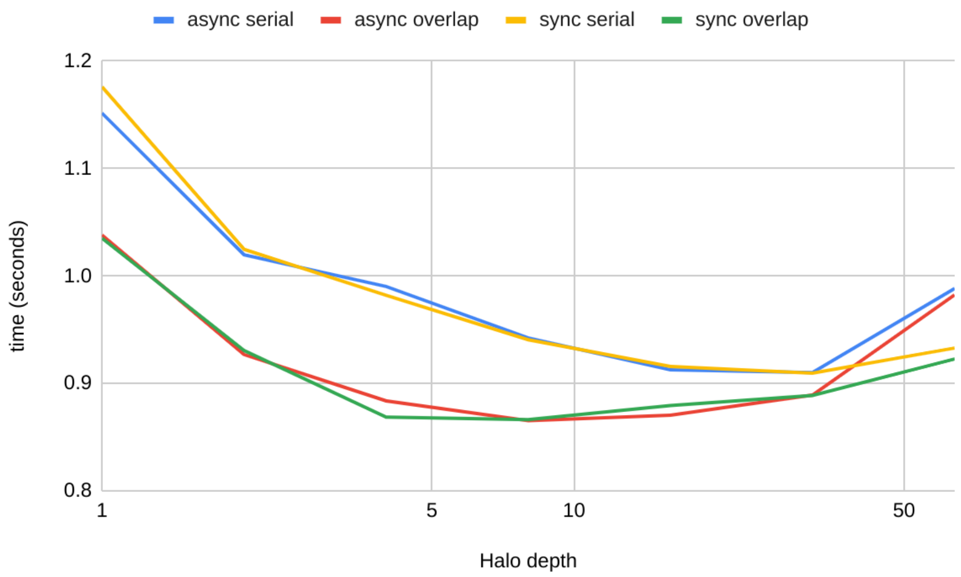

Figure 5.2: Execution times for side lengths of 10000 × 10000 on V100 nodes

For all grid sizes on the V100 GPUs, the results show that with a halo depth of one, the serial versions with synchronous and asynchronous halo exchanges perform similarly, and the overlap versions of synchronous and asynchronous halo exchanges perform similarly. However, the overlap versions tend to outperform the serial counterparts, even if only by a little. This is especially visible on the smaller grid sizes shown in Figure 5.1 and Figure 5.2, which shows the execution times with grid size 5000 × 5000 and 10000 × 10000 respectively. There we see a clear difference between the overlap and serial versions up until halo depth 64, where the asynchronous versions slow down faster than the synchronous versions.

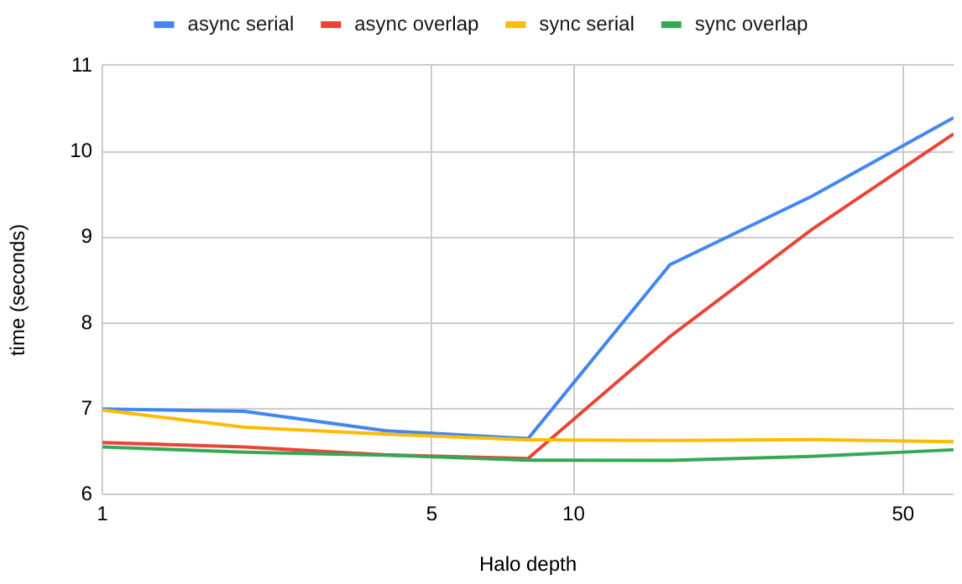

Figure 5.3: Execution times for side lengths of 30000 × 30000 on V100 nodes

When looking at the larger grid sizes, the results show that the versions with asynchronous halo exchanges perform significantly worse on the V100 nodes with the larger halo depth values. For example, for the runs with grid size 30000 × 30000, which are shown in Figure 5.3, there is a sharp increase in execution time at halo depths beyond eight for both the serial and overlap versions of the prototype when using the asynchronous halo exchange function. For the synchronous halo exchange on this grid size, the serial version is shown to be slowly increasing in performance all the way. In contrast, the overlap version performs best at halo depths eight and 16 before it slowly starts climbing. The synchronous overlapping version at halo depth eight was the fastest at 6.40 seconds, with a 1.09 speedup on the entire execution against the asynchronous serial version at halo depth one, which was the slowest.

Looking at the execution times for the A100 nodes, there is a steady decrease in time spent when looking at the versions with synchronous halo exchanges until halo depth eight, where similarly to the V100 nodes, the execution time tends to flatten out or increase.

In Figure 5.4, some aberrant results from the asynchronous versions are shown. While the synchronous version of the code has a steady decrease, the asynchronous versions slow down significantly at halo depth four and come back down at halo depth 32.

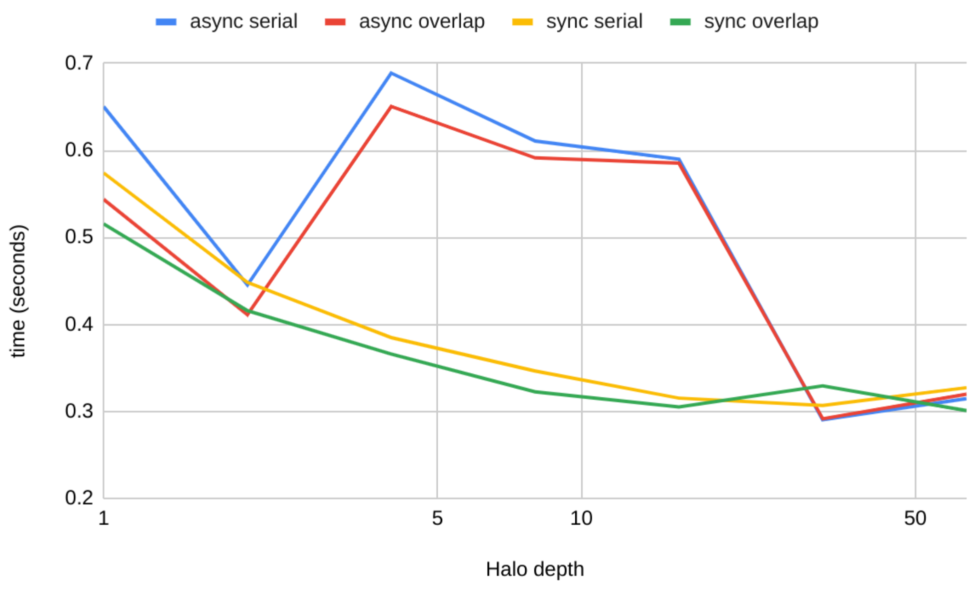

Figure 5.4: Execution times for side lengths of 5000 × 5000 on A100 nodes

Figure 5.5 shows the execution times on the 15000×15000 grid. We see a more expected result here than in Figure 5.4, with a more steady decrease in execution time as the halo depth decreases until the halo depth reaches eight. Still, however, the asynchronous versions slow down after halo depth one and come down again at halo depth eight.

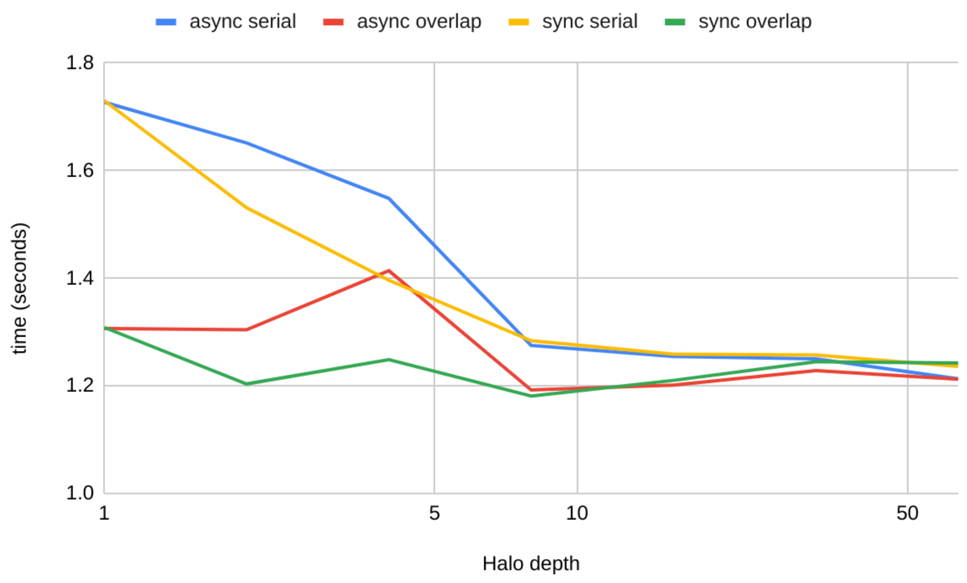

Figure 5.5: Execution times for side lengths of 15000 × 15000 on A100 nodes

In Figure 5.6, the execution times on the largest tested grid size on the A100 nodes are shown. Again the synchronous overlapping version performs the best here at halo depth eight with a 1.1736 speedup against the synchronous serial version at depth one. At this grid size, the asynchronous versions slow down at halo depths two and 32. The slowdown for the async overlap version is much less prominent at halo depth two than at halo depth 32 because the inner compute makes up a much more prominent fraction of the total computation time and can thus hide more of the increased message time.

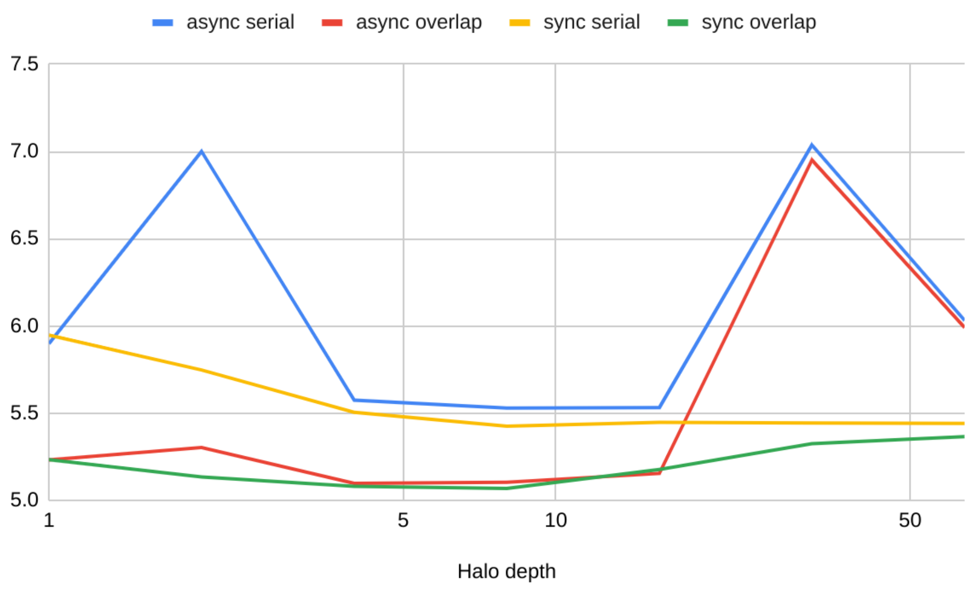

Figure 5.6: Execution times for side lengths of 35000 × 35000 on A100 nodes

5.3 Message time

Here the time results for the message passing stage are shown and discussed. This measurement only takes the time of the functions shown in Section 3.3.

The synchronous version of the halo exchange shown in Figure 5.7 shows a fairly consistent pattern across the different grid sizes. Where more time is spent sending messages with lower halo depths than with higher depths, most of the gains in halo exchange time happen up until halo depth eight. For example, with grid size 35000×35000, the synchronous halo exchange takes 0.287 seconds with a halo depth of one and 0.170 seconds with a halo depth of eight. That is a speedup of 1.680. For halo depths greater than eight, there is no gain in performance at that grid size.

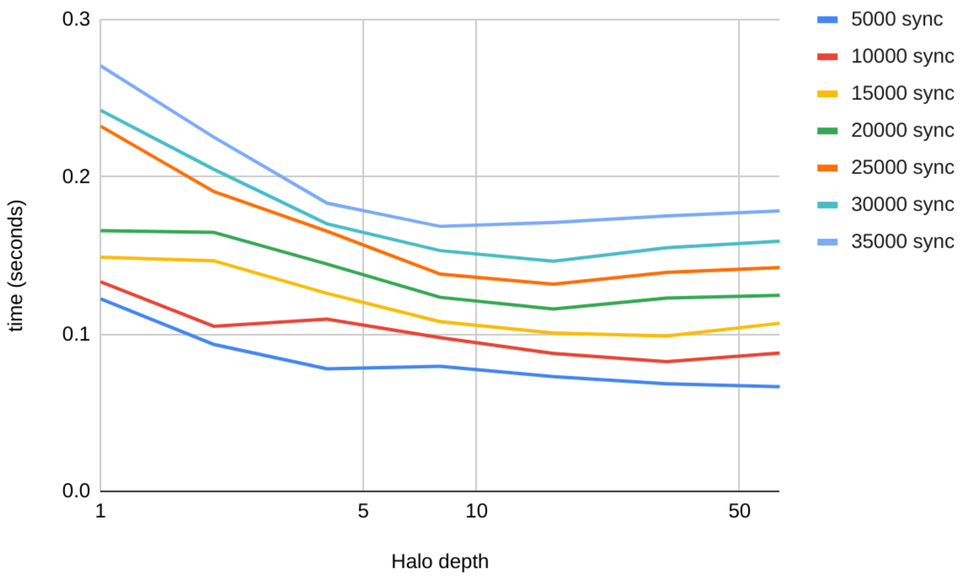

Figure 5.7: Message times for the synchronous runs on V100 nodes

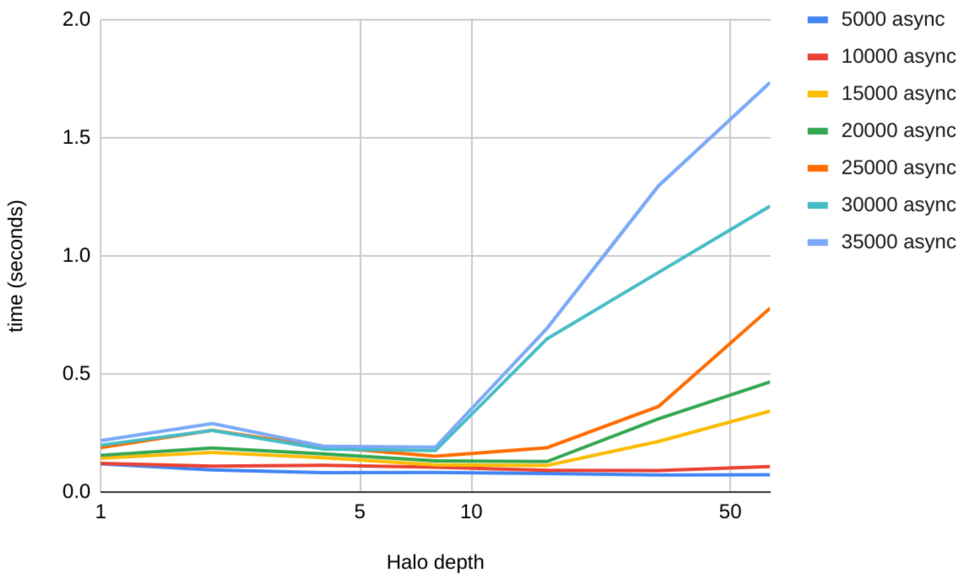

Figure 5.8: Message times for the asynchronous runs on V100 nodes

With the asynchronous version on the V100 nodes shown in Figure 5.8, the only significant performance gains for the message passing stage happened with small grid sizes. For the runs on the 5000×5000 grid, the asynchronous function takes 0.119 seconds at halo depth one and 0.072 seconds at halo depth 64. That is a speedup of 1.639. For grid size 10000×10000, there is a speedup of 1.333

between halo depths one and 32. We see a significant slowdown at higher halo depths as the grid sizes get larger. This slowdown is not at all present in the synchronous runs for any grid size. This slowdown is not present in the A100 runs, and since the GPU doesn't impact the halo exchange function, it shows that the CPU side hardware on the V100 nodes doesn't handle the eight simultaneous messages in the asynchronous function the same way as the CPU side hardware does on the A100 nodes.

The times for the halo exchange on the A100 nodes are shown in Figure 5.9 and Figure 5.10. Generally, We see a speedup for the synchronous versions as the halo depth increases.

When looking at the largest grid size, there is a halving in time spent, with a 2.01 speedup from halo depth one to 64 on the 35000×35000 grid for the A100 nodes. We know from the overall execution times that even though the message time is lowest at depth 64, the speedup at halo depth eight and 16 are more likely to be interesting since the overlap version often reaches its best performance. At halo depth eight, there is a speedup of 1.78 compared to halo depth one on the synchronous version.

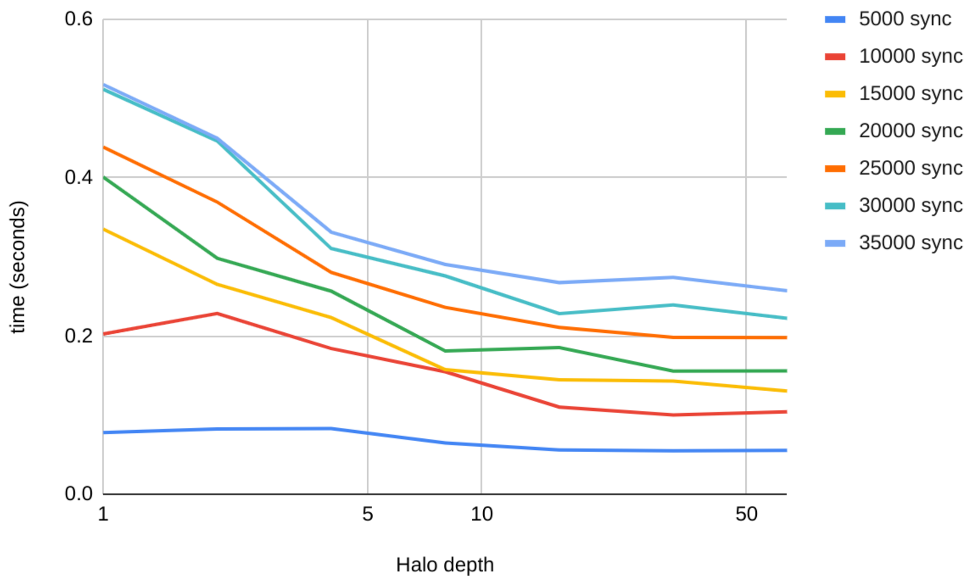

Figure 5.9: Message times for the synchronous runs on A100 nodes

The asynchronous times on the A100 nodes are shown in Figure 5.10. Here it is shown that the hardware consistently struggles at halo depth two. Especially with the larger grid sizes we see a significant slowdown. While the largest grid size has a substantial increase in message time at halo depths 32 and 64, that consistent increase in message time for the asynchronous function that was present on the V100 nodes is not present on the A100 nodes.

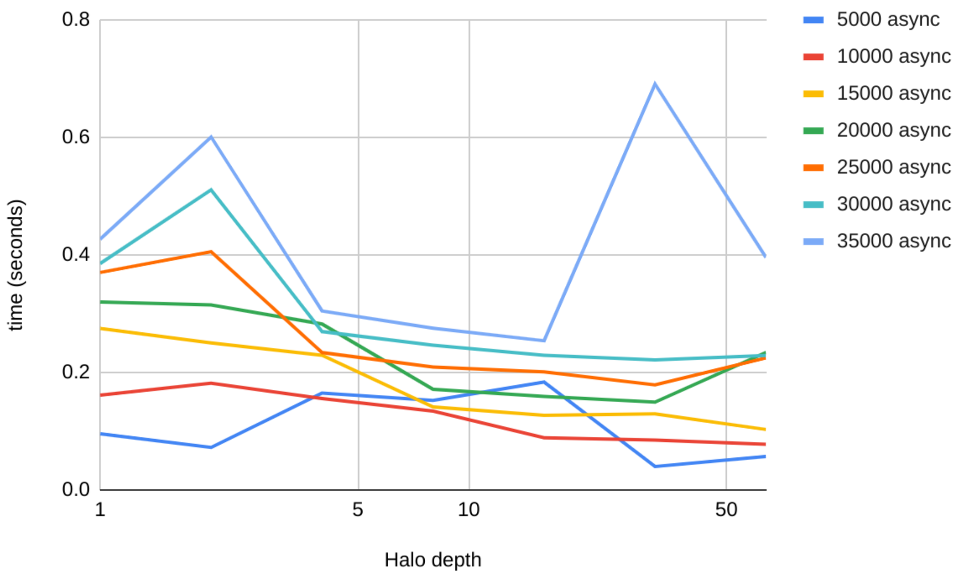

Figure 5.10: Message times for the asynchronous runs on A100 nodes

The synchronous version of the halo exchange tends to outperform the asynchronous version, especially after halo depth one where the message time decreases for the synchronous version but not always for the asynchronous version. The clear takeaway, however, is that the asynchronous version performs worse in unexpected ways on different hardware. On the A100 nodes, the asynchronous version had a drop in performance at halo depth two across most of the grid sizes that was not present in the synchronous runs. This phenomenon is not present in the V100 runs, but the V100 has a significant slowdown at higher halo depths on the larger grids that is not present in the A100 runs or the synchronous version.

The synchronous version consistently shows a more predictable response to the increasing halo depth in line with the theory proposed in Chapter 2. The theory suggests that when messages are grouped together, the overall time spent decreases due to an overhead reduction with the associated message handling.

So far, the desynchronisation measurements have not been included in the messaging times shown here. But it is still a part of the overall runtime and must be included to determine how well the overlap covers the communication. The desynchronisation times for the 35000 × 35000 runs on both the A100 and V100 nodes are shown in Figure 5.11. The async versions show the same patterns as the message times for the same data points. We recognise the spikes at halo depth two and 32 for the A100 runs, and the increase after halo depth eight for the V100 runs. This suggests that the slowdowns for the asynchronous versions of the halo exchange are not even across the nodes, and some nodes reach the barrier before the message passing stage much sooner and have to wait there.

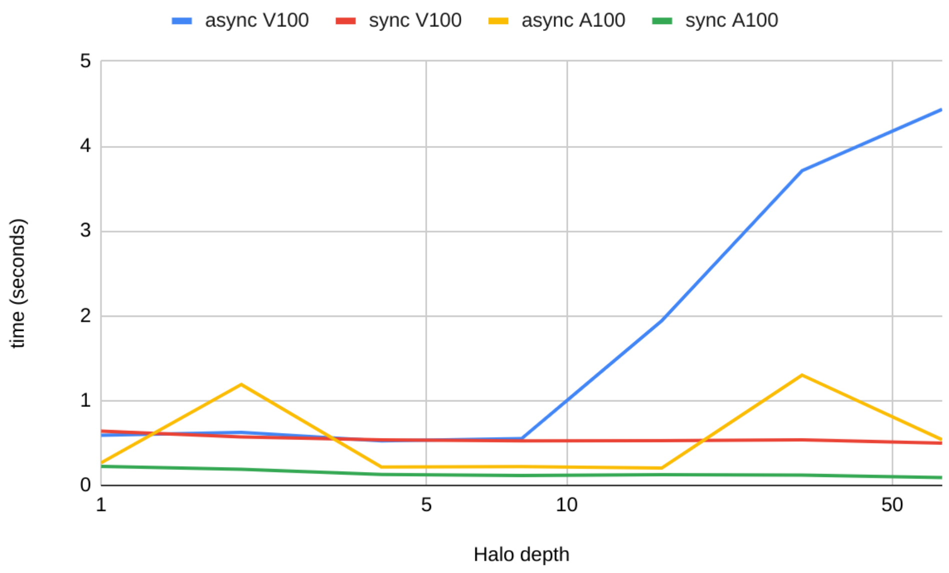

Figure 5.11: Desynchronisation times for 35000 × 35000 on A100 and V100 nodes

The overlapping version of the code covers a significant amount of the halo exchange communication with computations. Beyond halo depth eight, most of the overlap measurements begin to show slower execution times. This is because the shift from halo depth eight to 16 is when the inner compute time becomes less than the communication time, which means that the computations can no longer cover the message time.

5.4 Packing and Unpacking time

Here the results of timing the border fetching and border placing operations are described. Since both the kernels and transfers are timed together, we refer to the whole operation of fetching and placing as packing and unpacking respectively.

Since the packing operation happens at the start of the iterative loop and before the beginning of the inner compute on the overlap versions, it is entirely independent of both overlap and the nature of the halo exchange. Because of this, the results from all four versions are combined to look at the impact of increasing the halo depth across the different grid sizes in the same graph.

The unpacking operation will often happen simultaneously as the inner compute in the overlapping communication versions of the prototype. Therefore, the graphs showing the time measurements for this stage combine the asynchronous and synchronous communication versions but not the overlap and serial versions.

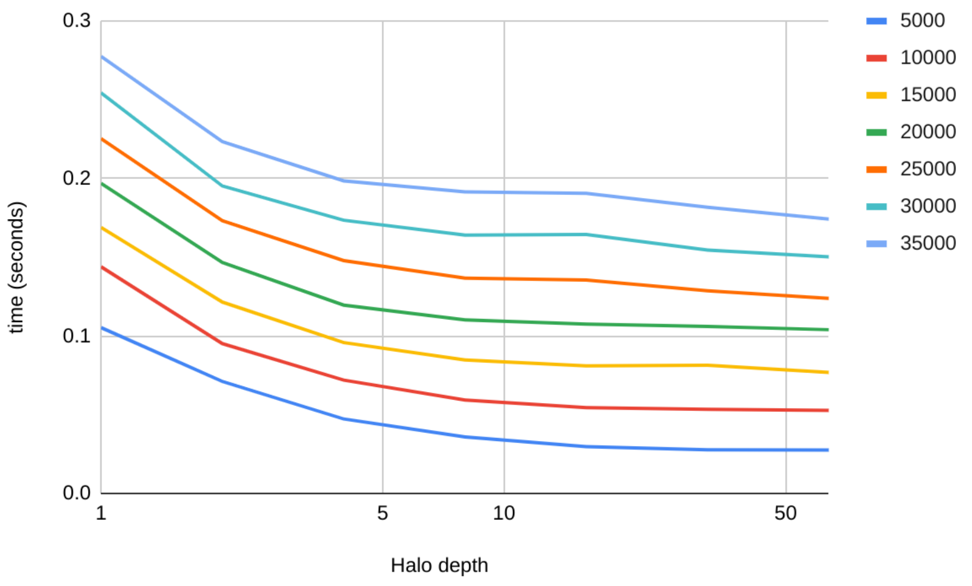

Figure 5.12: Packaging time on the V100 nodes for all grid sizes

Figure 5.12 shows the packing time for all grid sizes on the V100. Across all grid sizes, there is a speedup as the halo depth increases up to eight. For example, at grid size 35000 × 35000, there is a speedup of 1.47 going from halo depth one to eight. Beyond that, the performance flattens out or, in a few cases, has a slight slowdown. The slowdowns are not particularly significant. For grid size 5000 × 5000, the runs at halo depth 32 had a packing time of 0.0281 seconds, while the time at halo depth 64 was 0.0285 seconds, a 0.1% increase in time spent.

The times for the unpacking operation on V100 are shown in Figure 5.13 and Figure 5.14. As the figures show, the overlap versions of the code spend significantly more time unpacking at low halo depths. This is because the unpacking operations share resources with the inner compute. The dramatic reduction in time taken for those versions is then because as the halo depth increases, the unpacking operation spends less time waiting for the resources used by the inner compute operation.

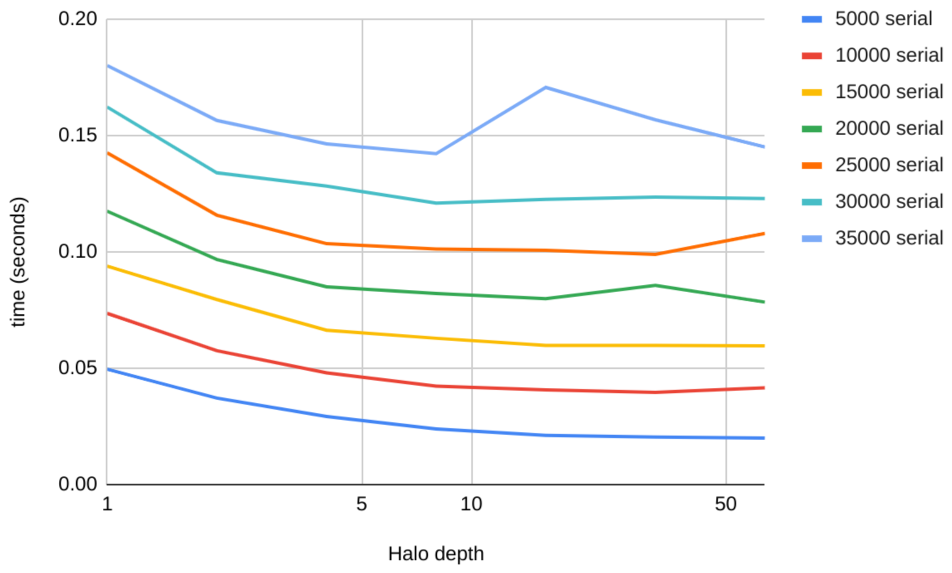

Figure 5.13: Unpacking time on the V100 nodes for all the serial runs

In the serial versions, there is a decrease in unpacking time as the halo depth increases, more similar to the packaging time. For example, on the V100 nodes, the runs on grid size 30000×30000 have a speedup between halo depths one and eight of 1.34.

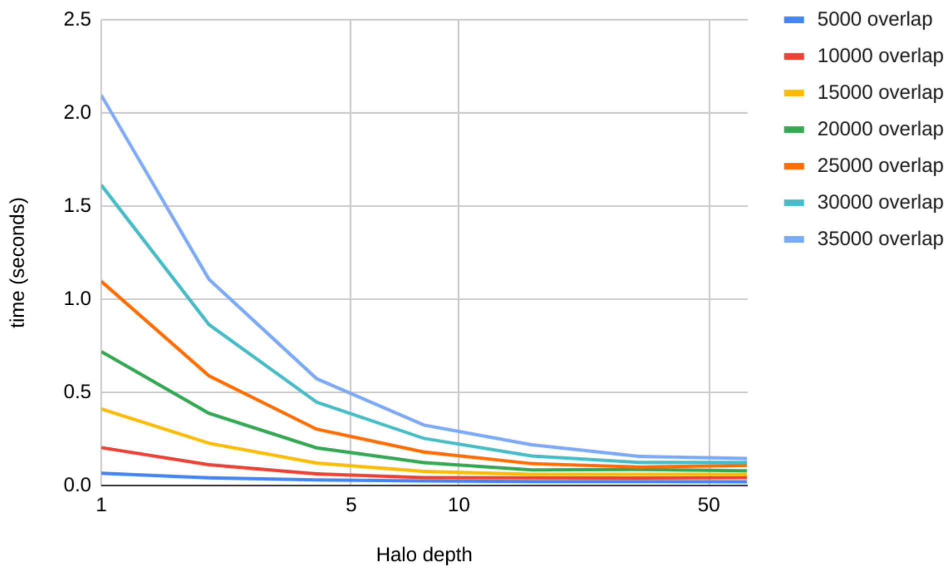

Figure 5.14: Unpacking time on the V100 nodes for all the overlap runs

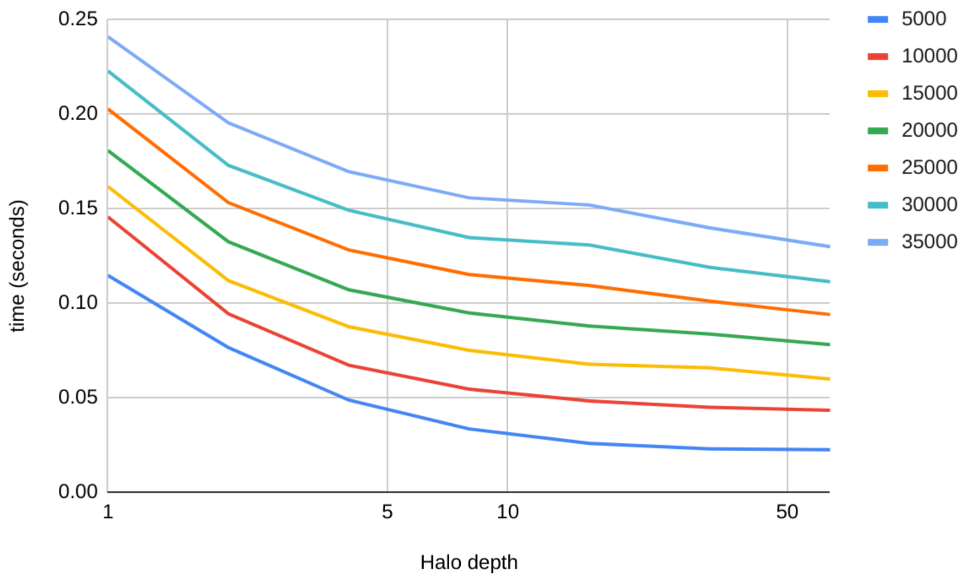

Figure 5.15: Packaging time on the A100 nodes for all grid sizes

In Figure 5.15, the timing results for the packing operation on the A100 nodes are shown. There is a decrease in time spent packing throughout the entire data set. Again, this speedup is entirely a performance gain since it is run in serial with all the other parts of the computation. In the V100 runs, the speedup was the most significant up to halo depth eight, which is not entirely dissimilar to the results from the runs on the A100 nodes. However, there are no speed decreases as the halo size increases on the A100 for any grid size. Also, for the A100, the speedup past halo depth eight is still generally higher than for the V100. For example, on the V100, the speedup on the 35000 × 35000 grid when going from halo depth eight to 64 was 1.099, while on the A100, it was 1.199.

For the serial unpacking operation, shown in Figure 5.16, the A100 results show a very similar pattern to the one on the V100, with the caveat of the operation being overall faster on the newer hardware. The overlap version is not shown in a figure here, it has the same decrease where the time spent is halved every time the halo depth doubles. But overall runs faster on the A100 than the V100.

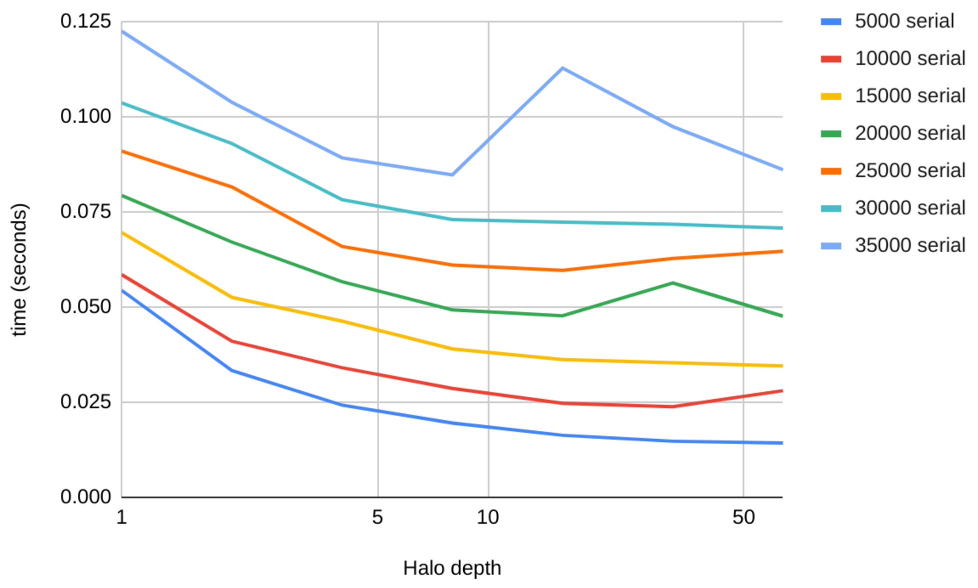

Figure 5.16: Unpacking time on the A100 nodes for all grid sizes on serial runs

These results of the benchmark show that as the halo depth increases, the time spent packing data and transporting data between the CPU and GPU is significantly decreased. Because the packing operation is always performed in serial, these performance gains will always impact the overall execution time.

5.5 Overlapping

Table 5.1 and Table 5.2 show how much of the message + desynchronisation + unpack time is covered on the V100 and A100 for the synchronous run on the 35000 × 35000 grid size. These are the times that are relevant to the overlap as they are the operations running alongside the inner compute. The increase in computation time associated with higher halo depths is implicitly included as it decreases the difference in execution times. We see that on the A100 nodes at depth one, 83.3% of the communication time is covered by the overlap. However, this coverage decreases as halo depth increases to 75.1% at halo depth eight, which, as discussed earlier, is the most performant setting. At halo depths beyond this, the share of the communication being covered is significantly reduced as the inner compute takes less time than the communication time. Between halo depths one and eight.

The V100 runs at this grid size have less coverage than the ones for the A100 nodes, with a 31.6% communication coverage on the most performant halo depth.

The amount of communication able to be covered by the overlap heavily depends on the grid size. For example, in the 10000 × 10000 grid on the A100, there is 63.9% coverage at halo depth one and 21.0% at depth eight.

Table 5.1: Table showing how much of the communication is covered by the overlapping synchronous version on the V100 nodes on the 35000 × 35000 grid.

Depth	Message + Desync time	Execution difference	% Decrease
1	1.0852	0.5224	48.1397
2	0.9532	0.3182	33.3786
4	0.8687	0.3062	35.2514
8	0.8422	0.2662	31.6126
16	0.8767	0.2957	33.7302
32	0.8770	0.2700	30.7901
64	0.8254	0.1204	14.5914

Table 5.2: Table showing how much of the communication is covered by the overlapping synchronous version on the A100 nodes on the 35000 × 35000 grid.

Depth	Communication time	Execution difference	% Decrease
1	0.8578	0.7146	83.3034
2	0.7389	0.6130	82.9539
4	0.5363	0.4247	79.1934
8	0.4760	0.3574	75.0933
16	0.4976	0.2707	54.3979
32	0.4828	0.1190	24.6493
64	0.4366	0.0749	17.1672

Chapter 6

Conclusions & Future Work

Stencils are a family of algorithms that update points of a multi-dimensional data mesh with the neighbouring values as inputs. Given how compute-intensive the stencil computations are they beg for better benchmarking tools, especially on GPUs that are becoming central accelerators for High-Performance Computing workloads.

In this thesis, we described our benchmarking tool for doing stencil computations on the GPU with halo exchanges using MPI. Three different optimisations were presented and tested: synchronous halo exchanges, deep halos, and overlapping communication and computation. The ways the different optimisations impacted the different parts of the program were measured and analysed for several relevant problem sizes ranging from 5000×5000 to 35000×35000, and benchmarked on two different popular GPU HPC architectures, the NVIDIA V100 and A100.

Because the speedups that were achieved were so dependent on the grid size, even when only looking at the communication parts of the results, it is challenging to answer the research questions laid out in the introduction with specific numerical performance gains. However, we did observe that these optimisations do work in general and give significant performance increases for the halo exchanges. For the largest problem size tested, the optimisations gave a 1.09 speedup when using nodes with V100 GPUs and a 1.17 speedup when using nodes with A100 GPUs.

We showed that deepening halos reduce the amount of communication overhead by batching communication together in exchange for an increase in the number of computations needed for each iteration. That reduction was shown for both communication between the GPU and CPU, and between different nodes in a compute cluster. These results showed that deepening the halos on their own, continue to provide increased performance as halo size increases until the decrease in communication time is eclipsed by the increase in computation time.

Note that deep halos are the only optimisation that affects the device to host border transfer, where increasing the halo depth to the 64 would decrease the time spent on the fetching operation by between 0.08 and 0.10 seconds on the V100 GPUs and between 0.09 and 0.11 seconds on the A100 GPUs. The biggest performance gains, in terms of wall-time, were for the biggest problem sizes.

Synchronised halo exchanges provided either slightly better performance or vastly better performance than the asynchronous version used in our implementation. The synchronous versions perform predictably. However, there are times when the asynchronous version unpredictably performs much worse to the point where it spends more time communicating than calculating. This may make it unreasonable to use it over the synchronous version in a practical use case.

Overlapping communication and calculations was shown to be a very impactful optimisation. It provided better performance than the serial version in almost every instance, and especially so on larger grid sizes. On the largest grid size with the newest hardware, the overlapping version was able to cover 83.3% of the communication. It was also shown that maximising coverage is not always going to give the best performance.

Our original idea for implementing the overlapping optimisation was to use POSIX threads to start the kernel in a separate thread. However, during development it was discovered that it was simpler and more efficient to not use multithreading on the CPU and just make use of the fact that CUDA kernel launches are asynchronous by default to achieve the overlap.

The synchronous halo exchange doesn't interact with the other optimisations in particularly interesting ways outside of it working as expected. The asynchronous version showed that at low halo depths, overlapping communication and calculation can hide unexpected slowdowns in communication between nodes in the cluster.

Deep halos provided the most speedup with the synchronous version of the halo exchange, a decrease which kept going for all halo sizes albeit with diminishing returns. Overlapping communication and calculations meant that the point where increasing halo depth was no longer when the increase in computation outpaced the decrease in communication time, but rather when the inner compute took less time than the communication. This means that the ideal halo depth for the overlap versions is lower than for the version where communications and computation is serialised.

6.1 Future Work

This thesis has focused primarily on the halo exchange part of 2D stencil computations. Therefore there are many things related to stencil computations that were not examined. Following are several suggestions.

Optimizing computations

Note that the computation part in itself was not explored in this work. There are implementations of stencil computations that should make better use of the warp-level and shared memory when performing the stencil computation. It would thus be interesting to explore a less naive implementation of the `calculate` kernel.

An implementation that uses shared memory to share data between warps could also change the efficacy of splitting the computation into an inner and outer compute.

Explor CUDA memory operations

There are memory operations in CUDA that were not explored here. For example pinned memory, which pins a region of host memory so that the GPU can access it directly [18], could be interesting to look at.

Extending the benchmark to 3D

A fairly straightforward expansion is extending the benchmark to 3D. There are many natural phenomena that cannot be modelled using only two dimensions, and problem domains of three dimensions are necessary. Going from 2D to 3D changes the relationship between the amounts of inner and outer points and how they are stored in memory, which could give different results.

Running on more nodes and devices

We scheduled jobs on Idun to run with more than four nodes, but those jobs were not run in time to be included here because the relevant hardware was in high demand during the semester in which this thesis was written. So all our results are from runs with four total nodes. This means that for the asynchronous halo exchange, four of the MPI messages were sent to the same neighbour, which could have changed the results to something different than what they would have been with eight unique neighbours. So it could be interesting to see how the benchmark scales to even bigger problem sizes on more nodes.

This benchmark was run with each node on the cluster running one MPI process using one GPU. Often nodes will have access to multiple GPUs. Extending the work to work on multiple devices per node would allow for benchmarking the impact these optimisations have on device-to-device data transfers, such as the "colocated" communication described in [9].

Bibliography

[1] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, Vienna Austria: ACM, Feb. 24, 2018, pp. 100–112, ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168824. [Online]. Available: https://dl.acm.org/doi/10.1145/3168824 (visited on 05/24/2023).

[2] L. Zhang, M. Wahib, P. Chen, J. Meng, X. Wang, T. Endo, and S. Matsuoka, *Revisiting temporal blocking stencil optimizations*, May 12, 2023. DOI: 10.1145/3577193.3593716. arXiv: 2305.07390[cs]. [Online]. Available: http://arxiv.org/abs/2305.07390 (visited on 05/24/2023).

[3] W. Elwasif, W. Godoy, N. Hagerty, J. A. Harris, O. Hernandez, B. Joo, P. Kent, D. Lebrun-Grandie, E. Maccarthy, V. G. M. Vergara, B. Messer, R. Miller, S. Opal, S. Bastrakov, M. Bussmann, A. Debus, K. Steinger, J. Stephan, R. Widera, S. H. Bryngelson, H. L. Berre, A. Radhakrishnan, J. Young, S. Chandrasekaran, F. Ciorba, O. Simsek, K. C. F. Spiga, J. Hammond, J. E. S. D. Hardy, S. Keller, and J.-G. P. C. Trott, *Application experiences on a gpu-accelerated arm-based hpc testbed*, 2022. DOI: 10.48550/ARXIV.2209.09731. [Online]. Available: https://arxiv.org/abs/2209.09731.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: 10.1109/JPROC.2008.917757.

[5] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters," 2010.

[6] R. Thakur, W. Gropp, and E. Lusk, *Optimizing noncontiguous accesses in MPI-IO*, Oct. 15, 2003. arXiv: cs/0310029. [Online]. Available: http://arxiv.org/abs/cs/0310029 (visited on 05/22/2023).

[7] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in CUDA unified memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Nov. 2019, pp. 50–57. DOI: 10.1109/MCHPC49590.2019.00014.

[8] H. Li, D. Yu, A. Kumar, and Y.-C. Tu, "Performance modeling in CUDA streams - a means for high-throughput data processing," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct. 2014, pp. 301–310. DOI: 10.1109/BigData.2014.7004245.

[9] C. Pearson, "Movement and placement of non-contiguous data in distributed gpu computing," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2021.

[10] A. Hammer, "Analyzing halo computations on multicore CPUs," Accepted: 2022-02-24T18:19:25Z, Master thesis, NTNU, 2021. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2981316 (visited on 05/24/2023).

[11] "Finite difference methods," in *Numerical Treatment of Partial Differential Equations: Translated and revised by Martin Stynes*, ser. Universitext, C. Grossmann, H.-G. Roos, and M. Stynes, Eds., Berlin, Heidelberg: Springer, 2007, pp. 23–124, ISBN: 978-3-540-71584-9. DOI: 10.1007/978-3-540-71584-9_2. [Online]. Available: https://doi.org/10.1007/978-3-540-71584-9_2 (visited on 05/29/2023).

[12] H. Hotta, M. Rempel, and T. Yokoyama, "High-resolution calculations of the solar global convection with the reduced speed of sound tech- nique. i. the structure of the convection and the magnetic field without the rotation," *The Astrophysical Journal*, 2014.

[13] A. Beresnyak, "Spectra of strong magnetohydrodynamic turbulence from high-resolution simulations," *The Astrophysical Journal Letters*, vol. 784, no. 2, p. L20, Mar. 2014. DOI: 10.1088/2041-8205/784/2/L20. [Online]. Available: https://dx.doi.org/10.1088/2041-8205/784/2/L20.

[14] J. Pekkilä, M. S. Väisälä, M. J. Käpylä, P. J. Käpylä, and O. Anjum, "Methods for compressible fluid simulation on gpus using high-order finite differences," 2017.

[15] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356162. [Online]. Available: https://doi.org/10.1145/3295500.3356162.

[16] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLoP '10, New York, NY, USA: Association for Computing Machinery, Mar. 30, 2010, pp. 1–9, ISBN: 978-1-4503-0127-5. DOI: 10.1145/1953611.1953615. [Online]. Available: https://doi.org/10.1145/1953611.1953615 (visited on 03/02/2023).

[17] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: 1912.05848 [cs.DC].

[18] A. Nukada, T. Suzuki, and S. Matsuoka, “Efficient checkpoint/restart of CUDA applications,” *Parallel Computing*, vol. 116, p. 103 018, Jul. 2023, ISSN: 01678191. DOI: 10.1016/j.parco.2023.103018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167819123000248 (visited on 06/11/2023).

Appendix A

Running guide

To run this code you need to have an MPI implementation installed and a CUDA toolkit installed on a machine with a CUDA-capable device.

To compile every version, run the `./compile.sh` script from the root folder

To compile only one version, simply enter the folder for that version and run `make`

Once the code is compiled, the code can be executed with 4 MPI processes by running the command

```
mpirun -np 4 output1
```

Or for a 20000*20000 grid with 100 iterations and depth of eight, with sparse output

```
mpirun -np 4 output1 -h 20000 -l 20000 -i 100 -d 8 -s 1
```

To run all the versions in succession, the `runner.py` script can be executed from the root folder.

Appendix B

Code snippets

```
for (int i = 0;i<i1;){
    fetchBorders(leftBorder, rightBorder, northBorder, southBorder,
        adjustedHeight, adjustedLength, depth, mpiBorders,
        horizontalMpiBorders);
    threadedSegmentStart = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD);
    threadedSegmentSum += MPI_Wtime() - threadedSegmentStart;
    messageStart = MPI_Wtime();
    syncronousMpiBorderExchange(leftBorder, rightBorder, leftBorderReceive,
        rightBorderReceive, northBorder, southBorder,
        northBorderReceive, southBorderReceive,
    neighbours, myHeight, depth, adjustedLength);
    messagePassing += MPI_Wtime() - messageStart;
    placeBorders(leftBorderReceive, rightBorderReceive, northBorderReceive,
        southBorderReceive, adjustedHeight, adjustedLength, depth,
        mpiBorders, horizontalMpiBorders);
    for(int j = 0; j<depth && i<i1;j++){
        calcStart = MPI_Wtime();
        launchKernel(problemDomain);
        calcSum += MPI_Wtime() - calcStart;
        i++;
    }
}
```

Listing 11: Iterative loop without overlap and synchronous halo exchange

```
void syncronousMpiBorderExchange
(float *leftBorder, float *rightBorder, float *leftBorderReceive,
float *rightBorderReceive, float *northBorder, float *southBorder,
float *northBorderReceive, float *southBorderReceive, struct neighbours
neighbours, int myHeight, int depth, int adjustedLength){
MPI_Status status1, status2;
MPI_Request sendRequest1, sendRequest2, recvRequest1, recvRequest2;
MPI_Irecv(rightBorderReceive, sizeof(float) * myHeight * depth,
    MPI_BYTE, neighbours.east, 0, MPI_COMM_WORLD, &recvRequest1);
MPI_Irecv(leftBorderReceive, sizeof(float) * myHeight * depth,
    MPI_BYTE, neighbours.west, 1, MPI_COMM_WORLD, &recvRequest2);
MPI_Isend(leftBorder, sizeof(float) * myHeight * depth,
    MPI_BYTE, neighbours.west, 0, MPI_COMM_WORLD, &sendRequest1);
MPI_Isend(rightBorder, sizeof(float) * myHeight * depth,
    MPI_BYTE, neighbours.east, 1, MPI_COMM_WORLD, &sendRequest2);
MPI_Wait(&recvRequest1, &status1);
MPI_Wait(&recvRequest2, &status2);
for(int id = 0; id < depth; id++){
    for(int jd = 0; jd < depth; jd++){
        northBorder[jd + id * adjustedLength]
            = leftBorderReceive[jd + id * depth];}}
for(int id = 0; id < depth; id++){
    for(int jd = 0; jd < depth; jd++){
        northBorder[jd + id * adjustedLength + adjustedLength - depth]
            = rightBorderReceive[jd + id * depth];}}
for(int id = 0; id < depth; id++){
    for(int jd = 0; jd < depth; jd++){
        southBorder[jd + id * adjustedLength]
            = leftBorderReceive[jd+id*depth+depth*myHeight-depth*depth];}}
for(int id = 0; id < depth; id++){
    for(int jd = 0; jd < depth; jd++){
        southBorder[jd + id * adjustedLength + adjustedLength - depth]
            = rightBorderReceive[jd+id*depth+depth*myHeight-depth*depth];}}
MPI_Irecv(northBorderReceive, sizeof(float) * adjustedLength * depth,
    MPI_BYTE, neighbours.north, 3, MPI_COMM_WORLD, &recvRequest1);
MPI_Irecv(southBorderReceive, sizeof(float) * adjustedLength * depth,
    MPI_BYTE, neighbours.south, 2, MPI_COMM_WORLD, &recvRequest2);
MPI_Isend(northBorder, sizeof(float) * adjustedLength * depth,
    MPI_BYTE, neighbours.north, 2, MPI_COMM_WORLD, &sendRequest1);
MPI_Isend(southBorder, sizeof(float) * adjustedLength * depth,
    MPI_BYTE, neighbours.south, 3, MPI_COMM_WORLD, &sendRequest2);
MPI_Wait(&recvRequest1, &status1);
MPI_Wait(&recvRequest2, &status2);  }
```

Listing 12: Synchronous halo exchange function

```
    void asynchronousMpiBorderExchange
        (float *leftBorder, float *rightBorder, float *leftBorderReceive,
        float *rightBorderReceive, float *northBorder, float *southBorder,
        float *northBorderReceive, float *southBorderReceive,
        struct neighbours neighbours, int myHeight, int myLength,
        int depth, int adjustedLength, MPI_Datatype MPI_corner,
        MPI_Datatype MPI_center_horizontal, MPI_Request *sendRequests,
        MPI_Request *recvRequests, MPI_Status *statuses){
    MPI_Irecv(leftBorderReceive, sizeof(float) * myHeight * depth,
        MPI_BYTE, neighbours.west, 1, MPI_COMM_WORLD, &recvRequests[0]);
    MPI_Irecv(rightBorderReceive, sizeof(float) * myHeight * depth,
        MPI_BYTE, neighbours.east, 0, MPI_COMM_WORLD, &recvRequests[1]);
    MPI_Irecv(&northBorderReceive[depth], 1, MPI_center_horizontal,
        neighbours.north, 3, MPI_COMM_WORLD, &recvRequests[2]);
    MPI_Irecv(&southBorderReceive[depth], 1, MPI_center_horizontal,
        neighbours.south, 2, MPI_COMM_WORLD, &recvRequests[3]);
    MPI_Irecv(northBorderReceive, 1, MPI_corner, neighbours.northWest,
        7, MPI_COMM_WORLD, &recvRequests[4]);
    MPI_Irecv(&northBorderReceive[adjustedLength-depth], 1, MPI_corner,
        neighbours.northEast, 6, MPI_COMM_WORLD, &recvRequests[5]);
    MPI_Irecv(southBorderReceive, 1, MPI_corner, neighbours.southWest,
        5, MPI_COMM_WORLD, &recvRequests[6]);
    MPI_Irecv(&southBorderReceive[adjustedLength-depth], 1, MPI_corner,
        neighbours.southEast, 4, MPI_COMM_WORLD, &recvRequests[7]);
    MPI_Isend(leftBorder, sizeof(float) * myHeight * depth,
        MPI_BYTE, neighbours.west, 0, MPI_COMM_WORLD, &sendRequests[0]);
    MPI_Isend(rightBorder, sizeof(float) * myHeight * depth,
        MPI_BYTE, neighbours.east, 1, MPI_COMM_WORLD, &sendRequests[1]);
    MPI_Isend(&northBorder[depth], 1, MPI_center_horizontal,
        neighbours.north, 2, MPI_COMM_WORLD, &sendRequests[2]);
    MPI_Isend(&southBorder[depth], 1, MPI_center_horizontal,
        neighbours.south, 3, MPI_COMM_WORLD, &sendRequests[3]);
    MPI_Isend(leftBorder, sizeof(float) * depth * depth, MPI_BYTE,
        neighbours.northWest, 4, MPI_COMM_WORLD, &sendRequests[4]);
    MPI_Isend(rightBorder, sizeof(float) * depth * depth, MPI_BYTE,
        neighbours.northEast, 5, MPI_COMM_WORLD, &sendRequests[5]);
    MPI_Isend(&leftBorder[myHeight*depth-depth*depth], sizeof(float)*depth*depth,
        MPI_BYTE, neighbours.southWest, 6, MPI_COMM_WORLD, &sendRequests[6]);
    MPI_Isend(&rightBorder[myHeight*depth-depth*depth], sizeof(float)*depth*depth,
        MPI_BYTE, neighbours.southEast, 7, MPI_COMM_WORLD, &sendRequests[7]);
    MPI_Waitall(8, recvRequests, statuses);
}
```

Listing 13: Asynchronous halo exchange function

Appendix C

All benchmark results

All results from the benchmark are here. Abbreviated column titles are Dp. for Depth, Ex. for Execution time, Pack for Packing time, Unpack for Unpacking time, I. Cal. for Inner calculation, O. Cal. for Outer calculation, P. Cal. for Primary calculation, Mess. for Message time, and Desyn. for Desynchronization time.

C.1 A100 results

Table C.1: 5000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.6504	0.1357	0.0546	0.0000	0.0000	0.1337	0.0906	0.1162
2	0.4452	0.0763	0.0332	0.0000	0.0000	0.1369	0.0713	0.0595
4	0.6889	0.0491	0.0245	0.0000	0.0000	0.1515	0.1667	0.2489
8	0.6108	0.0334	0.0197	0.0000	0.0000	0.1538	0.1414	0.2206
16	0.5900	0.0257	0.0164	0.0000	0.0000	0.1555	0.1808	0.1768
32	0.2903	0.0226	0.0148	0.0000	0.0000	0.1502	0.0396	0.0321
64	0.3146	0.0224	0.0143	0.0000	0.0000	0.1570	0.0547	0.0366

Table C.2: 5000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.5437	0.1295	0.0570	0.1323	0.0305	0.0000	0.1006	0.0911
2	0.4111	0.0756	0.0349	0.0695	0.0160	0.0681	0.0736	0.0600
4	0.6506	0.0490	0.0251	0.0378	0.0090	0.1138	0.1629	0.2357
8	0.5914	0.0334	0.0198	0.0190	0.0046	0.1339	0.1633	0.1941
16	0.5853	0.0256	0.0162	0.0096	0.0023	0.1463	0.1866	0.1747
32	0.2915	0.0226	0.0148	0.0045	0.0012	0.1463	0.0402	0.0371
64	0.3198	0.0224	0.0143	0.0022	0.0007	0.1546	0.0592	0.0414

Table C.3: 5000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.5741	0.1347	0.0542	0.0000	0.0000	0.1324	0.0772	0.0680
2	0.4483	0.0769	0.0334	0.0000	0.0000	0.1372	0.0825	0.0510
4	0.3848	0.0483	0.0240	0.0000	0.0000	0.1409	0.0817	0.0434
8	0.3464	0.0334	0.0193	0.0000	0.0000	0.1433	0.0690	0.0406
16	0.3152	0.0258	0.0162	0.0000	0.0000	0.1459	0.0546	0.0387
32	0.3068	0.0232	0.0147	0.0000	0.0000	0.1499	0.0508	0.0371
64	0.3272	0.0223	0.0143	0.0000	0.0000	0.1569	0.0629	0.0411

Table C.4: 5000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.5155	0.1300	0.0584	0.1323	0.0306	0.0000	0.0785	0.0824
2	0.4157	0.0770	0.0352	0.0701	0.0161	0.0686	0.0821	0.0541
4	0.3658	0.0482	0.0244	0.0352	0.0084	0.1051	0.0843	0.0422
8	0.3225	0.0333	0.0194	0.0178	0.0043	0.1253	0.0605	0.0386
16	0.3050	0.0257	0.0160	0.0090	0.0022	0.1367	0.0573	0.0342
32	0.3293	0.0231	0.0147	0.0045	0.0011	0.1455	0.0589	0.0511
64	0.3008	0.0224	0.0144	0.0022	0.0006	0.1532	0.0478	0.0351

Table C.5: 10000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.0301	0.1465	0.0587	0.0000	0.0000	0.4359	0.1565	0.1273
2	1.0738	0.0950	0.0413	0.0000	0.0000	0.4515	0.1904	0.2307
4	0.9674	0.0676	0.0342	0.0000	0.0000	0.4509	0.1626	0.2030
8	0.8284	0.0550	0.0288	0.0000	0.0000	0.4422	0.1275	0.1367
16	0.6833	0.0487	0.0248	0.0000	0.0000	0.4510	0.0821	0.0447
32	0.6767	0.0455	0.0237	0.0000	0.0000	0.4580	0.0778	0.0426
64	0.6933	0.0440	0.0278	0.0000	0.0000	0.4667	0.0805	0.0467

Table C.6: 10000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.8100	0.1449	0.1205	0.4373	0.0283	0.0000	0.1659	0.2165
2	0.8283	0.0942	0.0492	0.2209	0.0144	0.2207	0.1731	0.1967
4	0.7962	0.0674	0.0351	0.1112	0.0074	0.3336	0.1489	0.1502
8	0.8225	0.0551	0.0290	0.0560	0.0037	0.3926	0.1412	0.1610
16	0.6774	0.0486	0.0250	0.0281	0.0019	0.4241	0.0958	0.0504
32	0.6743	0.0454	0.0236	0.0141	0.0011	0.4427	0.0922	0.0416
64	0.6743	0.0440	0.0279	0.0071	0.0007	0.4612	0.0747	0.0400

Table C.7: 10000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.0805	0.1466	0.0584	0.0000	0.0000	0.4364	0.1880	0.1461
2	0.9791	0.0938	0.0407	0.0000	0.0000	0.4415	0.2351	0.1040
4	0.8533	0.0667	0.0339	0.0000	0.0000	0.4416	0.1843	0.0784
8	0.7874	0.0538	0.0284	0.0000	0.0000	0.4446	0.1621	0.0604
16	0.7042	0.0475	0.0246	0.0000	0.0000	0.4521	0.1065	0.0417
32	0.6951	0.0442	0.0239	0.0000	0.0000	0.4574	0.0976	0.0430
64	0.6945	0.0425	0.0282	0.0000	0.0000	0.4684	0.0903	0.0375

Table C.8: 10000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.8335	0.1442	0.1064	0.4362	0.0283	0.0000	0.2171	0.2040
2	0.8045	0.0940	0.0507	0.2230	0.0145	0.2226	0.2219	0.1209
4	0.7563	0.0667	0.0348	0.1113	0.0074	0.3338	0.1840	0.0762
8	0.7346	0.0538	0.0286	0.0559	0.0037	0.3918	0.1474	0.0695
16	0.6997	0.0476	0.0248	0.0281	0.0019	0.4254	0.1137	0.0548
32	0.6981	0.0441	0.0238	0.0141	0.0011	0.4452	0.1028	0.0534
64	0.7310	0.0424	0.0283	0.0071	0.0007	0.4622	0.1177	0.0539

Table C.9: 15000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.7274	0.1621	0.0697	0.0000	0.0000	0.9044	0.2729	0.2160
2	1.6515	0.1121	0.0527	0.0000	0.0000	0.9096	0.2571	0.2575
4	1.5487	0.0879	0.0466	0.0000	0.0000	0.9096	0.2244	0.2321
8	1.2751	0.0758	0.0392	0.0000	0.0000	0.9172	0.1372	0.0679
16	1.2545	0.0686	0.0362	0.0000	0.0000	0.9211	0.1310	0.0661
32	1.2503	0.0670	0.0351	0.0000	0.0000	0.9290	0.1315	0.0590
64	1.2132	0.0615	0.0339	0.0000	0.0000	0.9387	0.1023	0.0494

Table C.10: 15000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.3066	0.1608	0.2527	0.9080	0.0280	0.0000	0.2768	0.4554
2	1.3042	0.1120	0.1090	0.4562	0.0146	0.4561	0.2435	0.2891
4	1.4140	0.0884	0.0535	0.2284	0.0072	0.6845	0.2342	0.2927
8	1.1924	0.0759	0.0402	0.1147	0.0037	0.8026	0.1451	0.0854
16	1.2016	0.0689	0.0362	0.0573	0.0020	0.8633	0.1229	0.0772
32	1.2285	0.0670	0.0351	0.0288	0.0012	0.9012	0.1276	0.0690
64	1.2125	0.0608	0.0339	0.0144	0.0008	0.9257	0.1036	0.0619

Table C.11: 15000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.7305	0.1625	0.0695	0.0000	0.0000	0.9030	0.3367	0.1566
2	1.5312	0.1118	0.0524	0.0000	0.0000	0.9083	0.2770	0.1196
4	1.3964	0.0869	0.0461	0.0000	0.0000	0.9116	0.2095	0.0944
8	1.2840	0.0742	0.0388	0.0000	0.0000	0.9164	0.1555	0.0617
16	1.2588	0.0667	0.0362	0.0000	0.0000	0.9216	0.1425	0.0606
32	1.2575	0.0644	0.0357	0.0000	0.0000	0.9292	0.1388	0.0610
64	1.2364	0.0584	0.0352	0.0000	0.0000	0.9402	0.1276	0.0479

Table C.12: 15000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.3086	0.1612	0.2711	0.9137	0.0281	0.0000	0.3335	0.3820
2	1.2036	0.1114	0.1220	0.4590	0.0146	0.4585	0.2534	0.1638
4	1.2487	0.0866	0.0535	0.2295	0.0072	0.6880	0.2368	0.1235
8	1.1810	0.0739	0.0393	0.1148	0.0037	0.8037	0.1594	0.0619
16	1.2102	0.0665	0.0364	0.0574	0.0020	0.8654	0.1464	0.0625
32	1.2448	0.0644	0.0357	0.0287	0.0012	0.8982	0.1474	0.0707
64	1.2424	0.0584	0.0352	0.0144	0.0008	0.9278	0.1332	0.0617

Table C.13: 20000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.4848	0.1814	0.0793	0.0000	0.0000	1.5615	0.3191	0.2431
2	2.5562	0.1329	0.0673	0.0000	0.0000	1.5651	0.3301	0.3917
4	2.4425	0.1082	0.0570	0.0000	0.0000	1.5663	0.3032	0.3598
8	2.0215	0.0959	0.0495	0.0000	0.0000	1.5750	0.1757	0.0879
16	2.0066	0.0894	0.0475	0.0000	0.0000	1.5799	0.1667	0.0917
32	1.9736	0.0852	0.0557	0.0000	0.0000	1.5870	0.1459	0.0713
64	2.3402	0.0799	0.0468	0.0000	0.0000	1.6072	0.2433	0.3357

Table C.14: 20000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.9843	0.1793	0.4288	1.5682	0.0287	0.0000	0.3208	0.8941
2	1.9532	0.1326	0.2380	0.7856	0.0145	0.7855	0.2998	0.4010
4	1.9832	0.1080	0.1034	0.3934	0.0075	1.1785	0.2619	0.2705
8	1.8397	0.0955	0.0574	0.1970	0.0037	1.3816	0.1672	0.0950
16	1.9007	0.0889	0.0474	0.0985	0.0020	1.4803	0.1518	0.0992
32	1.9418	0.0852	0.0558	0.0493	0.0013	1.5373	0.1532	0.0816
64	2.2562	0.0799	0.0469	0.0249	0.0009	1.5824	0.2248	0.2957

Table C.15: 20000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.5000	0.1820	0.0793	0.0000	0.0000	1.5625	0.3943	0.1815
2	2.2475	0.1322	0.0668	0.0000	0.0000	1.5661	0.2885	0.1250
4	2.1427	0.1060	0.0563	0.0000	0.0000	1.5703	0.2517	0.1105
8	2.0254	0.0939	0.0490	0.0000	0.0000	1.5752	0.1969	0.0730
16	2.0047	0.0865	0.0479	0.0000	0.0000	1.5814	0.1889	0.0687
32	1.9726	0.0821	0.0570	0.0000	0.0000	1.5885	0.1508	0.0658
64	1.9765	0.0763	0.0484	0.0000	0.0000	1.6080	0.1532	0.0635

Table C.16: 20000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.0136	0.1800	0.4619	1.5697	0.0287	0.0000	0.4075	0.8027
2	1.8844	0.1320	0.2410	0.7865	0.0145	0.7864	0.3082	0.3207
4	1.8414	0.1057	0.1004	0.3943	0.0075	1.1818	0.2615	0.1313
8	1.8181	0.0935	0.0567	0.1972	0.0037	1.3829	0.1652	0.0772
16	1.8918	0.0863	0.0477	0.0985	0.0020	1.4813	0.1820	0.0618
32	1.9282	0.0818	0.0567	0.0494	0.0013	1.5395	0.1605	0.0612
64	1.9573	0.0759	0.0480	0.0249	0.0009	1.5849	0.1586	0.0638

Table C.17: 25000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.4616	0.2035	0.0910	0.0000	0.0000	2.4360	0.3686	0.2623
2	3.8056	0.1536	0.0819	0.0000	0.0000	2.4369	0.4551	0.6083
4	3.0423	0.1293	0.0664	0.0000	0.0000	2.4420	0.2348	0.1214
8	2.9989	0.1167	0.0611	0.0000	0.0000	2.4497	0.2240	0.1095
16	2.9668	0.1112	0.0588	0.0000	0.0000	2.4544	0.1904	0.1205
32	2.9196	0.1026	0.0612	0.0000	0.0000	2.4719	0.1730	0.0822
64	3.1663	0.0953	0.0632	0.0000	0.0000	2.4859	0.2331	0.2613

Table C.18: 25000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.8897	0.2009	0.6752	2.4401	0.0295	0.0000	0.3718	1.4789
2	2.8659	0.1535	0.3675	1.2242	0.0153	1.2225	0.3560	0.6687
4	2.7084	0.1295	0.1873	0.6130	0.0076	1.8340	0.2329	0.2636
8	2.7438	0.1162	0.0925	0.3071	0.0040	2.1474	0.1942	0.1499
16	2.8476	0.1110	0.0596	0.1535	0.0022	2.3041	0.2115	0.1278
32	2.8914	0.1028	0.0615	0.0771	0.0015	2.3984	0.1847	0.1149
64	3.0907	0.0953	0.0631	0.0386	0.0012	2.4477	0.2166	0.2412

Table C.19: 25000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.4691	0.2041	0.0910	0.0000	0.0000	2.4343	0.4465	0.1932
2	3.2361	0.1527	0.0813	0.0000	0.0000	2.4423	0.3452	0.1451
4	3.0503	0.1269	0.0654	0.0000	0.0000	2.4442	0.2713	0.0949
8	2.9818	0.1138	0.0610	0.0000	0.0000	2.4520	0.2420	0.0755
16	2.9575	0.1075	0.0605	0.0000	0.0000	2.4569	0.2063	0.0949
32	2.9341	0.0994	0.0644	0.0000	0.0000	2.4728	0.1901	0.0787
64	2.9466	0.0928	0.0661	0.0000	0.0000	2.4869	0.1957	0.0779

Table C.20: 25000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.8841	0.2021	0.7119	2.4406	0.0295	0.0000	0.4312	1.3758
2	2.7949	0.1526	0.3567	1.2240	0.0153	1.2223	0.3933	0.5724
4	2.7173	0.1265	0.1697	0.6135	0.0076	1.8358	0.2890	0.2358
8	2.7056	0.1137	0.0800	0.3070	0.0040	2.1477	0.2302	0.0908
16	2.7982	0.1072	0.0611	0.1535	0.0022	2.3043	0.2152	0.0771
32	2.8666	0.0992	0.0642	0.0771	0.0015	2.3972	0.2064	0.0709
64	2.9078	0.0924	0.0661	0.0385	0.0012	2.4487	0.2000	0.0741

Table C.21: 30000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.6453	0.2232	0.1037	0.0000	0.0000	3.5397	0.4038	0.2754
2	5.1758	0.1743	0.0934	0.0000	0.0000	3.5404	0.5382	0.7597
4	4.2575	0.1510	0.0784	0.0000	0.0000	3.5503	0.2817	0.1478
8	4.1908	0.1368	0.0727	0.0000	0.0000	3.5546	0.2497	0.1390
16	4.1764	0.1336	0.0710	0.0000	0.0000	3.5683	0.2358	0.1360
32	4.1528	0.1214	0.0694	0.0000	0.0000	3.5895	0.2230	0.1207
64	4.2147	0.1133	0.0683	0.0000	0.0000	3.6063	0.2258	0.1734

Table C.22: 30000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.0042	0.2222	0.9979	3.5459	0.0305	0.0000	0.3669	2.2522
2	4.1251	0.1742	0.4290	1.7749	0.0155	1.7730	0.4836	1.1665
4	3.8889	0.1511	0.3062	0.8912	0.0087	2.6651	0.2572	0.4466
8	3.8963	0.1366	0.1338	0.4418	0.0042	3.1170	0.2433	0.2216
16	3.9527	0.1333	0.0731	0.2213	0.0024	3.3470	0.2225	0.1429
32	4.0310	0.1209	0.0694	0.1127	0.0017	3.4757	0.2194	0.1162
64	4.1740	0.1124	0.0682	0.0561	0.0014	3.5514	0.2322	0.1824

Table C.23: 30000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.6837	0.2236	0.1036	0.0000	0.0000	3.5399	0.4998	0.2174
2	4.5115	0.1716	0.0925	0.0000	0.0000	3.5436	0.4494	0.1848
4	4.2309	0.1470	0.0781	0.0000	0.0000	3.5513	0.3032	0.1036
8	4.1889	0.1326	0.0733	0.0000	0.0000	3.5579	0.2800	0.1074
16	4.1323	0.1283	0.0736	0.0000	0.0000	3.5714	0.2389	0.0888
32	4.1386	0.1168	0.0742	0.0000	0.0000	3.5896	0.2404	0.0889
64	4.1217	0.1097	0.0732	0.0000	0.0000	3.6079	0.2282	0.0753

Table C.24: 30000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.0226	0.2220	0.9951	3.5474	0.0306	0.0000	0.5239	2.1171
2	3.9131	0.1710	0.5179	1.7756	0.0155	1.7736	0.4434	0.9087
4	3.8442	0.1470	0.2394	0.8913	0.0087	2.6658	0.3177	0.4124
8	3.8244	0.1323	0.1167	0.4418	0.0041	3.1172	0.2717	0.1429
16	3.9044	0.1277	0.0764	0.2212	0.0024	3.3463	0.2175	0.1030
32	4.0240	0.1163	0.0746	0.1128	0.0017	3.4770	0.2377	0.0894
64	4.0499	0.1095	0.0736	0.0561	0.0014	3.5523	0.2166	0.0709

Table C.25: 35000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	5.8984	0.2410	0.1225	0.0000	0.0000	4.7400	0.4187	0.2663
2	7.0007	0.1979	0.1043	0.0000	0.0000	4.7485	0.6905	1.1893
4	5.5743	0.1718	0.0895	0.0000	0.0000	4.7490	0.2989	0.2166
8	5.5292	0.1587	0.0840	0.0000	0.0000	4.7586	0.2687	0.2210
16	5.5314	0.1552	0.1110	0.0000	0.0000	4.7657	0.2626	0.2051
32	7.0378	0.1431	0.0949	0.0000	0.0000	4.7927	0.6774	1.3005
64	6.0317	0.1315	0.0855	0.0000	0.0000	4.8491	0.3994	0.5383

Table C.26: 35000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	5.2316	0.2407	1.3061	4.7430	0.0319	0.0000	0.4347	3.0821
2	5.3028	0.1969	0.6111	2.3786	0.0181	2.3766	0.5110	1.5057
4	5.0974	0.1713	0.3793	1.1869	0.0083	3.5642	0.3108	0.6090
8	5.1035	0.1583	0.1745	0.5953	0.0046	4.1647	0.2817	0.2796
16	5.1542	0.1544	0.1138	0.2977	0.0027	4.4686	0.2452	0.1378
32	6.9506	0.1423	0.0951	0.1496	0.0019	4.6436	0.7042	1.3355
64	5.9905	0.1309	0.0855	0.0754	0.0016	4.7720	0.3921	0.5823

Table C.27: 35000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	5.9476	0.2404	0.1225	0.0000	0.0000	4.7396	0.5107	0.2246
2	5.7473	0.1933	0.1033	0.0000	0.0000	4.7453	0.4449	0.1907
4	5.5049	0.1675	0.0889	0.0000	0.0000	4.7531	0.3184	0.1291
8	5.4251	0.1530	0.0855	0.0000	0.0000	4.7585	0.2730	0.1175
16	5.4470	0.1492	0.1146	0.0000	0.0000	4.7687	0.2572	0.1258
32	5.4440	0.1371	0.0999	0.0000	0.0000	4.7953	0.2624	0.1205
64	5.4405	0.1286	0.0867	0.0000	0.0000	4.8480	0.2586	0.0912

Table C.28: 35000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	5.2331	0.2415	1.3419	4.7437	0.0319	0.0000	0.5251	2.9566
2	5.1344	0.1931	0.6725	2.3789	0.0181	2.3769	0.4546	1.3358
4	5.0802	0.1673	0.3199	1.1874	0.0083	3.5660	0.3442	0.6210
8	5.0677	0.1526	0.1668	0.5955	0.0046	4.1669	0.3079	0.2293
16	5.1763	0.1487	0.1181	0.2978	0.0027	4.4712	0.2777	0.1267
32	5.3250	0.1364	0.0996	0.1497	0.0019	4.6446	0.2857	0.1291
64	5.3656	0.1283	0.0866	0.0753	0.0016	4.7710	0.2556	0.0966

C.2 V100 Results

Table C.29: 5000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.6098	0.1205	0.0496	0.0000	0.0000	0.1604	0.1191	0.0593
2	0.4631	0.0702	0.0371	0.0000	0.0000	0.1609	0.0942	0.0403
4	0.3899	0.0462	0.0289	0.0000	0.0000	0.1613	0.0828	0.0304
8	0.3662	0.0350	0.0236	0.0000	0.0000	0.1630	0.0808	0.0276
16	0.3484	0.0288	0.0209	0.0000	0.0000	0.1654	0.0785	0.0250
32	0.3374	0.0267	0.0199	0.0000	0.0000	0.1693	0.0726	0.0217
64	0.3444	0.0267	0.0194	0.0000	0.0000	0.1765	0.0725	0.0234

Table C.30: 5000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.5257	0.1161	0.0670	0.1655	0.0267	0.0000	0.1197	0.0669
2	0.4140	0.0698	0.0405	0.0825	0.0136	0.0809	0.0930	0.0387
4	0.3562	0.0463	0.0291	0.0403	0.0068	0.1212	0.0812	0.0257
8	0.3524	0.0350	0.0237	0.0203	0.0034	0.1429	0.0842	0.0264
16	0.3370	0.0287	0.0210	0.0101	0.0018	0.1553	0.0774	0.0241
32	0.3289	0.0267	0.0200	0.0050	0.0009	0.1641	0.0721	0.0201
64	0.3396	0.0266	0.0193	0.0025	0.0005	0.1737	0.0733	0.0228

Table C.31: 5000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.6106	0.1206	0.0495	0.0000	0.0000	0.1603	0.1208	0.0582
2	0.4622	0.0723	0.0372	0.0000	0.0000	0.1609	0.0930	0.0386
4	0.3859	0.0484	0.0296	0.0000	0.0000	0.1613	0.0777	0.0290
8	0.3632	0.0366	0.0243	0.0000	0.0000	0.1630	0.0782	0.0253
16	0.3449	0.0305	0.0214	0.0000	0.0000	0.1654	0.0721	0.0261
32	0.3360	0.0286	0.0209	0.0000	0.0000	0.1693	0.0677	0.0226
64	0.3432	0.0286	0.0205	0.0000	0.0000	0.1766	0.0672	0.0246

Table C.32: 5000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	0.5255	0.1160	0.0641	0.1645	0.0268	0.0000	0.1241	0.0657
2	0.4146	0.0721	0.0402	0.0822	0.0135	0.0809	0.0939	0.0368
4	0.3561	0.0484	0.0298	0.0403	0.0068	0.1213	0.0783	0.0261
8	0.3513	0.0367	0.0244	0.0203	0.0034	0.1429	0.0809	0.0266
16	0.3360	0.0306	0.0215	0.0101	0.0018	0.1554	0.0737	0.0244
32	0.3318	0.0286	0.0210	0.0051	0.0009	0.1642	0.0691	0.0228
64	0.3335	0.0284	0.0204	0.0025	0.0005	0.1737	0.0659	0.0213

Table C.33: 10000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.1516	0.1455	0.0735	0.0000	0.0000	0.6286	0.1186	0.0864
2	1.0198	0.0928	0.0568	0.0000	0.0000	0.6294	0.1089	0.0730
4	0.9902	0.0700	0.0471	0.0000	0.0000	0.6316	0.1148	0.0803
8	0.9423	0.0573	0.0417	0.0000	0.0000	0.6330	0.1043	0.0690
16	0.9126	0.0524	0.0397	0.0000	0.0000	0.6384	0.0922	0.0595
32	0.9101	0.0516	0.0384	0.0000	0.0000	0.6440	0.0906	0.0576
64	0.9884	0.0515	0.0406	0.0000	0.0000	0.6588	0.1071	0.1041

Table C.34: 10000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.0381	0.1424	0.2042	0.6264	0.0286	0.0000	0.1244	0.4105
2	0.9268	0.0931	0.1117	0.3154	0.0143	0.3149	0.1103	0.2089
4	0.8837	0.0701	0.0633	0.1578	0.0073	0.4739	0.1112	0.1067
8	0.8653	0.0576	0.0420	0.0791	0.0038	0.5541	0.1058	0.0640
16	0.8704	0.0526	0.0398	0.0395	0.0020	0.5989	0.0917	0.0559
32	0.8891	0.0519	0.0383	0.0197	0.0012	0.6241	0.0916	0.0560
64	0.9823	0.0519	0.0405	0.0098	0.0009	0.6487	0.1091	0.1065

Table C.35: 10000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.1761	0.1454	0.0736	0.0000	0.0000	0.6286	0.1348	0.0921
2	1.0248	0.0973	0.0583	0.0000	0.0000	0.6295	0.1040	0.0754
4	0.9819	0.0738	0.0490	0.0000	0.0000	0.6317	0.1086	0.0720
8	0.9406	0.0612	0.0429	0.0000	0.0000	0.6331	0.0967	0.0689
16	0.9158	0.0564	0.0418	0.0000	0.0000	0.6384	0.0856	0.0622
32	0.9095	0.0551	0.0409	0.0000	0.0000	0.6441	0.0834	0.0582
64	0.9329	0.0537	0.0426	0.0000	0.0000	0.6589	0.0873	0.0641

Table C.36: 10000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.0352	0.1427	0.2023	0.6268	0.0287	0.0000	0.1319	0.3996
2	0.9306	0.0974	0.1113	0.3156	0.0143	0.3149	0.1061	0.2109
4	0.8686	0.0739	0.0601	0.1578	0.0073	0.4740	0.1105	0.0914
8	0.8662	0.0612	0.0430	0.0791	0.0038	0.5542	0.0989	0.0656
16	0.8793	0.0564	0.0419	0.0395	0.0020	0.5990	0.0897	0.0606
32	0.8887	0.0551	0.0408	0.0197	0.0012	0.6243	0.0817	0.0587
64	0.9227	0.0540	0.0427	0.0098	0.0009	0.6487	0.0887	0.0633

Table C.37: 15000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.0278	0.1708	0.0940	0.0000	0.0000	1.3722	0.1432	0.1487
2	1.9404	0.1183	0.0780	0.0000	0.0000	1.3754	0.1518	0.1553
4	1.8716	0.0926	0.0654	0.0000	0.0000	1.3745	0.1454	0.1452
8	1.7852	0.0815	0.0615	0.0000	0.0000	1.3782	0.1138	0.1119
16	1.7718	0.0781	0.0579	0.0000	0.0000	1.3882	0.1101	0.1062
32	2.1147	0.0797	0.0577	0.0000	0.0000	1.4008	0.2012	0.3470
64	2.5477	0.0765	0.0574	0.0000	0.0000	1.4217	0.3299	0.6352

Table C.38: 15000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.8670	0.1672	0.4116	1.3584	0.0305	0.0000	0.1437	0.9859
2	1.8000	0.1191	0.2382	0.6875	0.0164	0.6879	0.1833	0.4765
4	1.7274	0.0942	0.1220	0.3436	0.0082	1.0315	0.1450	0.2715
8	1.6642	0.0837	0.0755	0.1723	0.0046	1.2065	0.1161	0.1375
16	1.6856	0.0805	0.0586	0.0864	0.0027	1.3020	0.1151	0.0958
32	2.1608	0.0800	0.0579	0.0432	0.0018	1.3571	0.2268	0.4107
64	2.6239	0.0767	0.0576	0.0215	0.0014	1.3995	0.3552	0.7079

Table C.39: 15000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	2.0291	0.1709	0.0938	0.0000	0.0000	1.3721	0.1502	0.1427
2	1.9190	0.1242	0.0811	0.0000	0.0000	1.3754	0.1464	0.1316
4	1.8381	0.0983	0.0673	0.0000	0.0000	1.3746	0.1274	0.1228
8	1.7854	0.0871	0.0643	0.0000	0.0000	1.3781	0.1079	0.1107
16	1.7688	0.0827	0.0616	0.0000	0.0000	1.3883	0.0999	0.1056
32	1.7742	0.0828	0.0620	0.0000	0.0000	1.4008	0.0993	0.1010
64	1.7990	0.0771	0.0618	0.0000	0.0000	1.4216	0.1081	0.1037

Table C.40: 15000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	1.8537	0.1667	0.4108	1.3585	0.0305	0.0000	0.1477	0.9725
2	1.7513	0.1242	0.2151	0.6867	0.0163	0.6879	0.1470	0.4846
4	1.6800	0.0980	0.1181	0.3435	0.0081	1.0315	0.1245	0.2479
8	1.6571	0.0870	0.0764	0.1723	0.0045	1.2065	0.1079	0.1364
16	1.6872	0.0829	0.0620	0.0863	0.0026	1.3020	0.1014	0.1051
32	1.7245	0.0829	0.0621	0.0432	0.0018	1.3571	0.0984	0.0960
64	1.7707	0.0771	0.0617	0.0215	0.0014	1.3996	0.1057	0.1003

Table C.41: 20000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.2332	0.1992	0.1175	0.0000	0.0000	2.4862	0.1525	0.1805
2	3.1920	0.1421	0.0948	0.0000	0.0000	2.4869	0.1826	0.2161
4	3.0695	0.1153	0.0837	0.0000	0.0000	2.4906	0.1594	0.1721
8	2.9960	0.1060	0.0799	0.0000	0.0000	2.4985	0.1297	0.1433
16	2.9843	0.1039	0.0770	0.0000	0.0000	2.5012	0.1278	0.1428
32	3.6921	0.1045	0.0830	0.0000	0.0000	2.5061	0.3249	0.6429
64	4.3841	0.1035	0.0746	0.0000	0.0000	2.5411	0.5231	1.1130

Table C.42: 20000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.0767	0.1945	0.7181	2.4522	0.0330	0.0000	0.1585	1.8348
2	2.9746	0.1431	0.4070	1.2431	0.0162	1.2432	0.1898	0.8805
4	2.9159	0.1170	0.2046	0.6216	0.0089	1.8682	0.1639	0.4890
8	2.9022	0.1079	0.1223	0.3105	0.0056	2.1863	0.1333	0.2898
16	2.8418	0.1063	0.0815	0.1559	0.0035	2.3455	0.1302	0.1431
32	3.5035	0.1052	0.0829	0.0774	0.0025	2.4280	0.2949	0.5624
64	4.0007	0.1048	0.0747	0.0387	0.0020	2.5015	0.4089	0.8821

Table C.43: 20000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.3086	0.1997	0.1175	0.0000	0.0000	2.4861	0.1700	0.2215
2	3.1914	0.1509	0.0987	0.0000	0.0000	2.4869	0.1624	0.2079
4	3.0811	0.1231	0.0863	0.0000	0.0000	2.4906	0.1440	0.1801
8	3.0214	0.1136	0.0844	0.0000	0.0000	2.4985	0.1246	0.1551
16	3.0050	0.1102	0.0829	0.0000	0.0000	2.5014	0.1160	0.1570
32	3.0171	0.1073	0.0882	0.0000	0.0000	2.5061	0.1240	0.1592
64	3.0331	0.1039	0.0822	0.0000	0.0000	2.5411	0.1230	0.1517

Table C.44: 20000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	3.1008	0.1946	0.7194	2.4523	0.0330	0.0000	0.1617	1.8449
2	3.0450	0.1507	0.3681	1.2425	0.0162	1.2433	0.1672	0.9804
4	2.8826	0.1230	0.1990	0.6215	0.0088	1.8684	0.1451	0.4770
8	2.8556	0.1134	0.1235	0.3105	0.0056	2.1865	0.1222	0.2584
16	2.8346	0.1100	0.0845	0.1560	0.0035	2.3458	0.1159	0.1416
32	2.9448	0.1072	0.0878	0.0774	0.0024	2.4281	0.1220	0.1629
64	2.9967	0.1037	0.0819	0.0387	0.0020	2.5018	0.1262	0.1514

Table C.45: 25000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.8830	0.2281	0.1428	0.0000	0.0000	3.8586	0.1877	0.3369
2	5.0046	0.1691	0.1141	0.0000	0.0000	3.8633	0.2738	0.4971
4	4.6530	0.1444	0.1019	0.0000	0.0000	3.8590	0.1917	0.2967
8	4.5447	0.1337	0.0981	0.0000	0.0000	3.8673	0.1523	0.2446
16	4.6375	0.1327	0.0967	0.0000	0.0000	3.8772	0.1768	0.3187
32	5.4622	0.1283	0.0946	0.0000	0.0000	3.9036	0.4053	0.8960
64	6.6466	0.1232	0.1041	0.0000	0.0000	3.9626	0.7231	1.6926

Table C.46: 25000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.5644	0.2224	1.1050	3.8005	0.0368	0.0000	0.1883	2.8704
2	4.5470	0.1685	0.6170	1.9303	0.0199	1.9324	0.2490	1.4450
4	4.4101	0.1440	0.3063	0.9648	0.0120	2.8953	0.1897	0.8005
8	4.3818	0.1335	0.1776	0.4827	0.0067	3.3844	0.1510	0.4769
16	4.5627	0.1336	0.1147	0.2421	0.0044	3.6350	0.1978	0.4335
32	5.0276	0.1268	0.0940	0.1210	0.0033	3.7817	0.3195	0.6727
64	6.9599	0.1246	0.1047	0.0607	0.0026	3.9007	0.8332	1.9633

Table C.47: 25000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	5.0249	0.2285	0.1424	0.0000	0.0000	3.8585	0.2297	0.4205
2	4.7286	0.1780	0.1174	0.0000	0.0000	3.8631	0.1888	0.2992
4	4.6855	0.1516	0.1051	0.0000	0.0000	3.8590	0.1642	0.3326
8	4.5643	0.1400	0.1045	0.0000	0.0000	3.8671	0.1379	0.2647
16	4.5515	0.1381	0.1048	0.0000	0.0000	3.8772	0.1325	0.2617
32	4.5430	0.1297	0.1034	0.0000	0.0000	3.9033	0.1393	0.2369
64	4.5942	0.1239	0.1118	0.0000	0.0000	3.9623	0.1423	0.2227

Table C.48: 25000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	4.6655	0.2225	1.0865	3.8006	0.0368	0.0000	0.2350	2.9150
2	4.4667	0.1772	0.5610	1.9291	0.0199	1.9326	0.1927	1.4837
4	4.3731	0.1515	0.2983	0.9648	0.0120	2.8955	0.1667	0.7892
8	4.4004	0.1397	0.1800	0.4827	0.0067	3.3846	0.1384	0.4930
16	4.3462	0.1375	0.1200	0.2421	0.0044	3.6353	0.1311	0.2839
32	4.4218	0.1299	0.1035	0.1210	0.0033	3.7818	0.1392	0.2338
64	4.5265	0.1236	0.1115	0.0607	0.0026	3.9007	0.1425	0.2176

Table C.49: 30000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	6.9935	0.2592	0.1622	0.0000	0.0000	5.7602	0.1947	0.4813
2	6.9671	0.1908	0.1321	0.0000	0.0000	5.7600	0.2564	0.5349
4	6.7406	0.1693	0.1253	0.0000	0.0000	5.7814	0.1799	0.4217
8	6.6485	0.1603	0.1169	0.0000	0.0000	5.7669	0.1746	0.3844
16	8.6766	0.1621	0.1177	0.0000	0.0000	5.7706	0.7299	1.8618
32	9.4751	0.1530	0.1178	0.0000	0.0000	5.7924	0.9589	2.4167
64	10.3928	0.1511	0.1163	0.0000	0.0000	5.8350	1.2159	3.0378

Table C.50: 30000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	6.6035	0.2497	1.6260	5.6680	0.0413	0.0000	0.2021	4.3423
2	6.5524	0.1906	0.9040	2.8806	0.0202	2.8803	0.2642	2.1918
4	6.4593	0.1691	0.4476	1.4460	0.0133	4.3351	0.1841	1.2501
8	6.4168	0.1607	0.2506	0.7194	0.0080	5.0464	0.1762	0.7307
16	7.8384	0.1618	0.1530	0.3600	0.0053	5.4102	0.5670	1.5007
32	9.0869	0.1541	0.1180	0.1798	0.0040	5.6112	0.9009	2.2684
64	10.2025	0.1510	0.1165	0.0898	0.0032	5.7438	1.2064	2.9539

Table C.51: 30000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	6.9822	0.2592	0.1624	0.0000	0.0000	5.7600	0.2416	0.4489
2	6.7820	0.1998	0.1359	0.0000	0.0000	5.7601	0.2013	0.4104
4	6.7005	0.1780	0.1313	0.0000	0.0000	5.7813	0.1679	0.3904
8	6.6360	0.1676	0.1252	0.0000	0.0000	5.7667	0.1525	0.3818
16	6.6282	0.1671	0.1276	0.0000	0.0000	5.7703	0.1460	0.3836
32	6.6384	0.1556	0.1294	0.0000	0.0000	5.7922	0.1540	0.3739
64	6.6132	0.1495	0.1295	0.0000	0.0000	5.8346	0.1581	0.3112

Table C.52: 30000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	6.5524	0.2498	1.5990	5.6678	0.0412	0.0000	0.2434	4.2877
2	6.4895	0.1999	0.8235	2.8796	0.0202	2.8805	0.2084	2.2659
4	6.4556	0.1777	0.4481	1.4461	0.0133	4.3353	0.1724	1.2478
8	6.3969	0.1674	0.2547	0.7193	0.0079	5.0464	0.1539	0.7228
16	6.3963	0.1668	0.1639	0.3600	0.0053	5.4105	0.1468	0.4697
32	6.4429	0.1556	0.1292	0.1798	0.0040	5.6115	0.1558	0.3571
64	6.5187	0.1497	0.1296	0.0898	0.0032	5.7439	0.1601	0.3056

Table C.53: 35000 async serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	8.9236	0.2834	0.1801	0.0000	0.0000	7.5056	0.2196	0.5916
2	8.8956	0.2187	0.1545	0.0000	0.0000	7.5275	0.2855	0.6245
4	8.6245	0.1936	0.1424	0.0000	0.0000	7.5121	0.1941	0.5240
8	8.6583	0.1879	0.1368	0.0000	0.0000	7.5276	0.1896	0.5519
16	10.6126	0.1882	0.1650	0.0000	0.0000	7.5387	0.7450	1.9414
32	13.1461	0.1813	0.1502	0.0000	0.0000	7.6005	1.4660	3.7111
64	14.2242	0.1746	0.1414	0.0000	0.0000	7.7438	1.6991	4.4346

Table C.54: 35000 async overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	8.5908	0.2719	2.1093	7.3984	0.0465	0.0000	0.2157	5.7663
2	8.5083	0.2181	1.1400	3.7642	0.0290	3.7646	0.2934	2.9580
4	8.3482	0.1939	0.5732	1.8775	0.0154	5.6348	0.1933	1.6733
8	8.3555	0.1878	0.3212	0.9399	0.0098	6.5863	0.1894	1.0110
16	9.9226	0.1878	0.2119	0.4710	0.0069	7.0680	0.6416	1.7676
32	11.7161	0.1810	0.1498	0.2361	0.0046	7.3629	1.1272	2.8537
64	14.2427	0.1743	0.1416	0.1193	0.0036	7.6227	1.7710	4.4996

Table C.55: 35000 sync serial

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	9.0193	0.2832	0.1801	0.0000	0.0000	7.5056	0.2645	0.6405
2	8.7932	0.2287	0.1585	0.0000	0.0000	7.5277	0.2226	0.5722
4	8.6428	0.2036	0.1505	0.0000	0.0000	7.5122	0.1806	0.5376
8	8.6122	0.1956	0.1477	0.0000	0.0000	7.5274	0.1685	0.5260
16	8.6445	0.1932	0.1765	0.0000	0.0000	7.5393	0.1717	0.5285
32	8.6954	0.1825	0.1634	0.0000	0.0000	7.6004	0.1753	0.5382
64	8.7728	0.1743	0.1487	0.0000	0.0000	7.7433	0.1793	0.4974

Table C.56: 35000 sync overlap

Dp.	Ex.	Pack	Unpack	I. Cal.	O. Cal.	P. Cal.	Mess.	Desyn.
1	8.4969	0.2719	2.0800	7.3980	0.0465	0.0000	0.2770	5.6630
2	8.4750	0.2286	1.0752	3.7631	0.0289	3.7646	0.2281	3.0436
4	8.3366	0.2030	0.5737	1.8777	0.0154	5.6352	0.1860	1.6631
8	8.3460	0.1947	0.3267	0.9399	0.0098	6.5866	0.1687	1.0088
16	8.3488	0.1932	0.2239	0.4710	0.0069	7.0679	0.1704	0.6475
32	8.4254	0.1823	0.1628	0.2360	0.0046	7.3633	0.1750	0.5083
64	8.6524	0.1740	0.1484	0.1193	0.0036	7.6229	0.1775	0.4962

NTNU

Norwegian University of
Science and Technology