UNIVERSITY OF COPENHAGEN

MASTER THESIS

Exascale modelling using GPUs

Author:
Kristian Lyck LOTZKAT

Supervisors:
Troels Haugbølle
Åke Nordlund

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Physics

in the

Astrophysics and Planetary Science
Niels Bohr Institute

June 10, 2022

Declaration of Authorship

I, Kristian Lyck LOTZKAT, declare that this thesis titled, "Exascale modelling using GPUs" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITY OF COPENHAGEN

Abstract

Faculty of Science
Niels Bohr Institute

Master of Science in Physics

Exascale modelling using GPUs

by Kristian Lyck LOTZKAT

The porting of the DISPATCH framework to execute on GPUs using OpenMP was last year initialized by porting the finite volume MHD solver to GPUs. Here the porting of DISPATCH is continued, with the main focus of solving selfgravity on the GPUs. Several implementations with different memory structures are made. The implementations are based on previous implementations, where bunches of tasks are transferred and updated simultaneously on the GPUs. A new offload implementation is made, which allows to use several devices. Due to problems with the existing code, implementation in the DISPATCH framework has not been possible. Instead, the implementations have been thoroughly tested in mockups as preparation for the implementation in DISPATCH. A suggestion to a new offloading system is given, which may provide better utilization of both the CPU and GPU hardware.

It is found that solving selfgravity on the GPUs is indeed possible. The simple nature of the SOR method does not allow for good utilization of the GPU hardware, but does in general perform better than the CPU-version, provided a good scheduling scheme of the bunches is used. Bunches should at least contain the same number of patches as there are streaming multiprocessors on the GPU. This seems to make the performance less dependent on the number of available CPU cores and provides the best results. The implementations have been compared to the CPU version and does indeed yield the same results. However, validation from running experiments is still needed once fully implemented in DISPATCH.

Acknowledgements

I would like to thank my supervisors, Troels Haugbølle and Åke Nordlund, for providing guidance and assistance when needed throughout the project. Their vast knowledge, both technical and theoretical, has provided a solid foundation for most of the work.

I would also like to give a thank to Sven Karlson, who participated as an extra supervisor during the first half of the thesis, providing great knowledge about the technical details behind hardware and compilers.

Lastly, I will give a great thank to Stibofonden for funding a trip to Kajaani, Finland, where I participated in a PRACE autumnschool, hosted by the CSC, about porting codes to GPUs using OpenMP. This provided a great opportunity to discuss the project with some of the people behind LUMI and the great experience to actually see the LUMI supercomputer.

Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

1 Introduction 1

2 High Performance Computing 3
- 2.1 Basic computer Architecture . . . 3
  - 2.1.1 Memory . . . 4
  - 2.1.2 The Central Processing Unit . . . 5
  - 2.1.3 The Graphics Processing Unit . . . 7
    - The execution model . . . 8
- 2.2 GPU programming . . . 8
  - 2.2.1 The programming model . . . 9
  - 2.2.2 Directive based programming . . . 10
    - OpenMP . . . 11
- 2.3 Compilers . . . 12
- 2.4 HPC systems - Supercomputers . . . 13
  - 2.4.1 LUMI . . . 13

3 DISPATCH 15
- 3.1 Key ideas . . . 15
- 3.2 Tasks . . . 16
- 3.3 Offloading . . . 17
- 3.4 Task scheduling . . . 17

4 Theory 19
- 4.1 Selfgravity . . . 19
  - 4.1.1 Iterative methods . . . 20
    - Jacobi iterations and Gauss-Seidel . . . 21
    - Red-black iteration . . . 21
    - Successive Overrelaxation . . . 22
  - 4.1.2 Multi-grid methods . . . 23
  - 4.1.3 Solving Poisson's equation in DISPATCH . . . 23
  - 4.1.4 Gravity at different scales . . . 24

5 Implementation 26
- 5.1 Mockups . . . 26
  - 5.1.1 poisson_mock.f90 . . . 26
  - 5.1.2 GPU-implementation - pre-steps . . . 26
    - Step1.f90 . . . 26
    - Step2.f90 and Step3.f90 . . . 27

Step4.f90 27
Step5.f90 28
Step6.f90 28
Profiling 29
5.1.3 GPU-implementation - final versions 29
Profiling 31

6 Results 33
6.1 Optimal setup 33
6.1.1 CPU-version 33
6.1.2 System size 34
6.1.3 Patch per bunch 35
6.1.4 Bunch per device 35
6.1.5 Cores per device 36
6.1.6 Patch dimensions 38

7 Discussion 40
7.1 Current results 40
7.1.1 Optimal setup vs. DISPATCH 40
7.1.2 SOR on GPU 41
7.2 DISPATCH offload implementation - improvement ideas 42
7.2.1 Execution flow 42
7.2.2 Asynchronous pipelining 44
7.2.3 Linked list of procedure pointers 45

8 Conclusion 47

A Bunch version 49
A.1 Device_handler_mod 49
Update 49
Assign device 49
A.2 Device_mod 50
Assign bunch 50
A.3 Bunch_mod 50
Copy to bunch 50
Copy from bunch 51
Update 51
A.4 SOR 52

Bibliography 54

List of Figures

2.1 The memory hierarchy of computer architectures, showing the typical capacity range and access time for each layer. Not all computers contain all the layers. The price and capacity increase as one moves up the hierarchy, whereas the access time decreases. 5
2.2 Basic architecture of a dual core CPU-chip and the memory layers. The main memory and L3 cache are places off-chip. The L2 cache is usually places on-chip shared between the cores. Each core contains a control unit, registers, an ALU and two L1 caches - one for data and one for instructions. 6
2.3 Basic architecture of a GPU. On the left is the GPU, mainly containing different memory layers, a workload distributor and several streaming multiprocessors (SM). On the right is an SM containing different kind of memory, warp schedulers, dispatch units and several cores . . 7
2.4 Programming abstractions for GPU hardware. The computational domain is decomposed into a grid, consisting of several blocks. Each block contains several threads which are executed in warps. 10

3.1 A 2D computational domain is distributed to MPI processes. Each MPI process owns the tasks within the blue dashed square, which is divided in two groups *boundary tasks*(yellow) and *internal tasks*(green). The MPI rank also holds information of boundary tasks from neighbouring ranks, which constitute a third group of *boundary tasks*(red). The exchange of tasks between ranks happens by simply changing a virtual task to a boundary task and vice versa. 16
3.2 Simplified flow chart of the intra-node execution in DISPATCH. Everything within the red dashed square is the execution flow when executing on CPUs and everything within the blue square is the execution flow when executing on GPUs. 18

4.1 Red/Black iteration. During the first iteration all the black cells are updated. During the second update all the red cells are updated, using the updated black cells. This procedure continues until convergence after N iterations. 22

5.1 Profile of `Step6.f90` using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device. 29

5.2 Profile of the *bunch* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device. 31
5.3 Profile of the *device* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device. 32
5.4 Profile of the *bunch* version using Nsight Systems, executing using eight cores and two GPUs. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device. 32
5.5 Profile of the *bunch* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 216 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device. 32

6.1 *Left*: Time per patch in *ms* vs. number of cores for different mesh sizes. *Right*: Time per cell in *ns* vs. number of cores using different patch sizes. 34
6.2 The time per patch in ms as a function of the total number of patches for different versions of the mockups. The GPU versions uses one device and eight cores, and the CPU version uses 16 cores. The performance seems to get better as the number of patches increase, which is likely due to the overhead of initializing tasks, transferring data etc. becomes less significant compared to the amount of computations. . . 34
6.3 The time per patch in ms as a function of the number of patches per bunch for all the GPU versions, using one device and eight cores. *Left* is with three bunches per device, *right* is with 6 bunches per device. In general, it is best to use a bunch size which is a multiple of the number of SMs on the GPU, rather than having the total number of patches to be a multiple of the bunch size. 35
6.4 The time per patch in ms as a function of the number of bunches per device for the bunch-, device- and Step6-version, using one device and eight cores. Solid lines is with $Level = 4$ and dashed lines is with $Level = 5$. 36
6.5 *Bunch version*: The time per patch in ms as a function of the number of cores using different number of bunches and bunch sizes. Plots on the *left* use 108 patches per bunch, plots on the *right* uses 216 patches per bunch. The two plots on the *top* are with one device, the two plots in the *middle* are with two devices and the two plots at the *bottom* are with 3 devices. 37
6.6 *Device version*: The time per patch in ms as a function of the number of cores using different number of bunches and bunch sizes. Plots on the *left* use 108 patches per bunch, plots on the *right* uses 216 patches per bunch. The two plots on the *top* are with one device, the two plots in the *middle* are with two devices and the two plots at the *bottom* are with 3 devices. 38

6.7 Time per cell vs. number of cores using different patch-dimensions and bunch sizes. For each run three bunches per device is used. *Left* is the times for the bunch version. *Right* is the times for the device version. *Top* uses one device, *middle* uses two devices and the *bottom* uses 3 devices. 39

7.1 Flowchart of the suggested execution flow. 43
7.2 Asynchronous pipelining of GPU instructions using OpenMP. The red square on top shows the current execution, where one target task is executed at a time. The green square in the middle is the asynchronous pipelining, where data transfers and kernel execution are executed concurrently. The blue square at the bottom shows the optimal execution, where threads continue execution on the CPU while the target tasks are executed on the device. 45

Listings

2.1 Example of using the `target` directive to offload data and kernels to the device . . . 11
5.1 Assigning memory locations to thread . . . 27
5.2 Pseudo-code example of `update` subroutine . . . 27
5.3 Update function with offload . . . 28
5.4 Call to `SOR_bunch` . . . 29
A.1 The update routine . . . 49
A.2 Assigning a device, bunch and bunch-slot to a thread . . . 49
A.3 Routine that copies to bunch, and updates when full and the device is free . . . 50
A.4 Routine that copies patch data to the bunch . . . 50
A.5 Routine that copies patch data from the bunch . . . 51
A.6 Routine that updates the bunch on the device . . . 51
A.7 SOR bunch routine . . . 52

List of Abbreviations

ALU **A**rithmic **L**ogic **U**nit
AMR **A**daptive **M**esh **R**efinement
API **A**pplication **P**rogramming **I**nterface
CG **C**onjugate **G**radient
CPU **C**entral **P**rocessing **U**nit
CUDA **C**ompute **U**nified **D**evice **A**rchitecture
DRAM **D**ynamic **R**andom **A**ccess **M**emory
DtH **D**evice **t**o **H**ost
FLOPS **Fl**oating point **O**perations **P**er **S**econd
GPU **G**raphics **P**rocessing **U**nit
GPGPU **G**eneral **P**urpose **G**raphics **P**rocessing **U**nit
HIP **H**eterogeneous computing **I**nterface for **P**ortability
HPC **H**igh **P**erformance **C**omputing
HtD **H**ost **t**o **D**evice
I/O **I**nput/**O**utput
LB **L**eft **B**ottom
LT **L**eft **T**op
LHS **L**eft **H**and **S**ight
MHD **M**agneto**H**ydro**D**ynamics
MPI **M**essage **P**assing **I**nterface
NUMA **N**on**u**niform **M**emory **A**ccess
OpenACC **Open** **Acc**elerators
OpenMP **Open** **M**ultiprocessing
OS **O**perating **S**ystem
RB **R**ight **B**ottom
RHS **R**ight **H**and **S**ight
RT **R**ight **T**op
SIMD **S**ingle **I**nstruction **M**ultiple **D**ata
SIMT **S**ingle **I**nstruction **M**ultiple **T**reads
SM **S**treaming **M**ultiprocessor
SOR **S**uccessive **O**ver**r**elaxation
SRAM **S**tatic **R**andom **A**ccess **M**emory

Chapter 1

Introduction

Testing theories describing astronomical phenomena is hard, as experiments are practically impossible to conduct and astronomers throughout history have been forced to rely on observations. With the invention of computers, astronomers have gotten a tool which allows to test theories through experiments in the form of computer simulations, which is now firmly established as one of the main methods to study the various dynamics of the universe.

These simulations may include several kinds of physics, such as hydrodynamics, magneto hydrodynamics, radiative transport, heating and cooling and selfgravity. For this purpose many multiphysics frameworks have been developed, such as ENZO (Bryan et al., 2014), Athena++ (Stone et al., 2020), WOMBAT (Mendygral et al., 2017) and Boxlib (Zhang et al., 2016).

The requirements for these frameworks increase with the available computing power provided by the great number of supercomputers around the world. The computing power have increased drastically over the years and we are entering the era of exascale computing, where supercomputers are able to perform 10^{18} computations per second. Supercomputers contain thousands of nodes and have traditionally been based on Central Processing Units (CPUs). The Supercomputer Fugaku, located in Japan, contains 158.976 nodes, each containing one CPU with 48 cores, making a total of 7.630.848 cores.

The massive number of nodes does indeed provide a huge amount of computing power and hence requires scientists to develop clever solutions for the frameworks, to exploit all of the available resources. A widespread approach is using block-based meshes or block-based adaptive mesh refinement techniques (Berger et al., 1989), allowing to distribute blocks of the computational domain to all the nodes. The main issue with either approach comes from the time stepping required to move forward in time. The time step has to be applied globally, requiring global synchronization between each time step, which constitutes an issue for the scalability of the frameworks. In a survey by Dubey et al., 2014 they conclude, that frameworks must eliminate the bulk synchronous model to overcome these scalability issues.

DISPATCH is a fairly new framework (Nordlund et al., 2018), which takes a different approach to overcome the issues with scalability. The computational domain is divided into several smaller tasks, which are distributed among the compute nodes, like the just mentioned methods. However, instead of using a global time step, each task uses a locally determined time step, thus requiring no global synchronization and allowing each MPI-rank to only communicate with a finite number of other ranks. This has shown to have almost unlimited scalability and provides a perfect tool for the upcoming exascale era (Nordlund et al., 2018).

GPUs have provided a new source of computing power and have become popular for supercomputers and as of June 2022, 7 of the top 10 most powerful computers in the world are based on GPUs, including the first real exascale machine Frontier

(*Top500: June 2022*). Among these is also LUMI, a new supercomputer located in Finland. LUMI is the most powerful computer in Europe containing 2560 nodes, each containing one CPU with 64 cores and four AMD MI250X GPUs.

Programs must be targeted specifically to GPUs, to utilize the full hardware. This provides yet another challenge to the frameworks, as they must be prepared to also run efficiently on heterogeneous systems. Directive based languages, such as OpenMP, have become very popular to accomplish this, as it can be implemented directly in the existing code, thus minimizing the additional code which must be made. DISPATCH is currently being prepared to run on heterogeneous systems using OpenMP.

In the current DISPATCH-setup, CPUs are used for the pre- and post-processing of the tasks, which are transferred to the GPUs in bunches, where all the computationally heavy updates take place. This requires some modifications to the existing solvers, as they must solve a whole bunch of tasks, rather than just individual tasks. It is currently only the finite volume MHD solver that has been prepared for this (Haarh, 2021).

Gravity is a central topic in physics and plays a big role in most of the dynamics of the universe. In grid-based simulation, iterative methods are used to solve for the gravitational potential. DISPATCH currently have two implementations, which may be used to solve for the gravitational potential, conjugate gradient (CG) and successive overrelaxation (SOR). At the core, these methods repeat the same computation several times, until reaching an acceptable solution. Even though the CG method in general performs better than the SOR method, the latter is simpler and thus seems better suited for GPU prototyping.

This thesis aims to further prepare DISPATCH for GPU execution, by preparing the SOR routine to update a bunch of tasks on the GPU. It is divided as follows: Chapter 2 contains an overview of various topics in high performance computing. This includes the fundamentals of computer architectures, a description of the various hardware, GPU programming, compilers and High performance systems. Chapter 3 gives a brief description of the key ideas and implementations of DISPATCH. Chapter 4 contains an overview of the theory behind selfgravity, including various techniques to solve the Poisson equation and a brief description of the algorithm used in DISPATCH. Chapter 5 describes the implementations which are used to prepare the SOR routine for GPU execution. In chapter 6, several tests of the GPU implementation are presented and compared to the CPU implementation. Chapter 7 contains a discussion of the results and key points to be aware of when implemented in DISPATCH. It further provides a discussion of some of the current offload implementations in DISPATCH and suggestions to further improvements and extensions. Finally the thesis is concluded in Chapter 8.

Chapter 2

High Performance Computing

Astrophysical simulations are becoming more complex as time moves forward and the available computing power is constantly increasing. This requires the development of frameworks, that have the necessary scalability to utilize the ever-growing amount of hardware. Scientists are thus required to have some knowledge about several tools, which allow them to accomplish this. This is the essence of High Performance Computing (HPC).

There are several things to be aware of when doing HPC. Especially two topics are of great importance: 1) how memory is distributed and shared among the available hardware and 2) the source of the computing power.

This chapter will briefly cover some of these topics. First the main components of computer architectures are described, including the memory system and the two main sources of computing power - the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). As the focus of this thesis is porting code to GPUs, a brief description of how to produce code for a GPU is given. Finally, a short description of compilers is given, as these may have a huge impact on how the GPU application works, especially when using directive based programming languages.

2.1 Basic computer Architecture

When doing *High Performance Computing*, the computer architecture has a great impact on the performance of the program. Many different architectures have been introduced through history, which all have had their pros and cons. The most popular architectures for simple computers, which most modern computers have adapted - or at least very similar ones, are the *Von Neumann* , *Harward*- and *Modified Harvard* architectures which mainly differ in the number of access paths between different hardware components. Independent of the architecture, almost every computer consists of three main hardware components - the *Central Processing Unit* (CPU), a memory system and an *input/output* (I/O) system - which each have to be tuned and optimized in order to achieve the best overall performance. Besides the three main components, most modern computers also contain some sort of *Graphics Processing Units* (GPU), either integrated on the CPU (*integrated GPU*) or as a separate hardware component, better known as *dedicated*- or *discrete GPUs*. As the name suggest, GPUs were designed to handle graphical work which required processing huge amounts of data, as quickly as possible. The great computing power of a GPU have made them popular for more general computing tasks, where they serve as accelerators in many HPC systems (Null et al., 2014).

High performance systems are not as simple as "normal" computers, as they contain much more hardware, making the architectures more complex. They may contain thousands of nodes, allowing for highly parallel execution. This allows for great

performance, but also introduces various issues, which the programmer must deal with. Before going into depth with this, a brief description of each of the just mentioned components will be given.

2.1.1 Memory

Any computer program needs a memory system to store data. Accessing memory is in general a very slow process compared to performing actual computations, usually referred to as the *processor-memory gap*. This is a potential bottleneck for the performance of a system, as it will lead to lots of idle time for the CPU while it waits for memory to arrive. New hardware for memory have been developed, such as *static random access memory* (SRAM), which is much faster to access than the *dynamic random access memory* (DRAM) which is used as main memory. The problem with the faster memory hardware is that it is much more expensive per byte. To account for this, the memory system consists of several layers - also known as the *memory hierarchy* see Figure 2.1. The idea is to have the fast memory hardware very close to the processor, working as a kind of temporary memory - called *cache* - which the processor can access very fast. The cache memory is much smaller in size - usually in the range of KB or MB, depending on the specific location - but usually contains the needed data. The higher levels contain a subset of the data of the lower levels and when the needed data is not present in one layer, the system will usually find it in the next layer in the hierarchy. In effect, this allows the memory system to seem having the capacity of the slower memory hardware (usually GB) at the lower levels of the hierarchy, but the access time of the faster hardware ($\sim$4-40 clock cycles) at the higher levels. This is all based on *the principle of locality* - the fact, that processors tend to access memory in a specific pattern. It is usually split up in two categories: *temporal locality*, the fact that recently accessed memory is likely to be accessed again and *spatial locality*, that is if address x have just been accessed, address $x + 1$ is likely to be accessed next. Thus, instead of only moving the needed data to the CPU, the memory system moves a block of memory to the higher level in the hierarchy, increasing the chance that the data, which is needed next, will be in the fast cache memory.

At the top of the hierarchy we have the registers, usually around 64 bits in size. They are located within the CPU, and only contain the most relevant data and addresses which are to be used for the current instruction. They are usually divided into sets, where some sets are for general-purpose others for special purpose. The next layers are the cache memory. There are several different kinds located at different places. Not all computers contain all of them. The L1 cache has the smallest capacity and is usually located within a core, to which it is local - meaning it can only be accessed by that specific core. Most modern computers contain two L1 caches, one for data and one for instructions. This decreases the probability of getting a cache miss, when the needed memory is not present. If a computer contains the L3 cache, the L2 cache is usually located on the CPU chip - shared among several cores and the L3 cache is located off-chip between the CPU and the main memory.

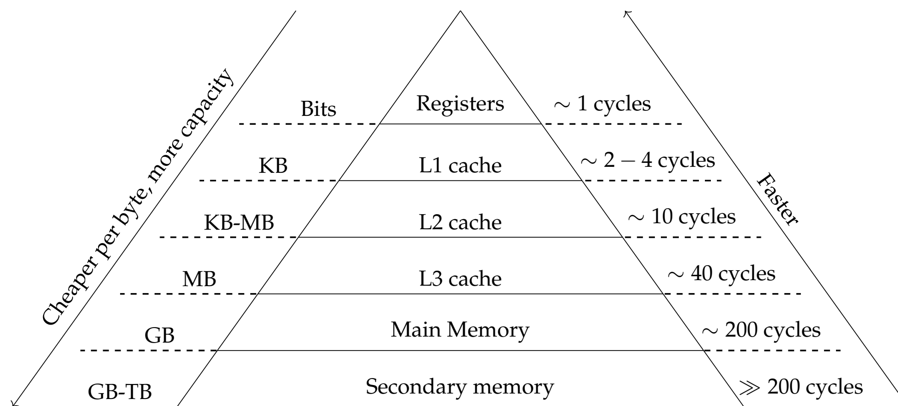

FIGURE 2.1: The memory hierarchy of computer architectures, showing the typical capacity range and access time for each layer. Not all computers contain all the layers. The price and capacity increase as one moves up the hierarchy, whereas the access time decreases.

Reading from memory is only one side of the story. Scientific programs also require a lot of writing to memory, which may cause some issues, especially when doing parallel computing using several cores or CPUs. If two CPUs contain the same cache line from main memory and CPU A computes a new value and updates it in its local cache, then the cache line of CPU B and the location in main memory will contain the wrong values. Memory has to be consistent throughout the memory hierarchy - or at least the memory in the caches should only contain the newest updated values, which is usually handled by some *cache coherence* protocol by the CPU. The protocol varies depending on the write protocol, but in almost all cases it will affect the performance, mainly through memory overhead and is thus important to be aware of.

The memory hierarchy is necessary to keep relevant memory as close to the CPU as possible. Using only one CPU with one core, this is fairly simple. Modern CPUs however usually contain several cores, and sometimes even several CPUs. This complicates things slightly as memory may be physically further away from some cores than others, introducing extra latency to access. An architecture where a global memory space is shared between all cores, but parts of it is more local to some cores than others, is referred to as a *nonuniform memory access* (NUMA) architecture, an example of a distributed shared memory system.

Some architectures are made of several "individual" computers, usually referred to as nodes. Each node may contain one or several CPUs and a local memory system, which cannot be accessed by other nodes. Nodes are then connected via some interconnection network, allowing to transfer data between nodes. This is an example of a distributed memory system and is what most modern supercomputers uses (Dumas, 2017).

2.1.2 The Central Processing Unit

The *Central Processing Unit* is the brain of the computer with the main purpose of processing data. It usually consists of three main components, the *Control Unit*, an *Arithmic Logic Unit* (ALU) and *memory*. The Control Unit controls which instructions are to be executed and makes sure the needed memory is ready for the execution. Doing so, it prepares the registers with the necessary data and decodes the instructions for the ALU, which then performs the instructions.

CPUs have improved a lot throughout the years and modern CPUs usually contain several cores, which allows the computer to run several processes concurrently - an example of a multicore processor is shown in Figure 2.2. The cores usually only perform one instruction at a time, thus older CPUs with one core would only allow one process to run at a time. However, today the *Operating System* (OS) allows each core to run several processes concurrently, by switching between various processes - also known as *context switching*. When one process stalls, e.g. because it is waiting for memory, the OS will switch to a different process which is ready for execution. Context switching is completely hidden from the user, to which it will seem as if several programs are running at the same time. With the introduction of threads - a sort of mini processes which a process can be subdivided into - the performance of CPUs was improved even further through multithreading - allowing multiple threads to execute in parallel. Threads share the same execution environment as their parent process and thus require fewer system resources. As a consequence, context switching between threads will generate less overhead compared to context switching between processes (Dumas, 2017).

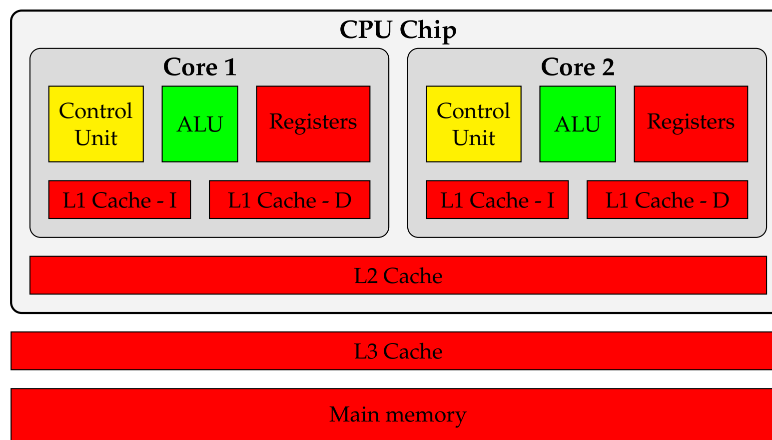

FIGURE 2.2: Basic architecture of a dual core CPU-chip and the memory layers. The main memory and L3 cache are places off-chip. The L2 cache is usually places on-chip shared between the cores. Each core contains a control unit, registers, an ALU and two L1 caches - one for data and one for instructions.

Most modern CPUs can also handle multiple instructions at once through *pipelining* - thus utilizing as much hardware as possible by dividing instructions into different stages, which are then executed concurrently. The CPU further allows multiple data to be computed at once - also known as *Single Instruction Multiple Data* (SIMD) - which are often very useful within scientific computations, as most data is stored as huge arrays, which undergoes several operations. Allowing multiple data to be computed in one instruction-call can increase the performance drastically. The technical details behind all these things, which allows a CPU to process data faster, is out of the scope for this thesis, and will not be described further. They are however very important to be aware of while programming, as they allow for different levels of parallelism - which can all be utilized to achieve better performance.

Through the just mentioned improvements, CPUs have become very powerful and modern CPUs found in most laptops are able to run with a clock rate well above 3 GHz, the best once up to around 5 GHz (executing more than 5 billion clock cycles per second). Each instruction performed takes a number of clock cycles to finish depending on the instruction. Reading from main memory may take about 50-200

clock cycles, whereas multiplication of two floating-point numbers only takes about 1 cycle. When comparing the performance of two CPUs, the clock-cycle may however be misleading. One of them might have a lower clock-frequency than the other but may be able to perform several instructions per clock-cycle, leading to a better overall performance. Because of this, the performance of a given hardware is often compared in FLOPS (*Floating point Operations Per Second*), which gives a better overall comparison (Robey et al., 2021). As an example, the theoretical peak performance of a 16-core AMD EPYC 7302 CPU at 3GHz is approximately 1.5 TFLOPS.

From the examples of number of clock cycles per instruction, it is obvious that reading and writing to/from memory takes much longer than performing actual computations (50-200 cycles vs. 1 cycle). This is commonly referred to as the *processor-memory gap*. As CPU speed has improved significantly over the years, read- and write-speed have improved very little. Despite that, engineers have found smart ways to account for this using a complex memory hierarchy, which is described in Section 2.1.1.

2.1.3 The Graphics Processing Unit

Graphics Processing Units were originally invented to accelerate numerical throughput for graphical tasks - hence the name - but have developed so they today are used in a wide variety of domains and are better described as accelerators - a special-purpose device that supplements the CPU. They are becoming increasingly popular as the main working horse of supercomputers and thus many vendors, such as NVIDIA and AMD, provide a big variety of powerful General Purpose GPUs (GPGPU). Each vendor uses slightly different terminology describing the hardware but have roughly the same architecture. For simplicity, I shall use the NVIDIA-terminology.

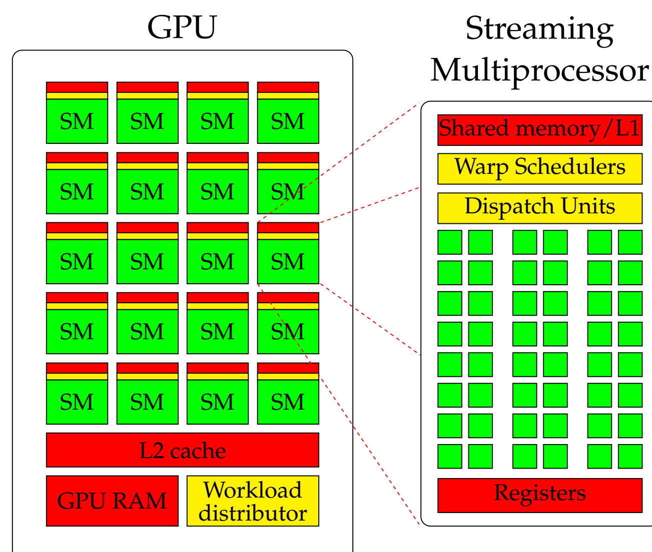

FIGURE 2.3: Basic architecture of a GPU. On the left is the GPU, mainly containing different memory layers, a workload distributor and several streaming multiprocessors (SM). On the right is an SM containing different kind of memory, warp schedulers, dispatch units and several cores

A simplified sketch of a GPU architecture (which does not apply to all GPU-models) is shown in Figure 2.3. Unlike the CPU, most of the hardware on a GPU is

made to process data and it does not contain the same amount of memory layers. It consists of GPU RAM, a workload distributor, an L2 cache and several *Streaming Multiprocessors* (SM). All instructions are executed by the SMs, rather than individual cores - which is one of the major distinctions between CPUs and GPUs. Each SM usually contains several warp schedulers, dispatch units and a dedicated L1 cache/shared memory, several cores and a dedicated register space, which is shared among all the cores (Robey et al., 2021). As an example, the NVIDIA A100 GPU contains 108 SMs, each containing 64 single-precision cores (32 double-precision cores), thus a total of 6912 single precision cores and a theoretical peak performance of 19.5 TFLOPS.

All memory layers on a GPU are much smaller than the equivalent memory on a CPU, but have a much higher memory bandwidth. The physical space is instead used for more cores. The main reason for the lack of memory, is that the memory hierarchy on a GPU is only designed for spatial locality, where it on the CPU is designed for both spatial and temporal locality. That is CPUs are designed to hide memory latency through a complex cache hierarchy, where GPUs are designed to hide latency through computations (Dumas, 2017).

The execution model

When a kernel is sent to the GPU, it is divided into several threads. Threads are then grouped together into *warps*. A warp usually consists of 32-64 threads, each operating on scalar data. Each SM can usually execute 2048 threads (32 warps) concurrently. In practice this number is however usually much smaller. All threads within a warp have to perform the same instruction, but may do it on different data. The operations do thus not classify as SIMD like on the CPU, but rather as *Single Instruction Multiple Threads* (SIMT). Some GPUs also contain vector hardware units, allowing to perform SIMD operations in addition to SIMT. Since all threads within a warp can only execute the same instruction, they are executed in lockstep and every thread must execute all paths. The warp will not terminate before all threads are completed, it is therefore of great importance to group similar threads as much as possible to utilize the hardware and obtain the best performance.

The cache on the individual SMs is usually incoherent. This may potentially lead to errors, but as we shall see in Section 2.2.1, the software usually makes sure that no two SMs work on the same data at the same time.

The warp schedulers allow several warps to run concurrently. When a thread within a warp gets a cache-miss, the warp will be put on a waiting list and a different warp is scheduled. With enough threads/warps available, this will hide memory latency, as the overall rate of computations will remain very high and thus not affect the performance (Robey et al., 2021).

2.2 GPU programming

From the previous section, it is obvious that the architecture of a GPU is very different from that of a CPU. This also means, that the programming model is very different. Programs optimized to run on CPUs will unlikely gain drastic performance improvements just by running on a GPU. The program has to be targeted the GPU in order to obtain this, which usually demands some sort of rewriting and rethinking of the program, in order to achieve the best possible performance. There are several methods to target the GPUs. One is native GPU languages such as CUDA (Compute

Unified Device Architecture) or HIP (Heterogeneous Computing Interface for Portability), respectively created by NVIDIA and AMD, which are both API models that target GPUs. The other option is to use directive based languages such as OpenACC or OpenMP.

Using one of the native languages allows for finer control of the performance, as one can target and utilize the specific hardware in use. The performance thus relies mainly on the programmer, rather than the compiler. The main problem with native languages is, that they do not allow much portability since none of the current native languages support all the hardware from the different vendors. They also require a separate source code, which may be very cumbersome to both produce and maintain. This requires the programmer to keep up to date with the ever-changing programming paradigms and requires a good understanding of the hardware and language. Though this may result in a great performing code, it is not preferable in most frameworks for two main reasons. 1) most modern frameworks are made to perform well independent of the system, as it may be run on many different systems with varying computing power and architectures. This is somewhat achievable on CPU-based systems, where the main headache is how memory is allocated and shared among nodes. To have consistent performance across GPU-systems, the framework must contain various versions, targeting hardware from the different vendors. Though this is possible, it will undoubtedly lead to a great increase in the extent of the framework code. 2) having specialized sections of code, which targets specific hardware requires a lot more maintenance, which can become a real headache if the framework run on all kinds of hardware. This will likely require updates in the code every time a new system is in use, in order to fine tune it to the specific hardware.

The lower maintenance approach is to instead use directive based languages, which were made to ease the creation of parallel code, or to port CPU-based code to GPUs, with (almost) no requirements to the programmer nor the system the code runs on. Directive based languages relies on directives, comment-like hints for the compiler, which then produces the necessary code upon compilation. Directive based languages thus relies more on the compiler, rather than the programmer and allows great portability between vendors. They are implemented directly in the CPU code and do not require a separate source code and are thus easier to both make and maintain. They do however not allow the same fine-tuned control of the performance (Robey et al., 2021). Compilers have improved a lot on this topic and some compilers can produce optimized code with performance similar to that of a CUDA program made by an intermediate programmer (Li et al., 2018).

As DISPATCH may be run on several kinds of hardware, it is thus favourable to port the code to GPUs using directive based languages and this will be the main focus of the next section. Since directive based languages do not explicitly use low-level languages, they do it implicitly as the compiler produces the code. It is thus preferable to have some understanding behind the GPU programming model to produce the best possible code.

2.2.1 The programming model

In Section 2.1.3 I described how the hardware executes several threads (warps) concurrently on the SMs. As the hardware contains several layers, which each provide various forms of parallelization, it is important to split up the data to utilize this. Using one of the native languages this is done explicitly by the programmer, whereas

using a directive based language it is done implicitly by the compiler. In either case, the basic programming model remains the same.

When doing GPU programming the CPU is referred to as the *host* and the GPU (or any accelerator) as the *device*, which is the terminology which shall be used in the rest of this thesis.

From a programming point of view, the computational domain is decomposed into a grid containing several independent blocks of data. A block consists of several threads which are then executed. The grid structure of the computational domain ensures that cache coherence is not a problem, as no two data elements will be in two different blocks. Each block may consist of up to 1024 threads, but usually they are smaller to obtain more memory per thread. Blocks have a local memory space, which is shared among the threads, which can thus only synchronize with threads within the same block. This means, that from a hardware point of view, blocks are limited to have all threads running on the same SM, where the threads are executed in warps, usually consisting of 32 threads (Robey et al., 2021).

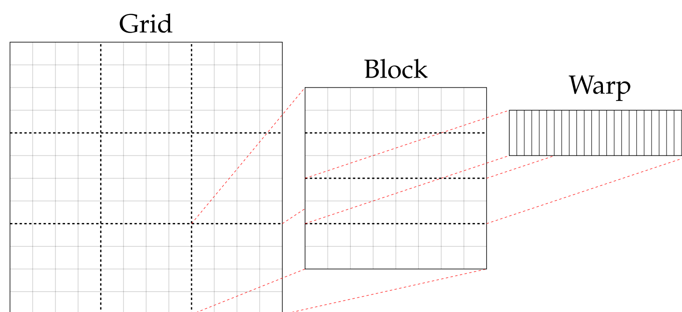

FIGURE 2.4: Programming abstractions for GPU hardware. The computational domain is decomposed into a grid, consisting of several blocks. Each block contains several threads which are executed in warps.

2.2.2 Directive based programming

Directives are language constructs, which through comment-like hints specifies to the compiler, how to handle specific regions of the code. The two main directive-based languages to support GPUs are OpenMP (Open Multi-processing) and OpenACC (Open Accelerators).

The first version of OpenACC was released in 2011, developed for heterogeneous computing and thus supported both CPU and GPU parallelism from the very beginning. The interpretation of the directives is handled by a compiler flag and it is thus not possible to produce parallel code for both CPUs and GPUs within the same program. It offers very fine-grained control, where the user can both control block-, warp- and thread-level parallelism. As it was developed mainly for GPUs from NVIDIA, the support to devices from other vendors, such as AMD, does not necessarily lead to the same increase in performance.

OpenMP on the other hand was first published in 1997 and was developed to turn sequential codes on shared-memory multiprocessor systems into parallel code. OpenMP did not support offloading to GPUs until 2015, where the OpenMP 4.5 version was released which included the `target` directive. Since offloading to a device is a directive of its own, OpenMP allows for both CPU and GPU parallelism within the same program. Furthermore, as the `target` directive was not developed for any

specific vendor, it is more independent of the hardware and thus offers similar performance across various vendors.

As mentioned, the directives work as hints to a compiler, which then produces the parallel code upon compilation. This means, that the level of support for various versions of the directive based languages will differ between compilers. Furthermore, the directive specifications may be interpreted differently across compilers, leading to different implementations (Diaz et al., 2019).

OpenACC is more well-developed, being older, but OpenMP has a broader industry support, and is expected to become more widely used in the future. Furthermore, OpenACC is not supported for all types of accelerators. Specifically, there is no compiler support for the AMD GPUs, giving 90% of the FLOPS of the LUMI supercomputer in Finland. It was therefore decided that DISPATCH should use OpenMP directives to enable efficient execution on GPUs.

OpenMP

OpenMP was meant to parallelize sequential code on shared-memory multiprocessor systems using directives or pragmas, but after the introduction of the `target` directive in OpenMP 4.5, offloading to GPUs have been supported. As the main focus of the thesis is porting code to a device, the primary focus will be on how one may use the `target` directive to do so. All the code is for Fortran, where the call to OpenMP directives happens with `!$omp`, where the C++ equivalent would be to use `#pragma`. To send a kernel to the device, one has to use the combination of constructs shown in Listing 2.1.

```
!$omp target teams distribute parallel do [clause, clause, ...]
do i=1,N
    do j=1,N
        !$omp simd [clause, clause, ...]
        do k=1,N
            ...
        enddo
    enddo
enddo
!$omp end target teams distribute parallel do
```

LISTING 2.1: Example of using the `target` directive to offload data and kernels to the device

Each construct has the following functionality:

- `target`: Executes the following code block on device
- `teams`: Creates a league of teams
- `distribute`: Distributes the work to the teams
- `parallel do`: Distributes the work to threads within a team
- `simd`: SIMD execution if the hardware allows it, otherwise it is ignored

Comparing this to the programming model, `teams distribute` roughly corresponds to the grid structure, where the work is divided into independent blocks and `parallel do simd` roughly corresponds to dividing work to threads within a block and executing them in parallel. The `target` directive creates a *target task*, which may be executed immediately by the thread or deferred to some later time.

The `clause` part of the directives allows the programmer to control how the compiler executes certain constructs. E.g. clauses such as `num_teams` sets the number of teams or `num_threads` that specifies the number of threads within a team. Though this may improve the performance on a specific hardware, it may result in worse performance when running on a different hardware. It is thus usually recommended to let the compiler optimize these parameters.

Other clauses control the data environment, such as `shared(list)`, `private(list)`, `firstprivate(list)` and `lastprivate(list)`. Using `shared(list)` will share the variables among all threads. `private(list)` creates a local variable for each thread that only exists within the parallel region. The variables are undefined upon entry and must be set within the parallel region. If they are defined prior to the parallel region, and one wishes to make a local copy for each thread with the given value upon entry, one must instead use the `firstprivate(list)` clause. If one instead wish to create a local variable for each thread, which will update the original value upon exit of the parallel region, one must use the `lastprivate(list)` clause. Variables defined before a parallel region will by default be shared and variables defined within a parallel region are by default private. It is good practice to specify all variables with one of the above clauses, as it makes the code less error prone.

If the loops are perfectly nested as in the example above, one may improve performance drastically by using the `collapse` clause, which collapses the nested loops into one iteration space.

The biggest impact on performance usually comes from the memory transfer between host and device. When entering the `target` region, scalars and statically allocated arrays are transferred to the device by default. Depending on the specific compiler, memory allocated on the heap is not necessarily transferred to the device by default. It is thus recommended to include the `map` clause, that specifies how the data should be mapped to and from the device. To obtain the best performance however, it is better to create data regions to avoid unnecessary data movement. This is handled by the `!$omp target data [map(to|tofrom|from)]` construct. This gives the programmer the control over the data movement and allows to run several kernels on the device, without having any transfers in between. This is referred to as a *structured* data region. It is also possible to use an *unstructured* data region, which is done by transferring data from host to device using `!$omp target enter data [map(alloc|to)]` and from device to host using `!$omp target exit data [map(from|release|delete)]`. Both constructs work in the same way, the unstructured region does however allow for more complex data management. One may also transfer specific data back and forth within a data region using `!$omp target update [to|from]`.

It is also possible to define data on the device, which exists throughout the duration of the program. This is done using `!$omp declare target (extended list)`. It is also necessary to declare functions or subroutines, which are called within a target region, with this directive. This is done by writing `!$omp declare target` as the first line in the subroutine/function (*OpenMP Application Programming Interface* 2015).

2.3 Compilers

Compilers are the link between the source-code and the computer-hardware, by translating instructions into machine-code, which can then be read and executed by the computer. They are especially important when using directive based languages, as the compiler thus not only translates, but also produces extra code based

on the directives. Many compilers exist, but not all support OpenMP, and the extent of the support across languages may differ. E.g. the GNU compiler, GCC, did not fully support OpenMP 4.5 for FORTRAN until GCC 11, released April 2021 - approximately 5 years after the OpenMP 4.5 release. The delay between the release of OpenMP versions and their support by compilers, may result in slightly different implementations across the compilers, as the specifications of the directives may be interpreted differently. Consequently, this may lead to variations of the specific behavior of a directive and varying performance across compilers. It is thus important to be aware of the level of OpenMP support of the compiler in use, and preferably have access to several compilers.

Diaz et al., 2019 evaluate the implementation of OpenMP 4.5 in different compilers (Clang, XL and GCC) on different systems. For most `target` directives the GCC compiler tends to have a greater overhead and in general performs worse than the other compilers. They conclude that:

> "by comparing release versions of different compilers, there seem to be more focus on reaching compliance with respect to the specifications, than there is with respects to drastically changing the possible runtime implementations, leading to the same overhead across multiple compiler versions" (Diaz et al., 2019)

Hence, despite that the evaluations were made using an older version of the GCC compiler than the one used in this thesis, similar performance for the different directives is to be expected.

2.4 HPC systems - Supercomputers

Modern supercomputers contain thousands of nodes. The hardware of a node may vary a lot but is typically made of a single multicore CPU with some interior NUMA architecture. Since GPUs provide much greater computation power, GPU based supercomputers have become very popular. These types of computers usually have much fewer, but more computationally powerful nodes, usually containing a single CPU and several GPUs.

As mentioned in section 2.1.1, nodes are connected via some interconnection network, which is used for communication and synchronization among nodes. Communication and synchronization of nodes is a key factor in high performance systems and may have a bad impact on the performance if not done properly. Communication is usually implemented using some Message Passing Interface (MPI). It is thus the programmer's responsibility to distribute memory and to schedule work on each node.

Communication and synchronization will create some overhead, which may partly be compensated by ensuring enough work is available on each node. In most cases there will however be situations where a lot of the hardware remains idle, while either waiting for data or at some synchronization point. For small systems this does not have a too significant impact, but as the systems become sufficiently big this is a potential issue for the performance of a program and requires great attention.

2.4.1 LUMI

LUMI is a supercomputer currently being built in Finland. It will become the third most powerful computer in the world and the most powerful in Europe. The GPU

partition, LUMI-G, will consist of 2560 nodes, each containing one 64-core AMD Trento CPU and four AMD MI250X GPUs, providing a potential peak performance above 550 PFLOPS/s (*LUMI's full system architecture revealed*).

LUMI was supposed to have been finished by February 2022, but is unfortunately delayed, with the new release being September 2022. Access have been given to an early access platform, which uses similar hardware to what will be used in LUMI. Tests were planned to be made on this system, to see how the code performed on a different system and with the CRAY compiler. Due to the late access and compiler issues, this has however not been possible to complete.

Chapter 3

DISPATCH

DISPATCH is a fairly new framework, which aims to solve the scalability issues other grid-based frameworks may suffer from (Dubey et al., 2014), by using locally determined time-stepping in each task, thus avoiding any global synchronization across MPI-ranks. This chapter gives a brief overview of the key ideas behind DISPATCH and the main features of the design.

3.1 Key ideas

DISPATCH is a relatively new task-based framework. For grid-based simulations the computational domain is split into small semi-independent tasks, in DISPATCH called *patches*. One of the key features is, that each patch contains data for several timeslots, which allows for individual and asynchronous time-stepping. Furthermore, each patch is small enough to fit into cache and allows for effective vectorization.

There are several MPI processes (usually one per. CPU socket) which 'owns' a subgroup of the tasks. An MPI rank owning a task means, that the specific rank is in charge of preparing and updating the task. The tasks within a rank may be divided into three groups. *Internal tasks* which only have neighbouring tasks, which are owned by the rank, *Boundary tasks* which are at the edge of the geometrical area, thus at the 'border' to a different ranks domain and *Virtual* tasks, which are owned by a different rank. The basic idea for a grid-based 2D problem is shown in figure 3.1, where each square symbolizes a task/patch. This model allows for mainly intra-node communication, and a single MPI process only has to communicate with a finite number of neighbouring ranks. With the asynchronous time-stepping, this allows for no global communication nor synchronization. This also means, that task scheduling is rank local and is handled via a time ordered ready queue, which ensures that tasks most in need for update are picked first. The framework makes sure that there are a lot more tasks than hardware threads, thus ensuring that threads keep busy throughout execution.

The exchange of ghost-zones between ranks happens through non-blocking MPI calls. With the described grouping of tasks within a rank, as a boundary task have been updated, *all* the patch data is sent to the neighbouring rank, instead of just the ghost-zone-data. This not only simplifies package creation, but also makes load-balancing a lot easier, as it only requires changing the status of a task from 'boundary' to 'virtual' on one rank, and from 'virtual' to 'boundary' on the neighbouring rank, hence no extra data exchange is needed.

The features just described, allows for potential unlimited OpenMP and MPI scaling.

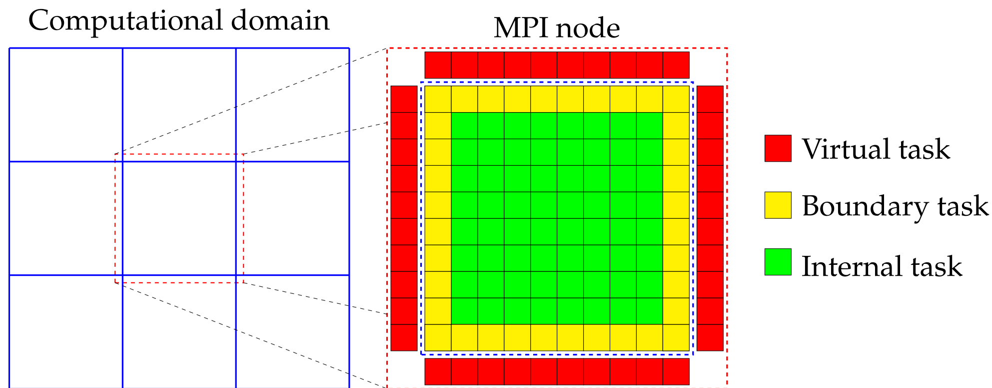

FIGURE 3.1: A 2D computational domain is distributed to MPI processes. Each MPI process owns the tasks within the blue dashed square, which is divided in two groups *boundary tasks*(yellow) and *internal tasks*(green). The MPI rank also holds information of boundary tasks from neighbouring ranks, which constitute a third group of *boundary tasks*(red). The exchange of tasks between ranks happens by simply changing a virtual task to a boundary task and vice versa.

3.2 Tasks

As mentioned, DISPATCH runs a simulation, by splitting the problem into several smaller tasks. A *task* is simply a derived type(object) in the FORTRAN language and is not to be confused with an OpenMP task. The framework relies heavily on inheritance and the basic task object extends to several other extended types, such as *patches*. A class is built up as a hierarchy. The base type is the *task* data type. It carries the basic information such as task ID, number of time slots, status flags and methods which extended types must implement, e.g. acquiring unique IDs for spawned tasks. For grid-based simulations a task is extended to the *patch* data type, which contains information concerning spatial properties, such as coordinate system, dimensions, guard zones and physical variables, and methods to measure intersections in space and time with other patches. Next is the *solver* data type, which specifies which solver to use for the specific task, which physical variables that are to be advanced in time and solver-specific parameters. The solver type is then extended to an *experiment* data type which adds experiment-specific procedures which e.g. set the initial and boundary conditions.

Tasks are organized into a *task list*, which is another derived type. The basic type of a *task list* is called a *node*, which contains pointers to the tasks and information about the neighbours which a specific task depends upon. The neighbour concept is thus not limited to spatial proximity, and a grid-based task(patch) can thus casually depend on a radiative transfer task and exchange the data needed for the update of each task. Hence, the neighbour concept is better described as a 'dependence scheme' rather than an actual neighbour, and a task thus only depends on a finite number of other tasks.

That tasks depend on each other, usually implies some sort of exchange of data before an update may be performed. A task updates using locally determined time steps, which are independent of the time steps of other tasks. This means that neighbouring tasks will usually not be at the exact same time. Hence, when exchanging ghost-zones, some interpolation or extrapolation in time is usually necessary. Since each task has access to multiple time slices, this is straightforward. Upon execution,

tasks ready for update are put in a time ordered ready-queue, which further ensures that the oldest tasks are updated first.

3.3 Offloading

Some parts of DISPATCH have been prepared for execution on GPUs and modules for offloading data have thus been developed. To limit the transfer back and forth between the CPU and the GPU (which is usually the main bottleneck for GPU applications), tasks are grouped together in what is referred to as a *bunch*. When a bunch is full, all the tasks are transferred and updated on the GPU. The number of tasks which a bunch may hold is set in the beginning of the program. This allows to allocate the needed memory on the GPU during the initialization of the program, where it stays allocated throughout execution. Before the kernel is sent to the GPU, the allocated memory on the device is updated with the new tasks using `!$omp target update`.

In situations where not enough patches are ready to completely fill a bunch, the program would not be able to continue further. To avoid this, a time limit between updates of bunches have been implemented. The time it takes to fill a bunch is measured during the first time it is used and works as an upper time limit between updates of a bunch throughout the program - making sure, that an update will be happen, even with the lack of ready patches (Haarh, 2021).

With this setup, solvers in DISPATCH must be prepared for updating a bunch rather than a single task. Currently it is only the DISPATCH/RAMSES solver which have been ported to run, using the just described bunching.

The current implementation only allows to use one device, even though several devices may be present.

3.4 Task scheduling

The task scheduling within DISPATCH is one of the key functionalities, which allows the good performance and scalability. For simplicity, I shall only focus on the task scheduling within a node.

As tasks become ready for update, they are placed in a time-ordered ready-queue. Whether a task is ready for update or not, mainly depends on the local time of neighbouring tasks. As mentioned, parts of the code have been prepared for GPUs and the task scheduling thus happens slightly different, depending on whether the code are to be executed using GPUs or CPUs. A flowchart of a simplified model of the intra-node task scheduling is shown in figure 3.2. When executing on CPUs the execution flow basically follows everything within the red dashed square and when executing on the GPU the execution flow follows the steps within the blue square. The main difference between the execution models is how, when and where a task is updated. If executed on CPUs(`offload=.false.`), the thread will prepare the task for update through a pre-update method. This is where the time step is computed and physical variables, such as the gravitational potential. After this the relevant solver is used to advance the task in time. After updating the task, a post-process method is called, where e.g. the gravitational potential is updated using the updated values from the solver. The thread then checks if neighbouring tasks have become ready due to the update, if so, they are placed in the ready queue. This also happens in the post-process step.

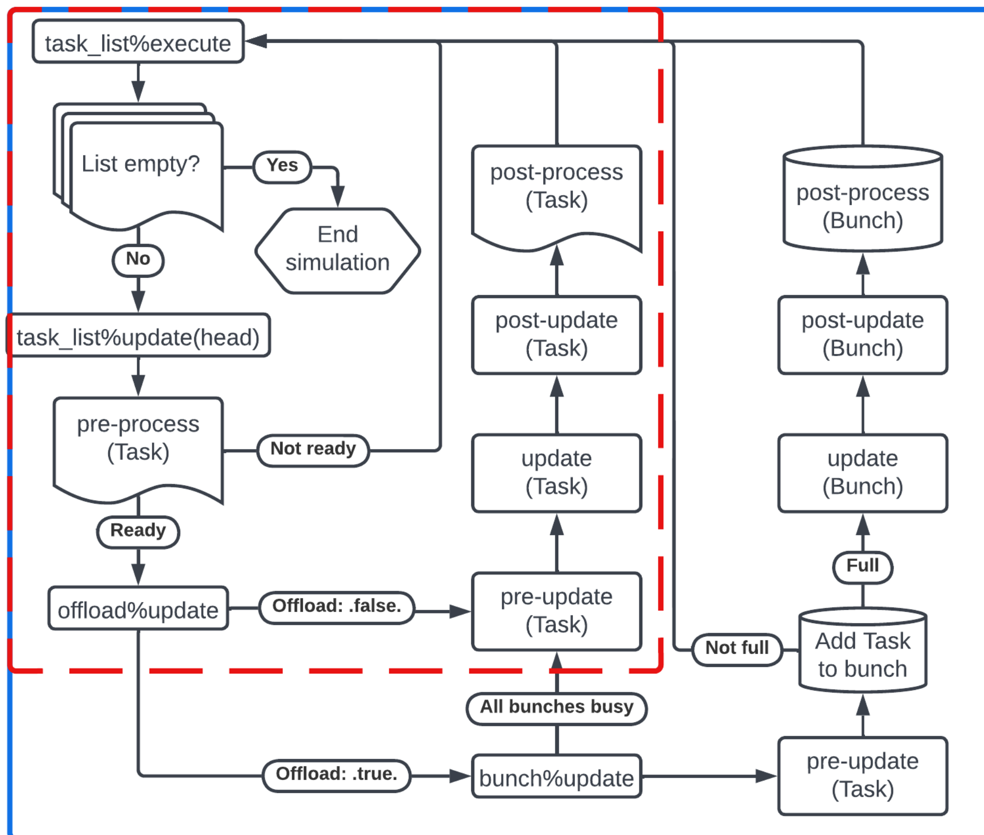

FIGURE 3.2: Simplified flow chart of the intra-node execution in DISPATCH. Everything within the red dashed square is the execution flow when executing on CPUs and everything within the blue square is the execution flow when executing on GPUs.

If executed on GPUs (`offload=.true.`), the thread will call `bunch\%update`, which checks whether all bunches are busy and if so, it will update the task on the CPU (if enabled). If that is not the case, the thread will go through the pre-update routine, before placing the task in the bunch. If the bunch is still not full, the thread will finish and continue to a new task. If the bunch is full, the bunch is transferred to the GPU and all the tasks are advanced in time. After the update, the updated values are copied from the bunch back to the tasks on the CPU and the post-update method is called, where the gravitational potential is computed using the updated values after which the post-process procedure is performed for all tasks.

The task scheduling may happen in different modes. The default mode lets threads within a node 'pick' the oldest task in the ready-queue and perform the update method. The check whether a task is ready or not, happens in the pre-process step. This requires critical regions, both when a thread pops or adds tasks from/to the ready queue. For current available hardware, the number of threads within a node is not big enough to create significant overhead during this process. A different mode makes a single thread in charge of removing and adding tasks to the ready queue. As tasks are removed, an OpenMP thread is spawned using `!$omp task` constructs, which then updates the task. This mode is more complex, but allows for better scalability, as no critical regions are needed, and may be preferable in the future, when the number of cores per node grows larger.

Chapter 4

Theory

Gravitation plays a role in most of the dynamics in the universe and is thus vital if one wants to make simulations of such. In this chapter, the basic theory that describes gravity is presented. The gravitational potential is described by the Poisson equation. Many numerical methods for solving the Poisson equation exist. For most computer simulations iterative methods have to be used, hence below is a brief overview of various iterative methods, which may be used to compute the gravitational potential given some mass distribution. Finally, a brief description of the algorithm used to solve selfgravity in DISPATCH is presented.

4.1 Selfgravity

The gravitational force between two objects in space is described by Newtons law of gravity. In a two-body system, the gravitational force applied on body one with mass m_1 located at position $\mathbf{r}_1 = (x_1, y_1, z_1)$, may be described as an interaction with a gravitational field, $\mathbf{g}_2(\mathbf{r}_1)$ exerted by body two with mass m_2 located at position $\mathbf{r}_2 = (x_2, y_2, z_2)$

$$\mathbf{F}_{12} = m_1 \mathbf{g}_2(\mathbf{r}_1), \qquad \mathbf{g}_2(\mathbf{r}_1) = -G m_2 \frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} \tag{4.1}$$

where G is the Universal gravitational constant. With only two bodies, the problem is rather simple and easy to compute. Often one is however more interested in systems containing several bodies, which are usually represented as point masses. Computing the net-force applied on a point mass with mass m located at position $\mathbf{r} = (x, y, z)$, exerted by N point masses with individual masses m_i at locations $\mathbf{r}_i = (x_i, y_i, z_i)$ would just be a sum of the force exerted by each point mass. In other words, the force applied on a point mass may be described as an interaction with a collective gravitational field, $\mathbf{g}(\mathbf{r})$, exerted by all the other point masses.

$$\mathbf{F} = \sum_{i=1}^{N} \mathbf{F}_i = m \sum_{i=1}^{N} \mathbf{g}_i(\mathbf{r}) = m\mathbf{g}(\mathbf{r}) \tag{4.2}$$

$$\mathbf{g}(\mathbf{r}) = -G \sum_{i=1}^{N} m_i \frac{\mathbf{r}_i - \mathbf{r}}{|\mathbf{r}_i - \mathbf{r}|^3} \tag{4.3}$$

Since gravitation is a conservative force, the gravitational field may be written as the gradient of a scalar potential

$$\mathbf{g}(\mathbf{r}) = \nabla\Phi(\mathbf{r}) \tag{4.4}$$

where Φ is defined as the gravitational potential. Furthermore, since the gravitational field of point mass i is independent of the gravitational field of point mass j,

the collective gravitational potential is simply the sum of the gravitational potentials for each point mass. The collective gravitational potential is thus

$$\Phi(\mathbf{r}) = -G\sum_{i=1}^{N}\frac{m_i}{|\mathbf{r}_i - \mathbf{r}|} \tag{4.5}$$

When simulating the dynamics of the universe, it is often inconvenient to work with collections of point masses and one instead uses a continuous mass distribution over the entire space. In 3D the mass located at position vector $\mathbf{r}'$ may thus be represented as $\rho(\mathbf{r}')d^3\mathbf{r}'$, where $\rho(\mathbf{r}')$ is the local mass density and $d^3\mathbf{r}'$ is a small volume element. Summing over the entire space and letting $d^3\mathbf{r}' \to 0$ yields a collective gravitational field of

$$\mathbf{g}(\mathbf{r}) = -G\int \rho(\mathbf{r}')\frac{(\mathbf{r}-\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|^3}d^3\mathbf{r}' \tag{4.6}$$

Taking the divergence on both sides and integrating from $-\infty$ to ∞ yields

$$\nabla\cdot\mathbf{g}(\mathbf{r}) = -4\pi G\rho(\mathbf{r}) \tag{4.7}$$

Using (4.4) finally yields

$$\nabla^2\Phi(\mathbf{r}) = -4\pi G\rho(\mathbf{r}) \tag{4.8}$$

where ∇^2 is the Laplacian operator. This is also known as the Poisson equation, which is used in numerical simulations involving selfgravity to solve for the gravitational potential based on some mass distribution, from which the gravitational forces can be computed.

Poisson's equation is a second-order partial differential, elliptical equation with Dirichlet boundary conditions. The term on the RHS is usually referred to as the source. In general, there is no analytically solution to the problem and one is thus forced to use numerical methods to find an approximate solution. Many different methods exists, which can be separated in two overall groups: *direct methods*, such as Fourier transforms, direct N-body calculations and *iterative methods*, such as relaxation methods e.g. Gauss Seidel and successive over relaxation (SOR).

DISPATCH has implemented two methods - conjugate gradient (CG) with preconditioner and SOR with Chebyshev acceleration. Both methods have similar costs, but since the SOR method is simpler it was chosen for the GPU implementation. Thus the CG method will not be explained further (but see Ramsey et al., 2018).

4.1.1 Iterative methods

The iterative methods are used on the differential expression of the Poisson equation (4.8). The problem may be solved in different ways, the simplest and most common being to introduce a pseudo-time τ and make the problem pseudo-time-dependent, allowing one to rewrite (4.8) as

$$\frac{\partial\Phi}{\partial\tau} = 4\pi G\rho - \nabla^2\Phi \tag{4.9}$$

This is basically a diffusion equation where one assumes that ρ is a constant of pseudo-time and Φ is a function of pseudo-time. The function is then iterated until $\partial\Phi/\partial\tau = 0$, where the Poisson equation is satisfied.

By discretizing pseudo-time and space, the difference equation to solve is (in 2D Cartesian coordinates)

$$\frac{\Phi_{i,j}^{n+1} - \Phi_{i,j}^{n}}{\Delta\tau} = 4\pi G\rho_{i,j} - \frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{(\Delta x)^2} + \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{(\Delta y)^2} \tag{4.10}$$

where Δx and Δy are the grid spacing, i and j are the indices in the x- and y-direction, $\Delta\tau$ is the pseudo-timestep and n and $n+1$ indicates the current and the updated potentials respectively. As for any diffusion problem, the grid size puts a limit to the allowed pseudo-time step. The maximum allowed time step in a problem with d dimensions and the same grid spacing in all directions, is governed by

$$\Delta\tau \leq \frac{\Delta x^2}{2d} \tag{4.11}$$

The potential, which is a solution to this problem, only depends on the chosen boundary conditions and the source term, coming from the density.

Jacobi iterations and Gauss-Seidel

The most simple method to solve (4.9) is using Jacobi iterations. In the Jacobi method, one usually uses the biggest allowed timestep, thus setting (in 2D) $\Delta\tau = 0.25(\Delta x)^2$. Assuming $\Delta x = \Delta y$ and rewriting (4.10) one thus arrives at

$$\Phi_{i,j}^{n+1} = 0.25\left(\Phi_{i+1,j}^{n} + \Phi_{i-1,j}^{n} + \Phi_{i,j+1}^{n} + \Phi_{i,j-1}^{n} - 4\pi G\rho_{i,j}(\Delta x)^2\right) \tag{4.12}$$

which allows one to compute the updated value, based on the neighbouring values and the source. This is repeated until convergence. That is (4.9) has reached a steady state and Φ is a solution to (4.8). Though this method is very easy to implement, it converges very slowly (scales like N^2) and it requires a separate array for storing updated values. Furthermore, the update of a single cell only depends on the neighbouring cells and thus only updates the shortest wavelengths efficiently. The method is thus very sensitive to the initial guess on the large-scale modes.

Rather than storing the updated values in a separate array, one might as well store the updated values directly in the original array and use them to update the neighbouring values. This would mean, that in (4.12) the $\Phi_{i-1,j}^{n}$ and $\Phi_{i,j-1}^{n}$ would instead be $\Phi_{i-1,j}^{n+1}$ and $\Phi_{i,j-1}^{n+1}$ respectively. This is known as the Gauss-Seidel method, which increases the convergence rate with a factor of about two. It is however still too slow to be used in most practical simulations (Bodenheimer, 2006).

Red-black iteration

The convergence rate may be further improved by realising that each cell only depends on the closest neighbours in the x-, y- and z-directions - the points are coupled as a checkerboard. This is known as red-black ordering, which allows one to update the black cells first, then use the updated black cells to update the red cells. This process is continued until convergence, as illustrated in figure 4.1. Like the Gauss-Seidel method, this does not require an extra memory array and thus requires less memory (Press et al., 1992).

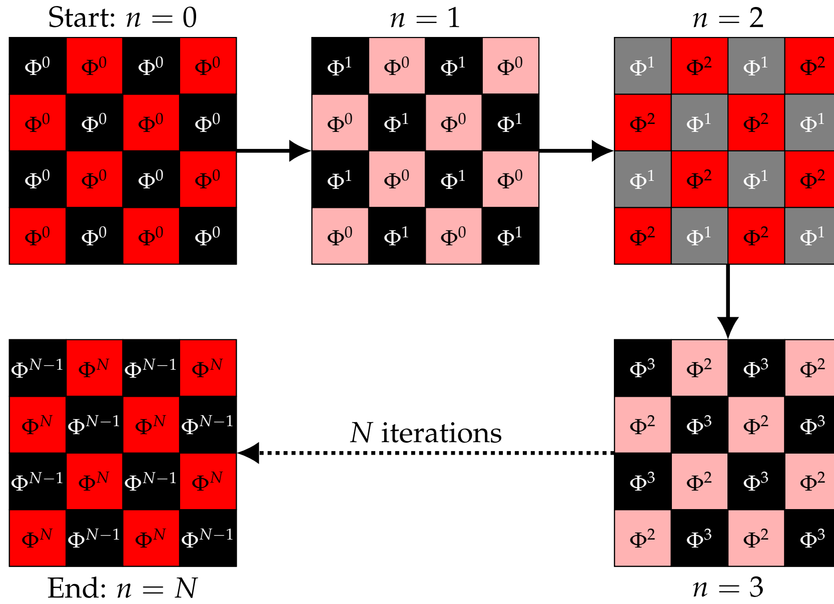

FIGURE 4.1: Red/Black iteration. During the first iteration all the black cells are updated. During the second update all the red cells are updated, using the updated black cells. This procedure continues until convergence after N iterations.

Successive Overrelaxation

Instead of performing a full update at each iteration, as is done with Jacobi and Gauss-Seidel, it is better to try to extrapolate each step, by introducing a factor ω and making the updated value a linear combination of Φ^n and Φ^{n+1} as shown in (4.13).

$$\Phi_{i,j}^{update} = (1-\omega)\Phi_{i,j}^{n} + \omega\Phi_{i,j}^{n+1} \tag{4.13}$$

Setting $\omega < 1$ is called underrelaxation and may be used to stabilize unstable methods. Setting $1 < \omega < 2$ is called overrelaxation, which accelerates the convergence rate. If ω is greater than two, the procedure becomes unstable and small-scale errors will grow. The optimal value of ω depends on the problem. For a uniform Cartesian grid, the optimal value is

$$\omega = \frac{2}{1 + (1 - r_s^2)^{1/2}} \tag{4.14}$$

where r_s is the spectral radius. With the same number of points and grid spacing in each direction, r_s is given by

$$r_s = \frac{1}{d}\sum_{i=1}^{d} \cos\left(\frac{\pi}{N_i}\right) \tag{4.15}$$

where d is the dimensions and N_i is the number of grid points in each direction of the grid. The method scales linearly with the number of grid points and thus often the method of choice when using iterative methods on a grid.

If using red-black iterations, one may further improve the convergence rate using Chebyshev acceleration (Press et al., 1992).

4.1.2 Multi-grid methods

Iterative methods are very powerful to find a solution to the Poisson equation. However, as they only solve efficiently at the shortest wavelength (the wavelengths at the scale of the grid), they are very sensitive to the initial guess of the large scale modes, which may be a problem in practice as the problem converges very slowly. To account for this, multigrid methods may be used as an extension to the iterative methods, which may speed up the convergence rate. To do so, a reformulation of the problem is usually used.

The Poisson equation may be written as a linear equation

$$\nabla^2\Phi = 4\pi G\rho \Leftrightarrow Av = S \tag{4.16}$$

where A is an extremely sparse matrix representing the discrete Laplace operator, v is the approximate gravitational potential and S is the source term. We may then define the error e as the difference between an exact solution for the gravitational potential, u, and the approximate solution. We further define the residual r as the difference between the source and Av. We thus have

$$\begin{aligned} e &= u - v \\ r &= S - Av \end{aligned}$$

Even though we do not know the exact solution, and thus not the error, we may write an equation for it

$$Ae = A(u - v) = S - Av = r \tag{4.17}$$

The idea is to create a series of coarser and coarser grids on top of the basic grid. The procedure is to perform a few iterations on the finest grid, thus smoothing out the error, and compute the residual. The residual is then restricted to a coarser grid, where one uses (4.17) to solve for the error and then updates the approximate solution and computes the new residual. This is repeated on increasingly coarser and coarser grids. After reaching the coarsest grid, the error is then prolonged back to the finest level, where the approximate solution is updated using the residual. If the solution is not good enough, the procedure is then repeated until convergence on the finest grid. Several procedures exist, the simplest being the V-cycle, where one propagates from the finest to the coarsest and back. Other common procedures are the F-cycle and the W-cycle (Bodenheimer, 2006).

4.1.3 Solving Poisson's equation in DISPATCH

Most problems in astrophysics have a large dynamic range, which in general requires a lot of global communication and synchronization, which makes simulations very expensive. Some methods have been developed, such as adaptive mesh refinement (Berger et al., 1989) which allows to use high resolution in areas where it is needed, thus keeping the cost down to a minimum for the specific problem. Despite the efforts, the methods still require a lot of global communication and synchronization to make all domains agree on the solution. This inevitable affects the scalability of the code and thus the performance for increasingly bigger problems. The structure of DISPATCH, described in chapter 3, allows the framework to overcome these problems, by splitting the domain into semi-independent patches with

data from multiple time steps. This is utilized to solve the Poisson equation in an effective manner with no global communication and blocking synchronization across domains.

Solving the Poisson equation in DISPATCH can be split up into three main steps. Firstly, up-to-date data of the mass distribution and the gravitational potential is 'downloaded' from neighbouring patches. When all the needed data is present, the Poisson equation is solved within the given patch, from which the gravitational acceleration is computed and used as a source term in the MHD solver. As the last step, the Poisson equation is solved again, now using the just updated density from the MHD routines, and the new solution is stored in the next time slot as a forward prediction - allowing neighbouring patches to fill their ghost zones and proceed with their updates. In this step the ghost zones are not yet known at the new time, so they are instead estimated by forward extrapolation in time of previous values. The first step of the next timestep repeats this solution, with by then up-to-date ghost zone values.

The download procedure either takes the form of prolongation or restriction, depending on the specific grid arrangement. In the first case linear interpolation in time and space is used on neighbouring patches to fill the ghost zones. In the second case, linear averaging of child patches with finer resolution is used to fill the ghost zones. Since each patch has individual time stepping, neighbouring patches will in general not be at the exact same time. Thus, to exchange ghost zones, some time interpolation or extrapolation is required.

The computational procedure is as follows. In periodic cases one has to renormalize the problem, by offsetting the density ρ in the source term by the global average, to avoid divergent solutions. The periodic problem is thus given by

$$\nabla^2\Phi = 4\pi G(\rho - \langle\rho\rangle) = 4\pi G\rho' \tag{4.18}$$

where $\langle\rho\rangle$ is the average density over the entire domain. The average density is computed from the initial conditions and does thus not require a global summation.

The equation is solved using iterative methods on the form

$$4\pi G R = \nabla^2\Phi - 4\pi G\rho' \tag{4.19}$$

where R is the residual - the adjustment required in mass to make Φ the exact solution. Though the goal is to reduce the LHS to zero, it is common practice to instead reduce its magnitude below some pre-set tolerance. In DISPATCH convergence is reached when R fulfills

$$\varepsilon = \frac{R}{\rho + \rho_0} \tag{4.20}$$

where ρ_0 is a floor value, characterising the desired tolerance in low density regions.

For non-periodic cases the procedure would be similar, but instead using (4.8) as the starting point, thus not offsetting by the global average density.

After computing the potential, the gravitational acceleration can be obtained from (4.4), which is then used in other solvers (Ramsey et al., 2018).

4.1.4 Gravity at different scales

Gravity is in theory a global problem, as all masses within a system affects one another, which is why the solution in general needs to be synchronized across the entire domain. However, in most cases the gravitational potential is smooth and slowly varying and a change in the mass distribution will only have a small effect on the

large-scale modes of the gravitational potential. As such, in DISPATCH the problem is solved locally in each patch, using boundary conditions from the neighbouring patches and the global synchronization is left out. The error on the solution is monitored via the residual. Increasing errors are usually due to too big changes in the mass distribution per time step, which may be solved by making the local time step smaller.

An argument for why this works may be provided by looking at how the gravitational potential changes at different scales. The gravitational potential depends on some mass distribution. The continuity equation tells us how the mass distribution of a fluid changes. Looking at the Fourier transform of the continuity equation, provides some information about how this happens at different scales. This yields

$$\partial_t \rho = -\nabla \cdot (\rho \overline{v}) \quad \xrightarrow{\mathcal{F}} \quad \partial_t \rho_k = i\overline{k} \cdot \rho_k \overline{u}_k \tag{4.21}$$

where ρ_k is the density at some scale, k is the wave number for some scale and $\overline{u}_k$ is the velocity of the fluid at that scale.

Doing the same for the Poisson equation and taking the time derivative yields

$$\nabla^2 \Phi = 4\pi G \rho \quad \xrightarrow{\mathcal{F}} \quad -k^2 \Phi_k = 4\pi G \rho_k \tag{4.22}$$

$$-k^2 \dot{\Phi}_k = 4\pi G i \overline{k} \rho_k \overline{u}_k \tag{4.23}$$

The magnitude of the relative change in gravitational potential at different scales is

$$\frac{\dot{\Phi}}{\Phi} = i\overline{k} \cdot \overline{u}_k \Rightarrow \left| \frac{\Delta \Phi_k}{\Phi_k} \right| = \Delta t \cdot |\overline{k} \cdot \overline{u}_k| \tag{4.24}$$

The relative change in the potential at some scale is thus proportional to the velocity of the mass at that scale. To put this into context, I shall look at two relevant setups - molecular clouds and a central potential.

Larsons scaling relationship (Heyer et al., 2009) roughly states that the velocity dispersion, u_L is proportional to the square root of the cloud size, L. The wavenumber is given by $k = 2\pi/\lambda$ with $\lambda \propto L$. Thus, for molecular clouds the velocity scales as $u_k \propto k^{-1/2}$. Using this in (4.24) yields that the relative change in the gravitational potential scales like $\Delta t \cdot k^{1/2}$.

At the other end of the scale we have a central potential, where the velocity is given by $u = \sqrt{GM/R}$ and thus scales as $u_k \propto k^{1/2}$. Inserting this in (4.24) yields that the relative change in the gravitational potential scales as $\Delta t \cdot k^{3/2}$. In either case we may conclude that the relative change on the greater scales is much smaller than that of the small scales, which supports the method used in DISPATCH, where synchronization of the global solution from small scales to large scales is ignored.

Chapter 5

Implementation

Instead of preparing the main code for the GPUs directly, my supervisors made a mockup-file of the Truelove experiment, which contained the main modules of DISPATCH, which were needed to run the experiment. This allowed me to prepare small prototypes for the various modules. The implementation was split up in several steps, implementing one part at a time. After having a working version, several improvements were made to obtain the final versions.

5.1 Mockups

5.1.1 poisson_mock.f90

The 'original' version which only uses CPUs. It consists of components from the `params_mod`, `patch_mod`, `poisson_mod` and `experiment_mod`, which are needed in order to run the experiment. The program makes one update of the gravitational potential Φ based on the density distribution ρ using the SOR method.

5.1.2 GPU-implementation - pre-steps

Step1.f90

As the first step I added a `bunch_mod` to the mockup. The idea was to utilize the already implemented parallelism of DISPATCH to fill bunches of data, which when full is transferred to the GPU and updated. This can easily lead to race conditions, where several threads try to write to the same place in memory simultaneously. To avoid this, I added two subroutines `init` and `init_thread`. The first subroutine initializes a 6-dimensional memory array indexed as `memb(x,y,z,iv,ip,ib)`, with `iv` being the variable index, `ip` the patch index and `ib` is the bunch index.

In the latter subroutine, a thread will enter a critical region where it will be assigned a location in the memory array, via a local patch ID, `lpid` and local bunch ID, `lbid`. An example is shown in listing 5.1. The critical region ensures that no two threads will be assigned the same location in the memory array, thus avoiding race conditions. When a bunch is full, the next thread will automatically go to the next bunch, until all bunches are full, at this point the thread will suspend the current OpenMP task, and work on a different OpenMP task, until the bunch is free.

The idea was to make an effective way of assigning threads a location in the memory array, while avoiding too much idle time and thus keeping as many threads occupied as possible. It was later discovered, that having the `!$omp taskyield` call within the critical region would keep the region closed for other threads to enter and get a location in the memory array. As it will be shown, this was solved by introducing extra queue-numbers, which allowed to move the `!$omp taskyield` call outside the critical region.

```
SUBROUTINE init_thread (self, lpid, lbid)
    class(bunch_t) :: self
    integer :: lpid, lbid

    !$omp critical (thread_init)
      lpid = pid
      pid = pid + 1
      if (lpid > nb ) then
        ib = ib + 1
        if (ib>num_bunch) ib = 1
        do while(.not. bunches(ib)%empty)
          !$omp taskyield
        enddo
        pid = 2
        lpid = 1
      endif
      if (lpid==1) bunches(ib)%empty = .false.
      lbid = ib
      !$omp end critical (thread_init)

      END SUBROUTINE init_thread
```

LISTING 5.1: Assigning memory locations to thread

Step2.f90 and Step3.f90

In `Step2.f90` a subroutine `copy_to_bunch` was added to the `bunch_mod` which transfers patch-data to the memory array, using the memory locations, `lpid` and `lbid`, assigned to the thread in the `init_thread` subroutine.

In `Step3.f90` two subroutines, `update` and `copy_from_bunch`, were added. The latter transfers the updated patch-data from the memory array back to the relevant patch. The `update` subroutine is called by all threads, however only the last thread in a specific bunch to make the call, will perform the actual update of the bunch and then transfer back the patch-data. A pseudo-code example of the method is shown in listing 5.2.

```
SUBROUTINE update(self, lbid)
    !$omp atomic capture
        bunches(lbid)%ready = bunches(lbid)%ready + 1
        lready = bunches(lbid)%ready
    !$omp end atomic

    if (MOD(lready,nb)==0) then
        update values in bunch
        call self%copy_from_bunch(lbid)
    endif
END SUBROUTINE update
```

LISTING 5.2: Pseudo-code example of `update` subroutine

Step4.f90

The previous steps worked as preparation of the CPU data, to be transferred easily to the GPU. The transfer can be made in many ways, where some are more effective than others. In general, it is very costly to allocate memory on the GPU. Using a memory array on the CPU to store all the patch data allows one to allocate the needed memory on the GPU in the very beginning of the program, using the

`!$omp target enter data map(alloc:memb)` directive. This makes a direct connection between the memory on the host and the device, allowing to update the device or host memory, without allocating memory at the same time, using the `!$omp target update` directive.

Furthermore, it is preferable to define various constants and functions, which are to be used on the GPU, from the very beginning. This can easily be done using the `!$omp declare target` directive.

In `Step4.f90` the subroutines `init_offload` and `finalize_offload` were introduced, which allocates and deallocates memory on the GPU, thus preparing the program for an effective data transfer.

Step5.f90

Now that the memory had been transferred to the GPU, I wanted to prepare the code for cases, where GPUs were not accessible. I thus added the two subroutines `update_host` and `update_device` which were to be called from the subroutine, depending on whether offload is on or off. Both subroutines will update the data in the memory array.

OpenMP only allows one kernel call per device. I thus had to add a variable, which tells the thread, which is to update on the device, whether the device is accessible or not. If the device is occupied, the thread will suspend the OpenMP task, until the device is free. The part of the code which takes care of this is shown in listing 5.3. Once again, this was done with `!$omp taskyield` inside a critical region, which was later changed to a queue system, in order to avoid idle time.

```
if (offload) then
    !$omp critical (device_ready)
    do while(GPU_active)
        !$omp taskyield
    enddo
    GPU_active = .true.
    !$omp end critical (device_ready)

    call bunch%update_device(lbid)
    GPU_active = .false.
else
    call bunch%update_host(lbid)
endif
```

LISTING 5.3: Update function with offload

Step6.f90

In `Step6.f90` the SOR-method was implemented to be run on the device. The function was defined using `!$omp declare target`, allowing it to be called from a `target`-region. The memory array on the device is firstly updated with the host data, after which the patches in the memory array are evenly distributed to all the SM's, which will then execute the `SOR_bunch` subroutine as shown in listing 5.4. When the kernel has finished, the memory array on the host is updated with the device data.

The implementation did execute on the GPU, but the performance was very poor. For small systems it performed worse than the CPU-version. For larger systems the performance was similar to the CPU-version. Furthermore, if the total number of patches were not a multiple of the bunch size, some patches were not updated, since a bunch is only updated when full.

```
1 !$omp target update to(memb(:,:,:,:,:,lbid), ds)
2 !$omp target teams distribute
3 do idx =1,nb
4   call bunch%sor_bunch(idx, lbid)
5 enddo
6 !$omp end target teams distribute
7 !$omp target update from(memb(:,:,:,:,:,lbid))
```

LISTING 5.4: Call to `SOR_bunch`

Profiling

A profile of the `Step6.f90` was made using Nsight systems, executing with eight cores and one GPU. The setting was set to use three bunches with 108 patches in each. The timeline is shown in Figure 5.1. The eight rows at the top show when the cores are active, indicated with black. The activity of the cores is very fractured and they seem to be idle a lot of the time. This is likely due to the `!$omp taskyield` inside the critical regions, especially in the `init_thread` routine, which do not allow threads to assign patches to bunches while all bunches are full.

The green and red lines at the bottom indicate data transfers between host and device - green indicating host to device(HtD) and red indicating device to host(DtH) transfers. The blue lines indicate the execution of a kernel on the device. The GPU seems to be active most of the time - either with a data transfer or kernel execution. The data transfers are not necessarily scheduled right before or immediately after the kernel execution, which is a consequence of the `target` directive being a task-generating directive. From the timeline below, it can be seen that sometimes several data transfers are executed in between kernel executions, and the DtH transfer does not necessarily happen right after, which makes the cores stay idle even longer. The time it takes to fill a bunch is much smaller than the time it takes to execute a kernel, which therefore hides the overhead in this specific example.

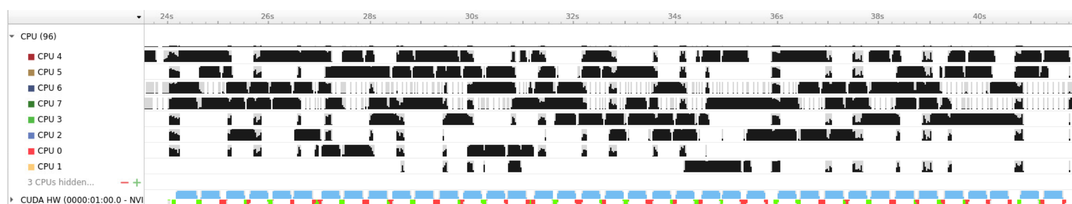

FIGURE 5.1: Profile of `Step6.f90` using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device.

A profile using Nsight Compute was made, to investigate the kernels in more detail. The `SOR_bunch` routine only obtained 17% of the peak performance throughput of the SMs and only issued two warps per scheduler - thus only having an occupancy of 12.5%.

5.1.3 GPU-implementation - final versions

Even though the SOR-routine did execute on the GPU, it was very inefficient compared to the CPU-version. I thus put a lot of effort into improving the performance.

Furthermore, as it is currently not possible to run on several GPUs in DISPATCH, I wanted to create a setup, which would allow this.

To do so I made a setup which uses the ideas of each step described in the previous section and mixed it with some of the current setup in DISPATCH. Three derived types were introduced. A *bunch* type, which contains memory arrays to store patch data and a patch list, which hold pointers to all the patches, which have data stored in the bunch. It furthermore contains procedures, which copies to and from bunches, copies to and from the device and updates the bunch on the device. The bunch memory is allocated on the device during initialization and updated with new patches during execution.

Next is the *device* type. It holds a number of bunches, and a device-ID which is used in the OpenMP calls to specify which device to use. It also keeps track of the number of active bunches and how many patches have been updated. Lastly is the *device handler* type. The device handler contains a device list of all the devices, the number of bunches per device and the size of the bunches. It contains procedures which assigns a device-ID, bunch-ID, bunch-slot and queue number to each thread, and updates the leftover patches, if necessary.

The SOR-bunch routine is moved to the `poisson_mod` and is called from the update routine of the bunch type. A procedure has also been added to the `poisson_mod`, which updates constants used in the SOR-routine on each device. This is done during the initialization of the program.

Several attempts were made to improve the throughput of the `SOR_bunch` routine. The source term was previously computed on the CPU and transferred to the GPU using the memory array. In the new version it was instead computed as part of the `SOR_bunch` routine. Small changes to the SOR iterations were added and several clauses were added to the parallel region, such as `schedule(static)`. The loops were also slightly changed, making them perfectly nested.

OMP-locks are used instead of critical regions, at places where `!$omp atomic` is not possible to use. To avoid having the `!$omp taskyield` call within a critical region (or a locked region) a queue number is introduced. This allows the device handler, to assign a device, bunch, and slot to threads, even though all bunches may be busy. The thread is not allowed to copy data to the bunch, before the queue number of the patch matches that of the bunch. The same is done for the execution on the device. When a bunch is full, a different queue number is given and the bunch is not allowed to call the target update, before the queue number matches that of the device.

The queue number seemed to be irrelevant, as long as there would only by one `!$omp target` call within a subroutine, however, when having several calls, it being data transfer or kernel execution, memory errors would occur. This may be due to the compiler and may not be an issue if compiling with a different compiler. Unfortunately, this was not possible to test, as I only had access to the GCC compiler.

To make sure all patches have been updated, a subroutine was added, which checks if a device has any active bunches and if so, force an update of the data it currently contains. This subroutine should be redundant in DISPATCH but is necessary in the mockup to allow to have bunch sizes, which the total number of patches in the system is not a multiple of.

Two versions were made, one which contained all the types just described, and thus had $N_{bunches}$ contiguous five-dimensional memory arrays, one for each bunch. The second version did not contain the bunch type, but instead the device type contained all the subroutines described for the bunch type. This version contained $N_{devices}$ contiguous six-dimensional memory arrays, one for each device. This was done to test how memory transfer would be affected when referencing a whole array

in the first version, compared to referencing a part of an array in the second version. The first version will be referred to as the *bunch*-version and the second as the *device*-version. Examples of the most important subroutines of the bunch-version can be found in Appendix A. The routines in the device-version are almost identical.

The memory layout of the memory arrays were made in the same way as they are in DISPATCH, thus having the following dimensions: `memb(x, var, y, z, np, nb)`, with `var` being the indices of the variables - such as the gravitational potential or density, `np` being the patch-slots and `nb` the bunch-slots. The bunch-version did not contain the bunch-slot.

Some issues occurred in the device-version when referencing parts of an array, using several `!$omp target` calls within the same subroutine. This was solved using a subroutine for each `target` call, which took the relevant part of the memory array as input, which were then used as input for the `target` call. This was also adopted to the bunch-version.

Profiling

Profiles of the two new versions were made using Nsight Systems, executing with eight cores and one GPU. The setting was set to use three bunches with 108 patches in each. The timeline for the *bunch* version is shown in Figure 5.2 and for the *device* version in Figure 5.3. The two timelines look very similar, and the two versions did indeed perform almost identically. Compared to the profile of `Step6.f90`, Figure 5.1, the cores are almost never idle. Though this may seem good, it is unlikely that each core is executing "relevant" work all of the time. A thread may assign a patch to a full bunch but is not allowed to copy data into the bunch before it is free. While waiting for this `!$omp taskyield` is used, to allow the thread to work on other available tasks. The behaviour of the `!$omp taskyield` may however vary. Optimally the task would be suspended and the thread would start working on another task - likely assigning a new patch to some bunch. At some point, there will be no new tasks to work on. At this point, the thread will continue to check whether a bunch is free, making at seem like the core is active, when in fact not doing any relevant work.

There is always some target task running, similar to the profile of `Step6.f90`. The kernels are however slightly faster, and the data transfer time has been reduced, due to the smaller memory array, because the source term is computed within the kernel, rather than on the host.

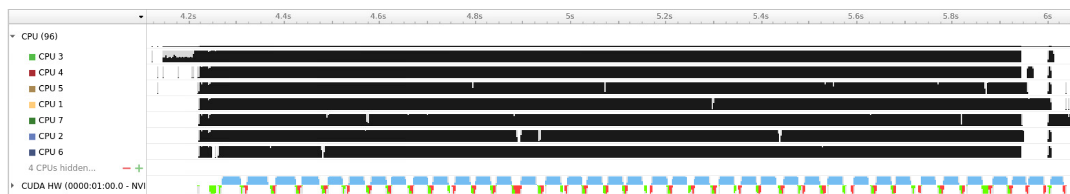

FIGURE 5.2: Profile of the *bunch* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device.

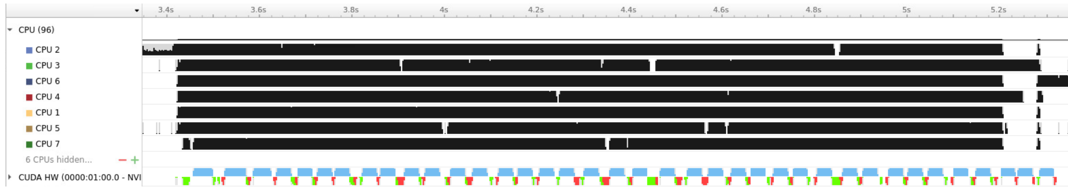

FIGURE 5.3: Profile of the *device* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device.

Profiles were also made using Nsight Compute. They were very similar to the profile of `step6.f90`, but had a slightly better throughput of just above 20%. If the patch size was increased to $64 \times 64 \times 64$ instead of $32 \times 32 \times 32$, the throughput was even better, almost reaching 50%. The overall time did however not seem to improve, compared to systems with the smaller patch size, but same total amount of cells. The poor utilization of the hardware thus seems to be a consequence of the SOR-routine.

Profiles were also made using two and three devices, with the same setup as above. The timelines of the bunch version can be seen in Figure 5.4 and Figure 5.5. When using two devices, both devices seem to be active almost the whole time, with very little time staying idle, which is exactly what one is looking for. Using three devices do seem to introduce extra overhead, making the devices stay idle. This seem to be due to not having enough cores, as using more cores the time the GPUs stay idle is reduced.

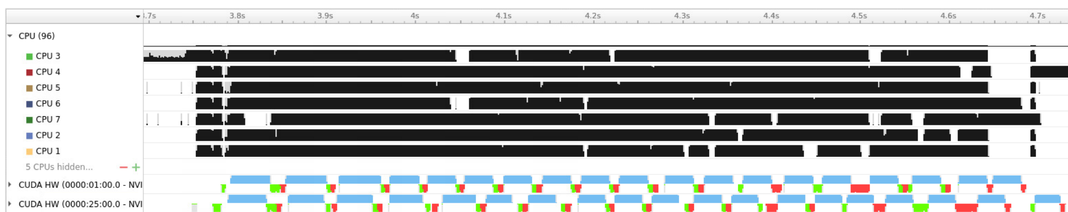

FIGURE 5.4: Profile of the *bunch* version using Nsight Systems, executing using eight cores and two GPUs. Three bunches containing 108 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device.

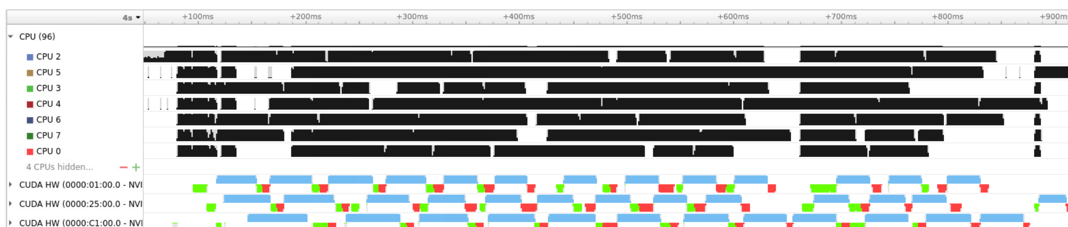

FIGURE 5.5: Profile of the *bunch* version using Nsight Systems, executing using eight cores and one GPU. Three bunches containing 216 patches each are used. The black lines indicate when a core is active. The blue lines indicate kernel execution on the GPU and the red and green lines are data transfers between the host and the device.

Chapter 6

Results

6.1 Optimal setup

Several tests were run on the mockups, to see how various settings affected the performance. All tests were run on the `astro2_gpu` partition of STENO having two 16-core AMD EPYC 7302 CPUs and four NVIDIA Tesla Ampere A100 GPUs. During the tests, only one of the CPUs were used, with different amounts of cores. The CPUs have a NUMA structure with 4 nodes containing 4 cores each. One thread per core was used.

Test were run using different system sizes, given by the total number of patches within the mesh. The number of patches is given $N_{patches} = 2^{3 \cdot Level}$. Each patch contains $32 \times 32 \times 32$ cells, with four of them being ghost cells - two on each side, unless otherwise stated. Each test was run five times, from which the mean time was computed. For each test the time per patch is compared, rather than the total execution time.

The GPU versions were run using eight cores and one GPU, unless otherwise stated. For the GPU versions, the time includes the assigning of a patch to a bunch, copying the patch to the bunch, transferring the bunch to the device, running the SOR-bunch routine on the device, transferring the bunch back to the host and copying the updated patch data back to the patches. It furthermore includes a routine that updates leftover patches in bunches after exiting the parallel region. The CPU version only contains a subroutine that computes the source term and the SOR routine.

6.1.1 CPU-version

To compare the tests with the CPU-version, several runs were made with various numbers of cores. The result is shown in Figure 6.1. The plot on the left is the time per patch with different mesh sizes, $Level = 4$(red) and $Level = 5$(blue). The plot on the right shows the time per cell using different dimensions of a patch, but the same total number of cells. The red is with 2^4 patches with dimensions $64 \times 64 \times 64$ and the blue is with 2^5 patches with dimensions $32 \times 32 \times 32$.

The total mesh size does not seem to affect the performance, however, how the cells are distributed across patches seem to have a big impact on the performance. Using 32 cells in each direction seem to perform better than using 64 cells in each direction.

Using 16 cores the time per patch is 0.44 ms.

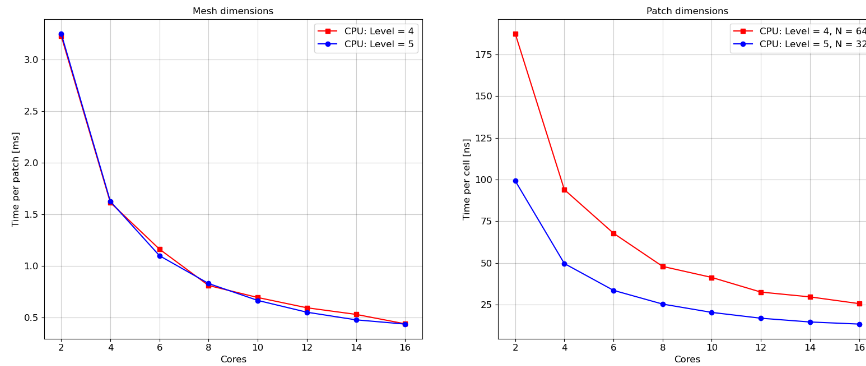

FIGURE 6.1: *Left*: Time per patch in *ms* vs. number of cores for different mesh sizes. *Right*: Time per cell in *ns* vs. number of cores using different patch sizes.

6.1.2 System size

It was tested how the different versions performed when increasing the number of patches in the system. The mesh sizes used were ranging from $8 \times 8 \times 8$ to $32 \times 32 \times 32$. The result is shown in Figure 6.2. For small systems, none of the versions perform well. There will be some overhead when initializing a parallel region, initializing tasks, transferring data to and from the device and sending the kernel to the device. For small systems, this overhead is too significant compared to the amount of computations.

As the system size increases, the overall performance becomes slightly better, which is likely due to the overhead becoming less significant compared to the computations. The performance improvement seems to be more significant for the GPU versions compared to the CPU version.

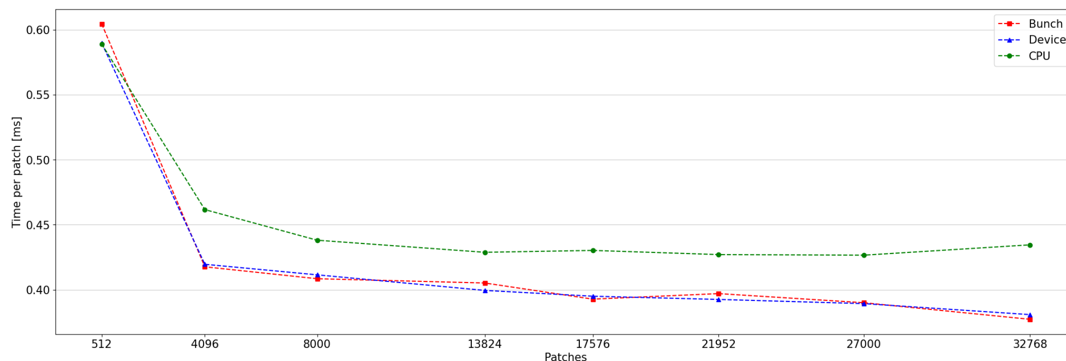

FIGURE 6.2: The time per patch in ms as a function of the total number of patches for different versions of the mockups. The GPU versions uses one device and eight cores, and the CPU version uses 16 cores. The performance seems to get better as the number of patches increase, which is likely due to the overhead of initializing tasks, transferring data etc. becomes less significant compared to the amount of computations.

6.1.3 Patch per bunch

The optimal bunch size was investigated for all versions. The result is shown in Figure 6.3. The tests were run using three and six bunches. The bunch sizes were either set such that the system size was a multiple of the bunch size, or the bunch size was a multiple of the number of SMs on the GPU.

In general, the performance is best using a multiple of the number of SMs on the GPU. Though this does require an extra update of a bunch which is not full after the parallel region, it still performs better. As the number of patches per bunch becomes larger, the performance is less hurt and performs better than the CPU version, but still worse compared to a multiple of the SMs.

To obtain the best performance from a GPU, one should utilize all the hardware throughout the program. When using bunch sizes that do not match the number of SMs, some of the SMs will get more data than others, making many of the SMs idle while the "leftovers" are updated, making the execution time of the kernel longer, while only updating a few extra patches. When using a multiple of the number of SMs, each SM will get the same amount of data, and will thus finish approximately at the same time. The size does not seem to have any significant impact on the performance, though using 108 patches seem to perform slightly better.

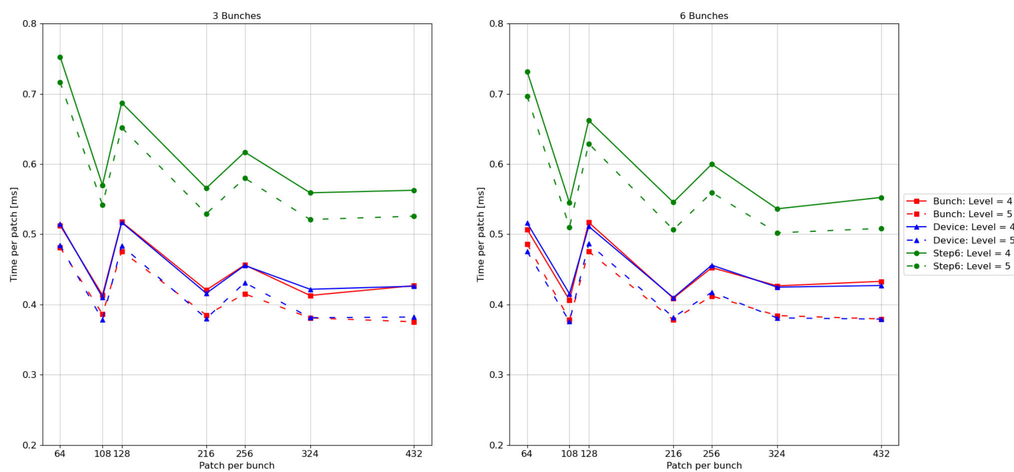

FIGURE 6.3: The time per patch in ms as a function of the number of patches per bunch for all the GPU versions, using one device and eight cores. *Left* is with three bunches per device, *right* is with 6 bunches per device. In general, it is best to use a bunch size which is a multiple of the number of SMs on the GPU, rather than having the total number of patches to be a multiple of the bunch size.

6.1.4 Bunch per device

It was tested how the number of bunches per device affects the performance. The tests were run using one GPU and eight cores. The size of each bunch was set to 108 patches, based on the tests of the optimal bunch size. The result is shown in Figure 6.4. The Bunch- and Device versions seem to perform more of less identically. For both versions it seems that the performance stays almost constant, when using two or more bunches per device, where they both perform better than the CPU version. There is a slight improvement when using more bunches, which on the other hand

also requires more memory and thus may not be beneficial, if using several GPUs. Using only one bunch harms the performance for all versions.

The Step 6 version seemed to have the best performance using only 2 bunches. This was not investigated much further, as it in general performed worse than the CPU version.

A small extra test was performed with the bunch version, where the time it takes to fill a bunch(FB-time) and the time from a bunch is full until it is ready for new data(BR-time) was measured. For a run using eight cores, one GPU, three bunches with 216 patches and $Level = 5$, the FB-time, excluding the time the thread waits for the bunch to be free, ranges from 6 ms to 25 ms, with an average of 10 ms. Including the time the thread is waiting for the bunch to open, the average FB-time is 26 ms. In general, the times to fill the bunches would be biggest the first time a bunch was used, and then get much smaller throughout the remaining updates. Though at some points, a big FB-time would appear randomly throughout the execution. Both cases of the big FB-time is likely explained by cache-misses. The average BR-time is 229 ms.

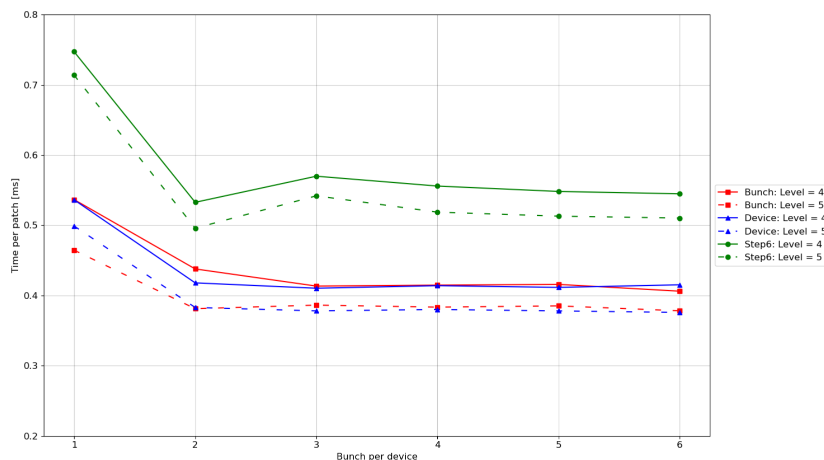

FIGURE 6.4: The time per patch in ms as a function of the number of bunches per device for the bunch-, device- and Step6-version, using one device and eight cores. Solid lines is with $Level = 4$ and dashed lines is with $Level = 5$.

6.1.5 Cores per device

It was tested how the number of cores affected the performance of the GPU versions. The tests were made using different number of bunches per device and two different bunch sizes, to see if it would have an impact when using more devices. The results for the bunch version can be seen in Figure 6.5 and for the device version in Figure 6.6. The tests were furthermore made on different mesh sizes, $Level = 4$ and $Level = 5$.

When using a smaller bunch size, the performance seems to be less affected on the number of cores and the mesh size. When using a greater bunch size, there seem to be a greater difference in the performance across mesh sizes. More cores are also needed to obtain the best performance.

Both versions perform almost identically. When only using one device, the greater bunch size seems to have less variations when using different number of cores. For the smaller bunch size and mesh size, the performance seems to perform slightly worse when not using all cores on the nodes, the difference is however very small.

Optimally, the performance should increase linearly with the number of devices. This is however not the case. Relative to using one device, the greatest speedup using two devices is approximately 2.01 for the bunch version using a bunch size of 108 and 1.93 using a bunch size of 216. For the device version it is 2.04 and 1.94 respectively, which is approximately the optimal speedup which one would expect. This trend does not apply when using three devices. In this case the speedup is only 2.33 and 2.25 for the bunch version and bunch sizes of respectively 108 and 216, and for the device version it is 2.30 and 2.25. Though it is faster, it does not reach the optimal speedup. Furthermore, when using more devices, the number of cores have a greater impact on the performance, compared to using one or two devices.

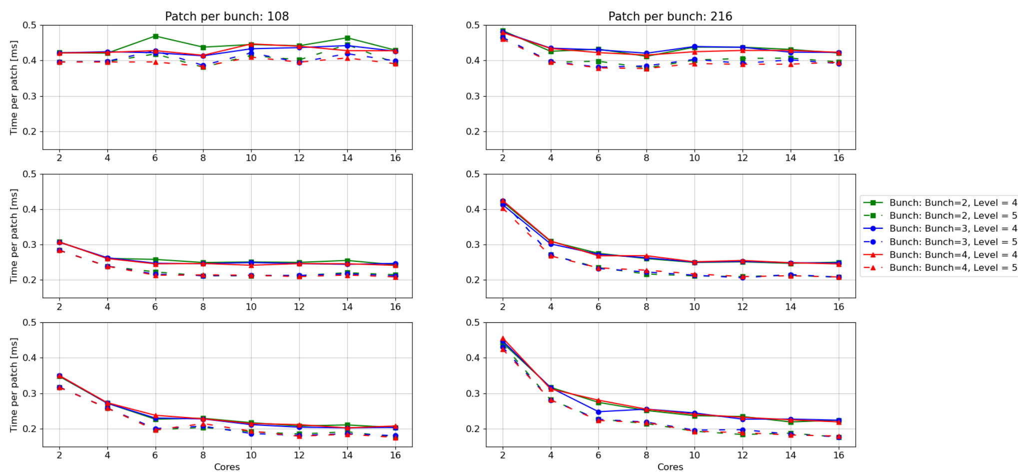

FIGURE 6.5: *Bunch version*: The time per patch in ms as a function of the number of cores using different number of bunches and bunch sizes. Plots on the *left* use 108 patches per bunch, plots on the *right* uses 216 patches per bunch. The two plots on the *top* are with one device, the two plots in the *middle* are with two devices and the two plots at the *bottom* are with 3 devices.

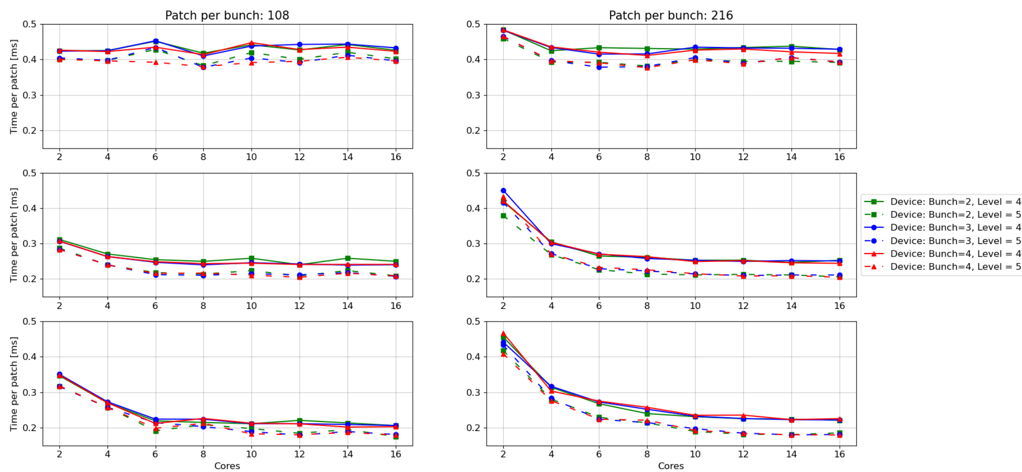

FIGURE 6.6: *Device version*: The time per patch in ms as a function of the number of cores using different number of bunches and bunch sizes. Plots on the *left* use 108 patches per bunch, plots on the *right* uses 216 patches per bunch. The two plots on the *top* are with one device, the two plots in the *middle* are with two devices and the two plots at the *bottom* are with 3 devices.

6.1.6 Patch dimensions

It was tested how the patch dimensions affect the performance. The tests were made using three bunches per device and bunch sizes of 108 and 216. The patch-dimensions was either $32 \times 32 \times 32$ using $Level = 5$ or $64 \times 64 \times 64$ using $Level = 4$, making it a total of 2^{30} cells. The time per cell is compared between the different versions.

In general, the performance does not seem to change significantly, especially when using enough cores. When only using one device, the greater patch dimensions seem to vary quite a bit, depending on whether all cores on a node are used or not. When using two or more devices, these variations seem to go away. In both cases, using a smaller bunch size seem to be more stable in performance and be less dependent on the number of cores.

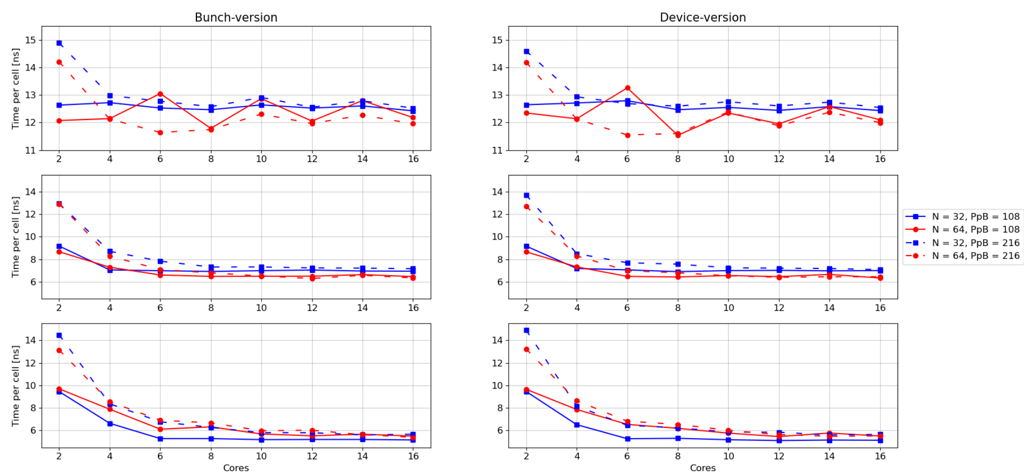

FIGURE 6.7: Time per cell vs. number of cores using different patch-dimensions and bunch sizes. For each run three bunches per device is used. *Left* is the times for the bunch version. *Right* is the times for the device version. *Top* uses one device, *middle* uses two devices and the *bottom* uses 3 devices.

Chapter 7

Discussion

7.1 Current results

Due to problems with the current offload implementation in DISPATCH, it has not been possible to implement the SOR-bunch routine in DISPATCH, and hence test it on some experiments to validate that it works and yields the same results as the CPU-versions. Tests were made in the mockups, yielding approximately the same results, only varying on the last digits. Running a full experiment with DISPATCH would indeed behave differently, as several kinds of physics would usually be computed during an update. The bunch module in DISPATCH also works in a slightly different way than the one used in the mockups, which will also have an impact on the performance. The results from the tests do thus not necessarily compare to what would be observed when run using the full DISPATCH code. They can however provide some indications on what to expect on the node-level and provide possible improvements ideas.

The two GPU versions perform almost identical. There thus seem to be no difference in having all bunches per device stored in one contiguous array or in $N_{bunches}$ smaller contiguous arrays. The latter is slightly easier to program and is almost identical to the current DISPATCH implementation and would thus seem preferable.

7.1.1 Optimal setup vs. DISPATCH

DISPATCH will distribute tasks across several nodes where the total number of tasks per node depends on the total system size. If the number of tasks per node becomes too small, the performance seems to be worse and one would thus like to ensure that the number of tasks per node is above some minimum, to obtain the best performance. The impact of the number of tasks seems to be slightly greater for the GPU-versions compared to the CPU version. The mockups only contain one mesh. In DISPATCH, a node may contain several grids at different levels for the same area. This automatically increases the number of tasks and would likely ensure that the number of tasks is well above 2^4, where the performance changes very little, when increasing the number of tasks.

In the mockups, only one update iteration is performed for a group of patches. In DISPATCH the number of tasks ready for update will vary throughout the execution and it will in general take longer to preprocess, update and postprocess a task. This will likely affect the optimal bunch size and the optimal number of bunches per device. The number of patches per bunch should however be some multiple of the number of SMs on the GPUs, to utilize the hardware best possible.

Optimally one would like the performance to increase linearly with the number of devices used. This does not seem to be the case, as the performance using two or three devices is very similar. To obtain this, one would need to fill $N_{devices}$ bunches

simultaneously, to keep all the devices busy throughout the execution. One thus need to have enough hardware threads on the CPU to accomplish this. A node on LUMI will contain one 64 code AMD Trento CPU and four AMP MI250X GPUs. There is thus an upper limit for how fast a bunch may be filled and thus how fast a bunch will be ready for update on the GPU. With the mockup versions, it thus seems likely, that it will be hard to utilize all GPUs on a node, which will likely also be the case in DISPATCH. In the current version, devices, bunches, and bunch-slots are assigned from the same device-handler in a round-robin-fashion for all threads, which may explain why the speedup does not increase linearly with the number of devices. Based on the results on the number of cores using one device (Figure 6.5 and 6.6), the GPU versions do not seem to need more than 2-4 cores, when using a bunch size of 108, to perform well. Thus, one may obtain better performance using several devices by grouping cores together, having each group assigning tasks to one device exclusively. This will indeed require some modifications to the current versions and the performance will likely be very dependent on the setup of an experiment. If the number of tasks per node is not big enough, one may end up having several non-full bunches (worst case one for each device) when the ready-queue is empty. It would thus seem of great importance to provide enough tasks per node, so that the ready-queue is never empty, which will potentially allow better utilization of the full hardware.

How cells are distributed does not seem to have any major impact on the performance of the GPU versions. It does however have a great impact on the CPU version. Using patch sizes of $64 \times 64 \times 64$ performs way worse than having patch sizes of $32 \times 32 \times 32$. Since DISPATCH may run on systems with varying hardware, it thus seems most beneficial to keep the patch sizes small.

7.1.2 SOR on GPU

In general, the GPU versions perform better than the CPU version. Using 16 cores, one device and three bunches, containing 108 patches each, the GPU version approximately performs 2.4 times faster than the CPU version using 16 cores. This includes all the extra functions in the GPU versions and hence the speedup for the SOR-routine is even greater. The peak performance of the CPU using 16 cores is about 0.77 TFLOPS, where it for one GPU is 19.5 TFLOPS. The maximum speedup that may be obtained is thus around 25. Based on the profiles, the occupancy of the GPU was only 12.5%, which would yield a speedup roughly consistent with the actual result.

Running the SOR method on GPUs provides poor utilization of the hardware. Patches within a bunch are distributed such that each SM gets at least one patch to update. Though up to 32 threads may be used at the same time, parts of the method only allows a single thread to be active, while the others remain idle. Furthermore, at each iteration of the update, data must be distributed to threads and they all have to synchronize at the end due to the `reduction` clause. This introduces some overhead that unavoidably has an impact on the performance. Increasing the patch-dimensions does seem to improve the utilization of the hardware, since more computations happen within one update iteration, and thus a greater part of the SOR-routine utilizes the full hardware of an SM. It does however not seem to improve the overall performance, comparing the time per cell (Figure 6.7).

The GPU hardware may be utilized better by refactoring the entire routine, having only loops running on the GPU. This would however introduce extra data transfers between host and device and would likely make each SM stay idle while waiting

to perform the next update. It is thus expected that the overall performance is better with the current setup, despite the poor utilization of the GPU hardware.

In general, it does however seem beneficial to perform the SOR routine on the GPU, as long as the number of bunches and the bunch sizes are set correctly. As mentioned initially, on CPUs the SOR and CG methods for solving the Poisson equation have similar performance, and the SOR was chosen for GPU implementation due to the simpler code. Since exactly the simple nature of the code probably is a main reason for the poor GPU utilization, in retrospect it seems likely that implementing the CG method would give a larger speedup

7.2 DISPATCH offload implementation - improvement ideas

7.2.1 Execution flow

In the current offload implementation of DISPATCH, two points of the current execution flow may be improved to obtain better performance. Firstly, the time it takes to fill a bunch, t_{fill}, is measured doing the first time a bunch is used. This time is then used as a maximum interval between updates, which together with some scaling factor, ω, is used to compute when the next update has to take place, t_{next}. Every time a thread copies data to a bunch, the current time, t_{curent}, is measured. If $t_{current} > t_{next}$ an update is forced with the current number of patches within a bunch, after which t_{next} is updated by $t_{next} = t + \omega t_{fill}$, where the t is the time right before t_{next} is updated.

This mechanism is added to avoid the program to stall in cases where no tasks are ready for update and is thus necessary for the bunch setup to work. It however does not seem to be the optimal way to avoid the program to stall. The main reason being that the time it takes to fill a bunch is usually much greater the first time, as the program is more prone to cache misses, making the interval unnecessarily big and the program may stall too long in cases where the ready-queue is empty.

Secondly, the thread which performs the actual update of the bunch also performs all the post processing of all the patches within it. There will be cases, where some threads have no tasks to execute, and thus stay idle - especially in combination with the mechanism described above. Hence it would seem preferable to spread post processing across several threads and utilize the hardware better.

To improve the above issues, a slightly different approach for the offload execution is suggested.

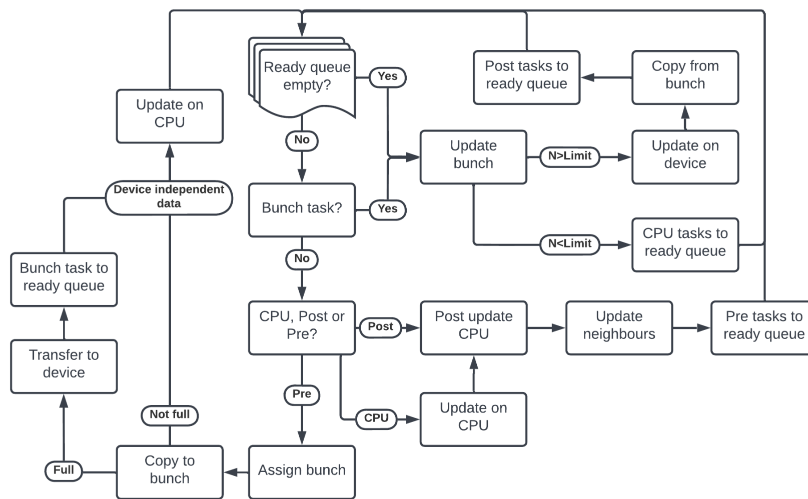

FIGURE 7.1: Flowchart of the suggested execution flow.

A task should be extended to have an *update status* which may take the form of *CPU*, *pre* or *post*. A new task type should further be introduced which is connected to the bunch type and controls the update on the device. Furthermore, a device handler-like method (as presented in the mockups) to assign a patch to a bunch, by providing a bunch-ID, rather than assigning a bunch within the routine which copies data to the bunch, the program can keep track of how many patches have been assigned to a bunch and how many patches have actually been copied to the bunch. The thread execution may thus take one of five different paths. A basic flowchart of the thread execution is presented on Figure 7.1.

The first path is tasks with the *pre* update status. Firstly, the thread is assigned a bunch, after which the patch is prepared and copied to the bunch. If the bunch is full, the bunch is transferred to the device and the bunch task is set to ready and placed as the first task in the ready queue, such that the next thread picking a task from the ready queue, will update the bunch. The main idea of this is, that since only a small part of the solvers in DISPATCH have been prepared for GPUs, thus when running experiments that solves many kinds of physics, most of the solvers must be called from the CPU. Some of the data may be independent of the data updated on the device and might as well be computed immediately by the thread, rather than leaving it to one thread after the device update. If the bunch is not full, or after placing the bunch task in the ready queue, the thread will thus update the values which are independent of the data updated on the device, before picking a new task from the ready queue.

The second path is the *bunch task*. The thread will simply update the bunch on the device and copy back the data to the patches. Instead of one thread performing all the post-update procedures for the patches within a bunch, post update status of each patch is changed to post, and placed back in the ready queue, allowing all threads to cooperate on generating new tasks that may be updated. The thread then picks a new task from the ready queue.

The third path is the case where a thread encounters an empty ready queue. In this case, the thread will force an update of the bunch, by first checking whether the number of patches within the bunch is above or below some limit. In the case of the number of patches being below the limit, the thread will generate CPU tasks, which are placed in the ready queue. If the number of patches is equal to or above

the limit, the thread will follow the same steps as for the bunch task. In both cases update should not be performed before the number of assigned patches matches the number of patches copied to the bunch. To make sure only one thread performs either of the updates, some control variable should be used, to inform other threads encountering the empty ready queue, that the update has been initialized.

The fourth path is tasks with the *post* update status. In this case the thread will perform the last updates on the CPU, after which the neighbours will be checked. If any of them have become ready, they are placed in the ready queue with the *pre* update status.

The fifth path is tasks with the *CPU* update status. In this case the thread will simply perform the remaining updates, that was supposed to be performed on the device. The update is only performed on one patch, rather than the whole bunch. After the update, the thread will follow the same steps as the tasks with the post update status.

This model will generate more tasks in the ready queue - also tasks which do not move the program forward in time. It is however very likely, that this model will utilize the hardware even better, as there would almost always be some task to execute. It would require some kind of execution hierarchy, making sure that the most important tasks are executed first. A possible hierarchy would be the *bunch* task at the top, tasks with the *post* update status, tasks with the *CPU* update status and tasks with the *pre* update status at the bottom, based on the assumption that it is preferable to generate tasks with the *pre* update status - that is, to fully complete the update of patch - which will then possibly allow neighbouring patches to move forward in time.

Using a hierarchy like this will likely create some overhead, when placing the tasks in the ready queue. It may thus be preferable to create extra ready queues, one for each update status and one for the bunch task, which are then checked in the hierarchy order.

Using the empty ready queue as the trigger for a forced update may lead to extra updates of non-full bunches, which will indeed execute slower than a full bunch - provided the optimal bunch size is used. One can imagine situations, where patches are placed in the ready queue shortly after the forced update have been called, which would have been added to the bunch with the current implementation. It is however the expectation, that the increased number of generated tasks and the task hierarchy will ensure, that there is a constant flow of tasks to be executed, with very little idle time for each hardware thread.

7.2.2 Asynchronous pipelining

A big effort was put into asynchronous pipelining, using a similar approach as Chikin et al., 2019, allowing data transfer and kernel execution at the same time. As it can be seen from the profiles, only one target instruction is executed at a time, as shown in the red square on the top of Figure 7.2. Optimally one would like a bunch to be transferred when it is ready, so that the data may be updated as soon as the GPU is free. That is, one would like to transfer a bunch to and from the device, while another bunch is being updated, as shown in the green square on the bottom of Figure 7.2. According to the OpenMP specifications, this should be possible using the `nowait` and `depend` clauses. The `nowait` allows the thread to continue execution on the device while the target task is executed, which allow it to call other target directives. Several attempts were made, but none of them with success. This may be caused by the fact that the `target` directives were called by different threads.

Attempts were therefore made, where one thread was in charge of updating all the bunches ready for update, thus making all the calls to the `target` directives. This did not work either, which seems to be due to lack of support of the compiler. It may thus work using a different compiler, or maybe for future versions of the GCC compiler, which will have extended support o

Only looking at the execution on the device, the benefit of this pipelining is likely minimal with the current setup, as the kernel execution usually takes much longer than the data transfer. However, allowing the thread to continue execution on the CPU while the target region is executed, the benefit may be much bigger. The idea is shown in the blue square on Figure 7.2, where the white boxes at the bottom show the thread execution. This is very simplified, as threads would usually work together to fill a bunch, the idea however remains the same. As the target task is being executed, the thread could be able to do other work, such as checking neighbor relations or assigning other tasks to bunches, instead of staying idle.

This model would require a `taskwait` call at the end, which would potentially make the thread stay idle while waiting for the target tasks to complete. A possible solution to this could be to seperate the update of a task into several steps, as described in section 7.2.1.

This model indeed requires the `nowait` and `depend` clauses to work properly, and thus do not seem possible with the current version of the GCC compiler.

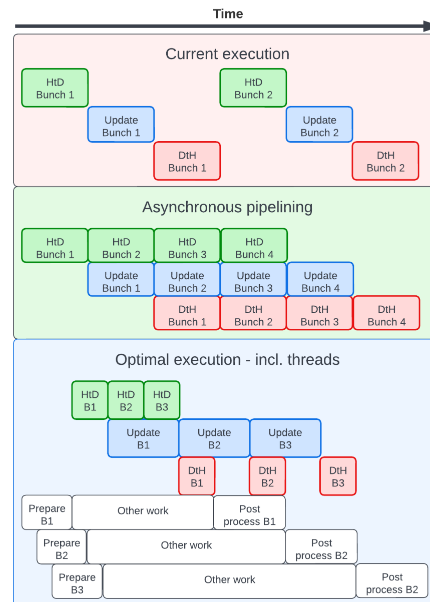

FIGURE 7.2: Asynchronous pipelining of GPU instructions using OpenMP. The red square on top shows the current execution, where one target task is executed at a time. The green square in the middle is the asynchronous pipelining, where data transfers and kernel execution are executed concurrently. The blue square at the bottom shows the optimal execution, where threads continue execution on the CPU while the target tasks are executed on the device.

7.2.3 Linked list of procedure pointers

With the current setup of DISPATCH, only the RAMSES solver has been prepared for the bunch execution, which makes the use of other physics modules straightforward, as they must be computed on the host. The number of different physics used will vary from simulation to simulation. All the solvers, except the RAMSES solver, will all be called within the pre- or post-update routines immediately before or after the update routine. As more solvers are prepared for bunch execution, one would optimally like to transfer the bunch to the device, distribute the patches to the SMs and then perform all the update routines within one kernel. With the current setup, this would require several update routines, depending on what physics is enabled and prepared for the bunch execution on the GPU. At some point, this will become almost impossible to keep track of and the code will be harder to maintain.

Instead, it may be beneficial to create a linked list of procedures, which is then called from the GPU. The update list would be made during the initialization, where

some variable from an input file could allow the programmer to control which routines to execute on the device and which to execute on the host. This would furthermore ease the future implementations of new solvers, which are ready for bunch-execution.

A few small tests were made to see if it is even possible to transfer procedure pointers to the device using OpenMP, which all seemed to be negative. Some further investigations should however be made, to finally conclude if this is due to OpenMP or some other errors in the test.

Chapter 8

Conclusion

It has been shown that porting the SOR routine to run on GPUs using OpenMP and use it to solve for the gravitational potential is indeed possible. The SOR routine itself makes it impossible to take full advantage of all the available hardware that a GPU offers. The main performance improvements came from improving the data transfers between host and device. A new system has been developed for this, which allows to execute on several devices and to effectively distribute tasks to bunches. In general, OpenMP has great support for porting code to devices, but does still seem to have minor flaws which complicates the process. Some issues are likely due to lack of support in the GCC compiler and are expected to work on other compilers with better support, or in future versions of the GCC compiler. It has not been possible to test different compilers.

Due to problems in the existing code, it has not been possible to make a proper implementation of the SOR routine in DISPATCH, hence it has not been possible to perform the necessary tests and validations and make it an integrated part of DISPATCH. Instead, the focus has been to thoroughly test the implementation in the mockups, to ease the implementation in the future and provide a better understanding of key points to obtain the best possible performance.

Two different versions of the mockup have been made, to test whether storing all bunches in one big contiguous memory array provide better performance than storing each bunch in its own contiguous memory array. It is found that both versions perform identically.

The new system follows the same ideas as the current implementation. Tasks are assigned a specific slot in one of the available bunches, which is then transferred to the device for update when full. It is found that the bunch size should be some multiple of the total number of streaming multiprocessors on the GPU, to obtain the best possible performance. Matching the bunch-size to the number of SMs on the GPU, makes the performance less dependent on the number of CPU-cores available and would seem preferable.

Using two devices does seem to improve performance with a factor of two, provided enough tasks are available for update. Using three devices only performs slightly better as two and seem to require several extra cores to avoid idle time on the GPUs.

Several ideas have been proposed, to improve the current offloading system, which builds on top of the system implemented in the mockups but could also apply to the current system. The suggestions aim to make better use of the hardware. The main cause of idle time is caused by not having enough tasks available to fill a bunch, forcing the program to stall until an update is forced after some time limit. Furthermore, the thread making the update performs all the post update procedures, which could easily be performed in parallel by several threads. It is thus suggested to introduce an *update status* to each task and to force updates when the ready queue

is empty, rather than some limit. The goal with the first point is to provide extra tasks for the CPU threads and make better utilization of these. The goal of the latter is to reduce the time the program stalls due to lack of available tasks and consequently provide new tasks much faster. Due to the problems with the current code and that the implementation of the SOR routine had the highest priority, it has not been possible to investigate this system further.

Asynchronous pipelining has also been tested, which could potentially provide better utilization of the CPU-cores. This is however not possible with the current version of the GCC compiler.

Finally, a new system for the update routine is suggested, making use of a linked list of procedures. This would make the implementation of other solvers, which are prepared for GPU-execution, easier and provide a simple way to control what physics to solve on the CPU and what to solve on the GPU. It would furthermore make the code easier to maintain, as it would avoid the need of having update routines for all combinations of solvers.

Appendix A

Bunch version

A.1 Device_handler_mod

Update

```
SUBROUTINE update(self, patch)
    class(device_handler_t) :: self
    class(patch_t) :: patch
    integer :: did, bid, pid, qid

    call omp_set_lock(self%lck_assign)
    call self%assign_device(did, bid, pid, qid)
    call omp_unset_lock(self%lck_assign)

    call self%device_list(did)%assign_bunch(patch, bid, pid, qid)

END SUBROUTINE update
```

LISTING A.1: The update routine

Assign device

```
SUBROUTINE assign_device(self, did, bid, pid, qid)
    class(device_handler_t) :: self
    integer, intent(out) :: did, bid, pid, qid
    logical, save :: first_time = .true.

    ! Get next slot in bunch and increment
    pid = self%next_patch
    self%next_patch = mod(self%next_patch,self%patch_per_bunch) + 1

    if (first_time) then
      first_time = .false.
      !$omp atomic update
      self%device_list(self%next_device)%active_bunches = &
      self%device_list(self%next_device)%active_bunches + 1
      !$omp end atomic

    else if (pid == 1) then
      self%next_device = mod(self%next_device, self%n_devices) + 1
      !$omp atomic update
      self%device_list(self%next_device)%active_bunches = &
      self%device_list(self%next_device)%active_bunches + 1
      !$omp end atomic

      if (self%next_device == 1) then
        self%next_bunch = mod(self%next_bunch, self%bunch_per_device) + 1

```

```
27          if (self%next_bunch == 1) then
28            self%next_qid = mod(self%next_qid, 100) + 1
29          endif
30        endif
31      endif
32
33      did = self%next_device
34      bid = self%next_bunch
35      qid = self%next_qid
36  END SUBROUTINE assign_device
```

LISTING A.2: Assigning a device, bunch and bunch-slot to a thread

A.2 Device_mod

Assign bunch

```
1  SUBROUTINE assign_bunch(self, patch, bid, pid, qid)
2      class(device_t) :: self
3      class(patch_t)  :: patch
4      integer :: bid, pid, qid, allocated, queue
5
6      call self%bunch_list(bid)%copy_to_bunch(patch, pid, qid, allocated)
7
8      if (allocated == self%patch_per_bunch - 1) then
9        call omp_set_lock(self%lck_update)
10       queue = self%next_update
11       self%next_update = mod(self%next_update,self%bunch_per_device*2) + 1
12       call omp_unset_lock(self%lck_update)
13
14       do while(.not. self%to_update == queue)
15         !$omp taskyield
16       enddo
17
18       call self%bunch_list(bid)%update()
19       self%to_update = mod(self%to_update,self%bunch_per_device*2) + 1
20
21       call self%bunch_list(bid)%copy_from_bunch()
22       !$omp atomic update
23       self%active_bunches = self%active_bunches - 1
24       !$omp end atomic
25     endif
26 END SUBROUTINE assign_bunch
```

LISTING A.3: Routine that copies to bunch, and updates when full and the device is free

A.3 Bunch_mod

Copy to bunch

```
1  SUBROUTINE copy_to_bunch(self, patch, pid, qid, allocated)
2      class(bunch_t) :: self
3      class(patch_t), target :: patch
4      integer :: pid, qid, i1, i2, i3
5      integer, intent(out) :: allocated
6
7      do while(.not. qid == self%queue)
8        !$omp taskyield
```

```
    enddo

    self%patch_list(pid)%patch => patch
    do i3 = lb(3), ub(3)
      do i2 = lb(2), ub(2)
        do i1 = lb(1), ub(1)
          self%memb(i1,iphi,i2,i3,pid) = patch%phi(i1,i2,i3)
          self%memb(i1,idensity,i2,i3,pid) = patch%density(i1,i2,i3)
        enddo
      enddo
    enddo
    self%ds(:,pid) = patch%ds

    !$omp atomic capture
    allocated = self%allocated
    self%allocated = self%allocated + 1
    !$omp end atomic

    if ((allocated == self%patch_per_bunch - 1)) then
      call self%copy_to_device(self%memb, self%ds)
    endif

END SUBROUTINE copy_to_bunch
```

LISTING A.4: Routine that copies patch data to the bunch

Copy from bunch

```
SUBROUTINE copy_from_bunch(self)
    class(bunch_t) :: self
    class(patch_t), pointer :: patch
    integer :: i, i3, i2, i1

    call self%copy_from_device(self%memb)

    do i = 1, self%allocated
      if (.not. associated(self%patch_list(i)%patch)) exit
      patch => self%patch_list(i)%patch
      do i3 = lb(3), ub(3)
        do i2 = lb(2), ub(2)
          do i1 = lb(1), ub(1)
            patch%phi(i1,i2,i3) = self%memb(i1,iphi,i2,i3,i)
          enddo
        enddo
      enddo
      self%patch_list(i)%patch => null()
    enddo

    self%allocated = 0
    self%queue = mod(self%queue,100) + 1

END SUBROUTINE copy_from_bunch
```

LISTING A.5: Routine that copies patch data from the bunch

Update

```
SUBROUTINE update_device(self, memb, ds)
    class(bunch_t) :: self
    real(kind=realkind), pointer :: memb(:,:,:,:,:)
```

```
    real(kind=realkind), pointer :: ds(:,:)
    integer :: N, lpid

    N = self%allocated

    !$omp target teams distribute device(self%device_id)
    do lpid = 1,N
        call poisson%sor_bunch(memb(:,:,:,:,lpid), ds(:,lpid), li, ui, n_bunch)
    enddo
    !$omp end target teams distribute
END SUBROUTINE update_device
```

LISTING A.6: Routine that updates the bunch on the device

A.4 SOR

```
SUBROUTINE sor_bunch (memb, ds, l, u, n)
  !$omp declare target
  real(kind=realkind) :: memb(:,:,:,:)
  real(kind=realkind) :: ds(:)
  integer :: l(3), u(3), n(3)
  integer :: i, j, k, sweep, ss, ione, jone, kone, iter, ii
  real(8) :: a, b1, b2, b3, res, numer, denom, rjac2, omega, error

  numer = 0.0
  denom = 0.0
  do i=1,3
    if (n(i) > 1) then
      numer = numer + cos(pi / n(i))
      denom = denom + 1d0
    end if
  end do
  rjac2 = (numer / denom)**2
  if (chebyshev) then
    omega = 1.0
  else
    omega = 2d0 / (1d0 + sqrt(1d0 - rjac2))
  end if

  ione = 1
  jone = 1
  kone = 1
  if (n(1) <= 1) ione = 0
  if (n(2) <= 1) jone = 0
  if (n(3) <= 1) kone = 0

  a = 0.0
  if (n(1) > 1) a = a + 2. / ds(1)**2
  if (n(2) > 1) a = a + 2. / ds(2)**2
  if (n(3) > 1) a = a + 2. / ds(3)**2

  a = 1. / a
  b1 = a / ds(1)**2 * ione
  b2 = a / ds(2)**2 * jone
  b3 = a / ds(3)**2 * kone

  do iter=1,max_iter
    error = 0.0
    do sweep=0,1
        !$omp parallel do collapse(2) reduction(max:error)  &
        !$omp schedule(static)  default(none) private(ss, i, res, ii) &
```

```
46         !$omp shared(memb, l, u, b1, b2, b3, a, fourPiG, idensity) &
47         !$omp shared(iphi, omega, floor, ione, jone, kone, sweep)
48         do k=l(3),u(3)
49           do j=l(2),u(2)
50             !$omp simd reduction(max:error) private(ss,i,res)
51             do ii=l(1),u(1),2
52               ss = modulo(j + k + sweep,2)
53               i = ii + ss
54               if (i > u(1)) cycle
55               res = b1 * (memb(i+ione,iphi,j,k) + memb(i-ione,iphi,j,k)) &
56                   + b2 * (memb(i,iphi,j+jone,k) + memb(i,iphi,j-jone,k)) &
57                   + b3 * (memb(i,iphi,j,k+kone) + memb(i,iphi,j,k-kone)) &
58                   - memb(i,iphi,j,k) - a * fourPiG * memb(i,idensity,j,k)
59               memb(i,iphi,j,k) = memb(i,iphi,j,k) + omega * res
60               error = max(abs(res / (memb(i,idensity,j,k) + floor)), error)
61             enddo
62           enddo
63         enddo
64         !$omp end parallel do
65       if (chebyshev) then
66         if (iter == 1) then
67           omega = 1.0 / (1.0 - rjac2 * 0.5)
68         else
69           omega = 1.0 / (1.0 - rjac2 * 0.25 * omega)
70         end if
71       end if
72     enddo
73 
74     error = error / (a*fourPiG)
75     if (error < tolerance) exit
76   enddo
77 
78 END SUBROUTINE sor_bunch
```

LISTING A.7: SOR bunch routine

Bibliography

Berger, M.J and P Colella (1989). "Local adaptive mesh refinement for shock hydrodynamics". eng. In: *Journal of computational physics* 82.1, pp. 64–84. ISSN: 0021-9991.

Bodenheimer, Peter (2006). *Numerical methods in astrophysics: an introduction*. CRC Press.

Bryan, Greg L. et al. (2014). "ENZO: AN ADAPTIVE MESH REFINEMENT CODE FOR ASTROPHYSICS". eng. In: *The Astrophysical journal. Supplement series* 211.2, pp. 19–52. ISSN: 0067-0049.

Chikin, Artem, Tyler Gobran, and José Nelson Amaral (2019). "OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries". eng. In: *Accelerator Programming Using Directives*. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 51–74. ISBN: 3030122735.

consortium, LUMI. *LUMI's full system architecture revealed*. URL: https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/. (accessed: 09.06.2022).

Diaz, Jose Monsalve et al. (2019). "Analysis of OpenMP 4.5 Offloading in Implementations: Correctness and Overhead". In: *Parallel Computing* 89, p. 102546. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.102546. URL: https://www.sciencedirect.com/science/article/pii/S0167819119301371.

Dubey, Anshu et al. (2014). "A survey of high level frameworks in block-structured adaptive mesh refinement packages". eng. In: *Journal of parallel and distributed computing* 74.12, pp. 3217–3227. ISSN: 0743-7315.

Dumas, Joseph D. (2017). *Computer architecture : fundamentals and principles of computer design*. eng. Second edition. Boca Raton: CRC Press. ISBN: 1-315-36711-4.

Haarh, Michael (2021). "Porting DISPATCH MHD to GPU using directive based programming". MA thesis. University of Copenhagen.

Heyer, Mark et al. (2009). "Re-Examining Larson's Scaling Relationships in Galactic Molecular Clouds". eng. In: *The Astrophysical journal* 699.2, pp. 1092–1103. ISSN: 0004-637X.

Li, Xuechao and Po-Chou Shih (2018). "An Early Performance Comparison of CUDA and OpenACC". In: *MATEC Web Conf.* 208, p. 05002. DOI: 10.1051/matecconf/201820805002. URL: https://doi.org/10.1051/matecconf/201820805002.

Mendygral, P. J et al. (2017). "WOMBAT: A Scalable and High-performance Astrophysical Magnetohydrodynamics Code". eng. In: *The Astrophysical journal. Supplement series* 228.2, pp. 23–23. ISSN: 0067-0049.

Nordlund, Åke et al. (2018). "dispatch: a numerical simulation framework for the exa-scale era – I. Fundamentals". eng. In: *Monthly notices of the Royal Astronomical Society* 477.1, pp. 624–638. ISSN: 0035-8711.

Null, Linda and Julia Lobour (2014). *The essentials of computer organization and architecture*. eng. 4. ed. Burlington, MA: Jones Bartlett Learning. ISBN: 9781284045611.

OpenMP Application Programming Interface (2015). Version 4.5. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

Press, William H. et al. (1992). *Numerical recipes in fortran 77 : the art of scientific computing : Volume 1 of Fortran Numerical Recipes*. eng. 2nd ed. Cambridge: Cambridge University Press. ISBN: 052143064x.

Ramsey, JP, T Haugbølle, and Å Nordlund (2018). "A simple and efficient solver for self-gravity in the DISPATCH astrophysical simulation framework". In: *Journal of Physics: Conference Series*. Vol. 1031. 1. IOP Publishing, p. 012021.

Robey, Robert and Yuliana Zamora (2021). *Parallel and High Performance Computing*. eng. Second edition. Manning Publications Co. ISBN: 9781617296468.

Stone, James M et al. (2020). "The Athena++ Adaptive Mesh Refinement Framework: Design and Magnetohydrodynamic Solvers". eng. In: *The Astrophysical journal. Supplement series* 249.1, pp. 4–. ISSN: 0067-0049.

Strohmaier, Erich et al. *Top500: June 2022*. URL: https://www.top500.org/lists/top500/2022/06/. (accessed: 09.06.2022).

Zhang, Weiqun et al. (2016). "BoxLib with Tiling: An Adaptive Mesh Refinement Software Framework". eng. In: *SIAM journal on scientific computing* 38.5, S156–S172. ISSN: 1064-8275.