UiO : **Department of Informatics**
University of Oslo

Auto-tuning Flood Simulations on CPUs and GPUs

Gard Skevik
master thesis autumn 2014

Auto-tuning Flood Simulations on CPUs and GPUs

Gard Skevik

1st August 2014

Abstract

In recent years, there has been a growing focus on how GPUs can be utilized for general purpose computations. However, this leads to less focus on the CPU as a computational resource. As a consequence, heterogeneous computers with both a CPU and a GPU may not utilize all its resources. To address this, I present a heterogeneous CPU and GPU implementation based on a high-resolution explicit scheme for the shallow-water equations on a single GPU (Brodtkorb et al. 2012) . I perform two levels of parallelism: First, a row domain decomposition method is used to decompose the computational domain to utilize both the CPU and the GPU in parallel. Secondly, the CPU code is multi-threaded to take advantage of all cores. Furthermore, systems of conservation laws often involve large computational domains where only parts of the domain has to be computed, e.g., water or other fluids. This can lead to imbalance in the workload if the computations between the GPU and the CPU are not balanced. To address this, I present dynamic auto-tuning methods that automatically tune the domain decomposition between the CPU and the GPU during runtime, as well as optimization techniques to skip computations for "dry" domain areas.

Contents

List of Figures

List of Tables

Listings

Preface

This thesis has been developed at SINTEF ICT at the department of Applied Mathematics. The work presented has been performed individually, but several algorithms implemented have been influenced by discussions with my supervisors and other master students that I worked closely with.

My implementations have been executed on two different systems, first, a desktop with an Intel Core i7-2600K CPU, 8 GB of RAM, and a Geforce GTX 480 GPU, and a laptop with an Intel Core i7 Q 740 CPU, 8 GB of RAM, and a Quadro Q1800M GPU. The desktop runs Ubuntu 12.04, while the laptop runs Ubuntu 12.10.

Acknowledgements: I would like to thank my supervisors André Brodtkorb, Franz Fuchs, and Martin Reimers for supporting me and providing feedback on my thesis.

Chapter 1

Introduction

This thesis explores auto-tuning techniques between CPUs and GPUs in a heterogeneous system. These techniques have been applied for shallow water simulations to automatically tune itself to achieve real-time performance. However, the general principles of the auto-tuning, including the implementation of a heterogeneous system applies well to any systems of conservation laws, e.g., the Euler equations [12] and MHD equations [28].

Shallow water simulations can be used to model a vast range of physical phenomena such as tsunamis, floods, storm surges and dam breaks. Such simulations are important for a number of reasons [4]: First off, to evaluate possible scenarios to help with creating inundation maps and emergency plans. Secondly, to gain a better view over ongoing events, for example simulating possible scenarios in real-time. These scenarios often involve a huge computational domain which yields high computing requirements to achieve such simulations in real-time.

I therefore present a shallow water simulator running on the CPU alongside the GPU, forming the basis of the heterogeneous system. The implementation is based on an existing simulator that runs on a single GPU [9]. The simulator has been implemented with a second-order accurate explicit finite-volume scheme for the shallow water equations. I first extend this scheme to a heterogeneous hybrid CPU/GPU system to utilize the CPU concurrently alongside the GPU. Secondly, I apply auto-tuning techniques between the CPU and the GPU. Section 1.1 continues by arguing for this field of study and presenting the motivation. Furthermore, the research questions that this thesis attempts to answer are presented in section 1.2. Finally, an overview of the thesis is given in section 1.3.

1.1 Motivation

The main motivation behind using a heterogeneous implementation of the shallow water simulations is the speedup potential over a single GPU implementation. Typically one would use GPUs to achieve a large speedup over the CPU, as done by many scientific papers in recent years [9, 16, 3, 41]. For shallow water simulations, its use of finite volume stencil computations is a primary motivation behind using GPUs, as stated in [9] and [35]. These types of stencil computations are highly parallel and well suited to implement on GPUs. The reason for this is that a GPU is optimized for net throughput, as it contains several SIMD processors that execute blocks in parallel [9]. As a result, many scientific papers rely only on the strength of GPUs to optimize certain parallel tasks, leaving the CPU idle [35, 1].

However, CPUs have also evolved into a more parallel nature by the addition of more and more cores [7]. Modern desktop servers can have CPUs with up to 15 cores [17] and servers therefore have a lot of computing power which can be utilized for speeding up shallow water simulations, especially if performed concurrently with the GPU. As a result, one can gain large speedup for computers with a weaker GPU, going from simulations with poor performance to real-time. I present performance results were the obtained peak performance was increased by 141% over a single GPU by utilizing a CPU together with the GPU. This makes it an exceptional resource for computations of parallel nature. This is supported by Lee et al. [25] which compared an Nvidia GTX 280 GPU and an Intel Core i7 960 CPU by running 14 different throughput kernels on both. After optimizing the code for both the CPU and GPU, the GPU had a performance advantage of only 2.5 times the CPU. These results indicate that the CPU is not necessary far worse than the GPU when it comes to throughput computing. By extending the current shallow water algorithms [9] to a hybrid CPU/GPU system and utilizing both these computational resources, one can perform the simulations faster or increase the spatial resolution and perform the simulations in the same amount of time.

1.2 Research question

There are many scientific papers that describe how to utilize GPUs in parallel algorithms to achieve real-time performance [9, 35, 15, 40, 1, 16]. However, most of this research focus only on the GPU, while papers describing how to effectively utilize both the CPU and the GPU as computational resources are lacking [6]. This is a problem as the performance of multicore CPUs and GPUs continues to scale with Moore's law, making it more common to use a heterogeneous architecture where both the CPU and GPU are utilized to attain the highest performance possible [33]. This thesis will try to identify this problem and poses two main questions:

1. How to implement a heterogeneous CPU/GPU simulator that provides good performance for systems of conservation laws?

2. What type of auto-tuning techniques can be applied to gain ideal load balancing between CPUs and GPUs in a real-world case?

The first question asks how one can implement a heterogeneous simulator effectively that utilizes the CPU in addition to the GPU. This mainly involves how the computational domain can be decomposed between the GPU and the CPU to achieve an effective decomposition that minimizes communication between them, and maximizes their ability to perform the computations in parallel. The second question asks how different auto-tuning techniques can be applied to improve performance for real-world cases. A typical flood simulation consists of a domain with both dry and wet areas. As the water propagates, one essentially wants the GPU and the CPU to receive a balanced distribution of the workload as they are not necessary equally powerful. This means the wet parts of the domain, i.e., the water should be distributed between these according to their computational power. Also, computing for dry areas are not necessary, which means it should be skipped to avoid wasting compute resources.

1.3 Organization of thesis

The following chapter provides relevant background information to this thesis. More specifically, a short introduction to GPUs and CPUs are given, in addition to the computing platform used by the single-GPU simulator [9]. Then, the mathematical background for the simulator is introduced. Finally, I explain the single-GPU simulator [9], and give an introduction to auto-tuning.

Chapter 3 goes into the details of implementing a heterogeneous simulator that utilizes both the CPU and the GPU. It first provides details about the multi-core CPU implementation. Then, an explanation of the domain decomposition is given. Additional techniques are also explained, i.e., *Ghost Cell Expansion* and *Early Exit*. Finally, I provide performance results of the heterogeneous implementation.

Furthermore, auto-tuning techniques are explained in chapter 4. I first discuss challenges related to dynamic auto-tuning. Then, a *Bounding box* technique is implemented on the CPU, as an alternative to early exit. This technique provides a way to only iterate through wet cells by surrounding all wet cells by a two dimensional bounding box. Finally, I focus on the implementation of two dynamic auto-tuning techniques. Both techniques are combined together to load balance the simulation between the CPU and the GPU. Lastly, performance results of these auto-tuning techniques are given.

Finally, I give my concluding remarks in chapter 5 and review the most important results from this thesis.

Chapter 2

Background

GPU development was traditionally driven by computer games where each pixel can be rendered individually to the screen using several processors to calculate the color of these pixels in parallel [7]. Their steady increase in performance and their ability to process parallel tasks soon made them attractive for high-performance computing. This gave rise to the field of GPGPU (General-Purpose computing on Graphics Processing Units) [25], using GPUs to perform computations that the CPU traditionally was used for. GPGPU became even more relevant as vendors launched programming models directly suited for this purpose, such as the CUDA computing platform which was introduced by Nvidia in 2006 [30]. As a result, GPUs became widely used processors for scientific computing.

Section 2.1 first provides a detailed overview of CPUs and GPUs, including the main differences between these two. In addition, a short introduction to CUDA is given in section 2.2. Furthermore, the shallow-water equations are discussed in section 2.3, together with numerical schemes to solve this system of equations. Then, in section 2.4, I introduce the existing single-GPU simulator [9] that this thesis is based on. I also give a closer description of how the explicit scheme for the shallow water equations is implemented. Finally, an introduction to auto-tuning is given in section 2.5.

2.1 Overview of GPUs and CPUs

As mentioned, GPUs became the primary processor for many computational problems that CPUs were traditionally being used for. However, this does not mean that the CPU should be excluded from these kind of problems, as it is perfectly capable of running many different type of applications [25]. In addition, it has evolved from a single-core to a multi-core processor, making it suitable for parallel tasks as well [25]. Still, there are several important differences between GPUs and CPUs that indicates what kinds of computational problems they are suited for. Section 2.1.1 will discuss some differences between these from an architectural perspective, while section 2.1.2 discusses the concepts of threads for both, highlighting the primary differences between them.

2.1.1 Architectural design

The architecture of the CPU and the GPU differs broadly. The CPU is designed with a latency oriented design in mind [34, 30]. As a result, it has complex logic for cache control where large caches are used to convert memory accesses with long latency into short latency cache accesses. In addition, it also has branch prediction to effectively reduce the latency of branches. A CPU devotes many of its transistors to this. This design orientation limits the number of processing cores that the CPU can pack on the same die due to the increasing area and power consumption [25].

The GPU on the other hand is designed as a throughput oriented device [34, 30] and in contrast to the CPU, it does not have complex cache control or branch prediction. Instead, GPUs are suited for data-parallel computations. This means that a large amount of threads executes GPU programs on many data elements in parallel, giving GPUs a lower requirement for control flow. In addition, the GPU has a relatively high ratio of arithmetic operations to memory operations. This enables it to hide memory latencies by performing calculations instead of using large caches like the CPU. GPUs also have caches, but these are smaller with the goal to achieve higher memory throughput, in turn giving it more efficient access to memory. Furthermore, the GPU is also accessed and controlled by the CPU, effectively forming a heterogeneous system. In addition, it executes asynchronously from the CPU which enables concurrent execution and memory transfers [7].

When it comes to throughput computing, the CPU is a good candidate because it has been the main choice of processor for traditional workloads [25]. Its design orientation has also made CPUs provide the best single thread performance for throughput computing workloads [25]. However, since the CPU has relatively few processing cores, it limits how much data that can be processed in parallel when compared to a GPU that has many parallel processing units suited for throughput computing [25].

2.1.2 Threads

Multi-threading by making use of more than one thread is essential to utilize modern CPUs. However, the concept of multi-threading is quite different between the GPU and the CPU.

For example, *context switching* between CPU threads are different than *context switching* between GPU threads [29, 30]. A *context* refers to the content of different registers and the program counter. For CPU threads the *context switching* is performed by the operating system. More precisely, this involves suspending the current running thread and store its *context* in memory, and then loading the *context* for the next thread into the CPUs registers. Finally, it resumes the thread by returning to the address indicated by the program counter. This is generally a slow and expensive operation. As a result, CPU threads are referred to as heavyweight entities.

In comparison, GPU threads are significantly more lightweight. The lowest unit of execution on a GPU is called a warp and consists of 32 threads. If a GPU has to wait for a warp of threads it can simply begin executing another one, effectively hiding latencies. There is no *context switching* involved since each *context* is maintained on-chip during the warps entire lifetime. As a result, there is no overhead related to *context switching* on the GPU.

Modern GPUs also support several thousand threads compared to a CPU [29]. For instance, modern NVIDIA GPUs can support up to 1536 threads concurrently per multiprocessor. A GPU with 16 multiprocessors therefore supports more than 24000 active threads concurrently. A CPU on the other hand is much more limited than this. Servers with four hexa-core processors can only run 24 threads concurrently or 48 if the CPU support Hyper-Threading.

2.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform by Nvidia for GPGPU programming [30]. GPGPU refers to the use of graphics processing units (GPUs) to perform general purpose computations. CUDA makes it possible to program towards Nvidia GPUs without having to implement the code in graphics APIs like OpenGL [18]. Programming on the CUDA platform is done through the use of C/C++ as the programming language. CUDA also includes extensions to the C language and runtime library to handle the GPU specific parts of the code.

2.2.1 Host and device model

In the CUDA programming model, the CPU and the GPU is known as the host and device, together forming a heterogeneous system [30]. A typical CUDA program consists of C/C++ code that is executed on the host and functions called kernels that are executed on the device. A kernel is an extended CUDA C function that is executed on the device N times in parallel by N numbers of threads. The host will usually execute sequential code while the device takes care of the parallel computations. The host is also responsible for calling CUDA kernels. For example, the code run by the host can consist of initializing data and writing to or reading from files. The device can then use several threads to execute a kernel that perform computations on these data.

The host and device also has their own memory spaces referred to as host memory and device memory [30]. The host memory is the computers RAM while the device memory is the global memory on the GPU. Since they do not share the same memory, an explicit data transfer is required to transfer data between the two memory spaces. This is performed by the host through CUDA runtime functions.

2.2.2 Thread hierarchy

CUDA organizes threads into a hierarchy consisting of *thread blocks* and *grids* [30]. A *thread block* can be viewed as a one, two or three-dimensional array that contains several threads, up to 1024 threads on modern GPUs. Similarly a *grid* can also be viewed as a one, two or three-dimensional array consisting of several *thread blocks*. As a result, a CUDA application can contain several *thread blocks* with multiple threads each. A kernel can be executed by multiple thread blocks in parallel. This makes it possible for several thousand threads to execute a certain kernel in parallel.

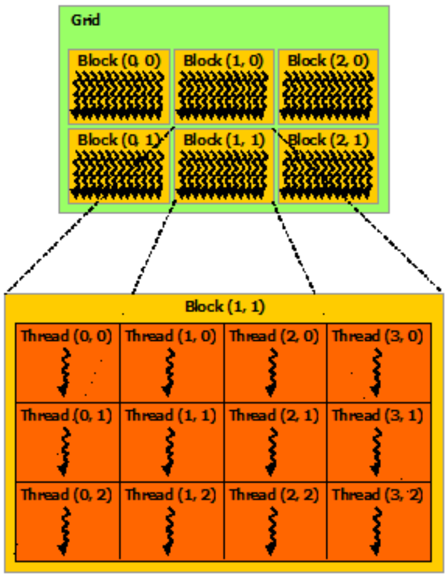


Figure 2.1: Overview of the thread hierarchy in CUDA. A *grid* contains multiple *thread blocks* and a *thread block* contains multiple threads. A kernel can be executed by several *thread blocks* in parallel. Original figure from CUDA C Programming Guide [30]

2.2.3 Memory hierarchy

There are multiple different types of memory spaces in a CUDA application [29, 30]. These are shared memory, local memory, global memory, and two additional read-only memory spaces called constant and texture memory. In addition, each thread also has its own registers.

Registers are the fastest memory that can be used by threads. This is where variables that threads allocate inside kernels are stored. For current high-end GPUs each multiprocessor can contain up to 64000 32-bit registers and as much as 255 registers per thread [30].

Shared memory is the second fastest type of memory and can be as fast as registers if accessed correctly. This memory is shared between all threads for a given thread block making it suitable for effective communication between threads inside a thread block. However, it is very limited in size with a maximum of 48 KB of space per multiprocessor [30].

Local and global memories are considerably slower because these memory spaces are located off-chip. Local memory is per thread meaning each thread has access to its own private local memory. The *nvcc* compiler

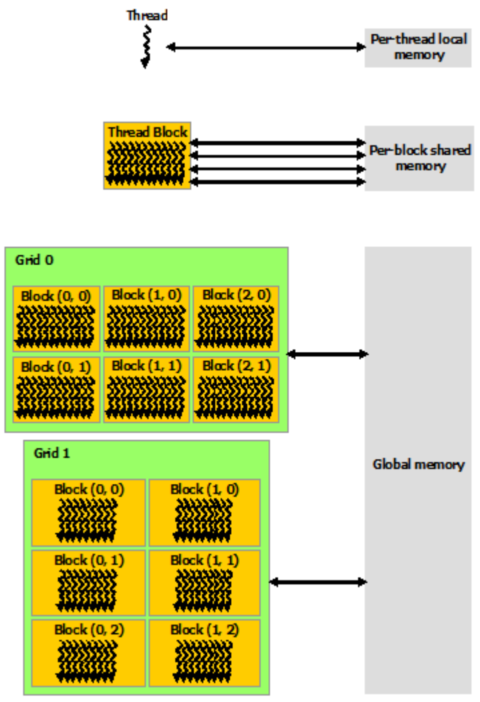


Figure 2.2: Overview of the most important memory spaces in CUDA. From the top, each *thread* has access to its own local memory. They also have access to shared memory which is shared by all threads inside a *thread block*. Finally, all *thread blocks* have access to the same global memory. Original figure from CUDA C Programming Guide [30]

determines what kind of variable will be stored in local memory. Usually, this is variables that are too large to fit in registers, for example structures or arrays. Global memory is the largest memory space, and is as its name suggests global, which means it is shared by the whole application. All threads and thread blocks inside a grid therefore have access to the same global memory. This kind of memory is normally allocated using runtime functions provided by CUDA.

Constant and texture memory are read-only memory spaces and are also shared by all threads. These usually have high access latency similar to global and local memory. However, constant memory can be as fast as registers if all threads of a warp access the same location in memory. Texture memory will get the best performance when threads from the same warp read texture addresses that are close together. All Nvidia GPUs has a constant memory size of 64 KB [30].

2.3 Mathematical background

The shallow-water equations are a set of hyperbolic partial differential equations. These equations can model physical phenomena like tidal waves, tsunamis, rivers and dam breaks. In two dimensions the equations can be written in conservative form:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \tag{2.1}$$

where h is the water depth, hu and hv is the momentum along the x-axis and y-axis on a Cartesian coordinate system and g is the gravitational constant. This can also be expressed in vector form:

$$Q_t + F(Q)_x + G(Q)_y = 0. \tag{2.2}$$

Here, Q represent the variables $\begin{bmatrix} h, hu, hv \end{bmatrix}$ while F and G is the flux along the x-axis and y-axis on a Cartesian coordinate system.

To numerically solve hyperbolic conservation laws, such as the shallow-water equations, one introduces explicit schemes defined over a grid where each grid cell can be updated independently of each other. Two classical schemes for solving this system of equations are the Lax-Friedrichs scheme [23] and the Lax-Wendroff scheme [24]. The Lax-Friedrichs scheme is a first-order scheme [15] while the Lax-Wendroff scheme is a second-order accurate scheme in both space and time [24, 15]. I will describe the Lax-Friedrichs scheme more closely since it is a good introduction to solving the shallow water equations. This scheme can be written in the following way [23]:

$$\hat{f}^n_{i-1/2} = \frac{1}{2}(f_{i-1} + f_i) - \frac{\Delta x}{2\Delta t}(u^n_i - u^n_{i-1}), \tag{2.3}$$

where f is the flux on a cell interface and u is either the water depth or momentum. The notation u^n_i means that u is a given value on the cell i for a timestep n. Another way to write this is $u^n_i = u(x_i, t_n)$. However, the Lax-Friedrichs scheme does not produce very good approximation qualities for solutions containing discontinuities since it smears the solutions as shown in [15]. To improve approximation, high resolution schemes can be used, for example the REA with piecewise linear reconstruction [26].

To correctly simulate tsunamis, dam breaks and flooding over realistic terrain, source terms for bed slope and bed shear stress friction has to be included in the shallow-water equations:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}, \tag{2.4}$$

where B is the bottom topography and C_z is the Chézy friction coefficient. This is expressed in vector form by extending (2.2) to:

$$Q_t + F(Q)_x + G(Q)_y = H_B(Q, \nabla B) + H_f(Q), \tag{2.5}$$

where H_B is the bed slope source term and H_f is the bed shear stress source term.

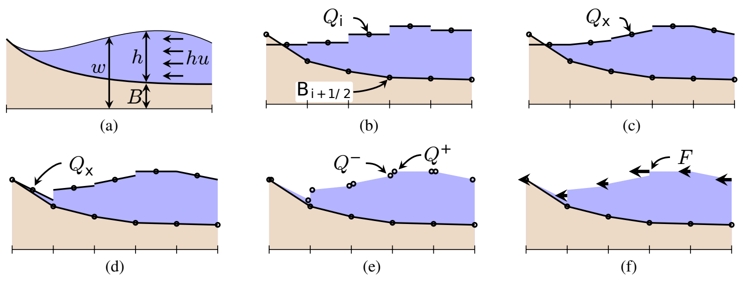


Figure 2.3: Illustration of the water flow and flux computations in one dimension. **a:** Water flow over a bottom topography represented by the continuous variables. **b**: Discretization of conserved variables Q at cell centers and the bottom topography B at cell intersections. **c**: Reconstruction of the slopes using the generalized minmod flux limiter. **d**: The slopes are modified to avoid negative water depth at the integration points. **e**: The conserved variables are reconstructed at the cell interfaces Q^+ and Q^-. **f**: Finally, from these values, the fluxes are computed at each cell interface by using the central-upwind flux function [21]. Original figure from Brodtkorb et al. [9]

From now on, I make a distinction between the water elevation w and the water depth h (see figure 2.3a), and use $Q = [w, hu, hv]^T$ as the vector of conserved variables. Therefore, I use Q_1, Q_2, and Q_3 to denote the water elevation, and water momentum along the x and y-axis respectively. It is important that the numerical scheme to solve this system of equations handles both dry and wet cells to simulate dam breaks and flooding. In addition, Brodtkorb et al. [9] required other properties as well, such as well-balancedness and second-order accurate flux calculations. The explicit Kurganov-Petrova scheme [22] fits well with these requirements and was therefore used in [9]. In this scheme, Q is given as cell averages, and the bathymetry B as a piecewise bilinear surface defined by the values at cell corners, see figure 2.4. Fluxes are computed across all cell interfaces. The spatial discretization of this scheme can be written:

$$\begin{aligned}\frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) \\ &\quad - [F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})] - [G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})]\end{aligned} \tag{2.6}$$

where H_f is the bed shear stress source term, H_B is the bed slope source term, and F and G are the fluxes across cell interfaces along the x-axis and y-axis on a Cartesian coordinate system.

2.4 Shallow water simulations

In this section, a detailed description of the state-of-the-art single-GPU simulator by Brodtkorb et al. [9] is given. Furthermore, their implementation of the explicit Kurganov-Petrova scheme for the shallow water equations is explained. To gain better knowledge of this work I recommend reading [9]. Finally, I also discuss related work for the shallow water equations.

2.4.1 Single-GPU simulator

The single-GPU implementation by Brodtkorb et al. [9] consists of three parts: A C++ interface, a set of CUDA kernels and an OpenGL visualization that renders the simulation to the screen.

The C++ interface handles data allocation, initialization and deallocation. It also moves data between the CPU and GPU as well as invoking the CUDA kernels on the GPU. The CUDA kernels handle the core of the simulation and are used to solve the numerical scheme. There are four different CUDA kernels:

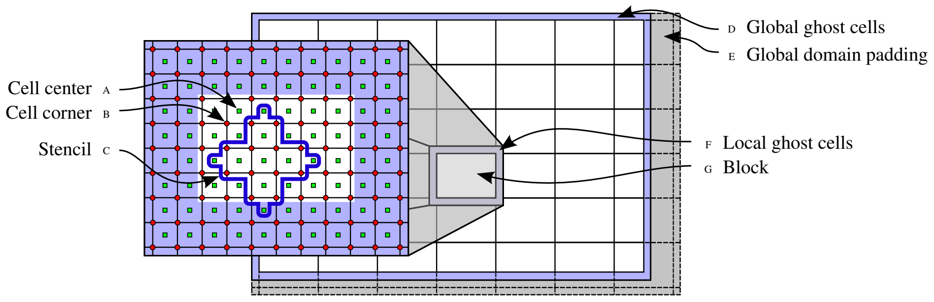


Figure 2.4: This overview shows the computational domain divided into *thread blocks* and cells. The data variables Q, H_B, and H_f are defined at cell centers while the bathymetry B is defined at cell corners. Original figure from Brodtkorb et al. [9]

Flux calculation: This kernel is responsible for the flux calculation across all cell interfaces in addition to the bed slope source term for all cells. It starts by reconstructing the bathymetry B at each interface midpoint along the x and y directions, making it aligned with Q. The rest of the steps performs the computations outlined in figure 2.3. First, the slopes of Q are reconstructed by using the generalized minmod flux limiter as shown in figure 2.3c. This limiter can be written:

$$\begin{aligned}(Q_x)_j &= MM\left(\theta\frac{Q_j - Q_{j-1}}{\Delta x}, \frac{Q_{j+1} - Q_{j-1}}{2\Delta x}, \theta\frac{Q_{j+1} - Q_j}{\Delta x}\right) \\ Q_x &= MM(\theta f, c, \theta b), \end{aligned} \tag{2.7}$$

where f, c and b are the forward, central, and backwards difference approximations to the derivative, while θ controls the numerical viscosity present in the scheme. Brodtkorb et al. [9] used $\theta = 1.3$ which was found to be the optimal value by Kurganov and Petrova [22]. Furthermore, the minmod function is defined as:

$$MM(a,b,c) = \begin{cases} \min_{(a,b,c)} & \text{if } a,b,c > 0 \\ \max_{(a,b,c)} & \text{if } a,b,c < 0\ . \\ 0 & \text{otherwise.} \end{cases} \tag{2.8}$$

However, instead of using branches as in (2.8), a branchless minmod slope limiter [15] was used as this avoids thread divergence on the GPU. When using this reconstruction, one can end up with negative water depth close to dry zones. As the eigenvalues of the system are $u \pm \sqrt{gh}$, the numerical scheme will not handle dry zones when the water depth is negative. To solve this, Brodtkorb et al. [9] simply modified the slopes (Figure 2.3d) to avoid negative water depth at the integration points. Furthermore, the conserved variables are then reconstructed on both sides of the cell interfaces (Figure 2.3e), from which the one dimensional fluxes, F and G from (2.6) are computed (Figure 2.3f) using the central-upwind flux function [21]. It also computes the bed slope source term, H_B from (2.6), for all cells. It then finds the net contribution to each cell by summing the fluxes with the bed slope source term. Finally, the flux kernel is also responsible for computing the eigenvalue at each integration point. Instead of storing one eigenvalue per integration point, it performs efficient shared memory reduction to find the minimum eigenvalue per block. These are stored in a per block buffer which is the input to the next kernel.

Maximum timestep: The second kernel computes the maximum timestep. This is computed by finding the minimum eigenvalue among all eigenvalues in the per block buffer computed in the previous kernel. The timestep, Δt is also limited by the CFL (Courant-Friedrichs-Lewy) condition [26]

$$\Delta t \leq \frac{1}{4} min \left\{ \Delta x / max_\Omega \mid u \pm \sqrt{gh} \mid, \Delta y / max_\Omega \mid v \pm \sqrt{gh} \mid \right\} \tag{2.9}$$

which ensures that the fastest propagation speed is a maximum of one quarter grid cell per timestep.

Time integration: This kernel computes the bed shear stress source term, more precisely H_f from (2.6). Finally, the maximum timestep is used to solve the equations forward in time.

Boundary conditions: The last kernel sets the values of global ghost cells on the data variables Q to apply boundary conditions. This has nothing to do with the numerical scheme, but is essentially used as a condition for handling the boundaries of the domain. One type of boundary condition implemented is *wall* which simulates a wall around the domain. The water elevation, and water momentum variables Q_1, Q_2, and Q_3 at the inner boundaries of the domain are copied into the ghost cells. Q_2 is also reversed in the horizontal direction while Q_3 is reversed in the vertical direction. As a result, the water flows in the opposite direction as soon as it hits the domain boundaries. In addition, there are also three other types of boundary conditions implemented. These are *fixed discharge*, *fixed depth*, and *free outlet*. I will not describe these further as they are not important for this thesis.

The numerical scheme is implemented both with first-order Euler and second-order accurate Runge-Kutta time integration. For second-order, each timestep is implemented as two substeps. The first substep runs all four kernels in order while the second substep only runs *flux calculation*, *time integration* and *boundary conditions*, see figure 2.5.

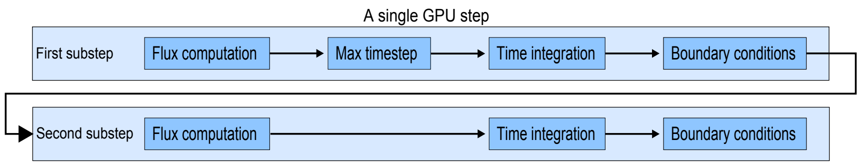


Figure 2.5: An overview of the GPU step function that calls the CUDA kernels. *Flux computation* computes the fluxes F and G together with the bed slope source term H_B. Furthermore, *Max timestep* finds the maximum Δt. *Time integration* computes the bed shear stress source term H_f and solves Q forward in time. Finally, *Boundary conditions* sets the global ghost cells on Q. First-order Euler only executes the first substep while second-order Runge-Kutta executes both substeps.

For the floating point precision of the scheme, single precision is used instead of double precision. The benefit is that data transfers and arithmetic operations can execute twice as fast, while data storage only takes half the space. Brodtkorb et al. [8] have shown that this is sufficiently accurate for the selected scheme. Furthermore, a good block partitioning is crucial for the performance of the numerical scheme on GPUs. Several factors that can affect the performance was found [9], more precisely: Warp size, shared memory access, shared memory size, number of warps for each streaming multiprocessor and global memory access.

The visualizer is implemented in OpenGL. It renders the terrain with a satellite image and the water surface with Fresnel equations to produce photo-realistic rendering. Alternatively it can render the data using a color transfer function in HSV color space which gives a good overview of the water depth.

2.4.2 Related work

There have been many publications on the shallow water equations, both on a single GPU and CPU, but also on multiple GPUs as well. For single GPU solutions, Seitz et al. [32] implemented a two-dimensional shallow water simulation on the GPU using the 2D predictor-corrector MacCormack method. They demonstrated a speedup of more than 100 times on large domain resolution comparing their GPU code to a serial CPU implementation in Fortran on same-generation hardware. de la Asunción et al. [10] implemented a one-layer shallow water system using a first order well-balanced finite volume scheme on GPUs using CUDA. Their CUDA code was implemented with both single precision and double precision. Furthermore, they compared their CUDA implementation to a serial and multi-core CPU version and a GPU version implemented in Cg on same-generation hardware. Their single precision CUDA code achieved a speedup of more than 140 and 40 times over the single-core CPU and multi-core CPU versions respectively. It was also faster than the

Cg GPU version. In addition, their double precision CUDA code achieved a speedup of 5.7 times over the multi-core CPU version.

Many authors have published multi GPU solutions as well. Acuña and Aoki [1] solved the shallow water equations for tsunami simulations using the CIP Conservative Semi-Lagrangian IDO Method. They decomposed their computational domain among multiple GPUs to utilize a multi-node GPU cluster. Their results showed high scalability, both on a single node with multiple GPUs and a cluster of GPUs with multiple nodes. Viñas et al. [40] also implemented shallow water simulations for multi-GPUs and discretized the domain using a first order Roe finite volume scheme. In addition, they included a transport equation to simulate transport of contaminants. As a result, their simulator is able to predict areas that are affected by the propagation of a discharge of pollutant. They performed their benchmarking on a heterogeneous cluster with two nodes, each node containing two M2050 GPUs and an Intel Xeon X5650 CPU with 6 cores and 12 threads via hyperthreading. Their fastest speedup achieved using all four GPUs was 78 times over a single Intel Xeon X5650 CPU.

2.5 Introduction to auto-tuning

Auto-tuning refers to how an application can automatically tune itself to gain increased performance. A common method is to auto-tune specifics parameters to adapt to the underlying hardware. This can increase performance on a number of different hardware configurations, and one avoids manually tuning on different hardware. In this section, I briefly discuss related work for auto-tuning between CPUs and GPUs in addition to stencil computations, as this is the primary focus for this thesis.

2.5.1 Auto-tuning stencil computations

Zhang and Mueller [42] created an auto-generation and auto-tuning framework for 3D stencil codes on GPUs. Their framework generates auto-tuned executable code based on a stencil specification as input. Their auto-generation and auto-tuning framework works in the following way: A stencil specification file is first provided by users as a stencil equation together with parameters such as its dimension and data type. Then, the framework parses this specification and detects necessary information such as names of input/output arrays, in addition to number of floating-point operations for the stencil. This information is used by the code generator to generate executable code that is auto-tuned to the best configuration of parameters based on run-time profiling. The auto-tuning framework performs a search over a parameterized search space for different parameters, such as the CUDA block size and whether to map read-only input arrays into texture memory or not. Their experimental results further showed the performance was competitive to manual code tuning.

Lutz et al. [27] presents a framework to automatically distribute and optimize stencil computations across multiple GPUs on the same node. Their primary focus is related to optimizing communication, more precisely, the exchange of halo cells. Therefore, they used six varied (in terms of number of stencil points, reads, writes, and floating point operations per point) stencil applications that requires communication between sub-domains: *Game of Life*, *Reverse Edge*, and *Swim* which are 2D stencils, and *Hyperthermia*, *Himeno*, and *Tricubic* which are 3D stencils. To perform communication optimizations, they increased the halo size to achieve less frequent communication at the cost of more computations. They found that the optimal halo size varied among the different stencils. In addition, they noted that using all GPUs available in a system is not always the most optimal solution. In some cases, the computational domain is too small to be decomposed into multiple sub-domains. In other cases, a large communication cost can negatively affect the scalability. As a result, they implemented an auto-tuning algorithm that tunes on these parameters, like *GPU selection and partitioning*, the *halo size*, as well as other parameters such as *volume orientation* and *swapping strategy*.

Their tuning for the *GPU selection and partitioning* decides which GPUs to use, including how many. It also decides how to assign partitions to them. Furthermore, the *halo size* was auto-tuned using three different strategies that changes the halo size dynamically as the application runs: *exhaustive search*, *hill climbing search*, and *dichotomic search*.
exhaustive search: The first simply tries every possible halo size and selects the best.
hill climbing search: The second starts at a minimum halo size and increases linearly until the performance decreases for several consecutive points.
dichotomic search: Since both of the previous searches linearly, they may not find the most optimal halo size if it is in the middle of the range. To counter this, their third strategy simply finds the optimal halo size within the best interval.

Their auto-tuning framework gave an average speed up of 1.83 times on two GPUs over a single GPU, and 2.86 times on four GPUs over a single GPU. The three different strategies for the halo size auto-tuning performed similarly. However, on larger input sizes, hill climbing search and dichotomic search performed slightly better than exhaustive search.

2.5.2 Auto-tuning between the CPU and the GPU

Song et al. [33] implemented a heterogeneous tile algorithm to utilize multi-core CPUs in addition to GPUs for dense matrix multiplication. This algorithm first partitions a matrix into tiles of size $B \times B$ and further divides this into smaller tiles of size $B \times b$. These tiles are assigned to the CPU and the GPU. They implemented an auto-tuning technique to decide how to partition the top-level tiles of size $B \times B$ to achieve good load balancing. This is done in three steps, but only the first two are the most important here. It starts with the two-level partitioning scheme that is used to further divide a matrix given the top-level tile size B where B is simply found by searching for the smallest matrix size that gives the best performance for the dominant GPU kernel. It then attempts to find the best partition of size $B \times B_h$ that will be cut from each top-level tile of size $B \times B$. Song et al. [33] further uses the following formula to find B_h:

$$B_h = \frac{Perf_{core} \cdot \#Cores}{Perf_{core} \cdot \#Cores + Perf_{gpu} \cdot \#GPUs} \cdot B, \qquad (2.10)$$

where $Perf_{core}$ and $Perf_{gpu}$ is written in Gflops and is the maximum performance of a dominant computational kernel of the algorithm on either a CPU core or a GPU.

Chapter 3

A shallow water simulator on heterogeneous architectures

The implementation of a heterogeneous simulator is a two-step approach. The single-GPU implementation [9] is first extended to run on a single CPU core, and then multi-threaded to utilize all CPU cores, forming the basis of a multi-core CPU implementation that can be utilized together with the GPU. To finish this, the computational domain has to be decomposed between the GPU and the CPU, giving each their own sub-domain. First of all, this involves the implementation of a domain decomposition technique. This creates additional challenges such as the propagation of water between the sub-domains. To solve this, a simple communication technique called ghost cell exchange is implemented. In addition, important concepts such as asynchronous execution are also taken into account to be able to execute the sub-domains in parallel.

Section 3.1 explains how the single-GPU implementation [9] is extended to the CPU. Several optimization techniques were performed on the CPU code, including multi-threading to take advantage of all cores. This is discussed in section 3.2. Then, the domain decomposition is introduced by giving an overview of several decomposition techniques in section 3.3, together with an implementation where the algorithmic details are explained. Additional techniques that can give better performance and help reducing the overhead of data transfers are also introduced. More precisely, section 3.4 explains a technique called Ghost cell expansion to reduce data transfer overhead, while section 3.5 explains an optimization technique called Early exit. Finally, I present extensive performance benchmarks for the heterogeneous implementation in section 3.6.

3.1 CPU implementation

The single-GPU implementation [9] already utilizes the CPU to perform the sequential parts of the code, for example initializing data which is done only one time at startup. The simulation itself which is highly parallel in nature is executed only on the GPU. Throughout the simulation the CPU only takes care of work like launching CUDA kernels and initiating memory copies between the CPU and the GPU. As a result, the shallow water simulation does not utilize the heterogeneous architecture to its full potential since the CPU can have several cores that are mostly idle. It is therefore important to extend the parallel parts of the code to the CPU so its parallel resources can be taken advantage of.

As discussed in [33] there are several features that have to be taken into consideration when extending the simulation to utilize the CPU as well. First off, the CPU and GPU each have their own memory space which requires an explicit memory copy to transfer data from one memory space to another. Secondly, these data transfers go through the PCI-Express bus that the GPU is connected to. As the speed gap between a GPU and its PCI-Express connection increases, the network will eventually become the bottleneck for the entire system. Finally GPUs are optimized for throughput while CPUs are optimized for latency. As a result, GPUs needs a larger computational domain size to be effectively utilized while smaller domain sizes apply better for CPUs.

To implement the shallow-water simulations on the CPU, I base my implementation on the four CUDA kernels explained in section 2.4.1. As a result, the CPU implementation uses the same numerical scheme as the GPU code, more precisely the Kurganov-Petrova scheme [22]. The CPU implementation is non-trivial since the GPUs data-driven programming model is quite different from the instruction-driven programming model on a CPU. However, the CUDA kernels contain for the most part standard C++ code which makes it easier since the CPU code will be implemented in C++. Listing 3.1 and 3.2 shows a simple implementation of a CUDA kernel and an equivalent C++ function. This should give an idea of the difference in implementation details between the GPU and the CPU code. The CUDA code in listing 3.1 uses the built in CUDA variables *blockIdx* and *threadIdx* to map each thread to a unique cell in the domain. The CPU code in listing 3.2 introduces loops to iterate over the two dimensional domain and perform computations for each cell at a time.

Listing 3.1: CUDA kernel for the GPU

```
__global__ void gpu_kernel(int x, int y)
{
    //Cell index variables for a thread
    int j = blockIdx.x * x + threadIdx.x;
    int i = blockIdx.y * y + threadIdx.y;

    //Computations for cell [i][j]
}
```

Listing 3.2: C++ function for the CPU

```
void cpu_function(int x, int y)
{
    //Loop through each cell [i][j] for a 2D domain
    #pragma omp parallel for private(i, j)
    for(int i = 0; i < y; ++i)
    {
        for(int j = 0; j < x; ++j)
        {
            //Computations for cell [i][j]
        }
    }
}
```

The CUDA kernels discussed in section 2.4.1 are implemented as C++ functions to run on the CPU. However, the computation of the minimum timestep can be performed as part of the flux computations. As a result, the CPU code only contains three functions, *Flux and max timestep computation*, *Time integration*, and *Boundary conditions*, as can be seen in figure 3.1.

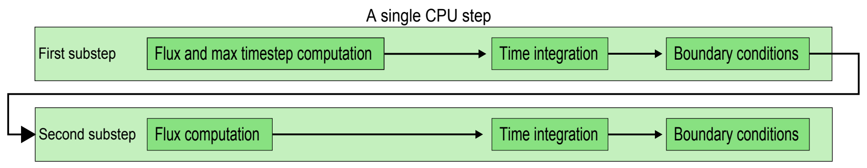


Figure 3.1: An overview of the CPU step function that calls the CPU equivalent versions of the CUDA kernels from figure 2.5. *Flux and max timestep computation* computes the fluxes F and G together with the bed slope source term H_B, and finds the maximum Δt. *Time integration* computes H_f and solves Q forward in time. Finally, *Boundary conditions* sets the global ghost cells on Q. First-order Euler only executes the first substep while second-order Runge-Kutta executes both substeps.

The maximum possible timestep is found by finding the minimum timestep. The CUDA kernel uses a minimum reduction to calculate the maximum timestep from the eigenvalues of each cell. For the CPU code, this computation is straightforward since each iteration can simply compare its eigenvalue with the previous iteration to find the minimum. However, this solution does not work well for a multi-core implementation since it requires a critical section to avoid race conditions. This can in turn decrease the performance. The most efficient solution found was to use the minimum reduction operator available in OpenMP [37] to first let each thread find its minimum value, and then compute the minimum among these.

3.2 Multi-core implementation and optimizations

Several optimization techniques have been implemented to improve the performance of the CPU implementation. First off, it was multi-threaded to fully utilize all CPU cores. In addition, several optimizations have been applied, both general code optimizations and optimizations that are more relevant to the CPU architecture. I also measure the speedup of the most important optimizations.

Multi-threading: The multi-threading was performed by using OpenMP [37]. There are several alternatives to OpenMP. For instance, Boost threads which is part of the Boost C++ library [2]. However, OpenMP is much less error prone since it lets the compiler do the work of decomposing the workload between all threads. Boost threads requires the programmer to create the threads and decompose the domain between these, using a significant amount of time to make sure the code is implemented correctly. Multi-threading using OpenMP works simply by adding the line *pragma omp parallel for* before the loops as shown in listing 3.2.

Kuhn and Petersen found that converting a threaded C program to OpenMP increased robustness without sacrificing performance [20]. They concluded that implementing SMP type parallelism was easier with OpenMP than hand-threading because the last mentioned would require several adjustments to the serial code contra the OpenMP version. First off, hand-threading would require the parallel regions to be moved into a separate subroutine. The programmer would also have to create a data structure off all the private variables for each thread. In OpenMP, these tasks mainly involve adding pragmas to let the compiler perform them. These results show that using OpenMP can save a significant amount of time. In addition, it does not necessarily decrease performance over programmer threaded code.

General optimizations: The flux function was found to be a large bottleneck, using around 90% of the total runtime. This result is similar to the GPU version of the code where Brodtkorb et al. found that the flux kernel used around 87.5% of the total runtime [9]. As a result, I first performed a change in the trade-off for the bathymetry between memory usage and performance. Then, I outline some additional optimizations on the multi-core CPU code.

Initially, for the flux computations, the bathymetry B is reconstructed at each timestep to make it aligned with Q. However, since B is constant through the whole simulation, this reconstruction can instead be performed a single time at the startup. The single-GPU code [9] performed this each timestep as a performance trade-off for less memory and bandwidth usage. However, the CPU is generally not limited by memory size as a CPU typically have access to more RAM compared to the amount of memory available on GPUs. Therefore, I perform this reconstruction a single time at the startup instead of every timestep, effectively improving the performance slightly by using more memory.

Then, the reconstructions of the conserved variables Q were optimized to fit the CPU architecture. As mentioned in section 2.4.1, the GPU code uses a branchless minmod slope limiter since this avoids thread divergence, resulting in better performance on the GPU. However, for the CPU, I utilized the branched version of the minmod function as defined in equation (2.8) since this yielded better performance.

The original GPU code [9] was also compiled with the fast math compiler flags, producing faster but less accurate floating point math, for example related to division and square root. However, their verification and validation experiments showed their solution to be accurate enough to correctly capture analytical solutions and real-world cases. Inaccuracy in floating point computations is often not among the larger problems as other errors such as model errors related to numerical schemes tend to drown the floating point errors [5]. I performed the same optimization on the CPU code, by compiling with *-ffast-math*, yielding faster floating point arithmetic and math functions, but less accurate results.

The numerical accuracy of compiling with *-ffast-math* was further verified. The verification was run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 20m$ in a square computational domain of $200m \times 200m$ with center at $x_c = 100m, y_c = 100m$. It was further run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum in x and y directions, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

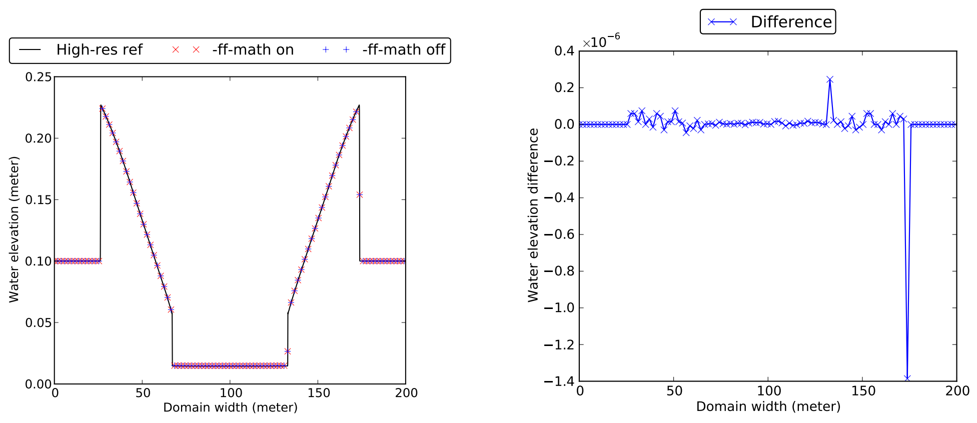
$$Q_1(x, y, 0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave.

Figure 3.2 shows the results from this verification, more precisely the water elevation at the centerline of the domain at a given timestep is provided in figure 3.2a. The high-resolution reference is run at a resolution of 4096^2 while the solutions with fast math enabled and disabled is run at a resolution of 1024^2. Figure 3.2b shows the numerical difference between enabling and disabling fast math, taken from the same centerline. As can be seen, the numerical difference in accuracy is very small. However, it becomes considerably larger at the shocks. Overall, utilizing *-ffast-math* still gives a very accurate approximation.

Performance gains: The speedup of each optimization, including the multi-threading was verified by performing a simulation on a large domain with a resolution of 4096^2. The simulations were run on the Intel Core i7-2600K. The results are provided in table 3.1, showing the speedup of each optimization with respect to the previous.

A reference version has also been provided, running only on a single core and with no optimizations enabled. The *multi-core* optimization (using four cores and 8 threads via hyper threading) resulted in a significant performance gain, running 4.8 times faster than the reference. The *minmod* optimization gained a speedup of 1.2 times. Finally, compiling with *-ffast-math* also resulted in a 1.2 times speedup.



(a) Overview of the water elevation (b) Numerical difference

Figure 3.2: Centerline plot of a second-order accurate Runge-Kutta simulation on an idealised circular dam break with radius $R = 20m$ placed in the center of a domain of size $(x, y) \in [0, 200]m \times [0, 200]m$. The left plot shows the accuracy by executing with fast math enabled and disabled compared to a high-resolution reference solution. The right plot shows the numerical difference in accuracy between enabling and disabling fast math. The difference is overall very small, but becomes larger at the shocks.

Optimization	Speedup
Reference	$1x$
Multi-core	$4.8x$
Minmod	$1.2x$
Fast math	$1.2x$

Table 3.1: Shows the speedup of each optimization with respect to the previous. The reference runs only on a single core and with no optimizations enabled.

3.3 Domain decomposition techniques

Domain decomposition is a task of decomposing a domain in N number of sub-domains where N can equal the number of GPUs and CPUs in the system. Each sub-domain can be computed on its own GPU or CPU. There exist many decomposition techniques that can be used when partitioning a computational domain into several smaller parts. Many papers have investigated such techniques, including [35, 33, 39, 31]. In this section, I will compare different techniques that have the potential to work well when decomposing between a GPU and a CPU.

First of all, there are several challenges related to decomposing a domain into sub-domains. One challenge that arises is the need for communication between the sub-domains. The water has to propagate between all sub-domains. To perform this, the sub-domains have to exchange rows of cells between each other. To address this issue, a communication technique that transfers data between the sub-domains has to be implemented. A second challenge is related to how the various sub-domains can be executed asynchronously of each other, and when to synchronize to make sure the global solution is correct. Finally, one also has to make sure that the timestep, Δt is computed correctly.

These challenges raise several requirements for a good decomposition technique. First off, it needs to be effective when it comes to communication between GPUs and CPUs. Secondly, the technique will have to work well in utilizing all CPU cores and GPUs in the system. Multiple techniques can be used to decompose a domain, for example row based decomposition, column based or even a tile based technique, illustrated in figure 3.3. The following section will discuss these different techniques, while section 3.3.2 will explain the implementation of domain decomposition and address the different challenges related to simulation on multiple domains.

3.3.1 Strategies for decomposition

Row based decomposition One technique is to decompose the domain horizontally in a row wise fashion resulting in N number of sub-domains that each contains several rows of cells. By assuming a global domain of size 600×600, any decomposition can be applied, for example two domains of size 600×300 or three domains of size 600×200. However, it is not limited to sub-domains of the same size. A configuration that contains two domains with a size of 600×400 and 600×200 is also valid. This can work well when the processors in the system are not equally fast. For example, a system with one GPU and one CPU where the GPU is faster and therefore needs to compute on a larger domain to achieve a balanced workload.

Sætra and Brodtkorb have implemented a row decomposition technique similar to this in [35] for a multi-GPU system where they decompose the domain into multiple sub-domains, each consisting of several rows of cells. They further emphasize two advantages with this technique, both related to the communication for transferring data between the sub-domains. The first advantage is that for data transfers between GPUs, it will transfer continuous parts of memory. The data transfers only have to include continuously rows. It is able to achieve this since all the data, row by row are allocated continuously in memory. The second advantage is the small amount of data transfers needed when exchanging cells between subdomains since each sub-domain has a maximum of two neighboring subdomains.

A very similar technique is also implemented in [39] by Venkatasubramanian and Vuduc for a hybrid CPU/GPU system that works for both multi-CPU and multi-GPU systems. Is uses the 2D Poison equation using the Jacobi's iterative method over a structured 2-dimensional grid. Their technique performs a decomposition of the domain into several blocks of rows. It first assigns a block to the CPU(s) while the rest of the blocks are divided between the GPUs.

After the domain decomposition, the computations are performed in three steps:

1: The first step computes one iteration of Jacobi for the blocks that were assigned to the GPUs.
2: The second step performs the same computation for the block that was assigned to the CPU(s). This is done simultaneously as the first step.
3: The last step simply exchange data between the CPU and GPU, more specifically the boundary rows.

They explain the performance of this technique more closely: Because of the overhead in step 3, this technique is only good if step 2 can do most of its computations before step 1 finishes. If the decomposition assigns more rows to the CPU, the time for step 2 increases while the time for step 1 decreases. The ideal decomposition is when step 1 and 2 finishes at almost the same time. Therefore a well-balanced decomposition is crucial for an implementation like this.

Column based decomposition The second technique decomposes the domain vertically in a column wise fashion. The way this works is similar to the row wise fashion. By assuming the same global domain of size 600×600 and decomposing this into two domains vertically, the result is two sub-domains of size 300×600 instead of 600×300 as in the horizontal technique. These two techniques look very similar on first impression. However there is one important difference between the two related to communication. The column based decomposition has to exchange columns on the right and left side of the domain. As a result, it is not possible to transfer continuous parts of memory. In addition, it requires many memory copies to transfer a single column. Therefore, this technique will not work as well in a GPU/CPU system as the row based decomposition since it requires a lot of small transfers over the PCI-Express bus instead of a few larger ones. To maximize memory throughout, one should in general minimize the amount of data transfers between the GPU and the CPU, especially minimize the amount of many small transfers since there is an overhead associated with each transfer [30].

Tile based decomposition Finally, it is possible to combine both of these techniques together and divide both in the horizontal and vertical direction, resulting in several square or rectangular sub-domains. Song et al. presents a technique similar to this in [33]. It introduces techniques to utilize all CPU cores in addition to all GPUs in heterogeneous multi-core and multi-GPU systems for dense matrix multiplication. They implement a heterogeneous tile algorithm. This algorithm uses a two-level partitioning scheme. As input it takes a matrix and divides this into several tiles, small tiles for CPUs and large tiles for GPUs. The reason for using two different tile sizes is that large tiles work better for GPUs performance wise, while for CPUs it is the opposite. The algorithm starts by dividing a matrix into large square tiles of size $B \times B$. Each of these top-level tiles are then divided into a number of smaller rectangular tiles of size $B \times b$ and one remaining tile. For example a 12×12 matrix can be divided into four tiles of size 6×6. Each of these tiles can further be divided into two 6×1 and one 6×4

rectangular tiles. A similar algorithm could be implemented to partition the computational domain of systems of conservation laws. However, it would include expensive communications, essentially having the same problems as the previous technique.

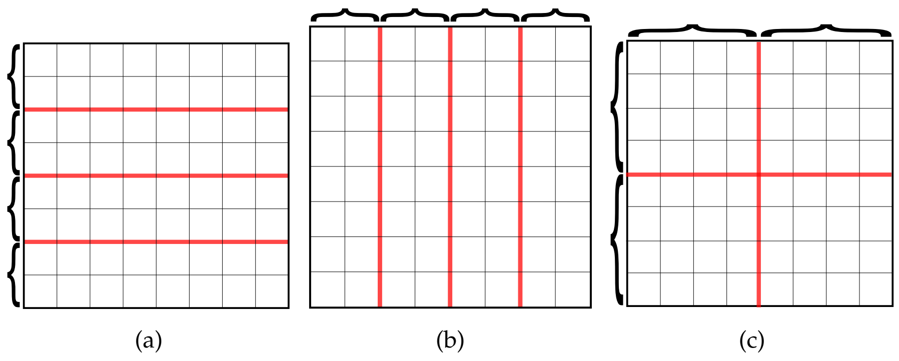

(a) (b) (c)

Figure 3.3: Illustration of the three different domain decomposition techniques. All techniques show how an initial domain can be decomposed into four sub-domains. The left figure applies row based decomposition technique to divide the domain into sub-domains with several rows of cells. The middle figure apples a similar column based decomposition technique to divide the domain into sub-domains of several columns. The right figure applies the tile based decomposition technique to divide the domain into several tiles.

3.3.2 Implementation

I have chosen to use the row based decomposition technique because of its advantages. As mentioned, this is also the technique used in [35] for a multi-GPU system. The advantages with this technique still applies for a GPU/CPU system since all data transfers will have to be transferred between system RAM and GPU global memory. The initial global domain can be decomposed into multiple sub-domains. However, to keep it simple, I first perform a static decompose with two sub-domains equal in size. For example, if the initial domain have a size of 400×400, this is decomposed into two sub-domains each of size 400×200. Using this decomposition, one sub-domain is assigned to the GPU and the other sub-domain is assigned to the CPU. Both sub-domains are updated independently of each other. The GPU updates its sub-domain by executing the CUDA kernels in figure 2.5 while the CPU updates its sub-domain by executing the C++ functions in figure 3.1. The implementation is performed for both first-order Euler and second-order Runge-Kutta time integration.

In the rest of this section I therefore focus on explaining all implementation details related to a multi-domain simulation given only two sub-domains. However, in the final parts of the section, more precisely in section 3.3.5, I will explain how it is extended to a simulation with N number of sub-domains. In addition, the section will also explain how the initial global domain can be decomposed into sub-domains of different sizes.

Timestep calculation As mentioned in section 3.3, one challenge related to simulations with multiple domains, is the calculation of the timestep, Δt. This value will most likely differ between the sub-domains. In [35], Sætra and Brodtkorb discuss two strategies for solving this. The first is simply to use a globally fixed timestep for the whole simulation. However, there is one problem with this strategy: The solution will most likely not propagate as quickly as possible since it requires a timestep which is smaller or equal than the smallest timestep allowed by the CFL condition. The second strategy is to compute the smallest timestep among the timesteps for each sub-domain. This is a better strategy since the solution will propagate as fast as possible. The only downside is that it requires synchronization to make sure all sub-domains have computed their own timestep before finding the smallest of these. This was deemed as the most suitable strategy and is therefore explained in greater detail below together with the technical details of running a multi-domain simulation.

Technical implementation details Listing 3.3 shows the implementation of a simulation that computes on a single domain. A single loop is used to iterate through all timesteps. The number of timesteps depends on the simulation length. At each timestep, a single *step* function executes the GPU and CPU code shown in figure 2.5 and 3.1. More precisely, it computes the fluxes and the maximum timestep. Then it integrates the equations forward in time, and finally performs boundary conditions on the boundaries of the domain.

Listing 3.3: A single simulation step for one domain

```
while(simulation is not finished)
{
    /*
    * Performs a step on a single domain
    */
    step();
}
```

In listing 3.4 this code is extended to a simulation that updates multiple domains. It uses the appropriate strategy for selecting the minimum Δt. Since this code updates multiple sub-domains instead of only a single domain, an array is used to store all the sub-domains. Similarly as in listing 3.3, a single loop is used to iterate through all timesteps. The second strategy for computing the minimum timestep requires us to first compute the fluxes and timestep for all sub-domains. Then, select the minimum of these timesteps, and finally perform time integration and boundary conditions on all sub-domains. As a result, the *step* function is instead split into two functions: *step1* and *step2*. *step1* computes the fluxes and the timestep while *step2* performs time integration and boundary conditions. At the start of each timestep, it performs ghost cell exchange between the sub-domains. This is further explained in section 3.3.3. Then, it iterates through all sub-domains and calls *step1* to compute fluxes and the minimum Δt for each sub-domain. The computations on these sub-domains are performed asynchronously. As a result, synchronization is performed to make sure all sub-domains have computed their Δt. More details regarding the asynchronous execution is given in section 3.3.4. When the synchronization have finished, the CPU iterates through all Δt values to find the minimum of these. This is then distributed to all sub-domains. Finally, it iterates through all sub-domains and calls *step2* to perform time integration and boundary conditions on each sub-domain.

Listing 3.4: A single step for a simulation with multiple sub-domains

```
while(simulation is not finished)
{
   runStep();
}

void runStep()
{
   //Ghost cell exchange
   exchange();

   //Each sub-domain computes fluxes and timestep
   for each subdomain
   {
      step1();
   }

   //Perform synchronization

   //CPU receives and finds the minimum timestep
   for each sub-domain
   {
      min(dt, getDt());
   }

   //CPU transfers minimum timestep to all sub-domains
   for each subdomain
   {
      setDt();
   }

   /*
   * Each sub-domain performs time integration
   * and applies boundary conditions
   */
```

```
    for each subdomain
    {
        step2();
    }

    //Perform synchronization
}
```

3.3.3 Ghost cell exchange

All sub-domains have global ghost cells forming overlapping regions. These ghost cells functions as boundaries that connect the sub-domains together since the stencil needs values from the neighboring sub-domain when computing the fluxes. See figure 3.4 for an illustration of this. This brings us to another challenge mentioned in section 3.3 related to simulations with multiple domains, more precisely the propagation of water between the sub-domains.

To correctly propagate the water between sub-domains, a communication technique that exchanges the ghost cells between the sub-domains has been implemented. It has to exchange a total of four rows of cell data between two sub-domains. The exact data to be copied for each of these cells are the water elevation Q_1, as well as the water momentum Q_2 and Q_3 along the x and y-axis. The technique is illustrated in figure 3.4. Each sub-domain contains internal cells as well as ghost cells around the internal cells. Cells are copied between the sub-domains in the following way: From the bottom sub-domain, the two internal rows at the north boundary are transferred to the bottom ghost cells on the top sub-domain. Likewise, from the top sub-domain, the two internal rows at the south boundary are transferred to the top ghost cells on the bottom sub-domain. The size of the ghost cell overlap between the sub-domains is decided by the stencil which uses two values in each direction. The global overlap therefore contains four rows of ghost cells, thus two rows for each sub-domain.

From a technical point of view the exchange is implemented by first downloading the ghost cells from all sub-domains to *pinned*, page-locked CPU memory. Pinned memory can be accessed directly by the GPU and can therefore be read or written with higher bandwidth than regular pageable memory [30]. The ghost cells are first downloaded from the GPUs sub-domain by using the CUDA runtime function *cudaMemcpy2DAsync* that copies data from the GPU to the pinned CPU buffer. Then, the ghost cells from the CPUs sub-domain are downloaded by using the C function *memcpy* and copy this directly to the pinned CPU buffer similar to the GPU data. Notice that the downloading from the GPU is initiated first using the *cudaMemcpy2DAsync* function. Since this function executes asynchronously, the CPU will not block while the data is transferred from the GPU to the CPU. Instead, it can proceed directly to retrieve the CPU data asynchronously along with the GPU data. At this point synchronization is performed to make sure all data is downloaded before uploading to both sub-domains. The uploading is performed similarly as the download. The data downloaded from the CPU to the pinned CPU buffer is first uploaded to the ghost cells on the GPUs sub-domain by using the CUDA runtime function *cudaMemcpy2DAsync*. Then, the pinned CPU buffer containing the data downloaded from the GPU is uploaded to the CPU using the C function *memcpy*. Again, these operations are executed asynchronously.

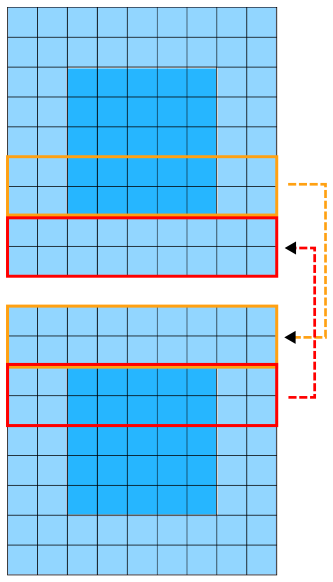


Figure 3.4: A multi-domain solution using two sub-domains. Each sub-domain has a resolution of 5 × 5 cells, or 9 × 9 including ghost cells. The dark blue cells indicate internal cells while the light blue cells represents ghost cells. At the start of each timestep, cell values are copied between both sub-domains. The cell values inside the red rectangle in the bottom sub-domain are copied to the ghost cells in the top sub-domain. Likewise, the cell values inside the yellow rectangle in the top sub-domain are copied to the ghost cells in the bottom sub-domain.

3.3.4 Asynchronous execution

For a solution with multiple domains where the sub-domains have been divided between the CPU and the GPU, it is critical that the computations are executed asynchronously to maximize the parallelism. For example given two sub-domains where sub-domain A represents the top sub-domain while sub-domain B represents the bottom sub-domain. A is assigned to the GPU while B is assigned to the CPU.

When executing a single timestep, the ghost cell exchange is first performed between A and B. As already described this ghost cell exchange is executed asynchronously. Then, the flux and timestep computations are performed for both sub-domains. It is important that as much of these computations are executed in parallel. From a technical point of view this means the GPUs sub-domain, in this case A will have to begin computing first. The CPU asynchronously executes the flux and timestep kernel for the GPU. When performing this asynchronously the GPU starts its computations as soon as the kernels are executed while the CPU immediately continues to perform the flux and timesteps computations for its own sub-domain *B* without blocking and waiting for the GPU to finish. This means as much as possible of these computations will be run in parallel.

When the CPU is finished with its computations, it synchronizes to make sure the GPU have finished and found the minimum timestep. Then, the CPU finds the minimum timestep of the sub-domains *A* and *B* and transfers this timestep to all sub-domains. This transferring is also best performed asynchronously. The CPU first transfers the minimum timestep to the GPU by using the CUDA function *cudaMemcpy2DAsync*. This function immediately returns enabling the CPU to set the minimum timestep for its own sub-domain while the timestep is still transferred to the GPU.

Then, the time integration and boundary conditions are performed for both sub-domains. Similarly to the flux and timestep computations, these will also have to be performed asynchronously which means the CPU first executes the CUDA kernels enabling the GPU to start time integration on its sub-domain *A* while the CPU continues executing time integration on its own sub-domain *B*.

Finally, it is completely valid that sub-domain *B* is assigned to the GPU instead of *A* and that sub-domain *A* is assigned to the CPU. In this case the CPU updates the upper part of the global domain while the GPU handles the bottom part. However, the GPU will still have to be executed first to enable asynchronous execution. In this case the simulation is started for the bottom part first instead of the upper part. Listing 3.5 is a modified version of listing 3.4 to show how the asynchronous execution works.

Listing 3.5: A single step for multiple sub-domains with asynchronous execution

```
while(simulation is not finished)
{
   runStep();
}

void runStep()
{
   //Asynchronous ghost cell exchange
   exchange();

   /*
   * GPU computes fluxes and timestep
   * CPU asynchronously computes fluxes and timestep
   */
   GPU.step1();
   CPU.step1Async();
```

```
    //Perform synchronization

    /*
    * CPU finds minimum timestep
    * CPU asynchronously transfers timestep
    * to all sub-domains
    */

    /*
    * GPU performs time integration
    * and boundary conditions
    * CPU asynchronously performs time integration
    * and boundary conditions
    */
    GPU.step2();
    CPU.step2Async();

    //Perform synchronization
}
```

3.3.5 Extending to N sub-domains and sub-domains of different sizes

The row based decomposition technique has also been extended to work with N number of sub-domains as illustrated in figure 3.5. As a result, there are several sub-domains in the middle that has two neighboring sub-domains instead of only one. A simulation with N number of sub-domains is mostly similar to a simulation with only two sub-domains. The only difference is related to the ghost cell exchange. As mentioned, the middle sub-domains have two neighbors. This means they have to exchange cells between both the upper neighbor and lower neighbor instead of only a single neighbor.

The solution has also been extended to work with sub-domains of different sizes. The main reason for this is that a simulation is implemented to run on both the GPU and the CPU in parallel. It is important that the GPU and the CPU receives a balanced amount of workload to maximize the parallelism. Otherwise, one of them will have to wait for the other to finish. Since a GPU often is faster than a CPU, it needs a larger computational domain than the CPU.

For example if an initial global domain is statically decomposed into two sub-domains A and B, where both A and B is 50% of the initial global domain. Assume A is assigned to the GPU while B is assigned to the CPU. Both receive the same amount of work. If the GPU is considerably faster than the CPU, it may perform the computations for a given timestep on A faster than the CPU performs the equal amount of work on B. As a result, the workload is highly imbalanced. This is problematic since a speedup of

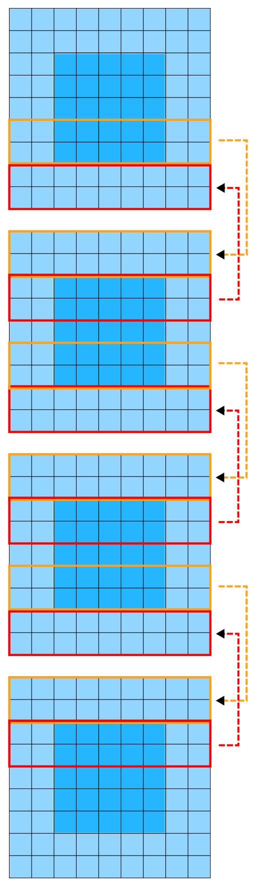


Figure 3.5: A solution with N number of sub-domains. In this case, all the middle sub-domains have to exchange between both the north and south neighbor since they are surrounded by sub-domains on both sides.

the simulation will most likely not be achieved and the GPU has nothing to compute on until the CPU has finished its own computations. A solution to avoid this is therefore to decompose the domain into two sub-domains that are not equal in size. For example, sub-domain A can contain 80% of the initial global domain while sub-domain B can contain 20% the initial global domain. As a result, the CPU will receive less work than the GPU and may finish computations for its sub-domain in almost the same execution time as the GPU. How large each sub-domain should be depends on the size of the initial global domain and how powerful the GPU and CPU used for the simulation are.

To decompose sub-domains into different sizes, some extra calculations have to be performed. I introduce a simple straightforward formula for this:

$$ws = \frac{ywp}{100}, \tag{3.1}$$

where ws is the workload each sub-domain receives specified in number of rows, y is the total number of rows for the initial global domain, and wp is the workload for each sub-domain given in percent. This is performed for each sub-domain which is seen in listing 3.6. It also handles any rest cases of the workload and gives this to the first sub-domain

Listing 3.6: Decomposition into sub-domains of different sizes

```
for each subdomain
{
    ws = y * wp / 100
    sum += ws
}

if sum is less than y
{
    rest = y - sum
    //First sub-domain receives the extra rest
}
```

3.4 Ghost cell expansion

As described in section 3.3.3, I implemented a communication technique that exchanges the ghost cells between the sub-domains to properly propagate the water. This exchange is performed between the GPU and the CPU. Such data transfers can often represent a bottleneck in different systems, mainly because the data has to be transfered over the PCI-Express bus between the CPU and GPU main memory. In [14], Gregg and Hazelwood argue that it is necessary to include the memory transfer overhead when reporting the speed of a particular GPU kernel. To support this they performed benchmarking on a set of kernels for several different GPUs and showed that when including the memory transfer times, it could take 2 to 50 times longer to run a kernel than the GPU processing time alone. They further specified that a faster GPU would be more affected by the PCI-Express overhead than a slower GPU with a similar PCI-Express bandwidth because the fastest GPU is able to perform computations faster than the slower GPU. This results in the overhead being more visible on the fastest GPU. To reduce the overhead related to data transfers, a technique called *Ghost cell expansion* (GCE) has been implemented. GCE has been implemented by several others [35, 11, 40]. A similar technique was also implemented in [19].

The technique is related to *ghost cell exchange* between sub-domains in a simulation with multiple domains. This exchange involves several data transfers between the CPU and the GPU, possibly creating a overhead. GCE is implemented by increasing the global domain overlap. For example, a global overlap of eight cells gives an overlap of four cells per sub-domain. Similarly, a global overlap of sixteen cells gives an overlap of eight cells per sub-domain. The increased overlap is implemented by creating slightly larger sub-domains. Increasing the global overlap therefore enables it to run N number of timesteps before having to swap the overlapping ghost cells between sub-domains, where N is decided by the size of the global overlap. Essentially GCE trades more computations for smaller overheads. For example, by increasing the global ghost cell overlap from four to eight cells, one can run two timesteps before having to exchange. Increasing the overlap again to sixteen cells gives four timesteps before having to exchange. The GCE technique results in fewer data transfers, but more data to transfer at a time. This means the total amount of data transferred over N timesteps are still the same for all global overlaps.

I use two formulas introduced by Sætra and Brodtkorb in [35] to explain this more closely. The time it takes to perform ghost cell exchange every timestep can be written:

$$w_1 = T(m) + c_T + C(m, n) + c. \tag{3.2}$$

Here m and n represents the domain dimensions, $T(m)$ is the time it takes to exchange ghost cells, c_T is the transfer overhead, $C(m, n)$ is the computation time for the sub-domain while c represents additional overheads. This formula can be extended to give the time when using GCE to exchange ghost cells every kth timestep:

$$w_k = T(m) + c_T/k + C(m, n + O(k)) + c. \tag{3.3}$$

Here the transfer overhead is divided by k and the increased overlap is introduced with $O(k)$ which effectively increases the size of each sub-domain.

The GCE technique has resulted in both positive and negative performance in several systems. Sætra and Brodtkorb implemented this technique in [35] for a multi-GPU system and benchmarked it on three different systems: A Tesla S1070 consisting of four Tesla C1060 GPUs, a SuperMicro SuperServer consisting of four Tesla C2050 GPUs, and lastly a desktop with two Geforce GTX 480 GPUs. Exchanging overlapping ghost cells each timestep gave the best result for the Tesla S1070 system. They argued that this was because of the overhead related to data transfers was small. However, a more positive result of GCE was seen on the Tesla C2050 system. They further argued that this was a result of the GPUs being faster, therefore making the data transfers overhead larger. The desktop with two Geforce GTX 480 GPUs had equivalent behavior as the Tesla C2050 system. Overall, the GCE technique resulted in only a small impact. However, when the overhead related to ghost cell exchange is larger, for example when exchanging across multiple nodes, Sætra and Brodtkorb expected the GCE technique to have a better performance effect.

Further experiments related to GCE has also been done in other publications. In [11], Ding and He proposed a GCE technique for reducing communications in cluster systems for partial differential equations (PDE) problems. They achieved a communication speedup of up to 170% with the GCE technique on an IBM SP and Cray T3E. Kjolstad and Snir [19] implemented a similar technique they called *Deep Halo* that can be used for stencil algorithms. The technique is similar to GCE in that it can be used to trade more computations for less frequent communication.

Figure 3.6 shows a standard ghost cell overlap of four cells. Figure 3.7 shows the GCE technique with an overlap of eight cells. The left part of 3.6 shows an initial global domain with 10 × 20 cells. This domain is decomposed into two sub-domains to the right, each of size 10 × 10 cells. The cells outside the black boundaries are the global ghost cells. Cells that contain water are marked in blue. The two sub-domains to the right also contain some light blue cells. These cells are the ghost cells which forms the overlapping region between the sub-domains. Each sub-domain contains two rows of cells from the boundary of the opposite sub-domain. In this case the global overlap is four cells, two on each sub-domain.

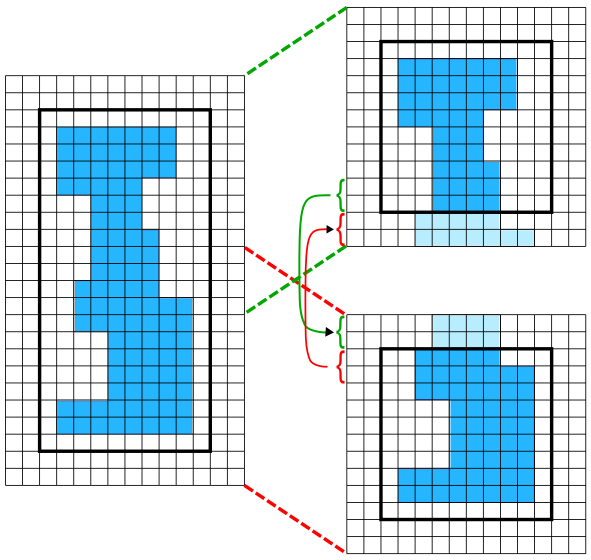


Figure 3.6: A normal ghost cell overlap of four rows, where each sub-domain contains two rows of ghost cells. As a result, the ghost cell exchange is performed every timestep.

Figure 3.7 shows how the GCE technique works. I have used the same initial global domain as in figure 3.6, but this time the global overlap is extended to eight cells, four on each sub-domain. As a result, both sub-domains have been extended in size, as illustrated by the new black line on the two sub-domains. Each of them now has a size of 10 × 12 cells. Both sub-domains now contain four rows of cells from the boundary of the opposite sub-domain. For the ghost cell exchange this means they now exchange twice the amount of data, but does so every second timestep instead of every timestep.

3.5 Early Exit

The early exit optimization aims to improve the runtime of the simulation by avoiding expensive flux calculations for dry cells in the domain since there is no point in updating these cells. This optimization was implemented on both the GPU and the CPU. The performance can be improved a

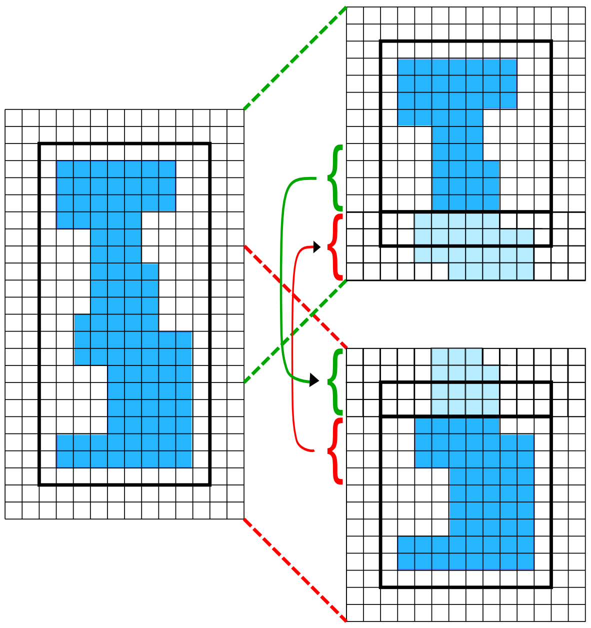


Figure 3.7: Ghost cell expansion with an overlap of eight rows, where each sub-domain now contains four rows of ghost cells. The ghost cell exchange is now performed every second timestep instead of every timestep, resulting in fewer data transfers, but larger transfers each time.

lot for larger domains with a lot of dry areas which is typical for flood simulations. However, as the simulation progress, more water will fill the domain and reduce the performance effect, but should still be quite good in most cases. On the negative side, early exit also involves more work since it needs to write and read from a buffer as well as performing shared memory reduction on the GPU. The time used for this should be minimal compared to the performance benefit. In this section, I first describe how early exit was implemented on the GPU and the CPU on a simulation with only a single domain. Then, I describe the changes performed to make it work on simulations with multiple domains.

Implementation of early exit on the GPU:
The flux and time integration kernel (see section 2.4.1) on the GPU both divide the domain into blocks where each block is computed in parallel by a CUDA block. Each block contains 16×12 cells that are updated by individual threads. The early exit optimization on the GPU involves skipping blocks that only contains dry cells. Since the flux kernel does not know if cells are dry or not, some additional computations has to be performed by the time integration kernel to mark blocks that can be skipped. In the time integration kernel, each thread in a CUDA block updates an independent cell for the block. If a cell contains water, the thread writes 1 to a shared memory buffer. Otherwise, it writes 0. Each CUDA block then performs shared memory reduction on its block to find the maximum value in the

buffer. This means that for a block to be defined as dry, all cells inside this block has to be dry. Finally, this value is written to a dry map which keeps track of blocks that are dry or contains water. It therefore contains 1 for wet blocks and 0 for dry.

Listing 3.7: Early exit optimization on the GPU

```
__device__ bool earlyExit()
{
   //Retrieve the index of the current block
   int bx = blockIdx.x;
   int by = blockIdx.y;

   //Get current block and its four neighbors
   current = Drymap[by][bx]
   top = Drymap[by-1][bx]
   bottom = Drymap[by+1][bx]
   left = Drymap[by][bx-1]
   right = Drymap[by][bx+1]

   //If current and all neighbors are dry, early exit
   if current == 0 and top == 0 and bottom == 0
         and left == 0 and right == 0
   {
      return true
   }
   return false
}

__global__ void flux_kernel(int x, int y)
{
   //Cell index variables for a thread
   int j = blockIdx.x * x + threadIdx.x;
   int i = blockIdx.y * y + threadIdx.y;

   if(earlyExit())
      return;

   //Computations for cell [i][j]
}
```

In the flux kernel each CUDA block reads from this dry map to determine if the block it is updating is dry. If the block and its four closest neighboring blocks are defined as dry, the computations for this block can be skipped. The algorithm for the GPU early exit optimization is shown in listing 3.7. The neighboring blocks has to be checked for dryness since each block in the flux function contains local ghost cells that overlap with the neighboring blocks. Figure 3.8 shows a domain with 6×5 blocks. The green color is used for land areas that are dry while the blue color is used for water. Several of the blocks contain only dry cells which mean these blocks are defined as dry. The upper blocks are both dry and wet and will therefore be defined as wet since they contain at least one wet cell. The stencil shows the four neighboring blocks that has to be checked in addition to the current block.

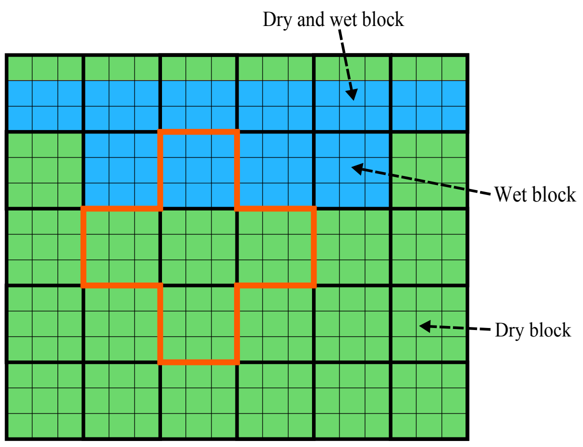


Figure 3.8: Early exit on the GPU: A simplified domain with 6 $\times$ 5 blocks. The green color is used for dry blocks while blue is used for wet blocks. Each block performs a lookup in the dry map to check if it is dry. The orange stencil marks the current block and its four neighbors that has to check for dry area.

Implementation of early exit on the CPU:
The early exit optimization on the CPU is performed in a similar way. However, as there is no notion of blocks since the flux and time integration functions (see section 3.1) simply loops through each cell in the domain to perform updates, the early exit is performed on a per cell basis. The time integration checks each cell for water. If it contains water, the cell is marked with 1 in the dry map, otherwise 0. Similarly to the GPU implementation, the flux function will read from the dry map, which is done for each cell in the domain. If the cell and its four closest neighboring cells are dry, the computations can be skipped for that cell. The algorithm for the CPU implementation of early exit is also provided in listing 3.8. The four closest neighbors have to be checked because each cell update involves a four point stencil operation that uses values from the neighboring cells. Figure 3.9 shows the same domain as used in figure 3.8. However, the domain is now divided into cells instead of blocks. The stencil shows the four neighboring cells that have to be checked in addition to the current cell.

The implementation of early exit on the CPU required a small redesign of the flux function to make it easier to implement it. Unfortunately, the redesigned function did not attain the same performance as the previous function, being slightly slower. The main reason behind this slowdown is the way it performs reconstructions of the cell values. The original flux function performs a single iteration over the domain one time at the start of the function to perform reconstructions for each cell before any

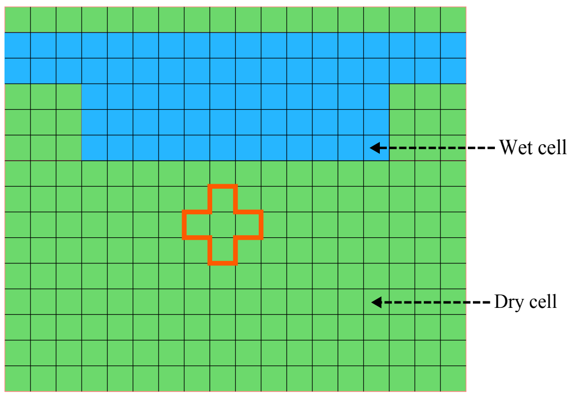


Figure 3.9: Early exit on the CPU: The same domain divided into cells instead. The green color is used for dry cells while blue is used for wet cells. Each cell performs a lookup in the dry map to check if it is dry. The orange stencil marks the current cell and its four neighbors that has to check for dry area.

computations at all. The new function performs the reconstructions in the same loop that iterates over the domain to update it. For each cell, it first performs reconstructions and then computations before handling the next cell. At any given cell it reconstructs the cell values needed for that cell before it performs computations for the cell. Since this cell needs values from its four nearest neighboring cells, it also has to perform reconstructions for the four neighboring cells as well as the current cell. When processing of the next cell starts, it also performs reconstructions for that cell and its four nearest neighbors meaning that for some cells, reconstructions are done at least twice which adds a considerably overhead for larger domains. As a consequence, I decided to keep the original function to use for simulations that does not utilize early exit and use the new code in a new function that is executed when early exit is utilized.

Listing 3.8: Early exit optimization on the CPU

```
bool earlyExit(i, j)
{
    //Get current cell and its four neighbors
    current = Drymap[i][j]
    top = Drymap[i-1][j]
    bottom = Drymap[i+1][j]
    left = Drymap[i][j-1]
    right = Drymap[i][j+1]

    //If current and all neighbors are dry, early exit
    if current == 0 and top == 0 and bottom == 0
           and left == 0 and right == 0
    {
        return true
    }
    return false
}

void flux_function(int x, int y)
{
    //Loop through each cell [i][j] for a 2D domain
    #pragma omp parallel for private(i, j)
    for(int i = 0; i < y; ++i)
    {
        for(int j = 0; j < x; ++j)
        {
            if(!earlyExit(i, j))
            {
                //Computations for cell [i][j]
            }
        }
    }
}
```

Multi-threading problem: Unfortunately, the early exit optimization introduces one problem related to load balancing of the multi-threaded CPU code. Each thread should get as equal workload as possible to make sure all CPU cores are busy most of the time. An imbalanced workload between threads can cause some threads to finish significantly before others resulting in idle CPU cores. As specified in [13] OpenMP assumes that all iterations of a loop have equal runtime. By default, OpenMP will therefore split a loop in large chunks that are equal in size. For example, given two threads, OpenMP splits a loop in a first half assigned to one thread and a second half assigned to the other thread. Since the early exit test skips computations for cells that are dry, this assumptions may not necessarily work well since iterations will either be completely skipped or computed. For example, given the domain in figure 3.9, OpenMP may divide the domain so that the first half that contains rows with a lot of water are assigned to one thread while the next half that contains only dry cells are assigned to the second thread. As a result, the second thread will skip computations for all its cells and finish significantly before the other thread.

According to [36] and [13] it is possible to adjust how OpenMP distributes the workload between threads by changing loop scheduling. For example, a different scheduling mechanism which can be used in the case of the early exit optimization is *schedule(dynamic)*. According to [36] and [13], the *dynamic* keyword will cause OpenMP to use a work queue that contains blocks of iterations. Each thread will retrieve a block of iterations from the top of the queue and execute these. When the thread is finished, it will repeat this process. The number of iterations that a block contains can be specified as an optional parameter. By using this scheduling technique for the early exit case each thread can retrieve a small amount of cells at a time and perform computations for these, and when finished repeat the process. As a result, it is more likely that dry and wet cells are divided more evenly between the threads.

Implementing early exit for multiple domains: The early exit optimization also introduces a problem on simulations that uses multiple domains. For example, given a simulation with two sub-domains A and B, where A contains water while the whole sub-domain B is dry. The dry map computed in the time integration of B will only contain dry cells which means all flux calculations will be skipped in the next timestep. However, at the start of the next timestep, before fluxes are computed, both sub-domains exchange ghost cell. As a result, water can have propagated over to sub-domain B from A. This makes the dry map computed for B in the previous timestep invalid since B no longer contains only dry cells. Some of the ghost cells can contain water which means the cells on either the lower or upper boundary has to perform flux computations.

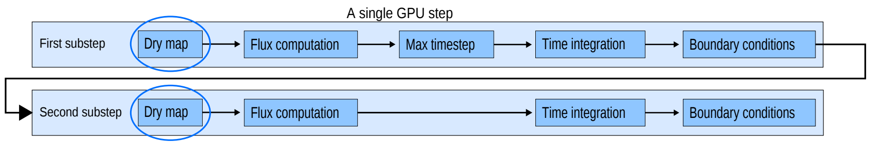


Figure 3.10: An overview of the GPU step function from figure 2.5 with the additional dry map computation for early exit. This computation is optional as indicated by the circle and is only executed if early exit is enabled.

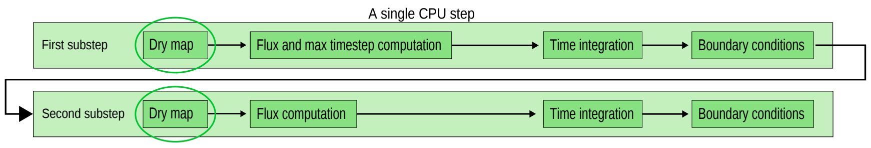


Figure 3.11: An overview of the CPU step function from figure 3.1 with the additional dry map computation for early exit. This computation is optional as indicated by the circle and is only executed if early exit is enabled.

There are several solutions for solving this. An easy solution is to make sure the boundary cells are always computed even if they are dry. However, this is not an optimal solution since it adds slightly more computations than needed and negatively affects the performance. A better solution is to create a new kernel and function for the GPU and CPU as shown in figure 3.10 and 3.11 which computes the dry map instead of computing this in the time integration. The new function is therefore executed after the ghost cell exchange but before the fluxes are computed. When updating the dry map it also has to check the ghost cells for water in addition to the internal domain.

3.6 Performance results

To evaluate the performance of the heterogeneous shallow water simulator, several benchmarks have been performed. These are performed on a desktop with an Intel Core i7-2600K CPU and a Geforce GTX 480 GPU. In section 3.6.4, I also compare this system with another system consisting of an Intel Core i7 Q 740 CPU and Quadro Q1800M GPU. First, I perform benchmarks for the CPU code in section 3.6.1. The multi-threaded code shows very good scalability when utilizing the CPUs four cores together with hyper-threading. Furthermore, the single-threaded and multi-threaded implementation of the CPU code is compared, including the use of early exit for both of them. The multi-threaded CPU code shows great performance increase over the single-threaded CPU code. This is more apparent for larger domains than smaller. In addition, the performance increases for the CPU on larger domain resolutions when early exit is utilized. Section 3.6.2 continues by showing performance results between the GPU and the multi-core CPU code. These results also include the use of early exit. The main results shows that the GPU is around 12 times faster on very large domain resolutions. The GPU also shows a solid performance increase for larger resolutions when early exit is utilized, but a decrease for smaller.

In addition, section 3.6.3 provides benchmarks for the GCE technique. The primary result is a linear decrease in performance when increasing the GCE overlap. As a result, using no GCE overlap therefore provides the best performance. Finally, in section 3.6.4, I provide benchmarks that shows the optimal workload balancing between the GPU and the CPU. These benchmarks are executed on the two different systems mentioned above, a desktop with an Intel Core i7-2600K and a Geforce GTX 480, and a laptop with an Intel Core i7 Q 740 and Quadro Q1800M. Solid performance is gained when utilizing the CPU together with a weaker GPU, as the system with an Intel Core i7 Q 740 and Quadro Q1800M shows a performance gain when 20% of the domain is assigned to the CPU. However, the results from the system with Intel Core i7-2600K and a Geforce GTX 480 does not show any relevant performance increase when utilizing the CPU together with the GPU, but it does not necessarily decrease performance either.

All benchmarks have been run using a known test case described in [38] that consider the domain as an idealised circular dam break surrounding a water column placed in the center of the domain. The initial conditions for the benchmarks are further described in each section.

3.6.1 Single-thread and multi-thread CPU performance

I first perform a benchmark that shows the scalability of the multi-threaded CPU code. This benchmark was run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 200m$ in a square computational domain of $2000m \times 2000m$ with center at $x_c = 1000m$, $y_c = 1000m$. It was further run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions

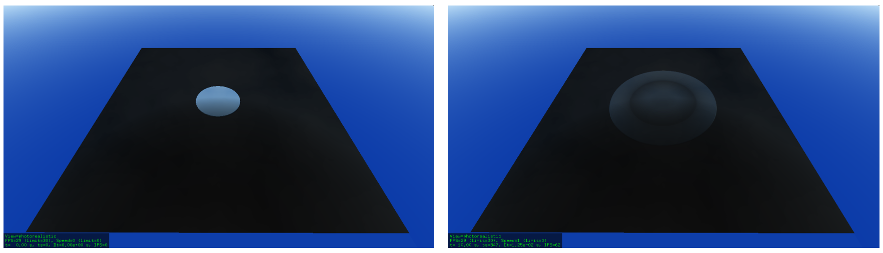

(a) (b)

Figure 3.12: Example of an idealised circular dam break. **Left**: At time $t = 0$, the circular dam instantaneously collapses which causes the water to flow in all directions. **Right**: The water flow at time $t = 10$.

for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 30$. The domain resolution used was 4096^2 to make sure all threads were fully utilized.

The results for this benchmark are provided in figure 3.13. The x-axis for this plot shows the number of threads used for the simulation while the y-axis shows the performance. Furthermore, it contains two graphs: One that shows the ideal scaling for up to sixteen threads, and one that shows the achieved scaling. The code scales very good from one to four threads across the four physical cores. However, the scaling is not perfect. There are mainly two reasons for this: First off, the threads stalls as a result of memory latencies when reading data from system RAM. This will negatively affect the performance since the threads will not execute perfectly in parallel. A second reason can be related to overhead when multi-threading. For example, a result of OpenMP using more time to decompose the workload between the threads as the thread count increases, but also because that each thread receives less work since there are more threads to use for the total workload.

As this CPU features hyper-threading technology, it benefits from an additional four threads. This means that each core has the ability to switch between executing two threads, giving a total of eight threads that can be utilized. However, only four of these threads can be executed in parallel since the CPU has four physical cores. As a result, when utilizing the additional four threads via hyper threading, it does not scale perfectly up to eight times. This is expected as hyper-threading does not give any more parallelism since there are, as mentioned, only four physical cores. The multi-threaded performance still increases to better than four

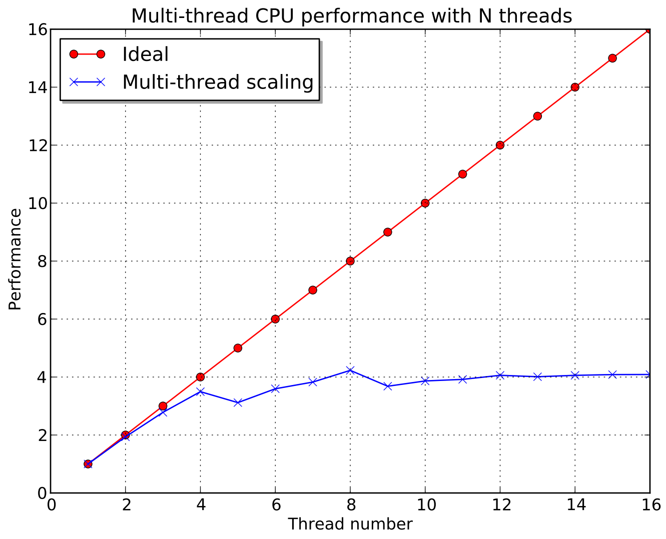


Figure 3.13: Multi-thread scaling using up to sixteen threads on an idealised circular dam break: The red graph shows the ideal scaling while the blue graph represents the scaling achieved with the Intel core i7-2600K (four cores and eight threads via hyper threading). It achieves better than perfect scaling when utilizing all eight threads available via hyper-threading, as only four of these threads run completely in parallel. This means it is able to hide memory latencies. No further performance is gained when increasing the thread number from eight to sixteen, but this is expected as the CPU can only utilize a total of eight threads.

times over the single-threaded performance. As a result, hyper-threading manages to perfectly hide the memory stalls as each core can switch to executing another thread when one of its threads stalls on a memory operation. Finally, the plot shows the performance when further increasing the number of threads to sixteen. No additional performance is gained from this as expected, since the CPU cannot execute any more threads in parallel. It is also apparent that executing with five, nine or thirteen threads gives a slight decrease in performance. I expect the main reason for this to be because not all cores are executing the same amount of threads.

In addition, I have performed a benchmark between the single-threaded CPU code and the multi-threaded CPU code. This benchmark was also run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 20m$ in a square computational domain of $200m \times 200m$ with center at $x_c = 100m$, $y_c = 100m$. Furthermore, it was executed with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 1m & \text{if } (x-x_c)^2 + (y-y_c)^2 \le R^2 \\ Q_1 = 0.1m & \text{if } (x-x_c)^2 + (y-y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 30$.

The benchmarking was performed with early exit enabled and disabled for both single-threaded and multi-threaded code. In addition, it has been executed on several different domain resolutions from 64^2 and up to 2048^2. I have provided two plots to show the results. Plot 3.14a shows the execution time between the single-threaded and multi-threaded code with early exit enabled and disabled for both. The y-axis shows the execution time in seconds. Plot 3.14b shows the relative increase or decrease in performance by enabling early exit, shown along the y-axis. In both plots, the x-axis shows the domain resolution. Furthermore, the execution times are also provided in table 3.2 and 3.3.

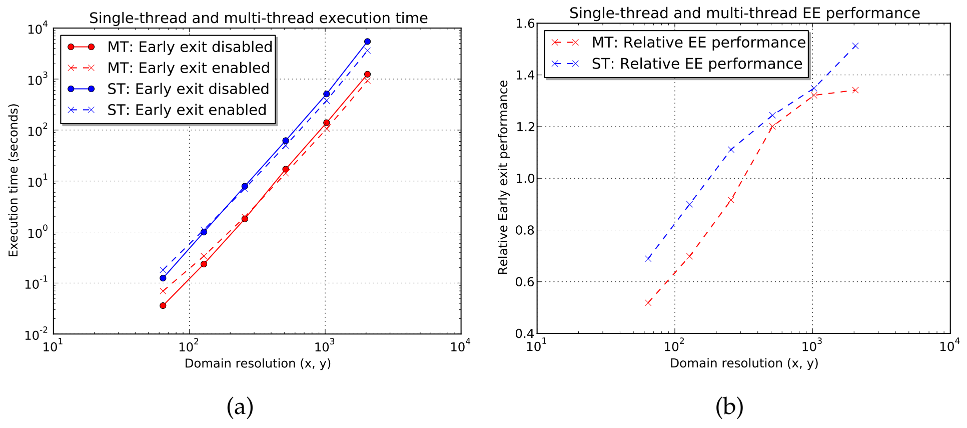


Figure 3.14: Performance comparison between a single-threaded and multi-threaded (4 cores and 8 threads via hyper threading) CPU simulation on an idealised circular dam break. *MT* and *ST* is short for multi-threaded and single-threaded respectively. By comparing the CPU code when early exit is enabled and disabled, the former becomes better as the domain resolution is increased. In the right plot, notice that the single-threaded code is able to better utilize early exit than the multi-threaded code.

There are several interesting findings based on the results. First off, a very good performance increase is shown when utilizing all cores on the CPU. However, it is noticeable that the performance increase is better when early exit is disabled. In this case, the average speed up for the multi-threaded code between the domain resolutions is almost 4. Overall, the results shows that the multi-threading scales well with the number of cores. However, when early exit is enabled, the speedup never goes above 4 for any domain resolution. In other words the performance gain from multi-threading is considerably lower. I expect the main reason behind this is that the cores are not utilized as well when early exit is enabled, mainly because of the problem discussed in section 3.5 related to imbalance in workload

Domain	CPU single-threaded	CPU multi-threaded	Speed up
64^2	1.2E-1	3.6E-2	3.3
128^2	1.0E+0	2.4E-1	4.2
256^2	7.9E+0	1.8E+0	4.4
512^2	6.2E+1	1.7E+1	3.6
1024^2	5.1E+2	1.4E+2	3.6
2048^2	5.4E+3	1.2E+3	4.4

Table 3.2: The execution times in seconds of the single-threaded and a multi-threaded (4 cores and 8 threads via hyper threading) CPU simulation with early exit disabled.

Domain	CPU single-threaded EE	CPU multi-threaded EE	Speed up
64^2	1.8E-1	6.9E-2	2.6
128^2	1.1E+0	3.4E-1	3.2
256^2	7.1E+0	2.0E+0	3.5
512^2	5.0E+1	1.4E+1	3.6
1024^2	3.8E+2	1.1E+2	3.4
2048^2	3.6E+3	9.3E+2	3.9

Table 3.3: The execution times in seconds of the single-threaded and a multi-threaded (4 cores and 8 threads via hyper threading) CPU simulation with early exit enabled.

between the threads.

Secondly, for smaller domain resolutions the speedup is not as great compared to the larger resolutions. This is the case both when early exit is enabled and disabled, where it is noticeable that the speedup is only around 3 times. This shows that the CPU cores are better utilized on larger domain resolutions compared to smaller. I expect the reason for this to be because of the overhead when OpenMP load balances workload between threads. This is less noticeable on larger domains since each thread receives a larger part of the domain, in turn leading to more computations.

Finally, the results also show that enabling early exit increases performance for larger domain resolutions, but decreases for smaller resolutions. I expect the main reason for this to be because of the overhead when enabling early exit. This overhead has less impact on larger resolutions due to the fact that there are many more cells to perform computations for. As a result, when most of the domain is dry, there are more cells that can take advantage of the early exit technique.

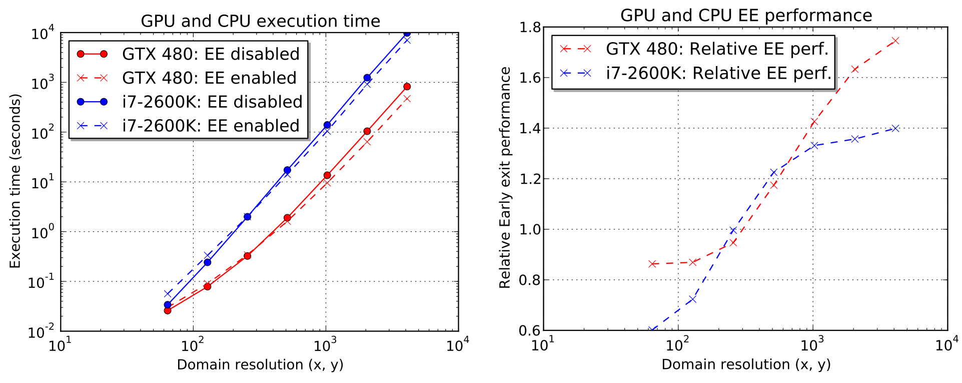
3.6.2 GPU vs CPU performance

A performance benchmark between the GPU and the multi-core CPU implementations has also been performed. This benchmark have been executed on a flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 20m$ in a square computational domain of $200m \times 200m$ with center at $x_c = 100m$, $y_c = 100m$. It was also run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x, y, 0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 30$.

Again, I have included the early exit performance by enabling early exit for both the GPU and the CPU. Several different domain resolutions have been executed, ranging from 64^2 to 4096^2. In addition, two plots are provided to show the results. Plot 3.15a shows the execution time between the CPU and the GPU with early exit enabled and disabled for both. The y-axis shows the execution time in seconds. Plot 3.15b shows the relative increase or decrease in performance when early exit is enabled, shown along the y-axis. For both plots, the x-axis shows the domain resolution. Furthermore, the execution times of the GPU and CPU are also provided in table 3.4 and 3.5.



(a) (b)

Figure 3.15: Performance comparison between a GPU and CPU (4 cores and 8 threads via hyper threading) simulation on an idealised circular dam break. Again, by comparing with early exit enabled and disabled, the former increases in performance as the domain resolution is increased. In the right plot, notice that the GPU is able to better utilize early exit, especially for larger domain resolutions, and gains a better performance increase than the CPU.

The results indicates that the GPU is only slightly faster on small domain resolutions like 64^2 and up to 12 times faster on larger resolutions like 4096^2 when early exit is disabled. It is also apparent that the GPU performs better than the CPU for each increasing domain resolution. This is a result of GPUs architecture; they are optimized for maximum throughput and are able to run thousands of threads in parallel while CPUs are optimized for latency. GPUs are therefore utilized better when used for computations on larger domains compared to smaller, while for CPUs it is the opposite. For larger resolutions like 1024^2 and upwards, this trend starts to even out. The main reason for this is that this particular GPU has reached its maximum potential in performance and is no longer able to run more threads in parallel for each increasing domain resolution.

These findings are also supported in [33] by Song, et. al for their heterogeneous tile algorithm explained in section 3.3.1. They found that large tiles will clobber a CPU core while small tiles do not obtain the best utilization of GPUs. Because of these findings, they used small tile sizes for CPUs and larger tile sizes for GPUs.

The early exit optimization also provides a good performance gain for both the GPU and the CPU implementations. However, the GPU gains a larger speedup compared to the CPU when using early exit. This means the early exit optimization gives better performance increase on the GPU or the overhead related to early exit is higher on the CPU. However, for both the GPU and the CPU, it is apparent that for small domain resolutions, enabling early exit gives a decrease in performance while it increases performance for larger domain resolutions. The main reason for this is the same as discussed in the previous section. As domain resolutions increase, the performance gain from using early exit also increases. This increase is relatively the same for both the GPU and the CPU. However, for very large domain resolutions, the GPU has a slightly better performance gain. The main reason for this is that the overhead related to early exit is slightly higher on the CPU.

Domain	CPU	GPU	Speed up
64^2	3.4E-2	2.6E-2	1.3
128^2	2.4E-1	7.9E-2	3.0
256^2	2.0E+0	3.2E-1	6.2
512^2	1.7E+1	1.9E+0	8.9
1024^2	1.4E+2	1.4E+1	10.0
2048^2	1.2E+3	1.0E+2	12.0
4096^2	9.8E+3	8.2E+2	12.0

Table 3.4: The execution times in seconds of the GPU and multi-threaded (4 cores and 8 threads via hyper threading) CPU simulation with early exit disabled.

3.6.3 Ghost cell expansion

The GCE technique has also been benchmarked by increasing the overlaps. The results are shown in Plot 3.16. The benchmark was run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 20m$ in a square computational domain of $200m \times 200m$ with center at $x_c = 100m$, $y_c = 100m$. In addition, it

Domain	CPU EE	GPU EE	Speed up
64^2	5.6E-2	3.0E-2	1.9
128^2	3.3E-1	9.0E-2	3.7
256^2	2.0E+0	3.4E-1	5.9
512^2	1.4E+1	1.6E+0	8.7
1024^2	1.0E+2	9.5E+0	10.5
2048^2	9.2E+2	6.4E+1	14.4
4096^2	7.0E+3	4.7E+2	14.9

Table 3.5: The execution times in seconds of the GPU and multi-threaded (4 cores and 8 threads via hyper threading) CPU simulation with early exit enabled.

was run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum in x and y directions, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 30$. The domain resolution used is 1024^2. The x-axis shows the percentage of the domain that is assigned to the CPU. The y-axis shows the number of the global GCE overlap. I have used overlaps ranging from 8, 16, 24 and up until 200. The z-axis shows the execution time measured in seconds.

Plot 3.16 shows a linear decrease in the performance when increasing the GCE overlap. In other words, the best result is obtained when using no additional GCE overlap. These results indicates that the overhead related to data transfers is small, therefore resulting in worse performance when increasing the overlap since this gives more computations.

To confirm this finding, I added the possibility to increase the execution time of the function that performs ghost cell exchange by introducing a constant delay. This delay is performed by calling the function *usleep* at the start of the ghost cell exchange function. The point of this is to simulate slower data transfers and run experiments to see if varying the GCE overlap will result in improved performance. I therefore performed an additional benchmark with a constant delay of 20 ms applied to the ghost cell exchange. This benchmark uses the same initial conditions as the previous one. The ghost cell exchange is now guaranteed to use at least 20 ms to complete in addition to the actual data transfers which results in an execution time slightly higher than 20 ms. The result of this can be seen in plot 3.17. I used the same overlaps as in plot 3.16, ranging from 8 to 200 on a domain with resolution of 1024^2. The plot shows that when increasing

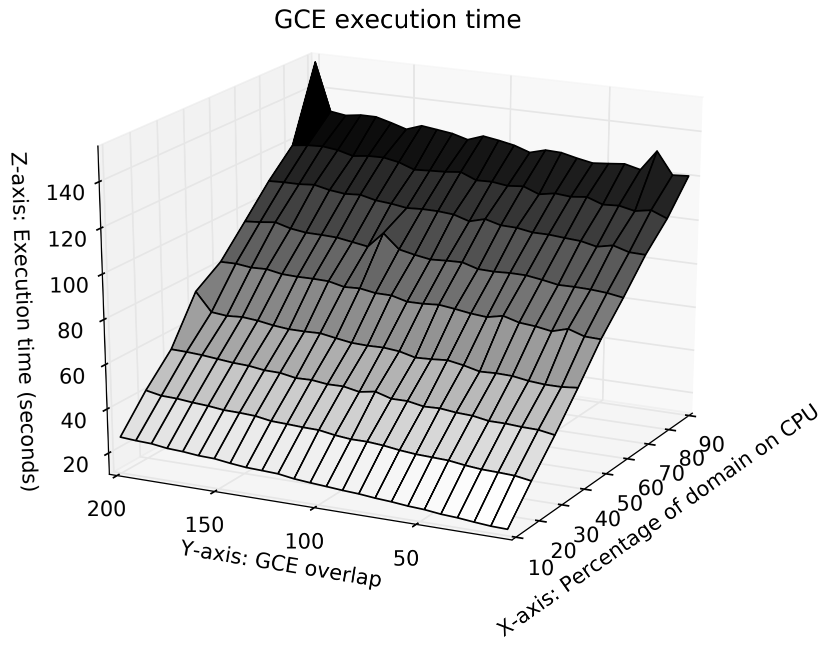


Figure 3.16: Execution time of ghost cell expansion on an idealised circular dam break with a domain resolution of 1024^2. It performs best when no additional GCE overlap is used. This is also true when the part of the domain assigned to the CPU is increased. Also notice the small spikes at points $(60, 128)$, $(90, 24)$, $(90, 200)$, and $(40, 200)$. These spikes shows configurations that performs considerably worse than the rest.

the global overlap, the performance improves. This is because there are fewer and fewer calls to the expensive exchange function as the overlap increase, and since the additional computations are not very expensive, the performance increases. This improvement in performance continues up to an overlap of about 72. At this point the additional computations starts to get noticeably expensive, which is why the performance gradually decreases. As a result, the ideal overlap in this case is around 72.

Both GCE plots also contains several small spikes where the simulations performs slightly worse. These are noticeable at points $(60, 128)$, $(90, 24)$, $(90, 200)$, and $(40, 200)$. These sudden spikes are related to the CPU code as they are only visible when parts of the domain have been assigned to the CPU. I expect the main reason for these spikes to be because that the specific domain resolution used in these cases does not fit well with the CPU architecture. In addition, I expect this can be optimized to run better.

To conclude, this shows that my original exchange function without any constant delay is sufficiently fast which is why plot 3.16 shows the best performance is gained when using no additional overlaps. As a result, this means that there are not much overhead related to data transfers between a CPU and GPU in the same computer. However, these findings are only supported for the desktop I benchmarked on.

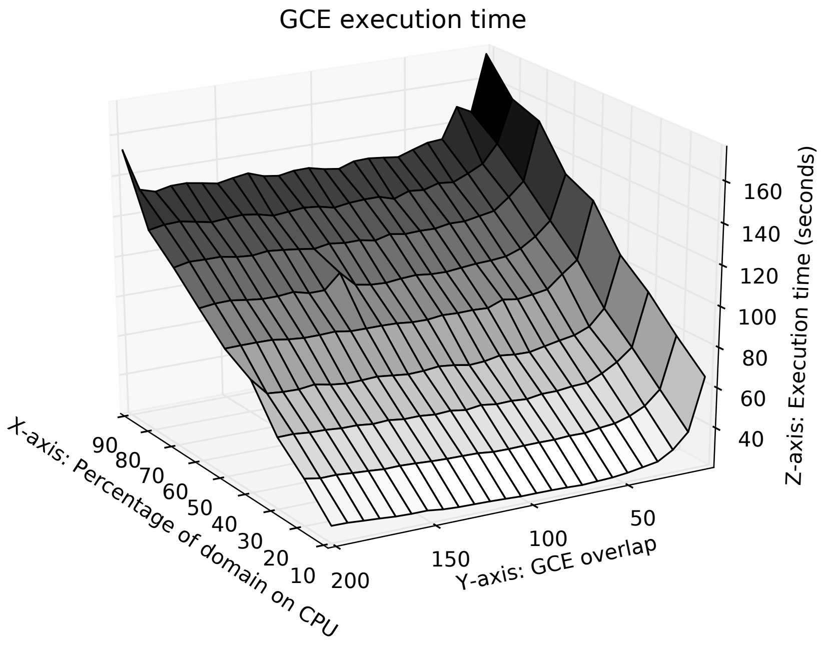


Figure 3.17: Execution time of ghost cell expansion on an idealised circular dam break with a domain resolution of 1024^2. It performs best when the GCE overlap is around 72. This is also true when the part of the domain assigned to the CPU is increased. Also notice the small spikes at points $(60, 128)$, $(90, 24)$, $(90, 200)$, and $(40, 200)$. These spikes shows configurations that performs considerably worse than the rest.

3.6.4 Static domain decomposition

So far, I have only shown performance results between the CPU and the GPU. In addition, it is also important to examine the performance of the heterogeneous shallow water simulator when utilizing both the GPU and the CPU simultaneously for computations. The goal is to be able to run a simulation faster using both these resources compared to only a single GPU. This performance benchmark therefore gives some insight into how the workload should be split between the GPU and the CPU for different domain resolutions to achieve the best execution time.

As mentioned earlier, this benchmark is executed on two different systems: The first system is a desktop that consists of an Intel Core i7-2600K and a Geforce GTX 480. The second system is a laptop with an Intel Core i7 Q 740 and Quadro Q1800M. The results for the first system is shown in plot 3.18 while the results for the second system is shown in plot 3.19. I will refer to these two systems as the **desktop** and **laptop**. The x-axis for both plots shows the percentage of the domain that is assigned to the CPU. The y-axis shows the domain resolution while the z-axis shows the execution time in seconds

The benchmark for the desktop was run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 20m$ in a square computational domain of $200m \times 200m$ with center at $x_c = 100m$, $y_c = 100m$. It was further run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 20$.

When the CPU is assigned around 5 to 10% of the domain, there is almost no improvement over only utilizing the GPU. However, a small improvement can be noticeable at larger domain resolutions. As the CPU is assigned even larger percentages of the domain, it is clear that the performance quickly decreases. The reason for this is simple. The GPU used is very powerful compared to the CPU. The results from section 3.6.2 shows that the Geforce GTX 480 is around 12 times faster than the Intel Core i7-2600K on large domain resolutions. The GPU should therefore be assigned most of the computational domain.

The benchmark for the laptop has been executed on the same test case as the desktop. However, the simulations were executed until time $t = 5$. The peak performance is achieved when 20% of the domain is assigned to the CPU. This applies for both small and large domain resolutions, but is more noticeable at the larger resolutions. When 25% or more of the domain is assigned to the CPU, the performance gradually decreases. As a result, the GPU and the CPU for the laptop are much more close to each other in performance than the GPU and the CPU for the desktop. Therefore, it is apparent that significant increases in performance can be achieved when simulating on a laptop or desktop with a weaker GPU, and not too strong compared to the CPU.

Also, for both systems, notice the spikes that shows configuration with a considerably worse performance. Both the desktop and laptop benchmarks contains large spikes at point $(40, 1800)$ and $(70, 2000)$. Plot 3.18 for the desktop also contain a smaller spike at point $(40, 800)$. I expect these spikes to be related to the CPU since they only occur when specific parts of the domain are assigned to the CPU. Most likely, this can be optimized to run better by modifying the CPU code.

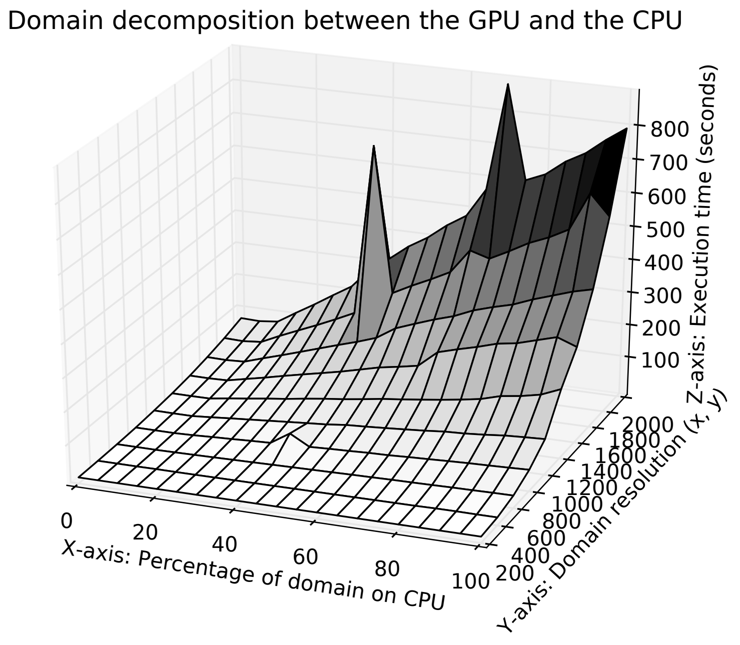


Figure 3.18: Desktop: Static decomposition between the Geforce GTX 480 GPU and Intel Core i7-2600K CPU on an idealised circular dam break with several different domain resolutions. Notice that when assigning around 5 to 10% of the domain to the CPU, the performance almost equals to only using the GPU. However, it quickly decreases when larger parts of the domain are assigned to the CPU.

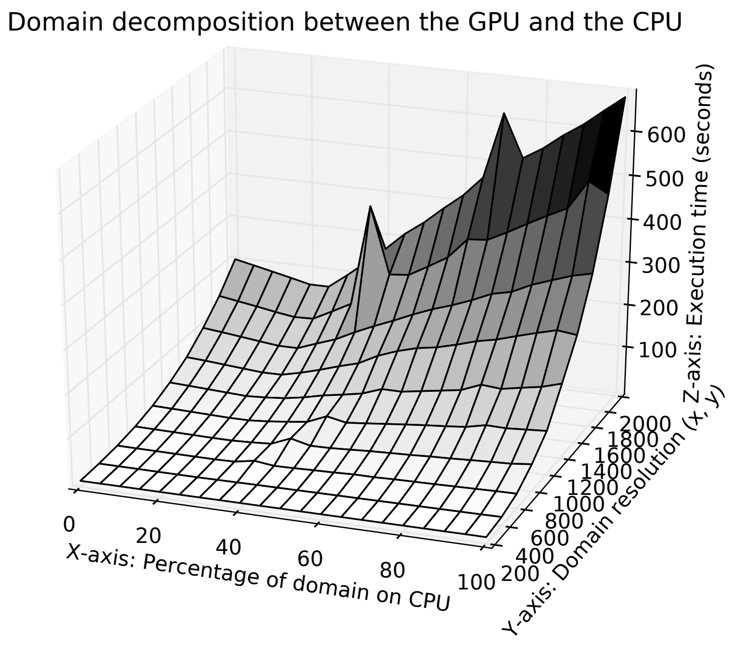


Figure 3.19: Laptop: Static decomposition between the Quadro Q1800M GPU and Intel Core i7 Q 740 CPU on an idealised circular dam break with several different domain resolutions. The peak performance is achieved when 20% of the domain is assigned to the CPU.

Chapter 4

Auto-tuning

In chapter 3, I presented a heterogeneous system that utilizes both the GPU and the CPU as computational resources. However, when performing computations using both of these resources, it is important that the workload between these is distributed such that ideal simulation speed is obtained. In this chapter, I focus on dynamic auto-tuning techniques on heterogeneous architectures to improve the heterogeneous system presented in chapter 3. In section 4.1, I first discuss auto-tuning for three different parameters that can be applied, more precisely *GCE* (section 3.4), *Early exit* (section 3.5), and *Domain decomposition* (section 3.3). In addition, I discuss challenges related to dynamic auto-tuning. Then, an alternative technique to the early exit optimization was implemented for the CPU, to specifically increase the CPUs performance related to auto-tuning. This technique is discussed in section 4.2. I continue by presenting the implementation details for the auto-tuning techniques. Section 4.3 explains the auto-tuning technique implemented for early exit while section 4.4 explains how auto-tuning is implemented to dynamically decompose the computational domain. Finally, in section 4.5 I present the performance results for these auto-tuning techniques. [1]

4.1 Dynamic auto-tuning

Dynamic auto-tuning refers to how an application can apply auto-tuning techniques during runtime. In other words it dynamically tunes the computational domain based on the hardware it is executing on. As a result, the domain automatically adapts to the underlying hardware given specific parameters that the auto-tuning is based on.

[1]Unlike the benchmarks from the previous chapter, this chapters performance section will feature some benchmarks with non-zero bathymetry, i.e., where the bathymetry either contains synthetic terrain values or terrain values from a real world case. When testing that these simulations were mathematically correct, it was discovered that the correct Δt value was not chosen when using additional overlap with the GCE technique. Through further testing, I believe this is related to a bug when initializing the bathymetry ghost cells among the sub-domains. However, since the benchmarks in this chapter are not using the GCE technique, this bug does not affect the results in any way.

As mentioned, there are several parameters that can be auto-tuned based on my heterogeneous implementation in chapter 3. The first is the *GCE* technique. However, as discovered in section 3.6.3, this technique did not improve the performance since increasing the ghost cell overlap negatively affected the performance. It was therefore best to always use no additional overlap. As a result, this parameter has not been dynamically auto-tuned. There are still two other parameters though, more precisely, *Early exit* and *Domain decomposition*. As the results from section 3.6.4 showed, the optimal static domain decomposition between the Intel Core i7 Q 740 and Quadro Q1800M was to run 20% of the computational domain on the CPU. Based on these results, one can argue that a load balanced workload for this system in shallow water simulations is to let the CPU perform computations on 20% of the rows with wet cells while the GPU performs computations for 80% of the rows with wet cells. To achieve this, I perform dynamic auto-tuning of *Domain decomposition* to dynamically adjust the domain decomposition between the GPU and the CPU as the water propagates. In addition, the early exit optimization can be used such that the computations for all the dry cells are skipped. However, since the early exit optimization adds an extra overhead, one does not want to use this when most of the domain contains wet cells. Therefore, this can also be auto-tuned dynamically to select whether early exit should be enabled or disabled.

When it comes to challenges, efficiency is of great importance for dynamic auto-tuning between the CPU and the GPU. Even though the auto-tuning technique only will be executed at given times throughout a simulation, it is still important that it is as effective as possible. Furthermore, it also needs to work well for heterogeneous architectures. For example, a technique that is implemented only with the CPU in mind does not necessarily translate into working for the GPU as well. There are several considerations that have to be taken into account. First of all, data transfers. An auto-tuning technique that requires large data transfers between the CPU and the GPU can be inefficient. In addition, computations related to the auto-tuning also has to be considered, since both the CPU and the GPU has to perform computations related to the dynamic auto-tuning as efficiently as possible. This can represent a challenge considering their architectural differences as described in section 2.1.

4.2 Bounding Box technique for the CPU

In section 3.5, I implemented an early exit optimization for both the CPU and the GPU. However, using early exit on the CPU raised an issue regarding the performance: For each timestep, an additional iteration over the domain has to be performed to compute the dry map that is used in the flux calculations (see section 3.1). It also has to perform the dry map lookup for each cell in the domain. As a result, if the CPUs domain only contains a small amount of wet cells, these computations can use more time than the actual numerical computations for each cell. This represents a quite expensive overhead which should be avoided. In addition, the time integration (see section 3.1) will not gain any speedup from using early exit since only the flux calculations benefits from this. These overheads also apply for the GPU, but the GPU is sufficiently fast as a result of its parallel nature. The overheads are therefore small compared to the execution time of the numerical calculations. As a result, there is no large negative impact to the performance compared to the CPU.

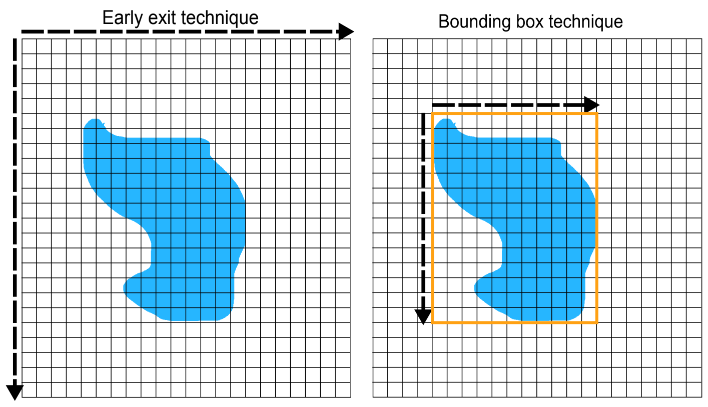


Figure 4.1: A domain with water represented by the blue cells. **Left image:** The flux function iterates through all cells in the domain, as indicated by the arrows. For each cell, a lookup in the dry map is performed to determine if it should perform early exit. **Right image:** The flux function only iterates through cells that are inside the bounding box, as indicated by the arrows. As a result, the additional overhead related to computing and reading the dry map is avoided.

To address this performance issue, I implemented a *Bounding Box* technique to replace the early exit optimization on the CPU. The wet cells are simply approximated by a two dimensional bounding box (see figure 4.1) defined by four edges. As a result, both the flux calculation and time integration can be performed by only iterating over the cells inside the bounding box, as every cell outside is guaranteed to be dry. The overhead for iterating over the domain to compute the dry map is also completely avoided. It still has to perform some additional computations as it has to compute the bounding box for each timestep, but this is less expensive as further stated below.

To compute the bounding box, I utilize the strategy in figure 4.2. The bounding box edges are extended along the water flow. This is achieved by taking advantage of the time integration to compute the bounding box. At the end of each timestep, time integration solves forward in time and makes sure the water propagates through the domain. As a result, it can guarantee which cells will be wet in the next timestep. Each of these cells can then store its x and y coordinate. Then, the minimum and maximum of these coordinates are computed as a reduction operation to compute the edges of the bounding box. The reduction is performed by utilizing the *min* and *max* reduction operators available through OpenMP [37] to correctly perform reductions in a multi-threaded environment. The computation of the bounding box are less expensive than the dry map computations for early exit since only cells inside the bounding box has to be considered.

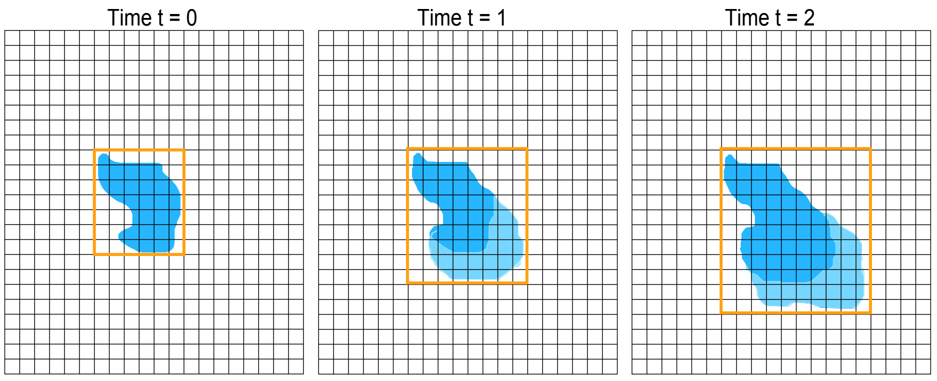


Figure 4.2: The strategy for computing the bounding box is shown at time $t = 0, 1$, and 2. The water only flows downwards to the right. The edges of the bounding box are extended in the direction of the flow.

4.3 Auto-tuning early exit

The results from section 3.6.2 showed that the early exit optimization provided a good performance increase for both the GPU and the CPU as long as the majority of the domain is dry. As specified in the previous section, a bounding box technique was implemented on the CPU since this was more effective. As a result, auto-tuning of early exit is only used by the GPU since the CPU utilizes the bounding box technique. When the majority of the domain contains water however, the flux computations have to be performed for most or every cell anyway. As a result, the early exit technique no longer provides a good performance increase since there are no cells to perform early exit on. Instead, it only provides a decrease in performance, mainly because of the additional overhead received when early exit is enabled. This makes the early exit optimization an excellent candidate for dynamic auto-tuning to ensure that it is enabled and disabled at the appropriate times depending on which configuration achieves the best execution time.

The auto-tuning implemented is a simple *probe* technique that executes a single step in the simulation and measures the execution time of this. The probe only executes kernels that perform computations related to early exit, more precisely the dry and flux kernels (see section 3.5 and 2.4.1). However, for second-order it has to execute the whole first substep, which includes the dry, flux, time integration and boundary conditions kernels (see also section 2.4.1 for these), since the dry and flux kernels in the second substep relies on these. For each sub-domain, two probes are launched every Nth timestep, one probe with early exit enabled and another probe with early exit disabled. The execution time is measured for both of them. When both probes have finished, the execution time is compared. If the probe executing with early exit is faster, then early exit is enabled, otherwise it is disabled. This ensures that a single timestep always executes at the fastest possible rate.

It is important to consider how often these probes should be launched, i.e., the value of N. First off, since the probes execute a single simulation step, they add a small overhead. However, this overhead is only noticeable if the probes are launched too frequently. Secondly, if the probes are launched to infrequently, the auto-tuning technique may fail to disable or enable early exit at a good rate, decreasing the overall performance.

4.4 Auto-tuning domain decomposition

To explain more closely how the dynamic auto-tuning technique will work, consider the following example as shown in figure 4.3: A real-world case with a dam break at the upper part of the domain. If this dam collapses, the water will propagate downwards. For example, approximately 20% of the rows with wet cells can be computed on the CPU and the other 80% on the GPU. This does not mean that the CPU is assigned a sub-domain that is 20% of the global domain, as become evident from figure 4.3. However, it performs computations on approximately 20% of the total water, but can still be assigned a larger computational sub-domain. To avoid wasting resources on computing dry parts of the domain, the auto-tuning of early exit presented in the previous section is utilized for the GPU, while the CPU utilizes the bounding box technique presented in section 4.2. As the simulation progress, water will continue to flow downwards, meaning the domain decomposition has to be dynamically changed to make sure the CPU computes on approximately 20% of the rows with wet cells at all times.

As a result, this requires a way to change the domain decomposition during runtime. A simple function to perform this is implemented and explained in greater detail in section 4.4.1. The auto-tuning itself basically consists of two auto-tuning algorithms: *Startup auto-tuning* and *Dynamic auto-tuning*. The first algorithm decides the computational workload between the CPU and the GPU. The second algorithm executes only at given times throughout the simulation to dynamically decompose the domain between the CPU and the GPU based on the computational workload that was found by the first algorithm. To perform the dynamic auto-tuning, I compare two different strategies. The first is a very simple naive approach that provides the baseline of the dynamic auto-tuning algorithm. The second strategy is a simple improvement upon the first, providing a better approach.

4.4.1 Dynamic domain decomposition

However, before discussing implementation details related to the auto-tuning algorithms, I will discuss how the domain can be decomposed during runtime. More precisely, this functions main purpose is to change

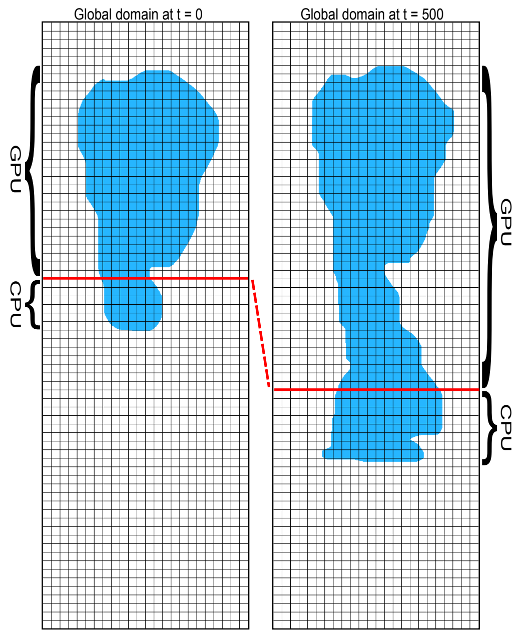


Figure 4.3: The dynamic auto-tuning technique for the domain. The red line shows the domain decomposition between the CPU and the GPU. The upper sub-domain is assigned to the GPU and the lower sub-domain is assigned to the CPU. **Left**: The domain decomposition at time $t = 0$. The CPU performs computations for 20% of the rows with wet cells, while the GPU performs computations for the other 80%. The CPU is assigned a larger sub-domain than the GPU, but still performs less work, since only the cells that contain water are computed. **Right**: The domain decomposition at time $t = 500$. The water has propagated further, which means that the domain decomposition is recomputed. The CPU still computes on 20% of the rows with wet cells while the GPU computes the rest.

the size of all the sub-domains. This is then utilized by the dynamic auto-tuning algorithm to change the computational workload on the fly for the CPU and the GPU. The following example gives an idea of how this works: Consider a simulation with two sub-domains, A and B that has been created by decomposing an initial global domain of dimension 1000^2. A has a size of 1000×600 while B has a size of 1000×400. In addition, A is assigned to the GPU while B is assigned to the CPU. During the simulation, the auto-tuning algorithm may decide that a dynamic change in domain decomposition is necessary. This technique is then utilized to change the size of each sub-domain. For example, A can be changed to 1000×800 while B can be changed to 1000×200.

This technique mainly performs three steps: The first step involves copying all the internal domain cells in each of the original sub-domains into a global buffer. The data that is copied consists of the water elevation Q_1, and the water momentum along the x and y directions, Q_2 and Q_3. Then, step two computes the new size of each sub-domain and then reconstructs the sub-domains with the new size. This reconstruction first deallocates the original sub-domains buffers and allocates new buffers based on the new size of the sub-domain. Then, all cell data (Q_1, Q_2, and Q_3) is copied from the global buffer into the newly reconstructed sub-domains buffers. Finally, the last step is performed to handle the boundaries. In other words, the ghost cells are initialized to the boundaries of the opposite sub-domains by performing ghost cell exchange, explained in section 3.3.3, between the sub-domains. Figure 4.4 gives an illustration of how these steps are performed.

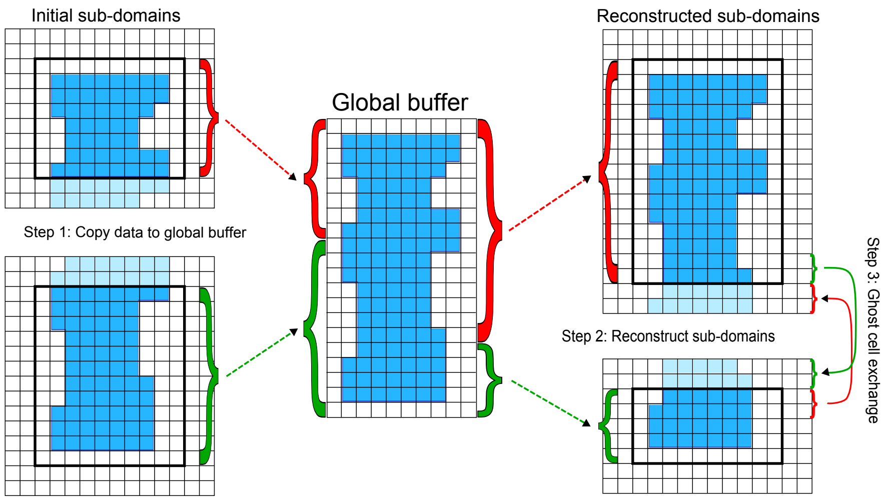

Initial sub-domains
Step 1: Copy data to global buffer
Global buffer
Reconstructed sub-domains
Step 2: Reconstruct sub-domains
Step 3: Ghost cell exchange

Figure 4.4: Changing the size of sub-domains during runtime. First, **step 1** copies the data in internal cells from the sub-domains into a global buffer. More precisely, the data copied is the water elevation Q_1, and the water momentum along the x and y directions, Q_2 and Q_3. Then, **step 2** reconstructs the sub-domains with new sizes and copies the cell data (Q_1, Q_2, and Q_3) from the global buffer to the new sub-domains. Finally, **step 3** performs ghost cell exchange to initialize the ghost cells.

4.4.2 Startup auto-tuning

At the startup of a simulation, this auto-tuning algorithm calculates the computational workload between the CPU and the GPU. The computed workload decides the amount of rows containing wet cells that they should should compute. For example, if the GPU is twice as fast as the CPU, it should compute on 66% of the rows with wet cells while the CPU computes on 33%. This algorithm is able to decide this at the start of a simulation by executing a single timestep for both the CPU and the GPU, and then recording the execution time measured in seconds. The computational workload W in percentage is then computed for both the CPU and the GPU based on their execution time by using the following formula:

$$
\begin{aligned}
W &= \frac{ny}{1}\left(1 - \frac{processor_t}{total_t}\right)\frac{100}{ny} \\
&= \frac{ny}{1}(1 - scaled_t)\frac{100}{ny} \\
&= \frac{ny}{1} inverted_t \frac{100}{ny},
\end{aligned}
\tag{4.1}
$$

where ny is the total number of rows in the global domain, $processor_t$ is the achieved execution time in seconds of either the GPU or the CPU, and $total_t$ is the total execution time in seconds of both the GPU and the CPU.

4.4.3 Dynamic auto-tuning

Naive method The naive method provides the baseline for how the dynamic auto-tuning algorithm can be implemented. It works by finding the middle point of the wet cells (see figure 4.5). This is simply computed by adding together the coordinates of all wet cells and dividing this by the total amount of wet cells. This point therefore represents the middle point of the water in the domain. As a result, both sub-domains can be given an equal amount of rows with wet cells. The rows with wet cells above the middle point can be distributed to the GPU. Similarly, the rows with wet cells below the middle point can be distributed to the CPU. Since both receive approximately the same amount of wet cells, it would only work well in systems that contains a CPU and GPU that are equally powerful. As this is rarely the case, this method is not ideal for most CPU/GPU configurations.

Optimized method The optimal method is similar to the naive method, but instead of computing the middle point, it computes a two dimensional bounding box around the wet cells similar to how the bounding box technique for the CPU was implemented in section 4.2. There are multiple ways to compute this bounding box.

First off, the CPU can perform it by iterating through its own sub-domain and the GPUs sub-domain. However, this requires the GPU to transfer information for its sub-domain to the CPU. More precisely it would have to transfer the water elevation Q_1 and the bathymetry B which is needed to be able to determine wet and dry cells in the domain. Transferring such amount of information can be expensive, especially when using larger domain resolutions. This could be made slightly more effective by letting the GPU determine wet and dry cells and store this information in a buffer. However, it would still have to transfer this buffer to the CPU to let it compute the bounding box. As a result, this only minimizes the transfers and is not an optimal solution if one wants to avoid large data transfers.

Therefore, another technique was implemented. The CPU and GPU can compute a local bounding box for its own sub-domains. This is easily performed on the CPU by iterating over the domain as explained earlier in section 4.2 to find the minimum and maximum coordinates in the x and y direction. For the GPU this is a more complex operation as a GPU thread would typically work on a single cell. Still, it can easily be performed on the GPU by utilizing efficient GPU reduction for each block through shared memory. Each thread is responsible for a single cell by storing its coordinates in shared memory if the cell is wet. Per block reduction is then performed to find the minimum and maximum coordinates in the x and y direction for all blocks. Since this results in a bounding box for each block, additional reductions has to be performed to further reduce this into only four coordinates representing a single bounding box.

When both the GPU and the CPU have computed their local bounding boxes, the CPU computes the global bounding box based on these as seen in figure 4.5. To perform this the bounding box computed by the GPU is first transferred to the CPUs memory. This means it does not avoid data transfers, but transferring a bounding box consisting of only four coordinates is more efficient and avoids large data transfers unlike the previous solution.

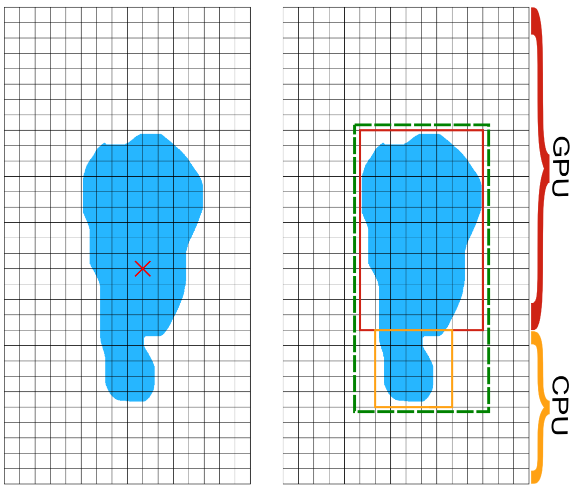
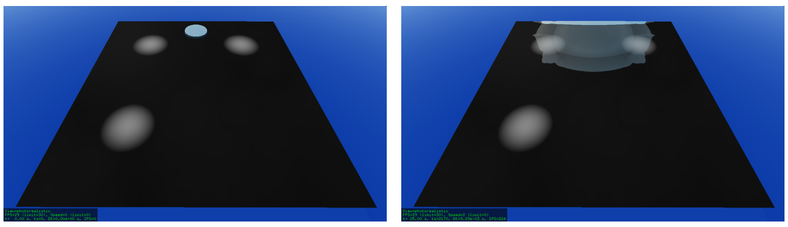


Figure 4.5: Two different dynamic auto-tuning methods. The water is marked in blue. **Left image**: Computes the middle point (coordinate) marked with the red cross among all wet cells. Each sub-domain can then be distributed an equal amount of rows with wet cells. **Right image**: The GPU and CPU compute their local bounding boxes marked in red and yellow. Then, the global bounding box marked in green is computed around these two. This makes it easier to compute a different distribution of rows with wet cells to the sub-domains. For example, the CPU can be assigned 20% of the lower part of the global bounding box.

The optimal method works better than the naive approach because it makes it possible to divide the wet cells unevenly between the sub-domains, instead of equally as the naive method. This is simply implemented by computing and assigning a given percentage of the bounding box to the GPU and CPU. More precisely, they are assigned the workload percentage of W that was computed in section 4.4.2 by using equation (4.1). At this point, a resize of the sub-domains are performed by utilizing the function described in section 4.4.1. The dynamic auto-tuning technique also has to be launched at specific timesteps similar to the early exit auto-tuning, i.e., each Nth timestep. If it is launched to frequently the overhead related to the auto-tuning computations and changing the sub-domains may become too large. If it is instead launched to infrequently it will fail to auto-tune the domain at a good rate which will decrease the overall performance.

4.5 Performance results

To start off, I evaluate the performance of the bounding box technique implemented for the CPU by comparing it with the early exit optimization on the Intel Core i7-2600K. It shows a solid increase in performance by around three times over early exit for large resolutions. Then, the performance of the auto-tuning techniques is evaluated. I first show the impact of auto-tuning early exit in section 4.5.2 on the Geforce GTX 480 GPU. The main results show that the auto-tuning technique provides the best performance on all domain resolutions. Finally, this auto-tuning technique is combined with auto-tuning of domain decomposition. The effect is shown by executing on two different cases, an idealised circular dam break (see figure 4.6), and a real world case, the Malpasset dam break (see figure 4.7) which collapsed in 1959 causing heavy casualties [9]. The benchmarks are further executed on the two systems used earlier, i.e., the desktop with an Intel Core i7-2600K CPU and a Geforce GTX 480 GPU, and a laptop with an Intel Core i7 Q 740 CPU and a Quadro Q1800M GPU.

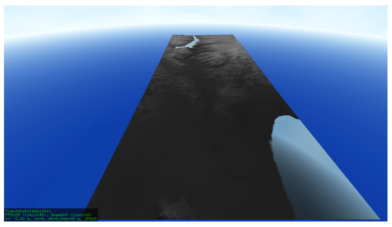


(a) (b)

Figure 4.6: Idealised circular dam with three bumps. **Left**: At time $t = 0$, the circular dam instantaneously collapses, causing the water to flow in all directions. **Right**: The water flow at time $t = 25$.

4.5.1 Bounding box technique

The bounding box technique was benchmarked on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 80m$ in a square computational domain of $1200m \times 1200m$ with center at $x_c = 600m, y_c = 600m$. It was further run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions


(a)

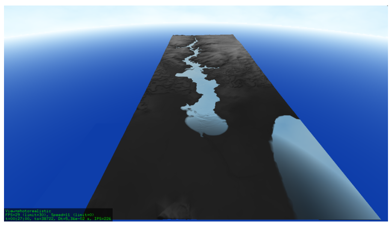

(b)

Figure 4.7: **Left**: Malpasset dam break at time $t = 0$. The water can be seen at the top of the valley before the breach while the sea is at the bottom right. **Right**: At time $t = 1620$ after the breach, the water follows the valley before reaching the sea.

for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum in x and y directions, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x, y, 0) = \begin{cases} Q_1 = 10m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 5$. A small time t is selected as the overhead when using early exit is more visible when a large area of the domain is dry. Therefore, it makes more sense comparing the early exit and bounding box techniques when the water has not yet propagated through large parts of the domain.

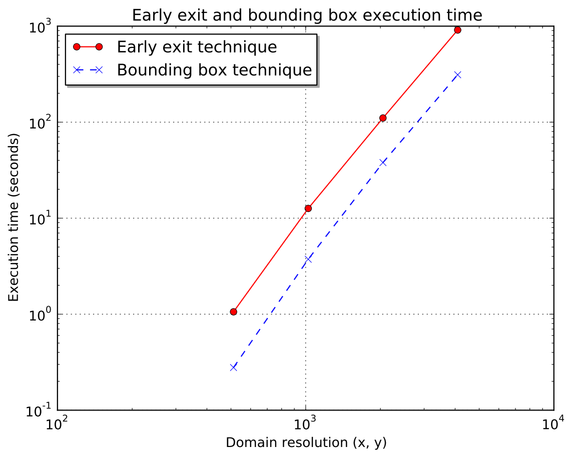


Figure 4.8: The execution time of the bounding box technique compared to early exit on an idealised circular dam break with four domain resolutions from 512^2 to 4096^2. The bounding box technique performs much better than early exit independent of the resolution.

Figure 4.8 shows the results by comparing on four domain resolutions from 512^2 to 4096^2. The simulation times are also provided in table 4.1.

The bounding box technique shows a solid performance increase over early exit. This is mainly because this technique avoids the overhead related to early exit, but also because it iterates over a smaller part of the domain as it only performs computations for cells that are inside the bounding box. However, the speedup also seems to be smaller when simulating on large resolutions. This is likely because the overhead related to early exit decreases as the resolution gets larger, which means early exit performs better on large resolutions.

Domain	Early exit	Bounding box	Speed up
512^2	1.4E+0	3.9E-1	3.7
1024^2	1.6E+1	5.0E+0	3.1
2048^2	1.3E+2	5.1E+1	2.6
4096^2	1.0E+3	3.5E+2	2.9

Table 4.1: The CPUs execution times in seconds for the bounding box technique and early exit on four domain resolutions from 512^2 to 4096^2. The speed up obtained is around three times on large resolutions, and can be even higher for smaller resolutions.

4.5.2 Auto-tuning early exit

The auto-tuning of early exit have been benchmarked on the GPU to measure the effect of applying this auto-tuning technique compared to executing a simulation without auto-tuning. The benchmark was run on a case with flat bathymetry and an idealised circular dam break surrounding a water column with radius $R = 300m$ in a square computational domain of $3000m \times 3000m$ with center at $x_c = 1500m$, $y_c = 1500m$. It was further run with wall boundary conditions and second-order accurate Runge-Kutta. The initial conditions for the bathymetry B was set to $B(x, y, 0) = 0$, while the water momentum, Q_2 and Q_3 were set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x, y, 0) = \begin{cases} Q_1 = 1m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0.1m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 1200$. I have executed with three different simulations: *Standard*, *Early exit*, and *Probe*. The first one have disabled early exit and therefore computes all cells. The second have enabled early exit. Finally, the last simulation utilizes the auto-tuning technique, which is performed every 500 timestep. All the simulations have been run on several domain resolutions ranging from 512^2 to 4096^2.

Plot 4.9 shows the improvement of the *Early exit* and *Probe* simulations over the *Standard* simulation. The x-axis shows the domain resolution used while the y-axis shows the percentage of achieved execution time

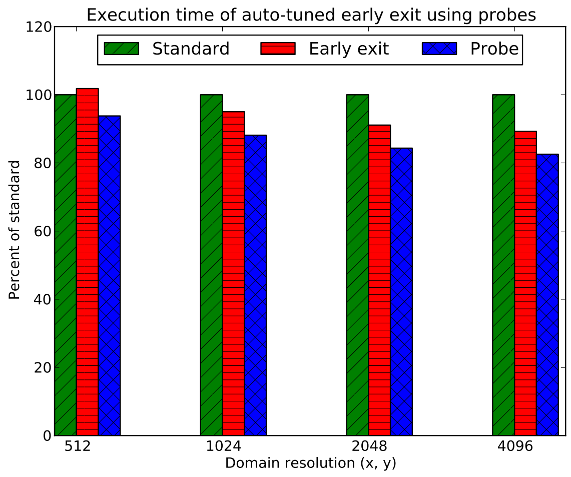


Figure 4.9: Shows the percentage of achieved execution time for *Early exit* and *Probe* relative to *Standard* on an idealised circular dam break (lower represents better execution time). Notice that *Probe* performs better than *Standard* for each increase in domain resolution as expected.

relative to the *Standard* simulation. A lower percentage is therefore better. The *Early exit* simulation performs better than the *Standard* simulation on all domain resolutions except for the lowest resolution. The performance improvement over the *Standard* simulation is also larger as the resolution increases. This is mainly because the overhead related to early exit is less noticeable when there are more cells in the domain that can take advantage of using early exit. As expected, the *Probe* simulation performs best on all domain resolutions. Again, it is apparent that the performance improvement over the *Standard* simulation grows larger as the resolution increases. This is mainly because the rate of which early exit is enabled and disabled works better on large resolutions.

Resolution	Standard	Early exit	Probe
512^2	1.5E+1	1.8E+1	1.4E+1
1024^2	1.2E+2	1.4E+2	1.1E+2
2048^2	1.0E+3	1.1E+3	8.7E+2
4096^2	8.9E+3	1.0E+4	7.5E+3

Table 4.2: The execution times in seconds between three different simulations, one with early exit enabled (*Early exit*), one with early exit disabled (*Standard*), and one with utilizing the auto-tuning technique (*Probe*).

4.5.3 Auto-tuning domain decomposition and early exit

Finally, several benchmarks have been performed too measure the effect of utilizing the full auto-tuning, i.e., on both *domain decomposition* and *early exit*. Two different cases have been used. First, a real-world case known as the Malpasset dam break. This dam break consist of 439×1099 cells, spaced equally by 15 meters, i.e., $\Delta x = \Delta y = 15m$. Furthermore, I simulate the first 4000 seconds after the breach using Euler time integration. The second case is an idealised circular dam break with three bumps in the terrain. This dam break surrounds a water column with radius $R = 66m$ in a square computational domain of $1000m \times 1000m$ located at the upper part of the domain with center at $x_c = 500m$, $y_c = 125m$. The case was further executed with wall boundary conditions using Euler time integration and with a resolution of $r_x = 1000$, $r_y = 1000$. The bathymetry B consist of three bumps, each with a radius of $R_B = 100m$ and center at $x_{c1} = 250m$, $y_{c1} = 250m$, $x_{c2} = 750m$, $y_{c2} = 250m$, and $x_{c3} = 250m$, $y_{c3} = 750m$ respectively, and is set to:

$$B(x,y,0) = \begin{cases} B = 5 * ({R_B}^2 - ((x - x_{c1})^2 - (y - y_{c1})^2))m \\ \quad \text{if } (x - x_{c1})^2 + (y - y_{c1})^2 < {R_B}^2 \\ B = 5 * ({R_B}^2 - ((x - x_{c2})^2 - (y - y_{c2})^2))m \\ \quad \text{if } (x - x_{c2})^2 + (y - y_{c2})^2 < {R_B}^2 \\ B = 5 * ({R_B}^2 - ((x - x_{c3})^2 - (y - y_{c3})^2))m \\ \quad \text{if } (x - x_{c3})^2 + (y - y_{c3})^2 < {R_B}^2 \\ B = 0m \\ \quad \text{otherwise} \end{cases} .$$

Furthermore, the water momentum, Q_2 and Q_3 were set to $Q_2(x,y,0) = Q_3(x,y,0) = 0$ throughout the domain, and the water elevation Q_1:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 10m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

At time $t = 0$, the dam is instantaneously removed, resulting in an outgoing circular wave that flows through the domain until time $t = 180$.

For each of these cases, I execute six simulations (described below) on a desktop and laptop. The desktop consist of an Intel Core i7-2600K CPU and a Geforce GTX 480 GPU while the laptop consist of an Intel Core i7 Q 740 CPU and a Quadro Q1800M GPU. I refer to these as the **desktop** and **laptop** from now on. The performance is shown in megacells computed per second throughout the simulation.

I first execute and compare three simulations where the early exit optimization is disabled, namely *Single GPU*, *Static*, and *Dynamic* [2].

[2]The *Dynamic* simulation utilizes the dynamic auto-tuning technique. This technique computes the bounding boxes for the GPU and the CPU and from these, the global bounding box, which auto-tunes the domain decomposition. For this to function correctly in the Malpasset case, it is important that the global bounding box only surrounds the water that flows down the valley, and does not include the sea at the bottom right (see figure 4.7). Since my implementation does not handle multiple global bounding boxes, I simply removed the sea from the Malpasset case for the benchmarking in this section. As a result, the simulation is mathematically incorrect, but this does not affect the results received in this section since the resulting water wave from the breach still propagates correctly.

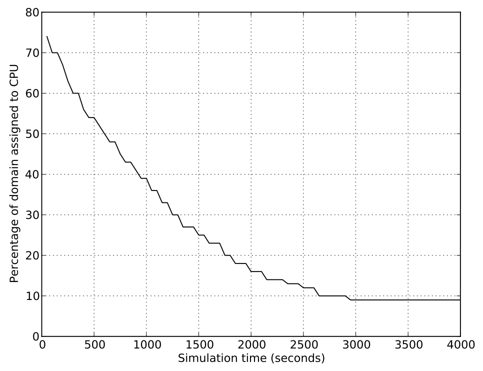
The first simulation computes all cells using a single GPU. The second simulation uses a static decomposition of the domain. For the desktop, the CPU is assigned 10% of the domain, and for the laptop, the CPU is assigned 20% of the domain. These were chosen because the results from section 3.6.4 showed that this was the optimal decomposition between the CPU and the GPU on the desktop and laptop respectively. The third simulation utilizes the dynamic auto-tuning of *domain decomposition* to balance the workload between the CPU and the GPU throughout the simulation. The workload is split in the same percentage as for the static decomposition. The CPU on the desktop is therefore assigned 10% of the rows with wet cells, while the CPU on the laptop is assigned 20% of the rows with wet cells.

I also execute the same simulations, but with the early exit optimization enabled, more precisely *Single GPU (EE)*, *Static (EE)*, and *Dynamic (EE)*. The first simulation utilizes a single GPU as last time, but with early exit enabled. The second simulation uses the same static decomposition of the domain, also with early exit enabled for both the GPU and the CPU. The last simulation utilizes the full auto-tuning, i.e., auto-tuning of *domain decomposition* and *early exit*. However, the CPU now uses the bounding box technique instead of early exit.

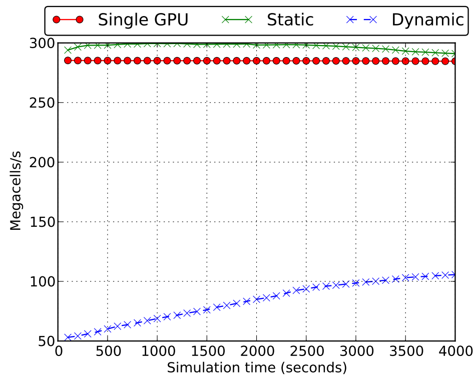
The dynamic auto-tuning algorithms are executed every 2000th timestep for the Malpasset case and every 500th timestep for the synthetic case. These values were chosen by experimenting with both cases to find an ideal value. Since the ideal value can be different from case to case, mainly depending on the Δt at any given time, it would be more ideal to perform an analysis of this value to decide what it should be set to for specific cases. However, this is outside the scope of this thesis.

Desktop: For the desktop, the performance of the Malpasset and idealised circular dam break can be seen in figure 4.10 and 4.11. For the three simulations with early exit disabled, the *Static* simulation at its peak performs approximately 5% better than the *Single GPU* for the Malpasset case and 9% better for the idealised circular dam break case. For both cases however, the *Dynamic* simulation gives very poor performance. As it utilizes auto-tuning of *domain decomposition*, the CPU will receive a large computational domain in the beginning of the simulation where the water lies in the upper region of the domain, on both Malpasset and the idealised circular dam break. This is the main reason for its poor performance, but also because both the GPU and the CPU computes on all cells, without utilizing its ideal optimization technique like early exit or bounding box. However, as the water flows down the valley in the Malpasset case, the CPU will receive a smaller computational domain, resulting in increased performance as the simulation progress. This also applies for the idealised circular dam break where the water flows in all directions, quickly filling the domain with water. However, I conclude that a static domain decomposition where 10% of the domain is executed on the CPU gives the best performance when early exit is disabled.

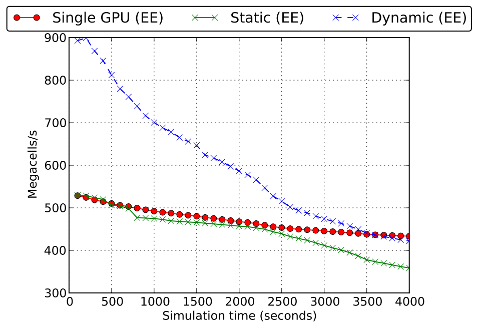
For the three simulations with early exit enabled, the *Single GPU (EE)* and *Static (EE)* is mostly very equivalent in performance. As the simulation progress however, it declines more in performance than the *Single GPU (EE)* simulation. The main reason for this is because of the overhead for early exit on the CPU. As more water flows into the CPUs computational domain, this overhead is slightly increased. The best performance is obtained by the *Dynamic (EE)* simulation as it utilizes the full auto-tuning, i.e., on both *domain decomposition* and *early exit*. For this simulation, the CPU uses its bounding box technique. The peak performance is almost 70% better than the *Single GPU (EE)* and *Static (EE)* for the Malpasset, and around 19% better for the idealised circular dam break. This clearly shows the strength of the two auto-tuning techniques when combined, and I conclude that *Dynamic (EE)* performs best when early exit is enabled.



(a) Malpasset: Percentage of domain assigned to the CPU.

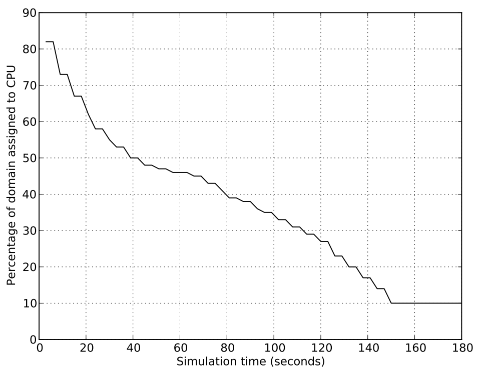


(b) Malpasset: Desktop performance with early exit disabled.

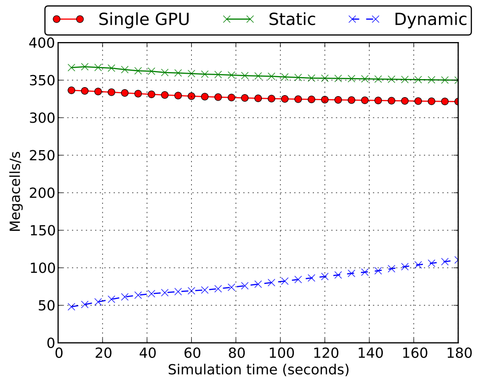


(c) Malpasset: Desktop performance with early exit enabled.

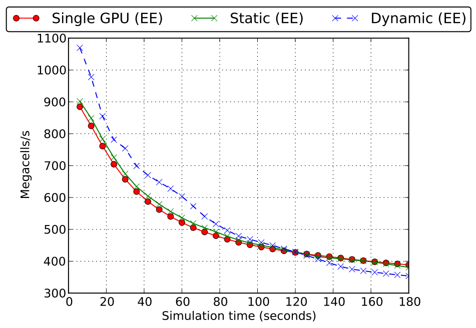
Figure 4.10: The performance shown in megacells per second for the Malpasset dam break on the desktop. Three simulations have been executed: *Single GPU*, *Static*, and *Dynamic*. **Left:** All simulations have been executed with early exit disabled. **Right:** The same simulations with early exit enabled and bounding box technique for the CPU used by the *Dynamic* simulation. **Top:** The percentage of assigned domain to the CPU throughout the simulation.



(a) Idealised circular dam: Percentage of domain assigned to the CPU.



(b) Idealised circular dam: Desktop performance with early exit disabled.



(c) Idealised circular dam: Desktop performance with early exit enabled.

Figure 4.11: The performance shown in megacells per second for the idealised circular dam break on the desktop. Three simulations have been executed: *Single GPU*, *Static*, and *Dynamic*. **Left:** All simulations have been executed with early exit disabled. **Right:** The same simulations with early exit enabled and bounding box technique for the CPU used by the *Dynamic* simulation. **Top:** The percentage of assigned domain to the CPU throughout the simulation.

I also show the overhead for auto-tuning in table 4.3. The execution time for the whole simulation is compared with the total auto-tuning time. To easily show the bottleneck of the dynamic auto-tuning if any, I split the total auto-tuning execution time in three parts: *Auto-tuning (BB)* which measures the execution time of the bounding box computation, *Auto-tuning (DD)* which measures the execution time when performing a dynamic change of the domain decomposition, and finally *Auto-tuning (Probe)* to measure the execution time of the probe technique that auto-tunes early exit on the GPU.

The execution time for the total auto-tuning is very small compared to the whole simulation, i.e., 1.1% of the Malpasset execution time and 3.0% of the idealised circular dam break execution time. Therefore, the overhead related to auto-tuning is very small, and should not give any noticeable

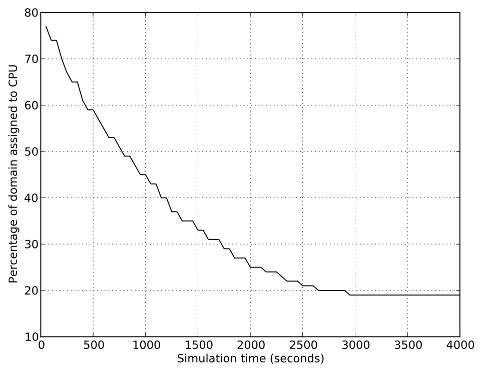
Execution times	Malpasset	Idealised circular dam
Simulation	7.5E+1	4.6E+1
Auto-tuning (total)	8.3E-1	1.4E+0
Auto-tuning (BB)	3.4E-2	6.8E-2
Auto-tuning (DD)	7.5E-1	1.2E+0
Auto-tuning (Probe)	4.8E-2	1.1E-1

Table 4.3: The auto-tuning overhead on the desktop for Malpasset and the idealised circular dam break. I show the execution time in seconds for the simulation and the auto-tuning algorithms. The dynamic domain decomposition is clearly the bottleneck. Also, notice that the execution time for the total auto-tuning only represents 1.1% of the Malpasset execution time and 3.0% of idealised circular dam break execution time.

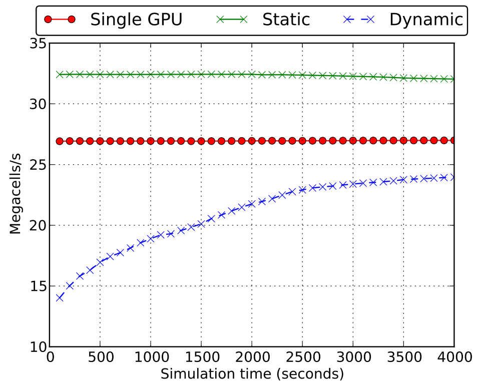
decrease in performance. The bottleneck for the auto-tuning however, is clearly the dynamic change for domain decomposition. Therefore, this could be optimized further, but I dont expect any noticeable improvement for the simulations execution time by performing such optimizations since the auto-tuning overhead is as shown very small.

Laptop: For the laptop, the performance of Malpasset and the idealised circular dam break can be seen in figure 4.12 and 4.13. I expect similar results for the laptop, but with even larger performance improvements since the performance gap between the CPU and the GPU is not as large as for the desktop. For the simulations with early exit disabled, the *Static* simulation performs better than the *Single GPU* simulation, with a peak performance of 18% better for the Malpasset case, and 23% better for the idealised circular dam break. This improvement is larger than the improvement gained for the desktop, as expected from the results in section 3.6.4. When it comes to the *Dynamic* simulation, it also gives poor performance for both cases similar to what it did on the desktop. However, at its peak, it is closer to the performance of the *Single GPU* than the desktop was, due to the CPU in the laptop being closer to its GPU performance wise compared to the desktop. To conclude however, the *Static* simulation also performs best on the laptop when early exit is disabled.

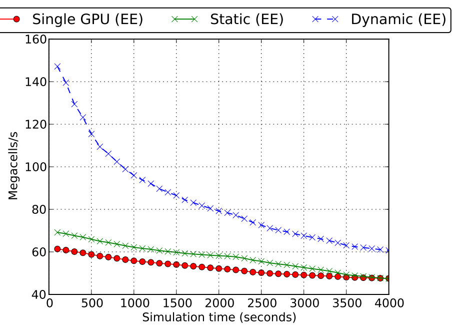
For the three simulations with early exit enabled, the *Single GPU (EE)* and *Static (EE)* simulations gives close to equivalent performance throughout the simulation, but the *Static (EE)* still performs slightly better than the *Single GPU (EE)* on both cases. However, unlike the desktop, it does not decline more in performance than the *Single GPU (EE)* towards the end. Again, I expect this is because the performance gap between the laptops GPU and CPU is smaller than for the desktop, making the CPU overhead related to early exit less dramatic in comparison. Similarly as before, the *Dynamic (EE)* gives the best performance, reaching a peak performance of 141% and 109% over the *Single GPU (EE)* on Malpasset and the idealised circular dam break respectively. This is even better than the desktop where the same simulations had a performance increase of 70% and 19%. As a result, it is clear that the auto-tuning techniques give even better performance for systems where the GPU and the CPU is more closer to each other in raw performance. This often applies for laptops as they are more often configured with a weaker GPU using less power than desktops.



(a) Malpasset: Percentage of domain assigned to the CPU.



(b) Malpasset: Laptop performance with early exit disabled.

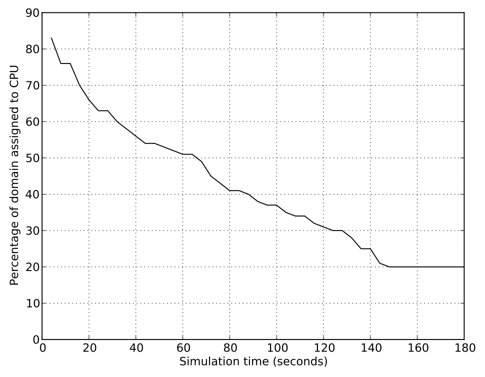


(c) Malpasset: Laptop performance with early exit enabled.

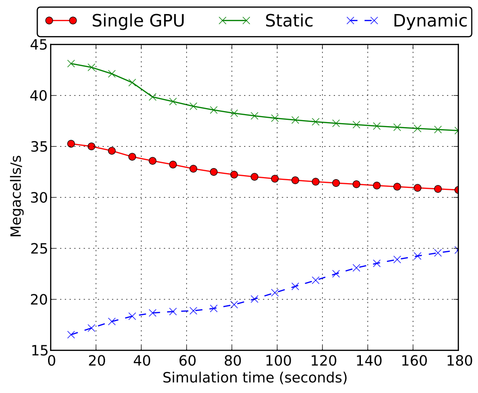
Figure 4.12: The performance shown in megacells per second for the Malpasset dam break on the laptop. Three simulations have been executed: *Single GPU*, *Static*, and *Dynamic*. **Left:** All simulations have been executed with early exit disabled. **Right:** The same simulations with early exit enabled and bounding box technique for the CPU used by the *Dynamic* simulation. **Top:** The percentage of assigned domain to the CPU throughout the simulation.

I also show the auto-tuning overhead for the laptop in table 4.4. Again, the execution time for the whole simulation is compared with the total auto-tuning execution time. I also split the dynamic auto-tuning in the same parts as above.

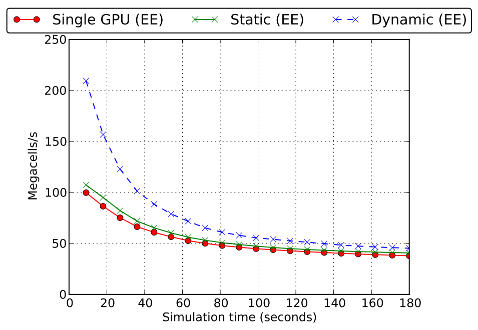
For the laptop, the execution time for the total auto-tuning is also very small compared to the whole simulation, i.e., 0.1% of the Malpasset execution time and 1.5% of the idealised circular dam break execution time. Therefore, this should not give any noticeable decrease in performance.



(a) Idealised circular dam: Percentage of domain assigned to the CPU.



(b) Idealised circular dam: Laptop performance with early exit disabled.



(c) Idealised circular dam: Laptop performance with early exit enabled.

Figure 4.13: The performance shown in megacells per second for the idealised circular dam break on the laptop. Three simulations have been executed: *Single GPU*, *Static*, and *Dynamic*. **Left:** All simulations have been executed with early exit disabled. **Right:** The same simulations with early exit enabled and bounding box technique for the CPU used by the *Dynamic* simulation. **Top:** The percentage of assigned domain to the CPU throughout the simulation.

Notice that the probe technique is the bottleneck for the Malpasset case while the dynamic change for domain decomposition is the bottleneck for the Idealised circular dam break, although the probe technique also uses a large amount of time for this case. Again, I reason that optimizing these will have a minimal effect on the simulations execution time and not make a noticeable difference in performance.

Malpasset and idealised circular dam break comparison: There is one interesting difference between these two cases as seen in figure 4.10c and 4.11c for the desktop, and figure 4.12c and 4.13c for the laptop. The performance decreases in an almost exponential fashion for the idealised circular dam break whereas its decreases more linearly for the Malpasset case. The exponential decrease is because the water flows in all directions,

Execution times	Malpasset	Idealised circular dam
Simulation	5.6E+2	3.5E+2
Auto-tuning (total)	7.3E-1	5.3E+0
Auto-tuning (BB)	3.7E-2	1.0E-1
Auto-tuning (DD)	7.8E-2	4.0E+0
Auto-tuning (Probe)	6.1E-1	1.2E+0

Table 4.4: The auto-tuning overhead on the laptop for Malpasset and the idealised circular dam break. I show the execution time in seconds for the simulation and the auto-tuning algorithms. Again, notice that the execution time for the total auto-tuning is only 0.1% of the Malpasset execution time and 1.5% of the idealised circular dam break execution time.

quickly filling the whole domain with water when the circular dam in the idealised circular dam break collapses. For the Malpasset case, the water follows the valley, meaning large parts of the domain will always be dry. Towards the end in the idealised circular dam break case on the desktop, the *Dynamic (EE)* simulation performs noticeable worse than the other simulations, especially compared to the *Single GPU (EE)* simulation. This is mainly because there are relatively few dry cells in the domain at that point, meaning the auto-tuning and computation of the bounding box on the CPU only adds unnecessary overhead. For the Malpasset case, this does not happen as there are always many dry cells in the domain. As a result, this should not be a major problem in most real-world flood simulations. However, this is not noticeable for the laptop, mainly because the performance gap between the GPU and the CPU is not as large.

Chapter 5

Conclusion

This thesis has proposed a shallow-water simulator for heterogeneous architectures by implementing a multi-core CPU implementation based on a single GPU simulator [9] for the shallow-water equations. Load balancing the workload between the computational domains have also been performed by implementing two different auto-tuning methods. By utilizing these methods and running the CPU in parallel with the GPU, I have shown performance gains of up to 70% and 141% from the single GPU simulator on a real-world case.

The computational domain was decomposed between the CPU and the GPU by implementing a row domain decomposition technique. This minimized the amount of communication as well as providing effective communication. I applied two different auto-tuning techniques to load balance the workload between the CPU and the GPU. The first technique auto-tuned the early exit optimization by dynamically deciding if early exit should be enabled or disabled. The second auto-tuning technique provided an algorithm that dynamically decomposes the domain based on the water flow. Both of these were combined to provide good load balancing.

In this thesis, the heterogeneous implementation and auto-tuning techniques have been applied to the shallow-water equations to simulate natural phenomena such as flood. However, my implementation is general in purpose and can be applied to any systems of conservation laws, for example the Euler equations [12] and the MHD equation [28]. The results from this thesis should therefore apply well for other conservation laws as well. For example, consider the Euler equations, which describe the dynamics of an ideal gas. I can use the same row domain decomposition technique to statically decompose the domain between the CPU and the GPU. Furthermore, both auto-tuning methods can also be applied similarly as for the shallow-water equations. For example, an early exit optimization and bounding box technique can be implemented to skip computations for cells that do not contain any kind of fluid. The dynamic auto-tuning for early exit can then be applied. In addition, auto-tuning of domain decomposition can also be applied to load balance the workload between the CPU and the GPU based on the fluid instead of the water flow.

Bibliography

[1] M. A. Acuña and T. Aoki. Real-Time Tsunami Simulation on Multi-node GPU Cluster. In *ACM/IEEE Conference on Supercomputing*. 2009.

[2] Boost C++ libraries. http://www.boost.org/, (visited on 2014-08-01).

[3] T. Brandvik and G. Pullan. Acceleration of a 3D Euler Solver using Commodity Graphics Hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*. 2008.

[4] A. R. Brodtkorb. Hyperbolic Conservation Laws on GPUs. http://babrodtk.at.ifi.uio.no/files/publications/brodtkorb_granada_2014_conslaws.pdf, (visited on 2014-08-01).

[5] A. R. Brodtkorb. Reproducible Science and Modern Scientific Software Development. http://www.sintef.no/project/eVITAmeeting/2013/Advanced_topics_in_reproducible_science.pdf, (visited on 2014-08-01).

[6] A. R. Brodtkorb. A MATLAB Interface to the GPU. Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2007.

[7] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. In *Scientific Programming*, 2010.

[8] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and Visualization of the Saint-Venant System using GPUs. In *Computing and Visualization in Science*, 2010.

[9] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation. In *Computers & Fluids*, 2012.

[10] M. de la Asunción, J. M. Mantas, and M. J. Castro. Simulation of one-layer shallow water systems on multicore and CUDA architectures. In *The Journal of Supercomputing*, 2010.

[11] C. Ding and Y. He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. 2001.

[12] Wikipedia. Euler equations (fluid dynamics). http://en.wikipedia.org/wiki/Euler_equations_(fluid_dynamics), (visited on 2014-08-01).

[13] R. Gerber. Getting Started with OpenMP*. https://software.intel.com/en-us/articles/getting-started-with-openmp, (visited on 2014-08-01).

[14] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 2011.

[15] T. R. Hagen, M. O. Henriksen, J. M. Hjelmervik, and K. Lie. How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine. In *Geometric Modelling, Numerical Simulation, and Optimization*. Springer Berlin Heidelberg, 2007.

[16] T. R. Hagen, K. Lie, and J. R. Natvig. Solving the Euler Equations on Graphics Processing Units. In *Computational Science–ICCS 2006*. Springer Berlin Heidelberg, 2006.

[17] Intel. Intel® Xeon® Processor E7 v2 Family. http://ark.intel.com/products/family/78584/Intel-Xeon-Processor-E7-v2-Family, (visited on 2014-08-01).

[18] Khronos Group. OpenGL – The Industry's Foundation for High Performance Graphics. http://www.opengl.org/, (visited on 2014-08-01).

[19] F. B. Kjolstad and M. Snir. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. 2010.

[20] B. Kuhn, P. Petersen, and E. O'Toole. OpenMP versus Threading in C/C++. In *Concurrency: Pract. Exper*, 2000.

[21] A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete Central-Upwind Schemes for Hyperbolic Conservation Laws and Hamilton–Jacobi Equations. In *SIAM Journal on Scientific Computing*, 2001.

[22] A. Kurganov and G. Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. In *Communications in Mathematical Sciences*, 2007.

[23] Wikipedia. Lax-Friedrichs method. http://en.wikipedia.org/wiki/Lax-Friedrichs_method, (visited on 2014-08-01).

[24] Wikipedia. Lax-Wendroff method. http://en.wikipedia.org/wiki/Lax-Wendroff_method, (visited on 2014-08-01).

[25] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation

of Throughput Computing on CPU and GPU. In *SIGARCH Computer Architecture News*, 2010.

[26] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[27] T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. In *ACM Transactions on Architecture and Code Optimization*, 2013.

[28] Wikipedia. Magnetohydrodynamics. http://en.wikipedia.org/wiki/Magnetohydrodynamics, (visited on 2014-08-01).

[29] NVIDIA. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide, (visited on 2014-08-01).

[30] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide, (visited on 2014-08-01).

[31] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. In *Computer Methods in Applied Mechanics and Engineering*, 2011.

[32] K. A. Seitz, Jr., A. Kennedy, O. Ransom, B. A. Younis, and J. D. Owens. A GPU Implementation for Two-Dimensional Shallow Water Modeling. In *ArXiv e-prints*, 2013.

[33] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems. In *Proceedings of the 26th ACM international conference on Supercomputing*. 2012.

[34] H. K. Stensland. INF5063 – GPU & CUDA. http://www.uio.no/studier/emner/matnat/ifi/INF5063/h13/Resources/inf5063-nvidia_gpu.pdf, (visited on 2014-08-01).

[35] M. L. Sætra and A. R. Brodtkorb. Shallow Water Simulations on Multiple GPUs. In *Applied Parallel and Scientific Computing*. Springer Berlin Heidelberg, 2012.

[36] The OpenMP ARB. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, (visited on 2014-08-01).

[37] The OpenMP ARB. The OpenMP® API specification for parallel programming. http://openmp.org/wp/, (visited on 2014-08-01).

[38] E. F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, LTD, 2001.

[39] S. Venkatasubramanian and R. W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *Proceedings of the 23rd international conference on Supercomputing*. 2009.

[40] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. A. García, M. J. Castro, and R. Doallo. A Multi-GPU Shallow Water Simulation with Transport of Contaminants. In *Concurrency and Computation: Practice and Experience*, 2013.

[41] P. Wang, T. Abel, and R. Kaehler. Adaptive Mesh Fluid Simulations on GPU. In *New Astronomy*, 2010.

[42] Y. Zhang and F. Mueller. Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012.