UiO : Department of Informatics
University of Oslo

Load-balancing multi-GPU shallow water simulations on small clusters

Gorm Skevik
master thesis autumn 2014

Load-balancing multi-GPU shallow water simulations on small clusters

Gorm Skevik

1st August 2014

Abstract

Historically, the use of graphics cards for scientific computing has yielded great performance. Order-of-magnitude performance gains has been obtained over the CPU (Owens et al. 2008) . This is mainly due to their massively parallel nature, delivering a possible performance gain over the traditional CPU. In this thesis, I extend a current single node/GPU shallow water simulator (Brodtkorb et al. 2012) to utilize a parallel environment composed of multiple nodes and multiple graphics cards, enabling larger and faster simulations. For this purpose, a row domain decomposition technique is implemented, dividing a global domain into several subdomains to be distributed between the different processing units. In an attempt to minimize communication latency between the units, a technique called *Ghost Cell Expansion* is also implemented. The main work of the thesis looks into load-balancing the workload between the units, in an attempt to achieve more efficient multi-GPU/node simulations. To load-balance the workload appropriately, several challenges arise. For example, one need to take into account the computational power of the graphics cards to correctly determine the amount of workload for each card. Also, one should take into account the underlying water placement, i.e., wet and dry cells of the domain throughout the simulation run. For this purpose, dynamic auto-tuning techniques are demonstrated and discussed. The implementation has been applied to the shallow water equations, but is general in use, and will work equally well for any systems of conservation laws. Finally, the implementation is thoroughly benchmarked on both multi-node and multi-GPU parallel environments.

Contents

List of Figures

List of Tables

Listings

Preface

This master thesis has been developed at SINTEF ICT at the department of Applied Mathematics. The work on the thesis has been performed individually. Also, several solutions and algorithms have been discussed and influenced by other master students in my research group.

The work has been carried out on two computers: The first with a 2.67 GHz Intel Core i7 920 CPU, 6 GB system memory and two GeForce GTX 480 graphics cards, each with 1.5 GB memory. The second computer has a 3.4 GHz Intel Core i7 2600K CPU, 8 GB system memory and two graphics cards. Different type of graphics cards have been used on this computer. First of all, a GeForce GTX 480 with 1.5 GB memory has been used. I also used a GeForce GTX 580 with 1.5 GB memory and a GeForce GTX 285 with 1 GB memory. This is specified as necessary. Both computers run Ubuntu 12.04.

Acknowledgements: I would like to thank my supervisors, André Brodtkorb, Franz Fuchs, and Martin Reimers for the support throughout the thesis. I especially thank André Brodtkorb and Franz Fuchs for reading through drafts of the thesis and giving feedback.

Chapter 1

Introduction

This thesis explores load-balancing between multiple GPUs and multiple nodes by investigating dynamic auto-tuning techniques. These techniques are applied for the shallow water equations, but are general in use, and can be used for other systems of conservation laws as well. Examples of such systems are the Euler equations [9] and MHD equations [22]. My implementation is based on a single-GPU shallow water simulator [7], which uses a second-order explicit finite volume scheme to compute the shallow water equations, usable to simulate dam breaks and flood scenarios. The primary motivation for simulating such scenarios are in preparation of events and in response to ongoing events [4], for example to create inundation maps and perform real-time visualizations to gain an overview of the water flow. I extend this simulator to run on single-node and cluster systems of multiple GPUs, increasing the ability to perform faster and more accurate large-scale simulations.

I first motivates the use of dynamic auto-tuning techniques on multi-GPU and multi-node systems in section 1.1. Then, I present two research questions in section 1.2 which are thoroughly addressed throughout the thesis. Finally, I give an overview of the thesis in section 1.3.

1.1 Motivation

The use of the GPU to speed up computations for systems of conservation laws have been widely discussed in many publications [15, 11, 3, 39, 7]. The current trend moves towards using multiple GPUs [2], motivating the use of this tremendous computing power provided to solve systems of conservation laws. As reported in [7], the current single-GPU shallow water simulator run an accurate simulation of the first 4000 seconds of the Malpasset dam break case in 27 seconds. By utilizing multiple graphics cards in a single node or across multiple nodes, it could compute the same simulation much faster or enable the use of larger or more accurate simulations. Several publications regarding the use of multiple graphics cards and multiple nodes have also been published [39, 38, 1, 35].

However, these publications do not look at auto-tuning to load-balance the workload between GPUs appropriately, leaving the workload

distribution static. This motivates the use of auto-tuning techniques to distribute the workload appropriately between the nodes and graphics cards. If applied correctly, this has the potential of giving a speed up over a static distribution. I show benchmarks showing a speed up over a static distribution for domains with large amount of wet cells. For load-balancing, it would be of great benefit to take the computational power into account. This way, it would be possible to distribute different amounts of workload to each graphics card depending on their processing power. This should work adequately on all GPUs, from low-end to high-end, as well as different generations.

For example, systems of conservation laws involve a computational domain that often contains areas where computations are not necessary. In shallow water simulations, this means that dry areas does not need to be computed, which implies that the wet areas should be distributed between the graphics cards. Dynamic auto-tuning techniques can be applied to change this distribution according to the GPUs computational power as the simulation progresses.

1.2 Research questions

From the above description and motivation, I will address the following questions in my thesis:

1. How can multiple graphics cards and multiple nodes efficiently be used to simulate systems of conservation laws?

2. What type of auto-tuning techniques can be used to achieve ideal load-balancing between the graphics cards?

The first question asks how the single-GPU simulator should be extended to efficiently support multiple graphics cards across multiple nodes. There are several challenges related to this. First of all, I need to look into domain decomposition techniques to divide a global domain into several subdomains. This way, it would be possible for each GPU to compute on a subdomain, decomposed from the original domain. The technique needs to minimize the amount of communication between the GPUs. It is also important that the solution scales well to N number of GPUs. Also, it is necessary to implement communication techniques to propagate the solution correctly between the subdomains. For this, one should look into techniques to minimize the communication overhead between the GPUs. This is especially relevant when the graphics cards are located on different nodes, as the data is transferred over the network for this case.

The second question asks how the multi-GPU cluster simulator can be implemented so that the domain is load-balanced between the graphics cards. Consider a typical flood simulation represented by a domain. Here, the domain consists of several wet and dry areas. For this, it is important to

implement a technique so that the GPUs only calculate on the wet areas of the domain. This is because computing on dry areas is not necessary and can be skipped to avoid wasting compute cycles. Also, to load-balance, all GPUs should compute on a specified amount of wet areas according to their computational power.

1.3 Organization of thesis

The thesis is organized around these main research questions. In the next chapter, I give a background to the most important concepts used throughout the thesis. I first give an introduction to the GPU as a computing platform, followed by a mathematical introduction to the Single-GPU shallow water simulator [7] which the implementation of this thesis is based on. The Single-GPU shallow water simulator is also introduced.

Chapter 3 gives a thoroughly explanation of my multi-GPU and multi-node shallow water simulator. I propose techniques for extending the single-GPU simulator to run on multiple GPUs. I first discuss techniques for dividing a domain into several subdomains. Two additional techniques, *Ghost Cell Expansion* and *Early exit* are also explained. Then, I explain how this was extended to run on a cluster of nodes. Finally, performance benchmarks for the multi-GPU and multi-node simulator are also presented.

Then, the dynamic auto-tuning techniques are presented in chapter 4. First, challenges related to the implementation of auto-tuning are briefly discussed. Then, I propose and discuss auto-tuning techniques for load-balancing between the graphics cards. The implemented technique is divided into two algorithms that works together to determine the final load-balance between the GPUs. Extensive performance benchmarks of the complete auto-tuning technique are also presented.

Finally, I give my concluding remarks in chapter 5. Here, I review the most important results.

Chapter 2

Background

Graphics cards were traditionally developed for graphics computations in computer games, more specifically rendering two dimensional images from geometric objects [5]. As a result, their architecture has traditionally been fully geared towards this task. However, their massively parallel architecture have made them suitable for high-performance computing, which led to the field of *GPGPU*, or General-Purpose computing on Graphics Processing Units [20]. *GPGPU* is the use of the GPU to perform computations that is normally run on the CPU. However, earlier graphics cards could only be programmed using a graphics API like OpenGL [12] and eventually shader (programming) languages like OpenGL Shading Language (GLSL) [13]. This made it hard to utilize their power for general-purpose computations, as this required that the algorithms were mapped to graphic primitives provided by these APIs and shader languages. To solve this, several graphics card vendors have developed frameworks for general-purpose computations on GPUs. For example, NVIDIA released the CUDA toolkit in 2006 [26]. This has simplified the implementation of scientific computations, for example for the use of systems of conservation laws. The shallow water equations used in this thesis are an example of such systems [10].

I will first give an overview of GPGPU programming on NVIDIA graphics processing units in section 2.1. The summary emphasizes the NVIDIA graphics card architecture and the CUDA framework to program on them. Here, a brief programming example is also given. Then, in section 2.2, I explain the mathematical background of the single-GPU shallow water simulator [7]. Here, the shallow water equations and numerical schemes for solving them are introduced. Finally, in section 2.3, a brief introduction of the single-GPU shallow water simulator used in this thesis is given. I summarize the most important parts of its structure and implementation.

2.1 An introduction to GPGPU programming

For GPGPU programming on NVIDIA graphics cards, NVIDIA provides the CUDA toolkit [26]. CUDA can be programmed in C, C++ and Fortran.

In contrast to CPUs, GPUs are specialized for high data throughput and parallel computing. Even with significantly lower clock frequencies they can process data much faster than the CPU. Various GPUs also have different compute capabilities following with a new core architecture. It is therefore important to take this into account when developing applications in CUDA, since newer graphics cards with newer compute capabilities might enable more or better functionality.

2.1.1 Data parallel computations

GPUs are designed to utilize a parallel programming model [28]. They excel when computations can be performed as data parallel computations. Their architecture, compared to a CPU, have much more of their transistors devoted to arithmetic processing and less to caching and control logic like branch prediction.

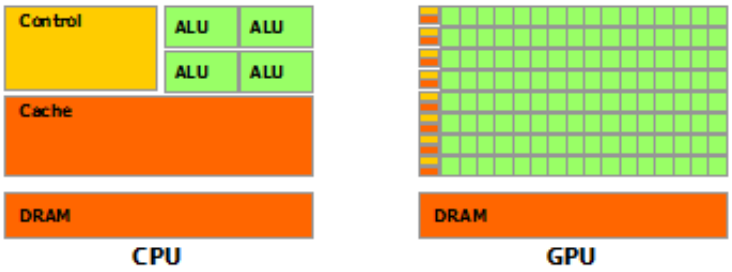


Figure 2.1: An overview of the CPU and the GPU architecture. As seen, the CPU devotes more of its transistors to cache and control logic, while the GPU dedicates most of its transistors to arithmetic units. Original figure from CUDA C Programming Guide [28].

GPUs are also designed to utilize the *Single Instruction, Multiple Data (SIMD)* parallel programming model [33]. SIMD expresses the idea that a single instruction is used on multiple data elements, and therefore all these elements can be processed in parallel. For CUDA, this means that each parallel thread can be mapped to a separate data element while performing the same instruction for all the threads. In CUDA, this is called *Single Instruction, Multiple Threads (SIMT)* [28].

2.1.2 The CUDA framework

The CUDA framework is programmed using functions called kernels [28]. Kernels are a CUDA extension to standard programming functions and are declared using the *__global__* keyword. They run on the GPU and is called from the CPU. It is also possible to declare GPU functions that are only called from the GPU. Such functions are declared using the keyword *__device__*. The kernels are run using a large number of threads that all execute the same kernel in parallel. These threads are again divided into blocks, called *thread blocks*. All the threads inside a block can communicate

with one another by using *shared memory*. It is also possible for threads in different blocks to communicate together. However, this must be performed with the *global device memory* on the GPU.

When a kernel is executed, the blocks are split into *warps*, where each warp contains 32 threads. Each of these warps are processed in a SIMD fashion which means that all the 32 threads runs the same instruction in parallel. In addition, all the blocks are processed by a *streaming multiprocessor (SM)*, which a GPU has many of. Thread blocks can be executed independently. This means they can be executed in parallel, as well as serially. This makes it easy to write code that scales with the number of cores on the GPU. When a kernel is executed the blocks are distributed between the available multiprocessors. The threads in a block execute concurrently and the SMs also have the possibility to execute blocks concurrently. If a given program uses a high number of multiprocessors it will run faster on a GPU with more of these than on a GPU with fewer multiprocessors.

A detailed guide of CUDA is given in the *CUDA C Programming Guide* [28].

The CUDA programming model

The CUDA model assumes a system consisting of a CPU, also called *host* and a graphical processing unit, referred to as the *device* [28]. This means that when programming in CUDA, the kernels with their related blocks and threads run on the device while the rest of the program runs on the host. The CPU and GPU is directly related to each other in that the host allocates all the data, then invokes and transfers this data to the device. The GPU acts as a co-processor which devotes all its power to processing this data. When it is done processing, the data is often copied back to the system memory again. For optimal performance, the serial parts of the code should therefore be processed on the CPU, while the parallel parts should be offloaded to the GPU [27].

Thread hierarchy

There are two main attributes to work with in CUDA [28]. These are *threadIdx* and *blockIdx*. These are used when parallelizing the code. Also, they are both three dimensional vectors. This makes it easy to implement parallel CUDA code for one-, two- and three-dimensional data structures and algorithms. These attributes can be used to identify the current thread and block running. They can therefore replace data elements in an existing serial code algorithm. This makes the code parallel so that each thread computes on a separate data element.

Figure 2.2 gives an overview of the thread and block relation in CUDA. As shown, each thread is grouped into a *block* or *thread block* as introduced earlier. Each of these blocks can in turn be grouped into a *grid*. A kernel can be executed by several blocks at the same time and each block can contain up to 1024 threads on modern GPUs [28]. This produces the possibility

of running several thousand threads for one kernel. The figure shows an example of a two dimensional grid and block structure.

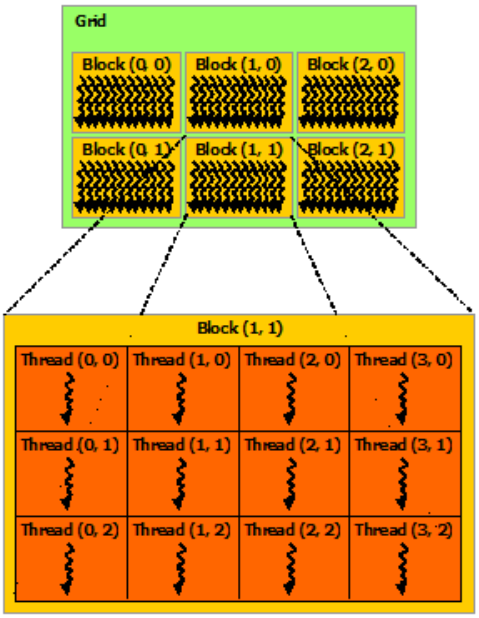


Figure 2.2: An overview of a block configuration in CUDA. Here, the threads are grouped into two dimensional blocks, while the blocks are grouped into two dimensional grids. Original figure from CUDA C Programming Guide [28].

CUDA memory types

There are different memory spaces available on graphics cards [28]. First of all, a graphics card has its own *device memory* which is a type of DRAM memory. This memory can be accessed in several ways; as a *global*, *constant* and *texture memory*, and is shared between all threads. Also, CPU threads can initiate data transfers to and from this memory.

The global memory is the main memory that all threads can read and write to. Constant and texture memory is optimized for constant variables and textures, in addition to being read only. Constant memory is cached in the *constant cache* while the texture memory is cached in the *texture cache*. In addition, each thread has its own *local memory* space, also a part of the device memory. This memory space is used for automatic variables per thread, when there are not enough free registers.

The second type of memory is *shared memory*. This memory is shared and accessible by all the threads in the same thread block, meaning that each block holds a separate part of the shared memory. This type of memory is also on-chip, which makes it a low latency memory useful for communication between threads in the same block. This will give a good speed up compared to using global memory instead.

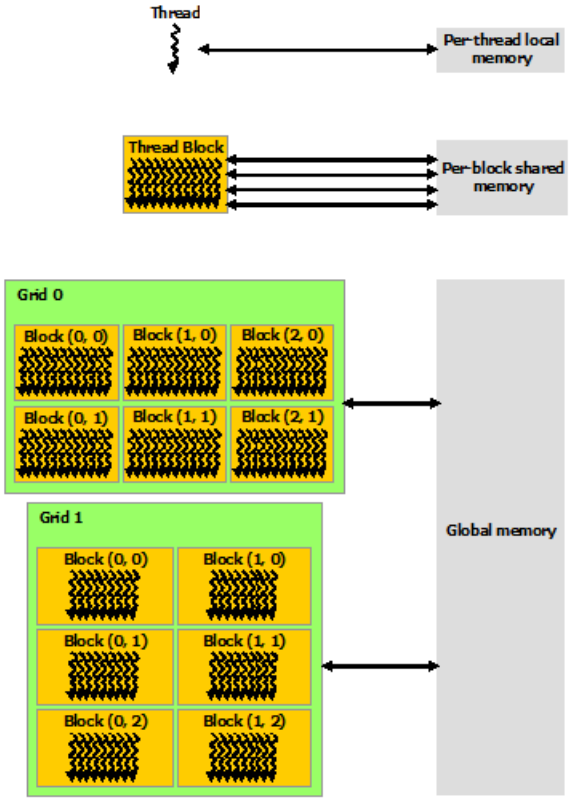


Figure 2.3: An overview of the memory spaces in CUDA. As seen, each thread has a per-thread local memory. Furthermore, each block can access an on-chip per-block shared memory, shared between all threads in the given block. Also, all blocks share access to the global memory, or DRAM memory. Original figure from CUDA C Programming Guide [28].

2.1.3 Programming in CUDA

To demonstrate the parallel capabilities of CUDA, I will here show and explain some modified example code from [28]. This will also serve as a brief practical introduction to CUDA.

As already stated, CUDA is programmed by making a kernel which contains the code to be executed on the graphical processing unit. Kernels are executed as standard function calls with the addition of the <<<, >>> brackets that specifies the *kernel configuration*. This configuration specifies the number of blocks and threads that will be used for the kernel execution. CPU code written in a supported programming language can easily be run on the GPU by only putting the code inside the kernel. Listing 2.1 shows some basic CPU code that loops through and adds elements of vectors of N size. The kernel is executed only one time, as a one dimensional configuration of 1 block and 1 thread.

Listing 2.1: Basic kernel example

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
```

```
{
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation with 1 thread
    VecAdd<<<1, 1>>>(A, B, C);
}
```

However, to utilize the GPU architecture effectively, the parallel possibilities in the code have to be identified. For example, loops can easily be unrolled and replaced with CUDA attributes that specifies which thread should compute on which data element in the code. Listing 2.2 is a modified version of the above code with the loop unrolled and replaced with parallel threads executing the same kernel and each computing on its own part of the vector. For this case, the code uses the attribute *threadIdx.x* which gives the ID of a given thread numbered from 0 to N-1.

Listing 2.2: Kernel executed in parallel

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Before a kernel call is made, it is also necessary to copy the CPU data the kernel will compute on, to the GPU. This is done using a CUDA memcpy function call. When the kernel is finished, the results from the computations can be copied back to the CPU. This is demonstrated in listing 2.3.

Listing 2.3: CPU-GPU transfers

```
int main()
{
    // Copy data to the GPU
    cudaMemcpy(A, srcA, A.size * sizeof(float),
                CudaMemcpyHostToDevice);
    cudaMemcpy(B, srcB, B.size * sizeof(float),
                CudaMemcpyHostToDevice);
```

```
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);

    // Copy results back to the CPU
    cudaMemcpy(dstC, C, C.size * sizeof(float),
               CudaMemcpyDeviceToHost);
}
```

2.2 Mathematical background

Shallow water equations are a system of equations of hyperbolic conservation laws. A hyperbolic conservation law is used in a physical system to describe a set of conserved quantities [21]. These equations can be used to model physical phenomena such as dam breaks, tsunamis, tidal waves and other forms of fluid motion. The shallow water equations can be written in conservative form:

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

Here, h denotes water depth, hu and hv is the momentum along the x-axis and y-axis on a Cartesian coordinate system while g is the acceleration due to gravity. An interesting note is that these equations are only applicable in cases where the vertical velocity is small compared to the horizontal velocity [7]. This however, applies in many cases for different types of fluid motion. In vector form, the equations (2.1) are given as

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

Here Q is the vector of conserved variables while F and G is the fluxes along the x-axis and y-axis on a Cartesian coordinate system, respectively.

For hyperbolic equations such as these, the standard way to solve is to introduce a domain divided into cells and calculate the fluxes using an explicit scheme. Two classical schemes for numerical solutions to such hyperbolic equations are the Lax-Friedrichs and Lax-Wendroff schemes [18, 19]. A brief summary of the Lax-Friedrichs scheme is given to serve as a basic numerical scheme to solve the shallow water equations (2.1). A more detailed introduction can be found in [10, 21]. The scheme can be written in 1 dimension as

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

Here, f is the flux on a cell interface and u is either the water depth or the momentum. The notation u^n_i represents the quantity u at grid cell i at timestep n. This could also be written $u(x_i, t_n)$. This quantity is here represented as the average of the cell.

The purpose of this scheme is to compute the Lax Friedrichs flux. The fluxes on the left and right cell interface are computed and the average of these is used to approximate the flux for a cell boundary. These types of operations, where a cell is updated according to the values of its neighbors are also called *stencil computations* [40]. As can be seen, the Δt is also needed. This is computed according to the Courant-Friedrichs-Lewy (CFL) condition [21]. This is necessary for the solution to converge. This condition is as follows:

$$\frac{\Delta t}{\Delta x} max|\lambda(u)| \leq 1. \tag{2.4}$$

Here, (Δx) is the cell size and $max|\lambda(u)|$ is the maximum absolute eigenvalue of the domain. The average cell quantity must also be updated according to

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{\Delta x}(\hat{f}_{i+1/2}^n - \hat{f}_{i-1/2}^n). \tag{2.5}$$

As can be seen, this equation updates the cell average with the computed Lax-Friedrichs flux at the next timestep $n+1$. One problem with the Lax-Friedrichs scheme is that it smears the solution [10]. To approximate it better, high-resolution schemes or REA with piecewise linear reconstruction can be used [21].

The shallow water equations (2.1) defined only works for domains with flat bathymetry. To correctly simulate dam breaks, tsunamis, tidal waves and other forms of fluid motion over realistic terrain, bed slope and bed shear stress friction terms needs to be added. The equations in conservative form is now defined as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{2.6}$$

Here, B is the bottom topography and C_z is the Ch*é*zy friction coefficient. In vector form, the equations (2.6) are given as

$$Q_t + F(Q)_x + G(Q)_y = H_B(Q, \nabla B) + H_f(Q). \tag{2.7}$$

Here, the additional terms H_B and H_f represents the bed slope and bed shear stress source terms, respectively. Figure 2.4 defines both the water depth h and the water elevation w. It is important to distinguish between these two. I therefore define $Q = [w, hu, hv]^T$ as the vector of conserved variables, and use this throughout the rest of the thesis. This means that Q_1 is the water elevation, while Q_2 and Q_3 is defined as the water momentum along the x- and y-axis of a Cartesian coordinate system, respectively. To numerically solve the equations (2.6) Brodtkorb et al. [7] required a numerical scheme that was well-balanced, conservative, second-order accurate in space and supported dry zones. The Kurganov-Petrova scheme [17] fits well with these properties and was therefore chosen as the numerical scheme. According to [6], the scheme also has a good utilization of the GPU architecture, in addition to being sufficiently accurate for single-precision arithmetic. This provides twice as fast data transfers and

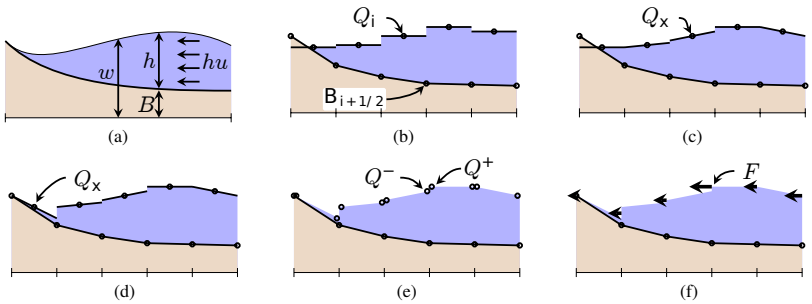


Figure 2.4: (a) Shallow water flow on a bathymetry in 1D and definition of variables. (b) Discretization of the conserved variables Q as cell averages. Here, B is defined at the cell intersections. (c) Reconstruction of slopes for each cell using the generalized minmod flux limiter. (d) Modification of slopes to avoid negative values for h. (e) Reconstruct intersection values from the left and right slopes. (f) Computing fluxes at each cell interface using the central-upwind flux function [16]. Original figure from Brodtkorb et al. [7].

arithmetic operations over double precision, in addition to that all memory storage only uses half the space [7]. The Kurganov-Petrova scheme can be written:

$$\begin{aligned}\frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) \\ &\quad - [F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})] - [G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})]\end{aligned} \tag{2.8}$$

Here, B is the bathymetry defined at the cell corners and Q is given as cell averages. H_f and H_B represents the bed shear stress and bed slope source terms, respectively. F and G represents the fluxes across the cell interfaces along the x-axis and y-axis on a Cartesian coordinate system.

2.3 The single-GPU shallow water simulator

The implementation of my work is based on the single-GPU shallow water simulator by Brodtkorb et al. [7]. Interested readers are referred to this article for a more detailed description. The original code is implemented and runs on a single NVIDIA GPU. As stated in [7], the GPU was chosen as the computational processor due to two main reasons. First, GPUs have evolved to more general computational units and are therefore widely used in scientific computing. Secondly, they are optimized for high data throughput and parallelism and can therefore process much more data than the CPU. Furthermore, the application is implemented in C++ and relies on CUDA [26] for solving the numerical scheme, while OpenGL [12] is used to visualize the results.

As mentioned, the numerical scheme chosen is the Kurganov-Petrova scheme. The main work for the scheme is to solve the shallow water equations (2.6) in 2 dimensions. This scheme maps well to the GPU and

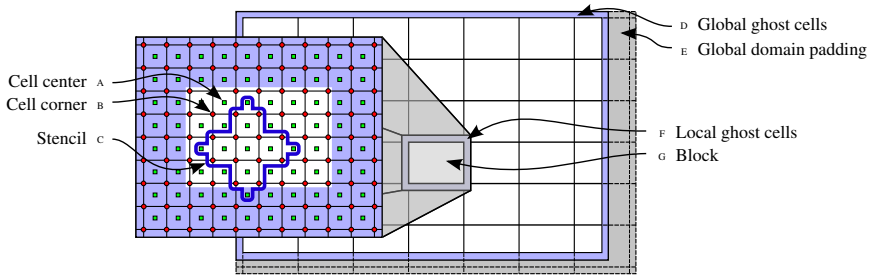


Figure 2.5: Figure shows a domain and how it is decomposed into CUDA blocks on the GPU. A block is composed of several cells, each cell corresponding to a CUDA thread, computing in parallel. Notice that the blocks also have local ghost cells. The data variables Q, H_B and H_f are given at cell centers (A) and B is defined at cell corners (B). Original figure from Brodtkorb et al. [7].

its parallel architecture because it is an explicit scheme defined over a grid. This makes it particularly effective to implement it on a GPU over the traditional CPU, as each cell in the domain can be computed independently and in parallel by a CUDA thread. Figure 2.5 outlines the domain block decomposition of the simulator. Here, a domain is decomposed into blocks and cells on the GPU, each cell corresponding to a thread. Each of the blocks and threads runs in parallel on the GPU. Brodtkorb et al. [6] also made use of single precision over double precision as this was proven to be accurate enough for the selected scheme. This provides twice as fast data transfers and arithmetic operations. Also, the application supports both the first-order accurate Euler time integrator and the second-order accurate Runge-Kutta time integrator.

The calculations for the numerical scheme are performed by four CUDA kernels, outlined in figure 2.6 in a single *step* function. These four kernels are, in order: *calculating the fluxes*, *finding the maximum timestep* Δt, *evolving the solution in time*, and finally *applying boundary conditions*. A full timestep with the second-order accurate Runge-Kutta time integrator need to run through this step two times.

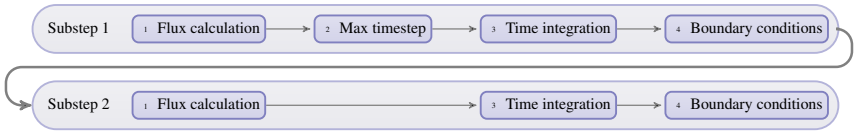


Figure 2.6: The single-GPU simulator *step* function. The first substep calculates the fluxes, finds the maximum timestep, evolves the solution in time, and applies boundary conditions, The second substep calculates the fluxes, evolves the solution in time, and finally, applies boundary conditions. Note that the first-order time integrator only runs through the first substep, while second-order runs through both. Original figure from Brodtkorb et al. [7].

Flux calculation: This kernel is responsible for computing the net flux from (2.8). As can be seen in (2.8), this involves first computing the flux at all cell interfaces, and then computing the bed slope source term, H_B for all cells. This is then summed to find the net flux. First, it reconstructs the value of B at each interface midpoint to make it aligned with Q. Then, it performs a reconstruction of the slopes for each cell using the generalized minmod flux limiter (see figure 2.4c). This reconstruction does not guarantee positive values at the wet/dry integration points for the water depth h. Therefore, modification of this variable is necessary to avoid negative values, as seen in figure 2.4d. If this was not done, the numerical scheme would not handle dry zones because the eigenvalues are $u \pm \sqrt{gh}$. Then, the kernel reconstructs the intersection values Q^+ and Q^- from the slopes (figure 2.4e). These values are then used to compute the fluxes F and G in (2.8) at each cell interface using the central-upwind flux function [16], seen in figure 2.4f. Finally, it computes the bed slope source term H_B in (2.8), and calculates the net flux by summing this with the computed fluxes. Also, the flux kernel is responsible for computing the maximum timestep per CUDA block, used in the timestep kernel to compute the final maximum timestep. This is done by computing the minimum eigenvalue per block.

Maximum timestep: This kernel inputs the per block timesteps calculated in the flux kernel and computes the maximum timestep Δt from these. The timestep is computed according to the CFL condition (3.1). The maximum timestep is the maximum possible timestep to use and therefore the lowest timestep.

Time integration: This kernel first computes the bed shear stress source term, H_f from (2.8). Then, the kernel evolves the solution in time.

Boundary conditions: This kernel applies the boundary conditions for Q used in the simulation. Boundary conditions are implemented using global ghost cells, with two cells in each direction because the scheme is second-order accurate. The application implements several types of boundary conditions. The one used in my implementation is the *wall boundary condition*. This condition is applied by inverting the momentum on the boundary cells so that the water reflects back on the domain.

The application also provides both visualizing with a photo-realistic view and a non photo-realistic view. The last method visualize the physical variables, like for example water depth h, using a color mapping function. Finally, the implementation of [7] was validated against several test cases, including a real world dam break, The Malpasset dam break.

Chapter 3

A multi-GPU and multi-node shallow water simulator

Extending the single-GPU simulator to perform as a multi-GPU cluster simulator can be performed in two main steps. First, the original domain should be decomposed into several subdomains, so that several graphics cards can compute on the domain in parallel. For this purpose, a *domain decomposition technique* has to be implemented. Also, to propagate the solution correctly between the subdomains, techniques for transferring data between the subdomains needs to be investigated. A common method is to exchange the ghost cells between the subdomains, forming an overlapping area. Secondly, this multi-GPU implementation has to be extended to execute on multiple nodes. This can be thought of as running a multi-GPU simulation per process, each process running on a separate node and communicating with the other processes over the network.

First, I explain and discuss several different domain decomposition techniques, emphasising their advantages and disadvantages. This is done in section 3.1. Next, in section 3.2, I propose my implementation of the multi-GPU simulator. Here, I explain the implementation of multiple subdomains on multiple graphics cards, as well as the ghost cell exchange to propagate the solution between subdomains. Then, in section 3.3, I move on to the multi-node implementation, explaining the extensions needed for the multi-GPU simulator to run on a cluster of nodes. In addition, several optimization techniques are demonstrated. First, a latency hiding technique for the ghost cell exchanges, called *Ghost Cell Expansion* is demonstrated and discussed. This is done in section 3.4. Secondly, an implemented optimization technique called *Early Exit* is shown in section 3.5. Finally, I present extensive performance benchmarks of the complete multi-GPU cluster simulator in section 3.6.

3.1 Domain decomposition techniques

Domain decomposition is the task of decomposing a single domain into N number of subdomains. These subdomains can in turn be distributed to different graphics cards, giving the ability to perform computations

in parallel. However, there are several challenges related to solutions consisting of multiple subdomains. First, one needs to determine how a domain should be decomposed. This affects the solution in numerous ways. For example, the number of neighboring subdomains is determined by the decomposition technique. This affects the number of ghost cell exchanges a subdomain needs to perform, and therefore the amount of communication between the GPUs. There exist several methods to decompose a domain, each with its advantages and disadvantages. There have been multiple publications where such techniques have been implemented [31, 34, 37, 1, 35]. Here, I discuss the techniques, with emphasis on how they solve the above challenges. Figure 3.1 shows an illustration of the techniques.

3.1.1 Row decomposition

I will first discuss the *row decomposition* technique, implemented by Sætra and Brodtkorb in their multi-GPU implementation [35]. The technique decomposes the global domain into N subdomains on whole rows so that each new subdomain consists of a set of rows from the original domain. Using this technique to decompose a domain with a resolution of 1000×1000 into two subdomains, each subdomain receives a resolution of 1000×500. It is also straightforward to decompose into subdomains of varying resolutions with this technique. For example, in the above example, the subdomains can also be initialized as 1000×700 and 1000×300 resolution respectively. This is advantageous when using several different GPUs, because each GPU can be set up with a subdomain suitably sized for their computational power.

Sætra and Brodtkorb also [35] emphasizes two other advantages with this technique. First of all, the technique enables the transfer of continuous parts of memory between the subdomains when performing a ghost cell exchange. This is because the domains are allocated linearly in memory, row by row, meaning that several rows can be packed into a single transfer. The second advantage is that each subdomain has at most two neighbors, therefore decreasing the amount of data transfers necessary when doing a ghost cell exchange.

3.1.2 Column decomposition

Instead of decomposing the domain row by row, it is also possible to decompose it along the columns. This technique is identical to the above, only that each new subdomain would consist of whole columns instead of whole rows. Decomposing a domain of resolution 1000×1000 into two subdomains would yield a resolution 500×1000 for each subdomain. This technique has similar advantages as *row decomposition*. First of all, it is simple to initialize subdomains of varying resolutions. Secondly, it only gives a maximum of two neighbors to perform ghost cell exchanges with, therefore minimizing the amount of subdomains to communicate with. However, a disadvantage is that it is not able to transfer whole

columns continuously in memory since the data is allocated row by row in memory. This is important, since the ghost cells have to be exchanged by transferring columns to the neighboring subdomains on the left and right side. To transfer a column, one would need to transfer one cell at a time. This would result in many small data transfers, compared to a single large data transfer. This is not optimal because of the overhead related to each transfer. Therefore it is better to pack these transfers into a single large transfer [28].

3.1.3 Tile decomposition

The third and final technique is to decompose the domain into several tiles. A similar technique is implemented by Song et al. in [34]. For example, a square domain can be divided into four evenly sized subdomains. I demonstrate it by decomposing a domain of resolution 1000×1000 into four subdomains. In this case, all subdomains would get a resolution of 500×500. It is also possible to set up these as varying resolutions. There exist several disadvantages with this technique. First of all, one would get several more neighbors to perform ghost cell exchanges with, increasing the amount of data transfers. Secondly, when performing ghost cell exchanges with the left or right subdomains, it still has to perform several smaller transfers for the columns. Communicating with the top and bottom subdomains however enables the transfer of rows as a single transfer.

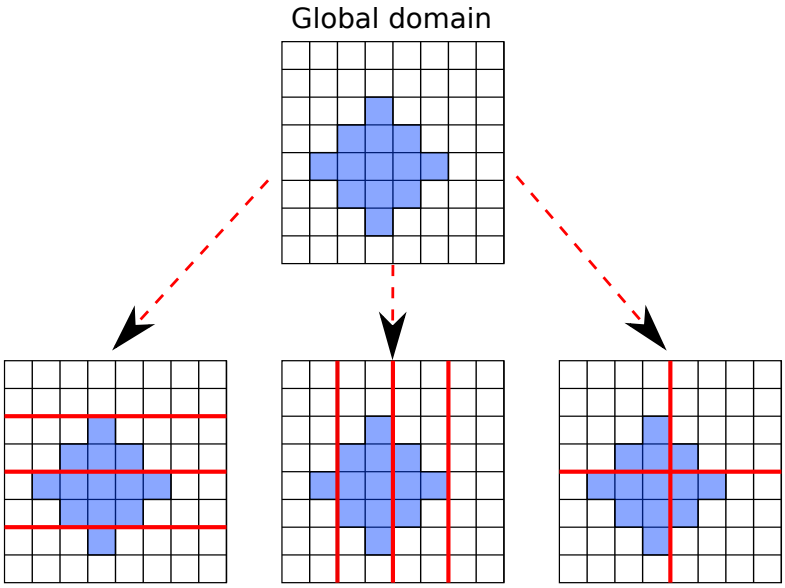


Figure 3.1: A global domain decomposed using three different domain decomposition techniques. The global domain is decomposed into four subdomains. **Left:** Row decomposition. **Middle:** Column decomposition. **Right:** Tile decomposition.

3.2 Multi-GPU implementation

A multi-GPU implementation involves decomposing an initial domain into several subdomains, distributing them to different GPUs. This should function equally as a single global domain, which means that communication between the graphics cards has to be performed. There are several challenges related to implementing a multi-GPU solution. First of all, the solution has to propagate correctly between the subdomains. This means that the water needs to flow between the subdomain boundaries as necessary. Secondly, the timestep Δt also has to be computed correctly between the subdomains. To correctly evolve the solution, all subdomains need to integrate using the same timestep Δt. These challenges, including solutions to them, will be thoroughly explored and discussed in this section.

3.2.1 Setting up two subdomains

To simplify, I first extended the code to only make use of two subdomains. For this, I made a new class that extends the single-GPU domain class. This class represents a subdomain in the multi-domain implementation.

In addition, a new class to manage the multi-GPU logic itself was made. It takes the number of subdomains to initialize, amongst other variables as input. The class has two primary functions, *swap* and *step*. The first function performs the ghost cell exchange, while *step* is the extended version of the original single-GPU step function. This function will handle all the new logic required for a multi-GPU simulation. This function is repeatedly called to evolve the solution in time until the simulation is finished. All the necessary functions and their implementation will be extensively described in this section.

Initializing the subdomains: To initialize the subdomains, a domain decomposition technique has to be utilized. Because of its advantages as described above, I implement the *row decomposition* technique. This technique decomposes a global domain into several subdomains, each consisting of a set of rows from the original domain. This is seen in the left illustration in figure 3.1. All subdomains are initialized as equally large parts of the global domain. For example, a domain could have a size of 1000×1000 cells. Utilizing the implemented decomposition technique, the total size for all subdomains will be equal to the original domain. This means that, for two subdomains, each subdomain would have a size of 1000×500 cells. Together, both of these would correspond to the original 1000×1000 domain. Each subdomain is also assigned to a different graphics card. This was fairly straightforward, as the only additional code was to call the CUDA function *cudaSetDevice* at the appropriate places. The function takes the GPU ID as the input. The ID is numerated in the range of 0 to $cudaGetDeviceCount() - 1$. The function *cudaGetDeviceCount()* returns the number of GPUs in the system. The function is called when initializing the subdomains, as well as for each *step* call and memory allocation of the

subdomains. The GPU ID is stored for each subdomain and used where necessary to change the GPU device.

Ghost cell exchange: Each of the subdomains will now be a separate simulator which computes its domain independent of the other. To evolve the solution correctly, the values of each boundary cell needs to be propagated properly between the domains. This is solved by copying boundary cells from both neighboring subdomains into the other subdomain. More specifically, I copy the two bottom rows of the upper subdomain to the ghost cells of the bottom subdomain. Then, I copy the two upper rows of the bottom subdomain to the ghost cells of the upper subdomain. This can be seen in figure 3.2. This is performed for the water elevation and water momentum data, the Q_1, Q_2, Q_3 buffers. The ghost cells of both subdomains are part of an overlapping area, functioning as a boundary condition which connects both subdomains.

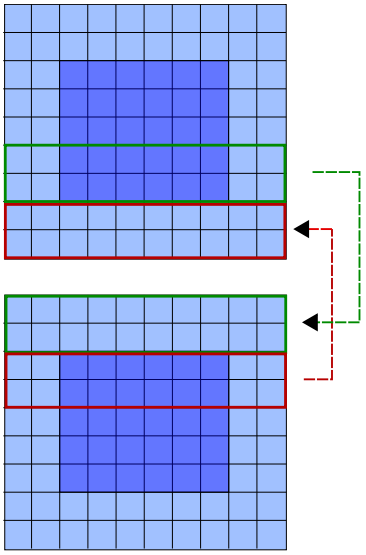


Figure 3.2: A ghost cell exchange for two subdomains, each with a 6×5 resolution. The internal cells are shaded in dark and ghost cells in light. The light shaded parts in the red and green rectangles are the overlapping region that connects each domain in the ghost cell exchange. The rows marked red in the bottom subdomain are copied to the ghost cells of the upper subdomain, while the rows marked green in the upper subdomain are copied to the ghost cells of the bottom subdomain.

The ghost cell exchange between the subdomains is done before each simulation timestep. This means that it is called at the start of each call to *step*. I copy between both subdomains, for each of the three data buffers, the water elevation Q_1, and the water momentum in the x and y directions Q_2 and Q_3. When performing a ghost cell exchange between the subdomains, the data will be transferred over the PCI Express bus. The swapping is performed with the function *cudaMemcpy3DPeerAsync*. This function does a peer-to-peer-transfer, meaning a transfer between two graphics

cards. There are also other alternatives to using this function. Another efficient alternative for older compute capabilities would be to download the necessary data from the source GPU to *pinned* CPU memory and upload this to the destination GPU. I use peer-to-peer memory transfers because this gives the fastest implementation for graphics cards supporting this [29].

As seen, the 3D version of the function was also used. The reason for this is that the data is allocated in a two dimensional fashion. The 3D method therefore provided the most elegant solution to access and transfer the data in a two dimensional fashion. Also, a 2D version of the function was not available in CUDA.

Finally, it is worth noting that I use the asynchronous version of the peer-to-peer function. This is done to give more efficient code, letting the CPU proceed to the next instruction, while the issued CUDA peer memory copy runs in parallel. In addition, it also gives the possibility of overlapping data transfers. The alternative would be that the CPU blocked, waiting for the peer-to-peer memory transfer to finish. This would waste computing cycles on the CPU and the CUDA invocations would not be able to overlap. To enable asynchronously use, a variable called a CUDA *stream* is used. This variable is sent in with each transfer. All calls made with the same stream is processed *in-order* as explained in [28]. When doing a peer memory copy between two given subdomains I issue it with the same stream used for kernel execution on the destination domain. Because of the *in-order* property this guarantees that the GPU will not start kernel invocations for the subdomain before the memory copy is finished.

Timestep synchronization: Each of the subdomains maintains its own timestep, Δt. To evolve the solution correctly, they need to agree on the same timestep. Sætra and Brodtkorb [35] discusses two methods for handling this. The first is to use a globally fixed timestep for both subdomains. For the solution to propagate correctly, this timestep has to be equal to or below the smallest timestep allowed by the CFL condition for the whole simulation. The second method is to synchronize the timesteps from both subdomains and choose the smallest. The reason for choosing the smallest timestep is because it is guaranteed to be an allowed timestep for both simulations. I chose to implement the second method. This method allows the solution to propagate as fast as possible. Using a globally fixed timestep could have given a slower simulation because the timestep could likely have been higher. It is also hard to estimate a large enough fixed timestep that evolves the solution correctly.

Sætra and Brodtkorb [35] also performed tests using both a fixed global timestep and synchronizing the timestep between multiple graphics cards. They found that the difference in performance was negligible for reasonable sized domains. As such, they concluded that synchronizing the timesteps was a viable method to use.

To synchronize the timestep, the current single-GPU *step* function (see figure 2.6) need to be divided in two. I extend this to my own implementation of *step*, shown in listing 3.1. The reason for this is that

the current function calls all the necessary kernel functions in order. It first computes the *fluxes*, then finds the *maximum timestep*, and solves the equations forward in time by executing *time integration*. Finally, it applies *boundary conditions*. I now need to divide this in half so that both domains can synchronize after finding their respective timestep. They can then continue to execute the second half using the same timestep. For this purpose, two new functions were made, *step1* and *step2*. Function *step1* will compute the fluxes and find the maximum timestep, while *step2* performs the time integration and applies boundary conditions. First, both subdomains will now run the *step1*. Then, they continue to synchronize their timesteps. This is performed by transfering the timestep from the GPU of each domain to the CPU. The CPU code iterates through and finds the smallest timestep. Then, it copies this timestep back to the respective GPU of both domains by iterating through them and calling the function *cudaMemcpyAsync*. Finally, the subdomains execute the *step2* function with the same timestep.

3.2.2 Extending to N number of subdomains

Having two subdomains set up and functioning correctly on two graphics cards, I would need to implement N number of subdomains so that an arbitrary number of subdomains could be initialized on an arbitrary number of graphics cards. For this, an extension was made for the ghost cell exchange. Because there is now more than two subdomains, the middle subdomains have two neighbors to exchange ghost cells with. This is shown in figure 3.3. The middle subdomains therefore need to perform a ghost cell exchange in both the north and south direction. The outer subdomains perform the ghost cell exchange as before.

In addition, the subdomains has to be distributed to N number of graphics cards automatically. The number of GPUs to use is inputted as an application argument. This is however restricted to the number of graphics cards available in the computer. I distribute each subdomain to the next available card. For example, if a simulation sets up three subdomains and two GPUs, the first and second subdomain is allocated for the first and second GPU respectively, while the last subdomain is allocated for the first GPU again.

3.2.3 Asynchronous execution

It is also necessary to make all subdomains run the *step1*, *step2* and related functions asynchronously. In the current code there was some synchronization that made the CPU code block at each subdomain. Since it performs calls to subdomains serially, it would therefore not go on to start execution for the next subdomain. This was fixed so that the CPU code would proceed correctly and all subdomains were able to run asynchronously. This is especially important when utilizing multiple GPUs, when each subdomain resides on a different graphics card. Here, it is important that the CPU code continue kernel execution for all

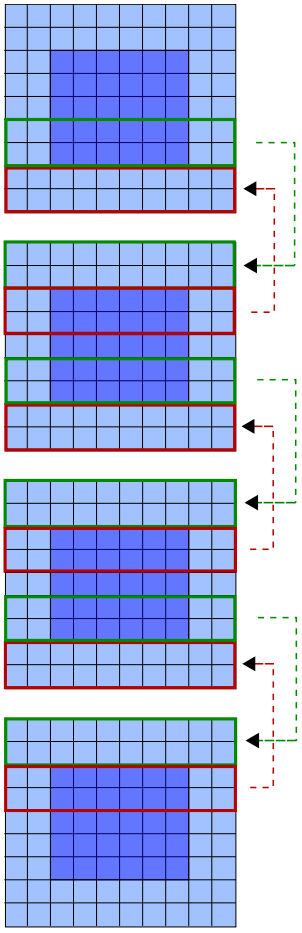


Figure 3.3: A ghost cell exchange for N number of subdomains, each with a 6×5 resolution. Again, the inner cells are shaded in dark and ghost cells in light. The light shaded parts in the red and green rectangles are the overlapping region that connects each subdomain in the ghost cell exchange. Notice that the inner subdomains need to swap two ways, while the outer subdomains only exchange in one direction.

subdomains, while the already executed subdomains run in parallel. As a result, all GPUs are able to run in parallel, providing a speed up over a single-GPU solution.

3.2.4 Extending to subdomains of different sizes

I have also implemented the ability to decompose the subdomains into different sizes. I explain and demonstrate this further with a case consisting of two subdomains A and B, each residing on a separate graphics card. When initializing these domains from the global domain, both could be set to half the size, letting each card compute on half the domain. For many hardware setups, this would be appropriate. However, in several cases it could be more advantageous to initialize the subdomains with different sizes. For example, if both graphics cards had different amount

of computational power, letting both subdomains have the same resolution would not yield optimal performance. This is because the high end card would always finish before the other, stalling the simulation. It would therefore be a better strategy to initialize the subdomains with different resolutions. For example, subdomain A residing on the high end card could get 70% of the original domain, while B residing on the other card could get 30%. This solution would yield better performance because both GPUs would compute on domains more ideal for their performance.

Another advantage with subdomains of different sizes is that they can be divided in different sizes depending on runtime parameters. This opens up for dynamically decomposing the domain into subdomains of different sizes at runtime. This has good potential of improving the performance of the implemented multi-GPU/node simulator.

Implementation: I extend the original domain decomposition by inputting a *workload* argument, describing the percentage of workload each subdomain should be given. I first translate the workload argument to an array where each index represents the workload in percent for a given subdomain. Next, the workload in amount of rows for each subdomain is computed. This is accomplished by looping through the workload array, applying the following formula on each subdomain:

$$wr = \frac{nywp}{100}. \tag{3.1}$$

Here, wr is the resulting workload in amount of rows for each subdomain, ny represents the amount of rows in the original global domain, while wp represents the workload in percent to be given to each subdomain. Rest cases are handled by giving the first subdomain the extra work. The offset, to correctly read from the global buffer is also calculated. Finally, each subdomain is initialized on the correct graphics card as before.

Finally, I show a pseudocode listing for the multi-GPU code, showing the basic structure and function calls of the *step* function.

Listing 3.1: Overview of the *step* function for multi-GPU

```
void step()
{
    if(one simulator)
        step() // Run the original single-GPU simulator
    else // For multi-GPU
        /*
        * Perform ghost cell exchange
        * between all subdomains
        */
        swap();

        /*
        * Call step1 for all subdomains,
        * running the flux and timestep kernels
```

```
        */
        for each subdomain
            step1();

        /*
        * Perform dt synchronization
        */

        /*
        * Get timestep from all subdomains
        * and find the smallest
        */
        for each subdomain
            getDt()

        /*
        * Set the smallest timestep for all subdomains
        */
        for each subdomain
            setDt()

        /*
        * Call step2 for all subdomains,
        * running the time integration and BC kernels.
        */
        for each subdomain
            step2();
}
```

3.3 A multi-node simulator

I have implemented a multi-node simulator running my multi-GPU shallow water simulator on several parallel processes. This is implemented using the *Message Passing Interface (MPI)* [24], a high performance API for communication. I use version 3.1 of the MPICH [25] implementation of MPI. It is then possible to initialize the shallow water simulator with several processes running a copy of the same executable. This enables it to take advantage of several computers, referred to as *nodes*, so that each process runs on a different node. Each process can also consist of a set of subdomains running on multiple graphics cards residing on the node. As such, I have a fully working shallow water simulator that utilizes multiple levels of parallelism; *multiple nodes*, *multiple graphics cards* in each node, and the parallelism on the GPU itself (see section 2.3 and figure 2.5). Enabling the use of multiple nodes also gives the possibility to use external computing power such as the *cloud* to compute simulations.

3.3.1 Design

The multi-node simulator is implemented as modularly as possible. I wanted the new code to run on top of the existing multi-GPU code so that the new code did not interfere too much with the existing code. This enabled me to write the code as new functions that runs in addition to the existing functions. This means that, when running a multi-node simulation, the application would proceed as before, calling the necessary *swap* and *step* multi-GPU functions. However, it would also call additional functions that handle multi-node specific logic, for example ghost cell exchange between subdomains at different processes. These functions runs completely separated from the current code.

Also, a multi-node version of the current code is, in reality, only several executables of the same code running in parallel on different computers. This means that each of the executables are a separate multi-GPU shallow water simulator running and computing on its subdomains completely separated from the others. The only exception being some synchronization and communication that needs to be done between the processes at specific locations in the code. Figure 3.4 outlines the multi-GPU cluster simulator.

This design has several advantages. By separating the code as much as possible I know that the existing multi-GPU code runs just as before and as such it is not possible to introduce new errors here. In addition, it made it much easier to implement the new code, as well as implementing new code for both the multi-GPU and multi-node version. If I make any changes to any of them it should not interfere with the other. Figure 3.4 outlines the multi-node simulator more closely.

3.3.2 Implementation

For the implementation, several extensions are made to the code. First, a new runtime argument *multi_node* is made. This can be set to *true* to run a multi-node simulator. This variable is used to check if the multi-node version is currently executing and then do the additional function calls to handle the multi-node logic. I also store the process *rank* (process ID) as given from MPI functions for each process. This is used so that processes can execute different code where they need to. Finally, *MPI_Init* and *MPI_Finalize* are called to handle allocation and deallocation of MPI.

To simplify, I made an assumption regarding how the subdomains were divided between the processes, and divide them serially. For example, if four subdomains and two processes are initialized, the processes will receive the first two and last two subdomains respectively. In contrast, it is not possible to give the first process the first and last subdomain while the second process gets the two middle subdomains. This would however give more flexibility. For example it would be easier to distribute the subdomains that contained more wet cells to the nodes that had more powerful hardware. However, this gave unnecessary complexity that is not necessary to deal with to compute simulations efficiently. I also figured out that I could get more or less the same results by simply changing the domain

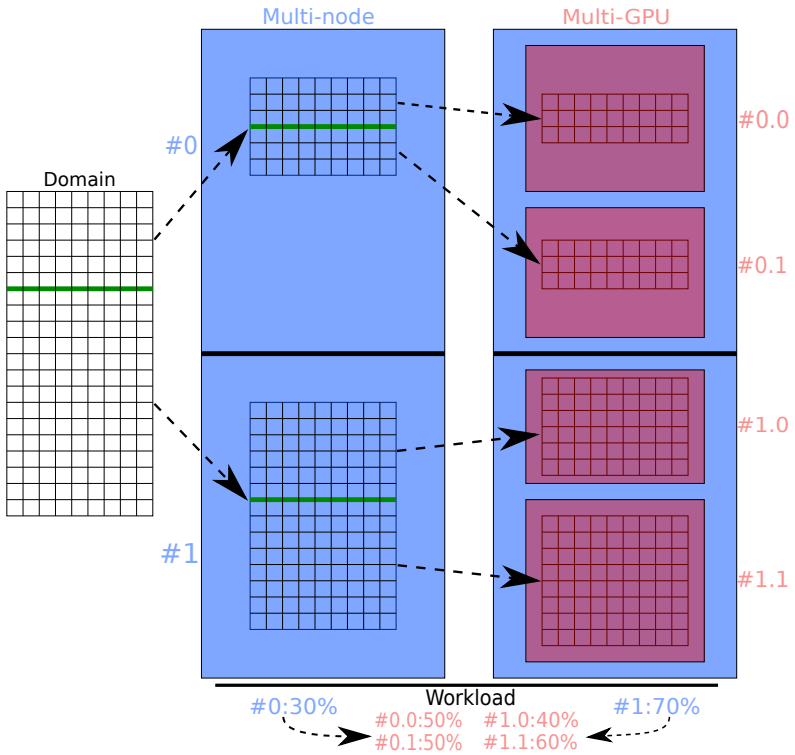


Figure 3.4: Shows the two implemented levels of parallelism of the shallow water simulator. Here, each blue box represents a node and each red box represents a GPU on that node. The figure shows a global domain that is distributed between two nodes so that each node in parallel compute on a separate subdomain. These subdomains are also computed in parallel between two GPUs on both nodes. As seen in the figure, each node and GPU have a different amount of workload of the global domain. The implementation supports N number of nodes and N number of graphics cards per node.

size dynamically during runtime. This functionality is also implemented as a part of the simulator. By doing this, it would be possible to compute larger domains on the nodes with more computational power.

Initializing the subdomains: For the code, I first of all modify the code to initialize the simulation correctly for a multi-node setup. The initialization sets up each subdomain as before and is therefore decomposed from a single domain. The only additional change is to check if multi-node is enabled, and distribute them appropriately between the processes. To make the implementation more flexible, I also made new runtime arguments to determine the number of subdomains and GPUs to run per process.

Ghost cell exchange: Additional changes were also made for the ghost cell exchange. The standard single-process exchange is performed exactly as before. The *swap* function is executed to exchange ghost cells between subdomains located in a given process. However, for the multi-node sim-

ulation, one also need to do additional ghost cell exchanges between subdomains located at the interfaces of each process. This means that I need to perform ghost cell exchanges between the processes. This is handled by a new function called *swap_mpi*. The main difference between these two functions is that the first function only do data transfers between graphics cards located in the same computer, while the last perform data transfers between different computers connected over a network. The function *swap_mpi* therefore takes longer time to perform. The function is implemented in a similar manner as *swap* with the exception that it uses MPI function calls to transfer the ghost cells across the network.

I implement the ghost cell communication asynchronously so that each MPI call is asynchronous. This is done so that the CPU can proceed and issue the next MPI calls, overlapping with the other MPI transfers. For each process, I handle communication at each interface of the subdomain. The subdomain data to send is first downloaded into three separate buffers called *send_cpuBuffer1*, *send_cpuBuffer2* and *send_cpuBuffer3* for each of the Q_1, Q_2, Q_3 buffers of the simulator. Next, a call the function *MPI_Irecv* is performed, receiving the three buffers from the other process into three buffers called *recv_cpuBuffer1*, *recv_cpuBuffer2* and *recv_cpuBuffer3*. Then, the first three buffers are sent with the MPI function call *MPI_Isend* to the other process. Finally, the received data in the final three buffers can be uploaded to GPU memory. Before this however, it is necessary to block on receive to make sure the process have received the ghost cells from the other process. If this was not done I would risk uploading buffers pointing to invalid data causing large errors in the implementation. This is done by calling the function *MPI_Waitall* for all receive requests.

Timestep synchronization: Finally, for the timestep synchronization, I also add additional code. The existing code finds the smallest timestep between all the subdomains of a single process. This code is first run per process, similar to the non multi-node implementation. This means that each process computes the smallest timestep for all its subdomains as before. The timesteps for all processes are then synchronized to find the smallest timestep between the processes. This is the only code that is added in addition to the existing synchronization. I implement this using MPI function calls.

There are several ways to implement this. One possibility would be to designate a master process which has the responsibility of receiving all timesteps from the other processes, compute the smallest and broadcast the value back to the other processes. The function calls *MPI_Isend*, *MPI_Irecv* and *MPI_Ibcast* could be used to handle this. Instead, I chose to use the function *MPI_Iallreduce* which encapsulates this functionality in one single function call. This method does a reduction of the values from all processes and sends the result back to all processes. The function can perform several types of reductions. I send in the argument *MPI_MIN* to do a minimum reduction. This allows me to find the smallest timestep between all processes in a single function call. Finally, each process can now upload this timestep to the GPU as before.

Finally, listing 3.2 shows the simulator function calls for multi-node. It shows the basic multi-GPU code run by each process, as well as the multi-node specific extensions.

Listing 3.2: Overview of the *step* function for multi-node

```
void step()
{
    if(one simulator AND no multi-node)
        step() // Run the original single-GPU simulator
    else
        /*
        * Perform ghost cell exchange
        * between all subdomains and all processes
        */
        swap();
        if(multi_node) swap_mpi

        /*
        * Call step1 for all subdomains
        */
        for each subdomain
            step1();

        /*
        * Perform dt synchronization,
        */

        /*
        * Get timestep from all subdomains
        * and find the smallest
        */
        for each subdomain
            getDt()

        /*
        * Find smallest timestep between processes
        */
        if(multi_node) MPI_Iallreduce()

        /*
        * Set the smallest timestep for all subdomains
        */
        for each subdomain
            setDt()

        /*
        * Call step2 for all subdomains
        */
```

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

3.4 Ghost Cell Expansion

To further improve the performance of the multi-GPU cluster simulator I have implemented a *latency hiding technique* called *Ghost Cell Expansion (GCE)*. The technique is an extension to the ghost cell exchange performed by the *swap* and *swap_mpi* functions. The technique has been thoroughly investigated by others [35, 8, 14] and have shown a good performance increase. The technique allows the implementation to handle less communication of data between the subdomains. The currently implemented solution does one ghost cell exchange per timestep. By increasing the ghost cell overlap between the subdomains, it would be possible to run more timesteps before doing a ghost cell exchange. The increase in overlap results in larger subdomains. This gives larger but fever data transfers, therefore decreasing the overhead related to transferring many small data packages. However, as stated in [8], disadvantages is the use of more memory and the extra computation costs for the slightly larger subdomains. Figure 3.5 shows a ghost cell exchange with no additional overlap.

Various authors have investigated this technique, reporting both a good increase in performance and no additional increase in performance. Sætra and Brodtkorb implemented the technique on a multi-GPU shallow water simulator [35]. They benchmarked the technique using three domain resolutions on three different systems. The first system was a Tesla S1070 system with four Tesla C1060 GPUs, the second system is a SuperMicro SuperServer consisting of four Tesla C2050 GPUs and the third system is a desktop computer consisting of two GeForce GTX 480 graphics cards. The first system, the Tesla C1060 system, gave no additional performance increase for large domains. They concluded that an overlap of 4, resulting in a ghost cell exchange each timestep, was most appropriate. The smallest domain gave a good performance increase when increasing the overlap. They argued that this was because the transferred data volume was so small, that the overheads became noticeable. The C2050 system showed a performance increase for all domain resolutions when increasing the overlap. They argued that this was because these GPUs were much faster, resulting in a significantly increase in communication overheads. A global overlap of 32 provided the best performance. The GeForce GTX 480 system showed equivalent behavior to that of the C2050 system, also having 32 as the most optimal global overlap.

Ding and He [8] also implemented the GCE technique to reduce communications in solving PDE problems on cluster systems. They benchmarked it on two systems, the Cray T3E and IBM SP. They reported a good speed up in communication, up to 170%, increasing the overlap.

To mathematically describe the technique, I introduce two formulas

described in [35]. The time taken for a single timestep running a ghost cell exchange every timestep can be written with the following formula.

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

Here, m and n are the domain dimensions, $T(m)$ is the ghost cell transfer time, c_T represents transfer overheads, $C(m,n)$ is the time it takes to compute on the subdomain, and c represents other overheads. Expanding this to using GCE to exchange ghost cells only every kth timestep, the time taken for a timestep is represented by

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

Here, the transfer overheads are divided by k and $\mathcal{O}(k)$ represents the additional time it takes to compute on the larger domain. I have implemented the technique for both the first-order accurate Euler time integrator and the second-order accurate Runge-Kutta time integrator. The overlap is decided by the stencil of the scheme. For the first-order accurate Euler time integrator, the overlap can be increased as a multiple of 4. The second-order accurate Runge-Kutta time integrator requires twice the overlap of first order.

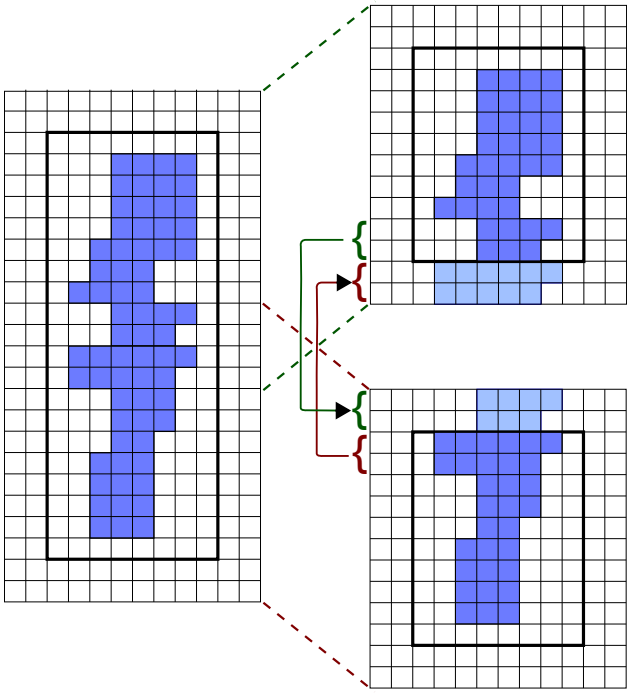


Figure 3.5: A ghost cell exchange with no additional GCE overlap. Here, showing two 8×10 subdomains. The light shaded parts are the overlapping region that connects each subdomain in the ghost cell exchange. No additional row of cells need to be defined and the ghost cell exchange is performed each timestep.

3.4.1 Implementation

To easily vary the level of overlap, I implemented a runtime argument to input the overlap. This is inputted as a global overlap. An overlap of 4 would give each subdomain an overlap of 2 ghost cell rows, representing no additional overlap. This is illustrated in figure 3.5. An overlap of 8 gives twice the overlap for each subdomain, illustrated in figure 3.6, and results in two timesteps per ghost cell exchange. Each additional overlap allows one more timestep before doing a ghost cell exchange. Also, for each new overlap, extra rows are required for all subdomains. The extra rows of cells is added when initializing the subdomains, and is determined by $overlapPerDomain - 2$. For example, using a global overlap of 8 would yield an overlap of 4 per subdomain interface. This gives two additional rows to add for each subdomain interface. This means that, the bottom subdomain add the extra rows at the upper part, while the upper subdomain add the rows at the bottom of its domain. All middle subdomains need to add the extra rows at both ends.

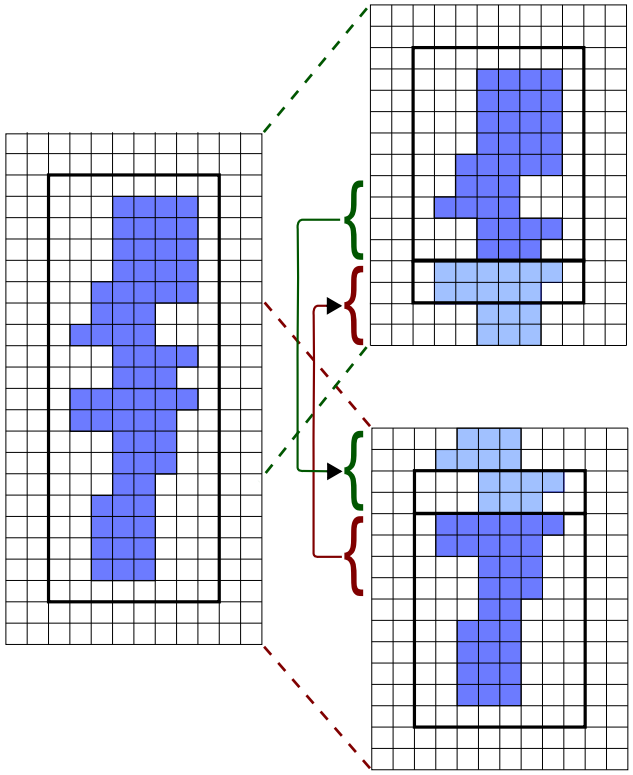


Figure 3.6: A ghost cell exchange with a global GCE overlap of 8. Here showing two 8×10 subdomains. The light shaded parts are the overlapping region that connects each subdomain in the ghost cell exchange. As seen, two additional row of cells needs to be defined for this overlap at each subdomain interface. The ghost cell exchange is here performed only each second timestep.

3.5 Early exit optimization

I have also implemented an optimization technique called *early exit*. The technique scans the domain by computing a *dry map* and use this to skip flux computations for parts of the domain that does not contain water. This has the potential of giving much better performance, especially when simulating real events such as flood events which contains a lot of dry areas throughout the simulation. The downside is the increased overhead related to computing the dry map. I first explain how I implemented this for a single domain, and then the extensions needed for the multi-GPU implementation.

3.5.1 Implementation

The technique is implemented on a per block basis. This means that the algorithm is applied on each CUDA block in the domain, computing whether it contains any wet cells, i.e., cells with water. One also needs to check the neighboring blocks, because of the local ghost cell overlap used between the blocks (see figure 2.5). A block is marked wet if the block contains at least one wet cell. If there are no wet cells it is marked dry. If the current block and all its neighboring blocks are marked dry, the flux computations can be safely skipped for this block.

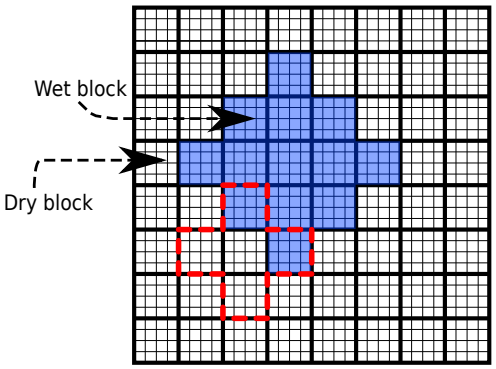


Figure 3.7: Illustrates the early exit technique on a 32×32 domain with 8×8 blocks, each with a resolution of 4×4 cells. Wet cells are marked in blue. The red dotted stencil shows the early exit check for the current block (middle) and its neighboring blocks. Here, early exit is not performed because some of the blocks are marked as wet.

I first implement this for a single domain simulation. For this, there are two parts of the code that need to be modified: the *flux kernel* and the *time integration kernel*, described in section 2.3. Both of these kernels are divided into blocks that run independently. The time integration kernel computes a dry map, used in the flux kernel to check whether to perform early exit. Since each thread represents a cell in the simulator domain, this can be done by performing a reduction on all threads. Each thread compute if its cell contains water or not, i.e., it contains a 0 (dry) or 1 (wet). A reduction to compute the maximum value is then performed. This value

is then inputted to the dry map by all blocks. This value will be 0 if all cells were dry and 1 if at least one cell was wet. To optimize the computations, the reduction makes use of shared memory on the GPU.

The dry map is inputted to the flux kernel where it is used to check if early exit should be performed. I implement this functionality in a new device function. The function scans the dry map, extracting the marked value of current block and its neighboring blocks. If all the values are 0, the function can return true to immediately return from the flux kernel, performing early exit. If at least one of the values are 1, the flux kernel need to continue its computations, since this means that at least one cell in at least one of the blocks was marked as wet.

To enable and disable use of the early exit optimization, I have implemented a runtime argument. It is implemented in such a way that when set to false, the early exit dry map is not computed and the flux kernel will not check whether to run early exit or not.

I demonstrate the technique in listing 3.3 by showing the early exit test in the flux kernel.

Listing 3.3: Overview of early exit in flux kernel

```
/*
 * Function checks if early exit should be performed
 * based on the dry map
 * bx,by = block ID
 */
template <bool early_exit>
__device__ bool early_exit_test(int bx, int by)
{
    // Check if early exit is disabled
    if(!early_exit) return false;

    // Get the value of the current block
    current = getValue(D, bx, by);

    /*
    * Get the value of all neighboring blocks
    * that exists from the dry map (D)
    */
    if(bx+1) right = getValue(D, bx+1, by);
    if(bx-1) left = getValue(D, bx-1, by);
    if(by+1) bottom = getValue(D, bx, by+1);
    if(by-1) top = getValue(D, bx, by-1);

    // Check each block value of the dry map
    if(current == 0 && right == 0 && left == 0
                && top == 0 && bottom == 0)
        return true; // Do early exit if dry
    else
```

```
        return false; // No early exit
}

/*
* Flux kernel running early exit
* and computing the fluxes
*/
__global__ void FGHKernel()
{
    // Get current block id
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Perform early exit
    if(early_exit_test<early_exit>(bx, by)) return;

    // Continue kernel here
}
```

3.5.2 Multi-domain extension

The current implementation of early exit works well when simulating only one domain. For my multi-GPU implementation, it will however not work. The reason for this is that the ghost cell exchange function executing at the beginning of each timestep modifies the domain buffers that the dry map was computed from. Specifically, the ghost cell exchanges propagate the water between the subdomains. This means that the computed dry map is based on data that no longer exist. For example, if a given cell was dry in the time integration kernel, it might now be defined as wet since the water might have propagated over from the other subdomains. The dry map can therefore not be safely used in the flux kernel. There are several strategies to solve this problem. A simple method is to always compute on the boundary cells. This might give a slighter less efficient implementation since the border blocks never perform early exit. Another strategy is therefore to make a dedicated early exit kernel. Here, I put the reduction algorithm from the time integration kernel. The computation of the dry map is therefore done in a separate kernel call. I run the early exit kernel before the flux kernel and after the ghost cell exchange call. This will ensure that the dry map for each subdomain is based on the newest updated domain buffers and can be safely used in the flux kernel.

3.6 Results

To validate the performance of the multi-GPU cluster simulator, several benchmarks have been performed. The plots for all benchmarks are produced using scripts made in the Python programming language [32]. The scripts run the necessary simulations which output the result data to

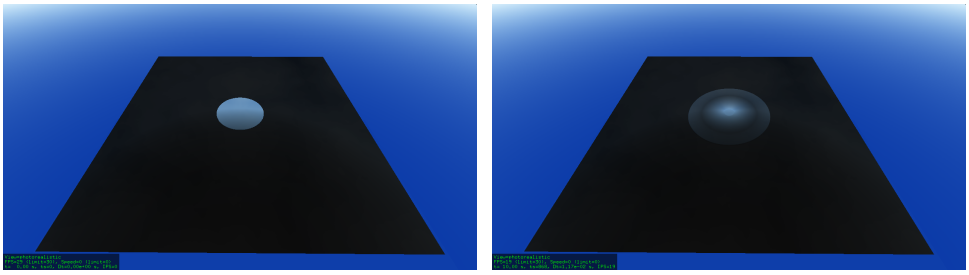


Figure 3.8: An idealised circular dam break in a square computational domain. This case describes a circular water column surrounded by a wall on a flat bathymetry. At time $t = 0$ (left), the dam instantly collapses, creating an outgoing circular wave, seen at $t = 10$ (right).

a file. The data is then read from the file and plotted using the library *matplotlib* [23].

Most benchmarks have been run on a system composed of a 2.67 GHz Intel Core i7 920 CPU, 6 GB system memory and two GeForce GTX 480 graphics cards with 1.5 GB memory each. When other systems are used, it is specified closer for each benchmark. To benchmark my implementation, a synthetic idealised circular dam break case have been used [36] as shown in figure 3.8. This case describes a circular water column placed in the middle on a flat bathymetry. I use this case for all benchmarks.

I first demonstrate the performance of the implemented multi-GPU and multi-node code in section 3.6.1. These benchmarks run multiple domains of varying resolutions (number of cells) on multi-GPU and multi-node systems. This is compared using one graphics card to verify the speed up. The benchmarks showed great scaling increasing the domain resolution on multi-GPU and cluster systems. Secondly, a benchmark of the *Ghost Cell Expansion* technique is shown in section 3.6.2. It is first demonstrated on a single-node system using two graphics cards. Then, it is shown on a cluster of two nodes. As the benchmarks showed, GCE had a negative performance impact, and in most cases produced linear decrease in performance increasing the overlap.

3.6.1 Performance of the multi-GPU and node implementations

Here, I demonstrate the performance of the implemented multi-domain code for both systems composed of multiple graphics cards and multiple nodes. The benchmarks are run on a domain initialized as a flat circular dam surrounded by a wall with radius $R = 200m$ in a square computational domain of size $2000m \times 2000m$ with center at $x_c = 1000m$, $y_c = 1000m$. The following is set as initial conditions: The bathymetry B is set to $B(x, y, 0) = 0$. The water momentum along the x-axis and y-axis, Q_2 and Q_3 is set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$. The water elevation Q_1 is set to:

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

The initial conditions were also set up to use wall boundary conditions with early exit disabled. At $t = 0$, the dam instantly collapses, creating an outgoing circular wave that propagates until $t = 60$. More detailed initialization parameters are described specifically for each benchmark.

Performance of the multi-GPU implementation

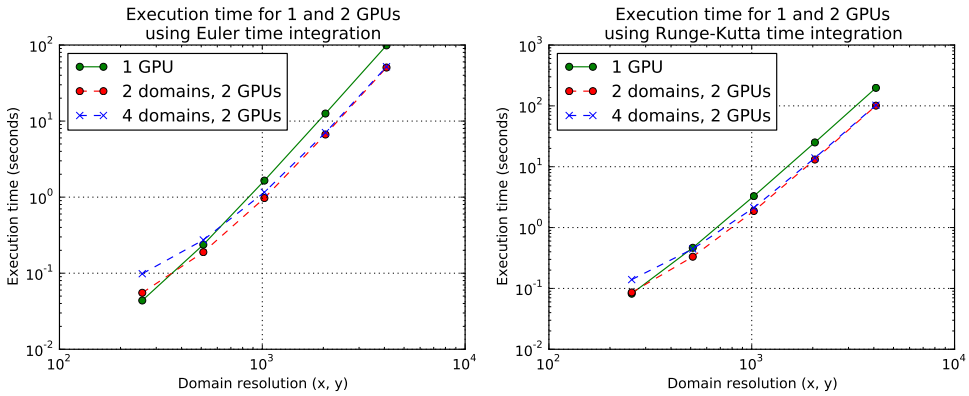
The first benchmark I run demonstrate the performance of my multi-GPU implementation versus the single-GPU implementation of the simulator.

The benchmark run simulations with a square domain with resolutions of $(2^n \times 2^n)$, where $n = 8 - 12$. The first simulation uses resolution 256×256 while the last simulation uses 4096×4096. For each simulation, the execution time is measured. These simulations are first run on a single GPU. Then, they are run using 2 GPUs. For the last case, each domain is first decomposed into two subdomains, one for each GPU. Then, the multi-GPU simulations are rerun, decomposing the domain into four subdomains so that each graphics card computes on two subdomains. For both cases, the domains are decomposed equally so that both cards receive an equal amount of workload.

The benchmark measures how long it takes to run a given domain on a single GPU versus using 2 GPUs. This is useful to see if the code is implemented correctly and also to verify that the simulation scales well and are able to take full advantage of all the graphics cards. Also, decomposing into four subdomains shows how well the multi-GPU setup performs when each graphics card computes on more than one domain. The benchmark is also executed two times, first with the first-order accurate Euler time integrator, then with the second-order accurate Runge-Kutta time integrator. I do this to validate that the multi-GPU implementation is implemented and behaves correctly for both time integrators. Both benchmarks are seen in figure 3.9.

First of all, the plots clearly show that the simulations on 2 GPUs runs faster than when using only 1 GPU. Simulations with large domain resolutions run close to twice as fast when using two graphics cards. This confirms that the solution behaves asynchronously, which means that the CPU do not block at specific places in the code. This is important, so that each GPU runs in parallel with the others. Also, both plots demonstrate equivalent behavior. This shows that both time integrators are implemented correctly with no synchronous code

Also, the performance gap between single- and multi-GPU increases as the domain resolution grows larger, i.e., larger resolutions gives better performance than smaller domains when running on two graphics cards. This behavior is expected, because when the domains are small, the first GPU may finish before the second starts to compute on its subdomain. This is because the CPU invokes kernel execution for the subdomains on the graphics cards serially. This means that it first invokes the first GPU,



(a) (b)

Figure 3.9: Performance benchmark running domain resolutions from 256×256 to 4096×4096 on an idealised circular dam break. These are run using 1 and 2 GeForce GTX 480 graphics cards. For the last case, each domain is decomposed into both two and four subdomains. (Left): Using the first-order accurate Euler time integrator. (Right): Using the second-order accurate Runge-Kutta time integrator. As seen, it shows good scaling when increasing the domain resolution on multi-GPU, doubling the performance for the largest resolutions.

and then proceed to the next GPU. As a result, there will be a small time gap between the invocations of GPUs. This can easily be seen in the plots, where the simulations on the smallest domain resolution do not run much faster using multiple graphics cards. For the lowest resolutions, it performs worse. However, this is not important since small domain resolutions are not candidates for multi-GPU simulations.

Finally, running four subdomains also showed the same improvement behavior as running two subdomains, doubling the performance on 2 graphics cards compared to single-GPU for largest domain resolutions. This demonstrates that running several subdomains per GPU scales equally well using large domain resolutions.

I also show the simulation times for both plots. The times for the Euler time integrator is outlined in table 3.1, while the Runge-Kutta time integrator is shown in table 3.2.

Domain resolutions	256^2	512^2	1024^2	2048^2	4096^2
1 GPU	4.4E-2	2.4E-1	1.6E+0	1.3E+1	9.9E+1
2 domains on 2 GPUs	5.5E-2	1.9E-1	9.7E-1	6.7E+0	5.1E+1
Speed up over 1 GPU	0.8	1.3	1.6	1.9	1.9
4 domains on 2 GPUs	9.8E-2	2.7E-1	1.2E+0	7.1E+0	5.2E+1
Speed up over 1 GPU	0.4	0.9	1.3	1.8	1.9

Table 3.1: The execution times (seconds) simulating different domain resolutions on one GPU and decomposed into two and four subdomains on two GPUs. Here, using the first-order Euler time integrator.

Domain resolutions	256^2	512^2	1024^2	2048^2	4096^2
1 GPU	8.2E-2	4.7E-1	3.3E+0	2.5E+1	2.0E+2
2 domains on 2 GPUs	8.6E-2	3.3E-1	1.9E+0	1.3E+1	1.0E+2
Speed up over 1 GPU	1	1.4	1.7	1.9	2
4 domains on 2 GPUs	1.4E-1	4.5E-1	2.1E+0	1.4E+1	1.0E+2
Speed up over 1 GPU	0.6	1	1.6	1.8	2

Table 3.2: The execution times (seconds) simulating different domain resolutions on one GPU and decomposed into two and four subdomains on two GPUs. Here, using the second-order Runge-Kutta time integrator.

Performance of the multi-node implementation

I also validate the implementation of the multi-node simulator. The benchmark run is similar to the above seen in figure 3.9. It is run on a small cluster setup with two computers connected over the same network. Both computers represent commodity-level desktop computers with two GeForce GTX 480 graphics cards.

Furthermore, the benchmark executes five simulations with domain resolutions of $(2^n \times 2^n)$, where $n = 8 - 12$. The first simulation uses resolution 256×256, while the last simulation uses 4096×4096. I here perform the benchmark using only the second-order accurate Runge-Kutta time integrator. First, the multi-node implementation is benchmarked. This is first done running the simulations using one graphics card on each node. Then, the simulations are run using all four GPUs across the two nodes. For this, the global domains are decomposed into one subdomain per graphics card. To compare with the non-MPI implementation, I also perform the same benchmark using only a single node. This is similar to figure 3.9b. This means that I first compute all five domains on a single GPU, then on two GPUs, decomposing the domains into two subdomains. This allows seeing how well the solution scales when using two nodes, compared to utilizing two graphics cards on a single node. In addition, the multi-node benchmark is compared with using only a single GPU. For all cases, each domain is decomposed equally so that all graphics cards receive an equal amount of workload.

As seen in figure 3.9, it was reported that a multi-GPU simulation was twice as fast as a single-GPU simulation for domains of sufficient resolutions. Ideally, for large enough resolutions, the multi-node simulation should also give twice the performance of a single-GPU simulation. Using all four GPUs, it should give four times the performance of the single-GPU benchmark. Utilizing four graphics cards across two nodes should also give twice the performance of utilizing only one card per node. The multi-node results is shown in figure 3.10.

First of all, I compare the multi-node plots with the plot using a single graphics card. As seen, the multi-node implementation runs slower for small domain resolutions. Increasing the domain resolution, it performs significantly faster. This behavior is expected, because of the overheads

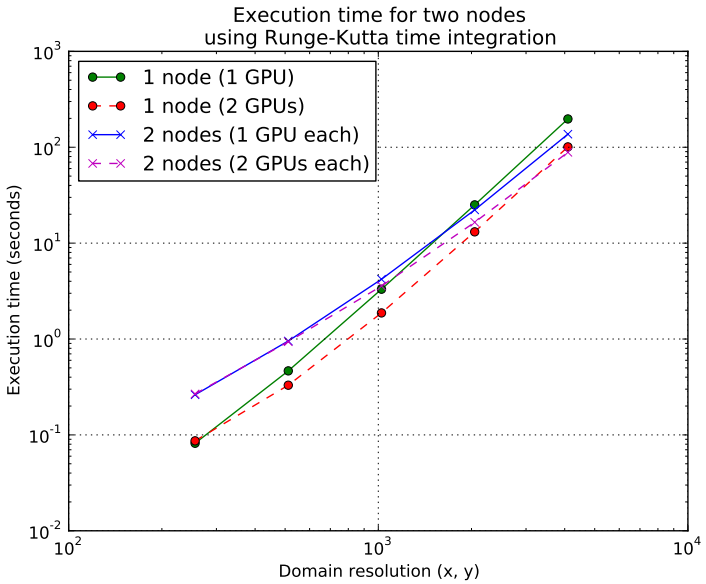


Figure 3.10: Performance benchmark running domain resolutions from 256×256 to 4096×4096 on an idealised circular dam break. These are first run on 2 nodes, utilizing up to four GeForce GTX 480 graphics cards in total, and secondly on a single node utilizing one and two GeForce GTX 480 graphics cards. It is run using the second-order accurate Runge-Kutta time integrator. The cluster benchmark of two GPUs performs worse for small domains. Increasing the resolution, it performs significantly better, showing a performance close to that of two GPUs on a single node. The cluster simulations with four GPUs also scale well, giving the best performance for the largest domain.

related to communication and synchronization of the subdomains when using MPI network transfers on the multi-node simulator. Because of this, computations on domains with low resolution using a single GPU finish before the multi-node simulation. Secondly, the cluster simulations with a total of 2 GPUs executes significantly slower on small domains, comparing against using 2 cards in a single node. However, as the domain resolution increase, the gap between them closes significantly. This shows that using two nodes scales equally well as using 2 GPUs in one node for large domain resolutions. I conclude that this behavior is also because of the overheads related to multi-node synchronization. Increasing the resolution will hide this overhead, causing the performance difference to decrease.

The cluster benchmark with four GPUs shows similar behavior to the one with 2 GPUs. For the largest domain, it runs 1.6 times as fast as using one card per node. This shows that the implementation scales well when increasing the number of GPUs per node. It also runs over twice as fast as the single-node/GPU benchmark for the largest domain. This shows that, for even larger resolutions it should give even better performance over a single-GPU simulation.

When using multiple computers, large domain resolutions are most

applicable to run as they require the high amount of memory found in cluster systems. As shown in my results, multi-node simulations perform sufficiently well on large domain resolutions. Also, to demonstrate the performance of multi-node simulations, I would have liked to run simulations with even larger domain resolutions. However, this was not possible as the graphics cards I had available did not have enough memory to handle these resolutions.

The simulation times for the plot are shown in table 3.3.

Domain resolutions	256^2	512^2	1024^2	2048^2	4096^2
1 node/1 GPU	8.2E-2	4.7E-1	3.3E+0	2.5E+1	2.0E+2
2 nodes/1 GPU each	2.6E-1	9.5E-1	4.2E+0	2.2E+1	1.4E+2
Speed up over 1 node/1 GPU	0.3	0.5	0.8	1.1	1.4
2 nodes/2 GPUs each	2.7E-1	9.4E-1	3.6E+0	1.6E+1	8.9E+1
Speed up over 2 nodes/1 GPU	1	1	1.2	1.4	1.6

Table 3.3: The execution times (seconds) simulating different domain resolutions on one and two nodes utilizing varying number of GeForce GTX 480 graphics cards. Here, using the second-order Runge-Kutta time integrator.

3.6.2 Performance of Ghost Cell Expansion

Here, benchmarks for the ghost cell expansion technique are shown. The benchmarks do performance measurements for both single- and multi-node systems. They measure the simulations execution time when increasing the overlap transferred in the ghost cell exchange function. All GCE benchmarks are run on a domain initialized as a flat circular dam surrounded by a wall with radius $R = 200m$ in a square computational domain of size $2000m \times 2000m$ with center at $x_c = 1000m$, $y_c = 1000m$. The following is set as initial conditions: The bathymetry B is set to $B(x, y, 0) = 0$. The water momentum along the x-axis and y-axis, Q_2 and Q_3 is set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$. The water elevation Q_1 is set to:

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

The benchmarks are also set up with wall boundary conditions and early exit disabled. At $t = 0$, the dam instantly collapses, creating an outgoing circular wave that propagates until $t = 30$. Furthermore, all benchmarks were performed using a varying number of domain resolutions, both 1024×1024 and 4096×4096 resolutions. This is specified closer for each benchmark. These domains are decomposed into two equal subdomains set up on two different graphics cards. I also use the second-order accurate Runge-Kutta time integrator. The benchmarks are executed with 50 different global overlaps, from $8 - 400$ respectively.

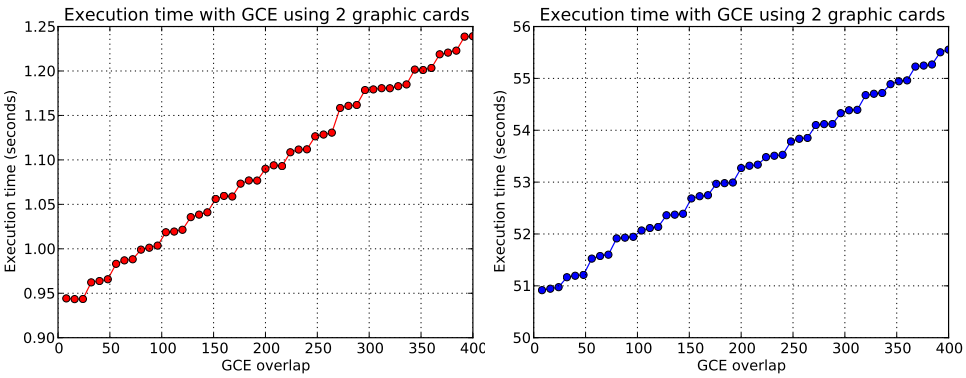


Figure 3.11: Performance benchmark of the ghost cell expansion technique on a single-node system with two graphics cards. Here, it is demonstrated using a domain resolution of 1024×1024 (left) and a domain resolution of 4096×4096 (right) on an idealised circular dam break. As seen, increasing the overlap gives a linear increase in the execution time. Also, notice the performance decreases each fourth overlap.

Single-node (multi-GPU) GCE benchmark

I first demonstrate the ghost cell expansion benchmark on a single-node system composed of two GeForce GTX 480 graphics cards. Increasing the GCE overlap should lower the overhead related to the data transfers, and therefore give a performance improvement. [35]. As mentioned earlier in section 3.4, Sætra and Brodtkorb demonstrated a performance increase when increasing the overlap.

My results can be seen in figure 3.11. Here, the left plot shows a domain resolution of 1024×1024, while the right plot shows a domain resolution of 4096×4096. The benchmark gives a linear decrease in performance when increasing the overlap. Both resolutions show the same behavior. This might be because the ghost cell exchange function is sufficiently fast that increasing the overlap does not matter at all. As a result, when increasing the overlap, and thus the size of the data transferred, it will run slower even though it runs larger and fewer data transfers between the graphics cards. I expect the reason that it is much faster than before, to be because of new CUDA driver updates, enabling new features of existing compute capabilities. For example, NVIDIA has introduced *GPUDirect* [29] for cards with a compute capability of 2.0 or above. This technology enables peer-to-peer transfers between graphics cards on the same system, in addition to optimizing memory accesses between graphics cards. This results in much faster memory copies. Also, as seen in the plots, each fourth overlap gives an extra increase in the execution time. This is likely related to the ghost cell exchange transfers or the kernel computations, and might be possible to optimize away.

Thus, as the results demonstrate, I conclude that using no GCE overlap at all (1 timestep per ghost cell exchange) is the most effective solution.

GCE benchmark with usleep

To validate the results above concerning the performance of the ghost cell exchange function, another benchmark was run. It was run with the same parameters as the above benchmark on the same system. However, in the ghost cell exchange function, a call to the function *usleep* was performed. This function delays the calling thread for a specified amount of time. This was done to make sure a ghost cell transfer used at least the time specified by the parameter to *usleep* to complete. By doing this, a constant delay was added to each exchange to simulate slower data transfers. Since a ghost cell exchange now runs significantly slower, this should give a performance increase when increasing the overlap. On this benchmark, only a domain resolution of 1024×1024 cells is run. This domain is decomposed equally between the two graphics cards.

The result can be seen in figure 3.12. The benchmark is executed with three different parameters for the *usleep* function: *1* (figure 3.12a), *10* (figure 3.12b) and *100* (figure 3.12c) milliseconds respectively. As seen, the performance improves when increasing the number of overlaps, similar to that reported by Brodtkorb and Sætra in [35]. The improvement is most noticeable for the 100 ms delay. Here, for the smallest overlap possible, the simulator has a runtime of about 27 seconds. Increasing the overlap, it constantly improves until it eventually flattens out. A visible effect is also noted when running with a delay of 10 ms. However, the performance given by the smallest and largest overlaps are only separated by a few seconds. Finally, performing data transfers with the smallest delay, one also gains a small performance improvement at the start. However, this is negligible compared to the others. Increasing the overlap above 50 gives a linear decrease in performance, similar to figure 3.11.

First of all, the results indicate that the ghost cell exchange needs to use at least 1 millisecond for the application to get any noticeable performance improvement at all. These results validates that my implementation of the ghost cell exchange function is sufficiently fast and that utilizing the ghost cell expansion technique does not give any additional performance improvement.

Multi-node GCE benchmark

Finally, I also run a ghost cell expansion benchmark on a small cluster composed of two nodes connected over the same network. Both nodes represent commodity-level desktop computers, both with a GeForce GTX 480 graphics card. This benchmark is similar to the benchmark seen in figure 3.11, except that it performs data transfers between graphics cards located at different nodes. I benchmark both the total execution time and the total execution time for all ghost cell exchanges for each simulation. This makes it easy to compare the performance change in different parts of the implementation when increasing the overlap.

The benchmark is run with the multi-node simulator implementation which uses MPI [24] to exchange ghost cells. I reason that the larger

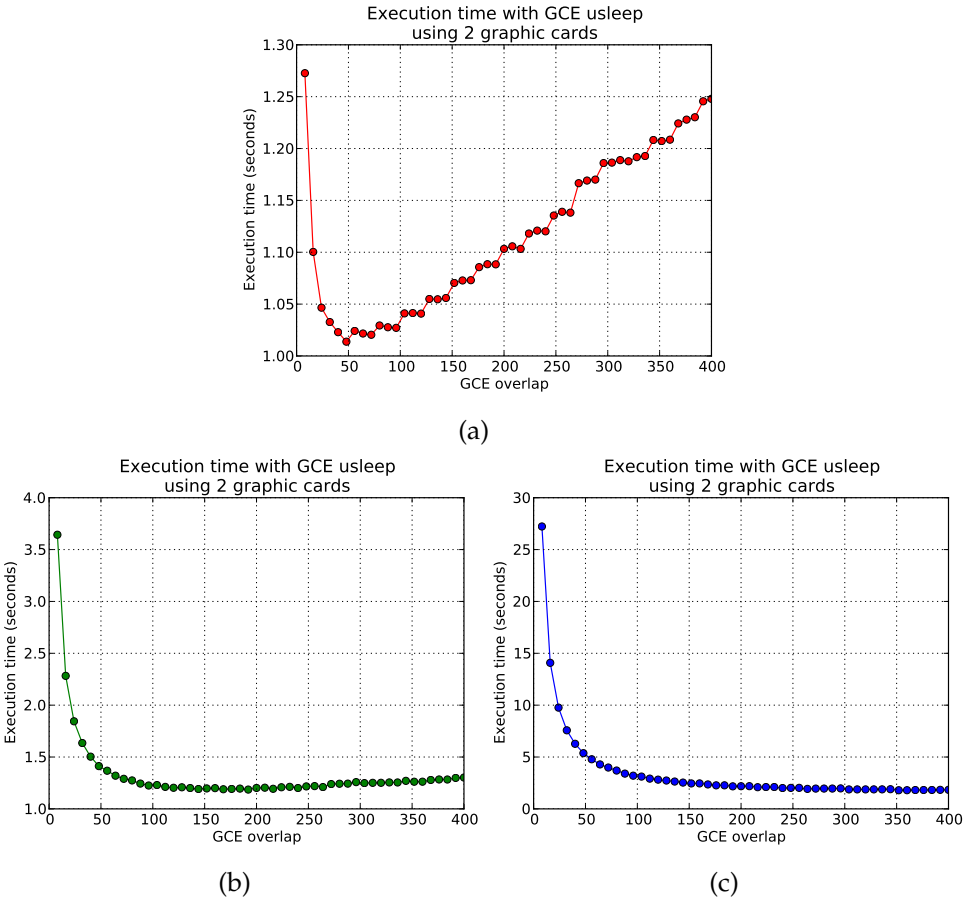


Figure 3.12: Performance benchmark of the ghost cell expansion technique on a single-node system with two graphics cards. It is run using a domain resolution of 1024×1024 on an idealised circular dam break. Here, a constant delay is added by using *usleep* (1 ms (top), 10 ms (left), and 100 ms (right)) to simulate slower ghost cell exchanges. As seen, for the largest delays, increasing the overlap gives a good performance improvement.

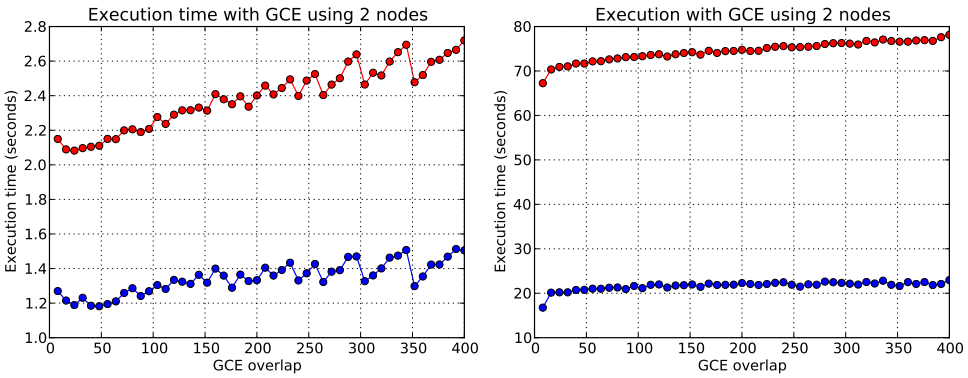


Figure 3.13: Performance benchmark of the ghost cell expansion technique on a cluster of two nodes. Here, it is demonstrated using a domain resolution of 1024×1024 (left) and a domain resolution of 4096×4096 (right) on an idealised circular dam break. Also, the top graph in both plots shows the total execution time of each simulation, while the bottom graph shows the total execution time for all ghost cell exchanges for each simulation. As seen, increasing the overlap gives an increase in both the total simulation execution time and the execution time for the ghost cell exchange.

overheads related to communicating over a network should give a performance increase with the ghost cell expansion technique. This view is also supported by Brodtkorb and Sætra in [35], as they expect GCE to have a greater effect when performing ghost cell exchange across multiple nodes.

I show my results in figure 3.13. The left plot shows a domain resolution of 1024×1024, while the right plot shows a domain resolution of 4096×4096. In both plots, the top graph shows the total simulation execution time and the bottom graph shows the total time execution of all ghost cell exchanges for each simulation. As seen, increasing the overlap does not give a performance improvement for cluster transfers either. Both domains show equivalent behavior, giving a linear decrease of simulation performance. The domain with the lowest resolution shows a negligible performance improvement at the start, before the performance decrease linearly. The benchmark for the largest domain also gives a small jump at the beginning when increasing the overlap from 8 to 16.

Also, the graph showing the total execution time of all ghost cell exchanges gives a similar result as the total execution time for both domains, showing a decrease in performance. From these results, I conclude that for each increase in overlap, an increase of execution time is added for the ghost cell transfer. The results also show a smaller increase of the execution time for ghost cell transfers compared to the total execution time of the simulation. This indicates that there also are other overheads related too increasing the overlap. I expect this to be related to the increase of the computational time caused by increasing the overlap, and thus the domain resolution. Also, for the plot of the 4096×4096 domain, the

ghost cell transfers show a constant behavior with minimal performance decrease. This means that, for large resolutions, other overheads have the largest impact of the performance decrease when increasing the overlap. This might be related to the increase in computational time.

As a final note, it would also be interesting to run both benchmarks using nodes connected across different networks to see if this would have any significant effect on the performance.

Chapter 4

Auto-tuning

In the previous chapter, I implemented a multi-GPU and multi-node shallow water simulator with a static load-balance of the workload. However, to perform load-balancing of the workload between the graphics cards, dynamic auto-tuning techniques can be utilized. Implementing these techniques can be performed in two main steps. First of all, the auto-tuning techniques themselves need to be implemented. For this, one needs to identify parameters for auto-tuning such that load-balancing is obtained. Secondly, to change the workload, and thus the size of each subdomain, a dynamic domain decomposition technique has to be implemented.

First, in section 4.1, I give an introduction to dynamic auto-tuning. Here, I discuss potential parameters to auto-tune on, i.e., early exit, domain decomposition, and GCE. I also discuss several challenges related to the auto-tuning algorithms. Then, I proceed to explaining the auto-tuning of early exit (see section 3.5) in section 4.2. Since auto-tuning relies on dynamic decomposition, I first give a description of this in section 4.3. I explain how I implemented both the multi-GPU and multi-node version of dynamic domain decomposition. I also provide a benchmark of the dynamic domain decomposition on multi-node. Then, I outline the implemented auto-tuning techniques in section 4.4. Finally, I present extensive performance benchmarks of the auto-tuned multi-GPU cluster simulator in section 4.5.

4.1 Dynamic auto-tuning

I first discuss the two parameters I auto-tune on in this thesis. Dynamic auto-tuning techniques are executed at runtime to change properties of the simulation. For example, the computational domain could be decomposed as the simulation progresses so that the workload is load-balanced between the devices. This could be accomplished by auto-tuning on *domain decomposition*. Such auto-tuning will distribute suitable sized parts of the domain to all GPUs according to their computational power. Another parameter to auto-tune is *early exit*. Running early exit for the whole simulation would give extra overhead when most of the domain is covered with water. As a result, one wants to enable early exit when most of the

domain is dry and disable early exit when most of the domain is wet. Here, it would not be necessary to perform an early exit check. This should therefore be auto-tuned so that the simulator automatically chooses to enable or disable early exit.

A third parameter to auto-tune on would be the *GCE*[1] overlap (see section 3.4). However, as my results in section 3.6.2 showed, the best performance was always obtained when running with the smallest possible overlap. Therefore, auto-tuning on this variable is not necessary.

Finally, there are several challenges related to the implementation of auto-tuning dynamic domain decomposition techniques. First of all, because the auto-tuning executes at runtime, it needs to be sufficiently fast. This especially applies to the synchronization and communication when decomposing the domain. It is necessary to design algorithms that can efficiently decompose the domain without too large costs related to data transfers, for example between nodes in a cluster system. It is also important to consider the interval the auto-tuning should be run on, meaning how often it should be run. The water should have propagated far enough from the last auto-tune run to get an appropriate change in workload, justifying the overhead of auto-tuning. This again depends on the Δt. If run too often, it might cause large overheads, resulting in negligible performance improvements for auto-tuning. Also, the change in workload would be less noticeable. A better solution would be to let the water propagate further before auto-tuning. The auto-tune interval will need to be experimented with to find a proper value. Also, the variable could potentially be auto-tuned. However, this is outside the scope of this thesis.

4.2 Auto-tuning of early exit

I auto-tune the early exit algorithm, so that the application executes the fastest possible kernel at runtime. This means that I decide whether the application should run the early exit kernel (see section 3.5) and the flux kernel (see section 2.3) with or without the early exit optimization at runtime. The motivation for this is that it is not necessary to run the early exit optimization at each timestep, especially because it gives extra overhead related to running the early exit kernel. A domain where most cells contain water will likely cause the flux kernel to perform computations for most of the cells while still running the early exit kernel. This is strictly not necessary.

I explain two methods to solve this. First of all, I could measure the time taken to run the early exit and flux kernel at each timestep. Then, I could run the fastest of these at the next timestep. This method would be able to always run the fastest possible kernel at each timestep without extra

[1]The benchmarks in this chapter uses domains with bathymetry values defining a realistic terrain. However, a bug related to GCE sometimes caused an incorrect timestep Δt. It was likely related to the decomposition of the bathymetry between the subdomains. However, since I do not use GCE for these benchmarks, the bug does not affect my results.

overhead. This method works well when running a single subdomain. However, for multiple subdomains, it is not appropriate. The reason for this is that the multi-domain implementation runs asynchronously. Therefore, it is hard to measure the time correctly for each subdomain. I therefore employ another method. I implement a probe that I run at given intervals, for example each hundredth timestep. This probe executes a full step (see figure 2.6) with the necessary kernels at the end of each timestep, first with the early exit optimization disabled, then enabled. I measure the time taken for each run and choose the fastest. Choosing to disable early exit, the early exit kernel will not be run. Enabling it will make the simulator call the early exit kernel computing the dry map, in addition to performing the early exit test in the flux kernel. It is also worth noting that this probe requires the simulator to perform two extra steps at the given interval, therefore producing extra overhead. However, the method allows me to create synthetic steps that I can synchronize on and therefore correctly measure the time taken for the kernel calls.

I use an application argument to enable auto-tuning of early exit and the interval it should be run on. More precisely, it contains the number of timesteps to perform before running the probe. This gives complete flexibility to decide how often the probe should be run. To get the best possible performance it is important to consider the value for this argument. Running the probe too often might cause large overheads related to the extra kernel calls. If run at too few intervals however, it might cause the simulator to run with a non-optimal kernel. For example, it might run with early exit even when the water has propagated through the majority of the domain.

4.3 Dynamic domain decomposition

I have implemented the ability to dynamically decompose the domain. This performs *domain decomposition* (see section 3.1) dynamically so that the size of each subdomain can be changed while the simulator is running.

The technique adjusts the subdomain sizes dynamically according to a new workload computed at runtime. The new workload could for example depend on the current amount of wet and dry cells, i.e., the placement of water in the global domain. There are several advantages with this technique. First of all, consider a case with a global domain decomposed into two subdomains A and B, each residing on a different graphics cards and early exit enabled. It would be an advantage that both cards computed about the same amount of wet cells. If there is much water residing on the top half of the global domain and none on the other half (see figure 4.5), it should not be decomposed at the center. This is because it would result in an inadequate workload distribution, causing the graphics card of subdomain A to do all the processing, while the other GPU would not have anything to calculate on. This would in turn lead to an ineffective multi-GPU solution. The ideal solution to this would be to decompose the domain so that both subdomains contain an equally amount of wet

cells. Secondly, one should also perform dynamic decomposition to accommodate for graphics cards with different computational power, for example high-end and low-end graphics cards. This could be done in a fashion so that high-end graphics cards always compute on the domain with the largest amount of wet cells.

4.3.1 Dynamic domain decomposition on multi-GPU

First, I have implemented this for a single node utilizing multiple graphics cards. The dynamic domain decomposition changes the size of each subdomain according to a new *workload*. It performs three main steps: **1:** It downloads the water elevation Q_1 and water momentum Q_2 and Q_3 of all subdomains to a global buffer on the CPU. **2:** It composes and initializes the new subdomains according to the new workload, and **3:** It re-initializes the ghost cells by performing a ghost cell exchange. Figure 4.1 defines two equally sized subdomains and gives an illustration of the steps decomposing to new subdomain sizes.

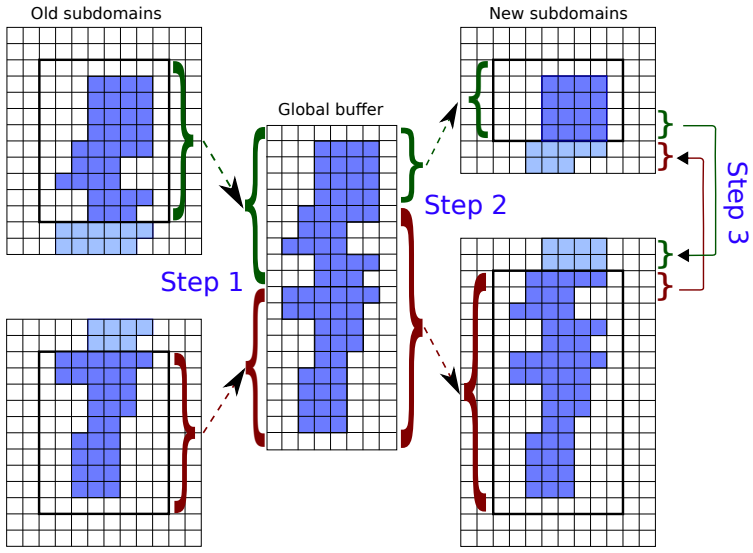


Figure 4.1: A dynamic domain decomposition performed between two GPUs on a single node, here shown with a subdomain per GPU. **Step 1:** First, the GPU subdomains are copied to a global buffer residing on the CPU. This means that the water elevation Q_1 and the water momentum Q_2 and Q_3 buffers are transferred into three global buffers. **Step 2:** Then, the new subdomains are initialized with the new workload. This is done by transferring the data in the global buffers back to the graphics card memory. **Step 3:** Finally, the ghost cells are re-initialized by doing a ghost cell exchange.

Step 1: The first step is to download the water elevation Q_1 and water momentum Q_2 and Q_3 of all subdomains to a single global buffer residing on the CPU. Only the interior cells are downloaded, as the ghost cells can be reconstructed later using the values of the inner cells.

Step 2: Next, I compose the new subdomain sizes according to the new workload. This is done as a standard domain decomposition (see section 3.2). More precisely, I calculate the correct size and offset of each subdomain. I also deallocate the old subdomains, freeing up data on the graphics card memory. Finally, the new subdomains are initialized according to the new size. The ghost cells are also reconstructed by using the inner values of the domain.

Step 3: At this point, the additional GCE ghost cells are re-initialized. This is done by performing a ghost cell exchange (see section 3.2), which ensures that the ghost cells are correctly reconstructed at the end of each dynamic decompose.

4.3.2 Dynamic domain decomposition on multi-node

Furthermore, I implement dynamic domain decomposition on the multi-node simulator. This is similar to the above method in that it performs a dynamic decomposition according to a new workload. It is important to take into consideration that there are several processes operating in parallel on multiple nodes. This required several changes from the multi-GPU version.

Naive version

First of all, I tried to implement it in a similar manner as the original version. This proved to be easy, as I for most parts followed the original design with some small modifications. The domain decomposition is performed and run per process and the processes synchronize as necessary during execution. Because there are now several processes, the allocation of the global buffer must be done by sending the subdomain data between the processes. The task of initializing the global buffer is allocated to a master process which receives the necessary subdomain data from the other processes.

Step 1: All processes first transfers the water elevation Q_1 and water momentum Q_2 and Q_3 of all subdomains to a per process buffer on the CPU. Next, the processes synchronize this. This is done sending the buffers to process 0, designated as the master process. Process 0 receives this and inserts it to the global buffer. It also transfers its own subdomains to this buffer. Then, process 0 broadcasts the global buffer back to the other processes. The network transfers utilize MPI function calls.

Step 2: Next, each process initializes its new subdomain sizes according to the new workload, by calculating the correct offset and size. The old subdomain objects are also deallocated. Then, each process initialize its new subdomains on the correct graphics card. The ghost cells are reconstructed based on the inner cells.

Step 3: Finally, the additional ghost cell overlap is re-initialized by performing a ghost cell exchange using swap_mpi (see section 3.3).

This method was straightforward to implement. The main problem was its long execution time, especially for large domains. I identified

the bottleneck to be the network transfers between the nodes. Because each process needs a copy of the global buffers, they had to synchronize domain data between them as described in *step 1*. This communication gave very large data transfers over the network and proved to be very slow. As a result, each dynamic decompose of multi-node gives a large halt of the execution. Therefore, I reason that the execution time of performing a dynamic decomposition on multi-node might be too large. I therefore designed and implemented a better solution, described next.

Optimized version

To solve the long execution times for the naive version I re-implemented the function using another technique. The main focus should be to minimize the amount and size of the data transfers between the processes. Conceptually, a dynamic decompose can be thought of as an extended version of the ghost cell exchange between the subdomains. Consider a case where a global domain is decomposed into two equal sized subdomains A and B. Then, a workload change occurs such that A gets 70%, while B gets 30%. This implies that a swap is performed so that A receives 20% of the global domain from B, and therefore extends itself to handle this. B does not received anything and therefore only deallocates the data it sent to A. By following this design, each process can handle all its subdomains as one single domain. Then, each of these process domains can handle its decomposition individually, sending and receiving only the data necessary with the other processes. This dramatically reduces the amount of data sent. I follow with an in depth explanation of the method.

Step 1: First, each process downloads the water elevation Q_1 and water momentum Q_2 and Q_3 of all its subdomains to a buffer. They also compute the workload for their domain.

Step 2: Next, the processes can perform the domain swapping, illustrated in figure 4.2. They first compute the amounts of rows to swap. This is done according to the new computed workload and the old size. The swapping itself is identical to the implemented ghost cell exchange function *swap_mpi* (see section 3.3), with the exception that it also needs to manipulate the length of the domains. First of all, when a process sends a part of its domain, it needs to deallocate this part from its own domain. Also, when a process receives data, it also needs to allocate new space for this in its domain.

Step 3: Then, each process handles the decomposition of its subdomains, decomposed from the new process domain. This is run completely per process without any synchronization between the processes. It computes the workload for each subdomain and initializes the new subdomains for each GPU.

Step 4: Finally, the ghost cells are also re-initialized. This is done by performing ghost cell exchanges with *swap* and *swap_mpi*.

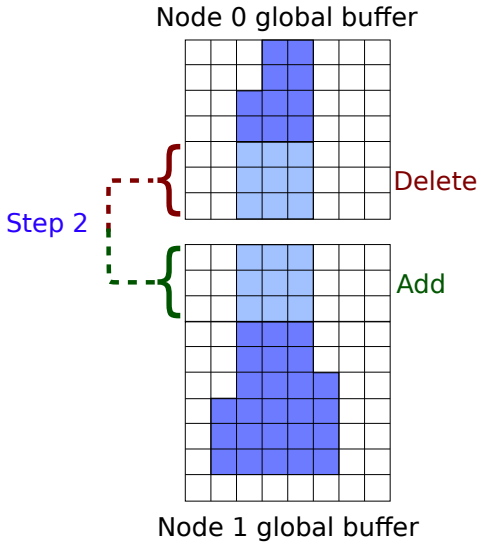


Figure 4.2: A dynamic domain decomposition performed between two nodes. Both domains initially have a resolution of 8×7, then a swap is performed so that the bottom domain get a part of the upper domain. **Step 1** (not shown)**:** This step is similar to step 1 in figure 4.1. This means that each process copy its subdomains to a global buffer. **Step 2:** Then, a swap according to the new workload is performed between the global buffers (seen in this figure). As seen, this swap changes the domain size for both nodes. **Step 3** (not shown)**:** Then, these buffers are copied to the GPU subdomains, done per process. This is similar to step 2 in figure 4.1. **Step 4** (not shown)**:** Finally, the ghost cells are re-initialized by doing a ghost cell exchange, also similar to step 3 in figure 4.1.

Benchmarking multi-node dynamic decomposition

Also, I run a performance benchmark of the MPI transfers to validate the performance of the dynamic domain decomposition on multi-node. This is necessary to check that my MPI implementation for dynamic decomposition is sufficiently fast, especially for large domain resolutions. The benchmark is implemented by sending data with different sizes using MPI [24] and the *SCP* command, and then measuring the time of the transfers. The data sizes used are 2, 4, 10, 100, 200 and 500 MB of data. The SCP transfers are performed by sending a file of the specified size. For this purpose, the file first needs to be created. This is done with the command *dd if=/dev/zero of=<file> bs=1024 count=<kB>*. To make a file size of 10 MB, $1024 * 10$ is sent in to the argument *count*. The file is then sent using the command *scp <file> <user@host:path>*.

The MPI transfers are measured by doing a standard MPI ghost cell exchange. For this purpose, a dedicated function was made. Instead of performing the swap both ways as usual, it is only performed one way identical to the SCP transfers. This means that only one node sends data and only one node receives the data. The function allocates the necessary buffers to send on the CPU with the same size as the SCP transfers. Also,

I do not measure the MPI function calls in the dynamic decomposition. However, since the MPI function calls are identical in the decomposition and ghost cell exchange functions, one can assume the performance to be the same for both of them. For both type of transfers, I measure the time and compute the speed of each transfer. The benchmark is seen in figure 4.3.

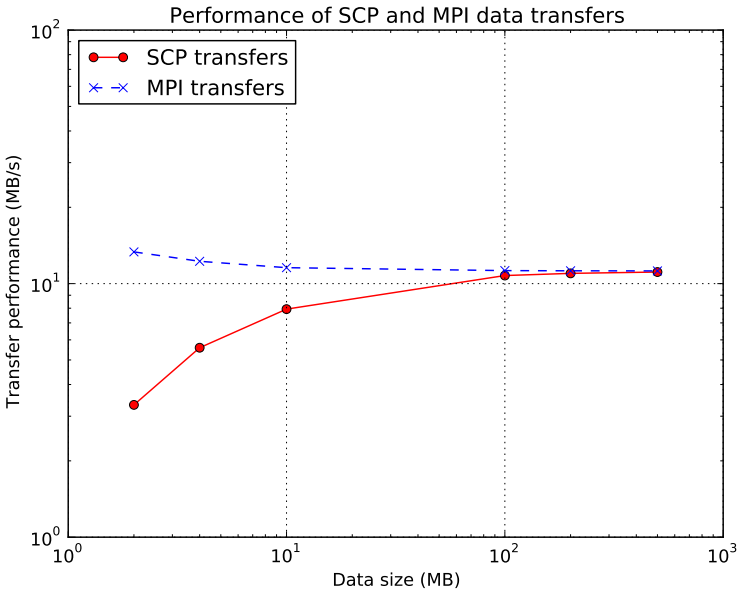


Figure 4.3: Performance of SCP and MPI data transfers using a varying number of data sizes. Both type of transfers are performed in a send/receive transfer from a given node to another node. The MPI transfers shows good performance compared to the SCP transfers.

First, for SCP, data transfer speed is increased when increasing the size of the data sent, reaching peak performance when sending about 100 MB of data. This shows that there is a small overhead related to these data transfers. The MPI transfers show a constant speed for all data sizes. Also, the plot shows MPI gives equal or better performance for all data sizes. This shows that my MPI implementation is sufficiently fast.

4.4 Auto-tuning dynamic domain decomposition

I now proceed to describe the implementation of the auto-tuning of dynamic domain decomposition. This considers underlying parameters of the application to determine the size of each subdomain. Consider a standard dam break simulation run by two subdomains on two graphics cards. Initially, when the dam breaches, the water flows downwards before it eventually covers most of the domain. To efficiently compute the simulation, both graphics cards should compute on the amount of wet cells according to their computational power throughout the simulation. This mean that the workload should be *load-balanced* between

the GPUs. To perform this, dynamic auto-tuning techniques are utilized. These techniques auto-tunes on underlying parameters of the application at runtime to determine the size of each subdomain. The auto-tune techniques are divided in two different algorithms: The *initial auto-tuning* and the *runtime auto-tuning*. Both algorithms are utilized together, each corresponding to a different part of the auto-tuning technique.

First, I describe the *initial auto-tuning* which is executed a single time at the beginning of the simulation. Here, auto-tuning is performed on the graphics card hardware. More precisely, the *computational power* of each graphics card. The auto-tuning algorithm decides the amount of workload according to the computational ability of the graphics cards. This means that each GPU will compute on a domain size that suits its computational ability. Second, I describe the *runtime auto-tuning*. This auto-tuning algorithm executes at specified intervals during runtime and computes the workload according to the water placement in the domain. As the water propagates through the domain, the algorithm resizes the subdomains as necessary such that all graphics cards compute on an appropriate amount of wet cells. The algorithms uses the workload computed by the *initial auto-tuning* to determine the correct amount of wet cells for each card. As a result, all GPUs compute on the amount of wet cells according to their computational power.

It is also important to mention that auto-tuning is run together with the early exit technique. This is very important to achieve the correct load balancing. If early exit was turned off, the graphics cards would calculate on dry cells too, therefore not achieving the desirable load balancing.

4.4.1 Initial auto-tuning

The initial auto-tuning auto-tunes on the computational power of each graphics card to correctly load-balance the workload. This is especially advantageous when utilizing multiple different graphics cards. Consider a case using two GPUs, a low-end and a high-end. The high-end card should compute on a larger amount of wet cells than the low-end card. For example, it can compute on a workload of 70%, while the low-end card can compute 30%. The workload calculated by this form of auto-tuning decides the percent of wet rows each graphics card should compute on for the rest of the simulation.

There are several strategies to perform this type of auto-tuning. An obvious strategy to perform such auto-tuning is to run synthetic cases at the start. These could be run with several different combinations of workload between the subdomains. Since all timesteps runs in approximately the same execution time, it is only necessary to run a single timestep. One could then measure the execution time of each run. The workload of the fastest run would be the workload to use for the main simulation. This method works well when running two subdomains. However, there are several disadvantages with this method. First, the amount of workload combinations to run becomes too large when increasing the number of subdomains. This might consume too much time, making the technique

too costly for practical use. Second, the technique does not necessarily give the most optimal workload-balancing. This is because the cases are run with a predefined amount of workload and because of this one does not test the workloads in-between.

A second strategy to perform such auto-tuning is to only run a single simulation case for each graphics card. This means that a single case is run with equal workload for all graphics cards, as opposed to running several cases with different combinations. One could measure the execution time for each run. Since the execution times represents how fast each GPU is, using this to compute a workload for each GPU is straightforward. This method proves superior to the first method as it gives you the most optimal workload possible. In addition, it only requires running the *step* function (see figure 2.6) a single time for each graphics card, minimizing the time used to perform the auto-tuning. Because of these advantages, I chose to implement this technique.

Implementation

The implementation of this for multi-GPU and multi-node is similar. The synthetic cases are run at the start of the simulation. I run a single synthetic case on each graphics card used. The cases are set up as a single-domain case and are performed by running one single timestep of the step function. The execution time used by each run is measured. Then, the workload is calculated for each GPU by using the execution time for that GPU. This is done according to the formula:

$$\begin{aligned} W &= \frac{ny}{1}(1 - (\frac{GPU_T}{sum_T})\frac{100}{ny} \\ W &= w_r\frac{100}{ny}. \end{aligned} \tag{4.1}$$

Here, W is the resulting workload in percent, the percent of wet rows the given GPU should compute on. GPU_T is the execution time, sum_T is the sum of all execution times, ny is the global domain size along the y-axis, and w_r is the workload in rows as calculated by the first part. The formula gives the workload for each graphics card. When performing this for the multi-node version, each process runs the synthetic cases on its own node. Then, the execution times are synchronized with a master process that performs the work of calculating the workload. The workload is then broadcasted back to all other processes.

4.4.2 Runtime auto-tuning

The runtime auto-tuning computes the workload according to the water placement in the domain. Figure 4.5 shows a simulation with three GPUs that auto-tune and load-balances the workload at runtime. To auto-tune correctly, the algorithm needs to know where the water is at any given timestep. A possible solution would be to compute the middle point of the cell coordinates with water. Given a domain divided up into cells, each

cell has a state of dry or wet, it would be straightforward to compute the middle point. This could be done by finding the average of all wet cell coordinates. The left illustration in figure 4.4 outlines this method. This method is not very flexible. For example, it is hard to utilize it for more than two subdomains without extending the technique. Also, each graphics card is given equal workload with this technique, since they receive the wet cells above and below the middle point.

Another method, is to compute the two dimensional bounding box of the water surface in the domain, seen in the right illustration in figure 4.4. This method is more flexible because it is easier to distribute the water between multiple subdomains. Also, it makes it possible to assign a different amount of wet cells to each subdomain. This can be done by for example assigning 30% of the bounding box to one subdomain, and 70% to another. Most water surfaces can easily be approximated by a bounding box which makes this an acceptable method to compute the workload of wet cells. Because of these advantages, I chose to implement the bounding box technique.

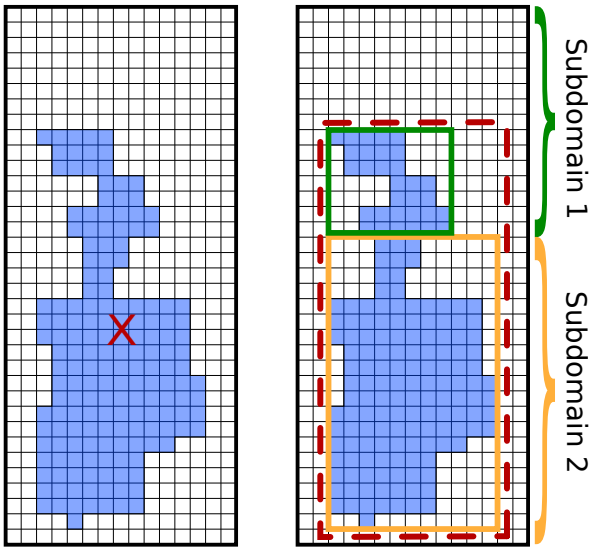


Figure 4.4: Two dynamic auto-tuning techniques. Wet cells are marked in blue. **Left:** Calculating the middle point. The X marks the middle point m, used to distribute the water equally between the subdomains. **Right:** Calculating the bounding box. Here, two subdomains first calculates their local bounding boxes, the green and yellow rectangles respectively. Finally, these are used to calculate the global bounding box (the red dashed rectangle) for the global domain. The global bounding box is then used to distribute the water between the subdomains. The last technique makes it easy to distribute a given percent of the bounding box so that each subdomain might get different amount of wet cells.

Bounding box computation: There are several methods to implement the computation of the bounding box of a domain for. A two dimensional

bounding box consist of four edges and is defined as a structure of four values representing the edges; the maximum and minimum x and y coordinates of the wet cells in the domain. To compute the bounding box, one need to be able to calculate the water depth for all cells, which defines if the cell is wet or dry. This can be done by using the water elevation Q_1 and bathymetry B. A straightforward method would be to transfer this data to a single buffer, and then compute the bounding box for this buffer. To compute the bounding box, the code would find the maximum and minimum coordinate (i, j) (cell index) of each wet cell. The problem with this method, especially for the multi-node version, is that it might be considerably slow. This is because the water elevation Q_1 and bathymetry B of all the subdomains for all nodes would need to be transfered to a single node, therefore doing several large network transfers. However, it is strictly not necessary to transfer the data of all subdomains to a single node. A better method would be that each node downloaded its own Q_1 and bathymetry B data, completely separated from the others. Each node can then computing the bounding boxes for its subdomains. Second, the processes could proceed to synchronize their bounding boxes, ending up with the final global bounding box for the global domain. This way, the nodes would only need to send and receive the four values defining a bounding box, providing a much faster method since it avoids large data transfers.

A third technique would be to calculate the bounding boxes using a reduction kernel on the GPU instead of downloading and doing the computations on the CPU. This technique would be faster for both single- and multi-node simulators because transferring the water elevation Q_1 and bathymetry B to the system memory is not necessary at all. Each GPU runs a kernel for its subdomains, doing a reduction to compute the maximum and minimum values defining the local bounding box of the subdomain. Finally, the bounding boxes are synchronized to compute the global bounding box for the global domain. This means that only the variables defining the bounding box is downloaded to system memory. The technique is illustrated in the right illustration in figure 4.4. Here, the bounding boxes for two subdomains are calculated, and then merged into a global bounding box for the global domain. Because of the obvious advantages, I chose to implement this method.

Implementation: The implementation of runtime auto-tuning is similar for both the multi-GPU and multi-node simulators. The auto-tuning is performed at a given interval, which means that it is performed at specified timesteps throughout the simulation. The interval is specified by setting the value of a runtime argument. The multi-GPU and multi-node simulators are implemented similarly.

Step 1: The first step is to compute the bounding box of each subdomain. This is done using a reduction kernel. The kernels primary purpose is to compute the bounding box for the subdomain it performs on. The kernel employs a reduction algorithm, computing the maximum and minimum coordinate values for each wet cell in the subdomain. To speed

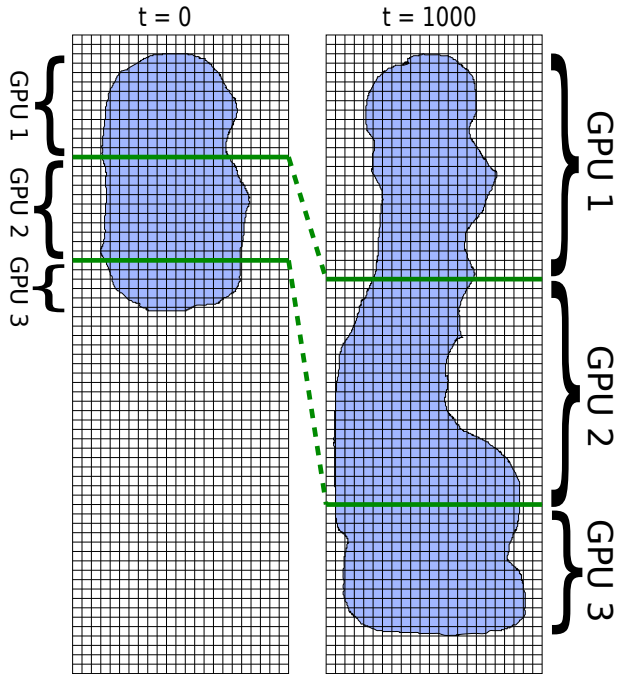


Figure 4.5: An example of auto-tuning dynamic domain decomposition on a dam break simulation running on 3 GPUs. The workload is load-balanced throughout the simulation run. Wet cells are marked in blue and the green lines shows the domain decomposition between the GPUs. The two first GPUs have a workload of 40% of the water, while the last GPU has a workload of 20%, determined by their computational power. At time $t = 0$, the dam collapses, creating a flood that flows downwards in the domain. Here, GPU 3 has the largest subdomain. However, it calculates on fewer wet cells than the others. At time $t = 1000$, the simulator re-autotunes, performing a new decomposition between the GPUs. Notice that all GPUs still calculate on the correct amount of cells according to the workload. For better performance, the auto-tuning algorithm should execute more frequently than shown in this figure.

up the computations, the algorithm utilizes shared memory (described in section 2.1). First, each thread computes the state of its cell. The threads with wet cells continue and store the coordinates of its cell. The reduction algorithm is then performed between all threads with wet cells, performing a reduction to find the maximum and minimum cell coordinates. The final values will be the four values defining the four edges of bounding box for this subdomain.

Then, the bounding box for the global domain is calculated. This is performed with a similar reduction as the kernels, only on the CPU instead. It checks each subdomains local bounding box to find the maximum and minimum coordinates.

Step 2: Next, the new workload for each subdomain is calculated. This is done by computing the workload as the amount of wet rows of the global bounding box according to the *initial auto-tuning* workload. More precisely,

each GPU is assigned a number of rows from the global bounding box. The number of rows is defined as W percent of rows as calculated in equation (4.1). By doing this, each GPU will compute on an amount of wet cells that is suitable for its computational power.

Step 3: Finally, a dynamic decomposition (see section 4.3) is performed according to the new auto-tuned workload.

The multi-node implementation employs a similar auto-tuning algorithm. This means that each process computes its local bounding boxes. Then, these are synchronized so that the computation of the global bounding box is designated to a master process. The master process also computes the workload and broadcasts this back to the other processes.

4.5 Results

To validate the performance of the implemented auto-tuning techniques on the multi-GPU cluster simulator, several benchmarks have been performed. I have run benchmarks for both the multi-GPU and the multi-node simulator. The benchmarks have been run on standard desktop computers, which I specify more closely when explaining the benchmarks. They were further performed using two different simulation cases. The first case used is a synthetic idealised circular dam break case as shown in figure 4.6. The second case is a real world dam break, The Malpasset dam break, which collapsed in 1959 [7]. It can be seen in figure 4.7. The case used is specified for each benchmark.

First, I performed a benchmark of the early exit optimization, shown in section 4.5.1. It is demonstrated both with and without the auto-tuning probe. Enabling early exit showed a slight performance improvement over disabling early exit, while auto-tuning of early exit showed a good performance increase over both disabling and enabling early exit. Secondly, benchmarks demonstrating the performance of the auto-tuning domain decomposition are run. These are shown in section 4.5.2. The benchmarks showed that load-balancing the multi-GPU and multi-node simulator was an appropriate strategy for domains with a large amount of wet cells. However, for domains with many dry cells, a static load-balance with early exit proved best.

4.5.1 Performance of Early Exit

I have performed a benchmark of the early exit optimization technique. The benchmark demonstrates the performance of the implemented early exit technique, including the auto-tuning of it. It was run on a system composed of two GeForce GTX 480 graphics cards. The benchmarks are run on a domain initialized as a flat circular dam surrounded by a wall with radius $R = 200m$ in a square computational domain of size $2000m \times 2000m$ with center at $x_c = 1000m$, $y_c = 1000m$. The following is set as initial conditions: The bathymetry B is set to $B(x, y, 0) = 0$. The water momentum along the x-axis and y-axis, Q_2 and Q_3 is set to $Q_2(x, y, 0) = Q_3(x, y, 0) = 0$. The

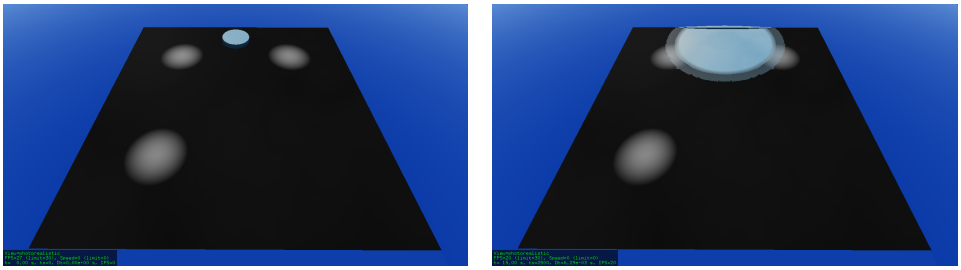


Figure 4.6: An idealised circular dam break in a square computational domain. It describes a circular water column surrounded by a wall on a bathymetry with three bumps. Here, the water column is placed at the upper part of the domain. At time $t = 0$ (left), the dam instantly collapses, creating an outgoing circular wave, seen at $t = 15$ (right). The water propagates until it covers the whole domain.

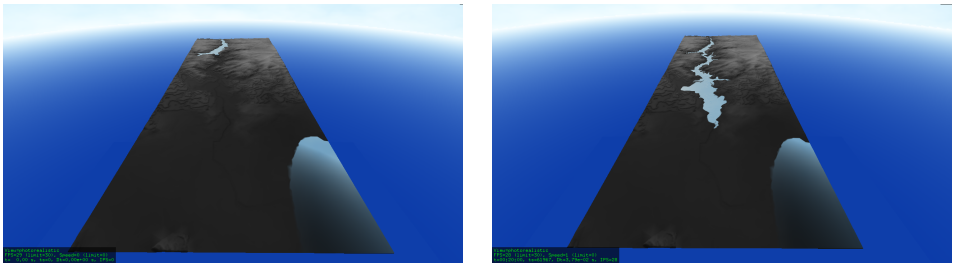


Figure 4.7: The Malpaset dam break. At time $t = 0$ (left), the water can be seen in the upper part pf the domain, while the sea is at the bottom right. Then, a breach occur, creating a flood that flows through the valley towards the bottom of the domain, seen at $t = 1200$ (right).

water elevation Q_1 is set to:

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

It was also set up with wall boundary conditions. At $t = 0$, the dam instantly collapses, creating an outgoing circular wave that propagates until $t = 1200$.

Furthermore, the benchmark run five multi-GPU simulations with two subdomains set up on two GPUs. Each subdomain has a resolution of $(2^{\times}2^n)$, where $n = 8 - 12$. It is run with the second-order accurate Runge-Kutta time integrator. The benchmark is run three times: *Standard*, *Early exit* and *EE auto-tune*. *Standard* runs all simulations with early exit disabled, computing on all cells in the domain. *Early exit* performs early exit of dry cells, while the last, *EE auto-tune*, runs the auto-tuning probe of the early exit algorithm. Here, the interval is set to 500. The simulation time was set to 20 minutes to achieve a more realistic simulation length. The results can be seen in figure 4.8.

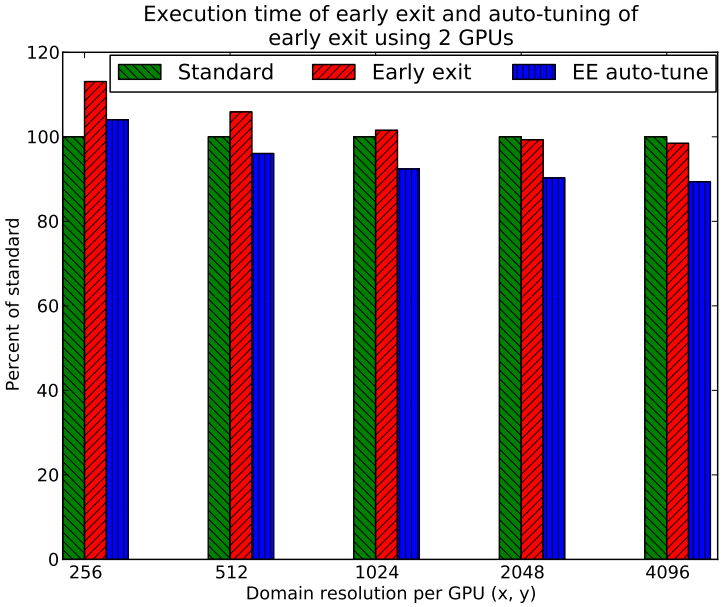


Figure 4.8: Early exit performance benchmark showing the percentage of achieved execution time for *early exit* and *EE auto-tune* relative to *standard* on an idealised circular dam break. Lower percent is better. It is run on a multi-GPU setup of two GPUs, each GPU running subdomain resolutions of 256×256 to 4096×4096. As seen, auto-tuning the early exit technique gives a performance improvement for each increase in resolution.

I first compare the *Standard* simulation against the simulation with *Early exit*. The first performs better for smaller domain resolutions, while *Early exit* improves and performs better for domains with larger resolution. This is likely because of the increased number of cells to skip flux calculations for, decreasing the impact of the overhead related to *Early exit*. However,

Early exit only runs slightly better for larger resolutions. This is likely because the water has covered the domain for most of the simulation. In effect, using *Early exit* only gives extra overhead related to running the early exit kernel. Running larger resolutions should give better performance results for *Early exit* since it would skip flux calculations for the majority of the simulation.

Finally, the *EE auto-tune* is shown. Ideally, this should give the best results since it is an attempt to enable and disable use of early exit at the appropriate times. For the smallest domain resolutions, it performs slightly better than the two other methods, with the exception of the lowest resolution. Compared to *Early exit*, it has a good speed up improvement increasing the domain resolution.

Comparing against *Standard*, *EE auto-tune* does not perform significantly better for small resolutions. Because of the low resolutions, *EE auto-tune* will mostly produce extra overhead. Because of this, the performance difference is negligible compared to *Standard*. Simulating on larger resolutions however will give a significant speed up. This is because it now manages to use both techniques appropriately without too large costs related to the overhead of early exit and the auto-tuning probe. For the largest domain resolutions, it gives a solid speed up of 10 minutes over *Standard*. Finally, tuning the auto-tune interval for specific simulations could produce even better results for *EE auto-tune*. For example, if the domain resolution is large and the water covers the domain for most of the simulation, increasing the interval could give better performance.

The simulation times are shown in table 4.1.

Domain resolution	Standard	Early exit	EE auto-tune
256×256	3.0E+0	3.4E+0	3.2E+0
512×512	1.6E+1	1.6E+1	1.5E+1
1024×1024	1.1E+2	1.1E+2	9.8E+1
2048×2048	8.2E+2	8.1E+2	7.4E+2
4096×4096	6.4E+3	6.3E+3	5.7E+3

Table 4.1: The execution times (seconds) with early exit disabled (standard), enabled (early exit) and early exit auto-tuned (EE auto-tune). Here, using a varying number of domain resolutions on two GPUs and use of the second-order Runge-Kutta time integrator.

4.5.2 Performance of auto-tuning domain decomposition

I have performed extensive performance benchmarks of the auto-tuning dynamic domain decomposition technique for both the multi-GPU and multi-node simulators. I run the benchmarks with two different cases, the Malpasset dam break (see figure 4.7) and an idealised circular dam break (see figure 4.6). The Malpasset dam break has been run by simulating the first 4000 seconds after the breach. It was run using the first-order accurate Euler time integrator. The circular dam break is run as an idealised

circular dam surrounded by a wall with radius $R = 133m$ in a square computational domain of size $2000m \times 2000m$ with center at $x_c = 1000m$, $y_c = 250m$. The following is set as initial conditions: The bathymetry B consists of three bumps, each bump has a radius of $R_B = 200m$. The first bump has a center at $bx_{c1} = 500m$, $by_{c1} = 500m$, the second bump has a center at $bx_{c2} = 1500m$, $by_{c2} = 500m$, while the last bump has a $bx_{c3} = 500m$, $by_{c3} = 1500m$. The bathymetry B is set to:

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

The water momentum along the x-axis and y-axis, Q_2 and Q_3 is set to $Q_2(x,y,0) = Q_3(x,y,0) = 0$, while the water elevation Q_1 is set to:

$$Q_1(x,y,0) = \begin{cases} Q_1 = 40m & \text{if } (x - x_c)^2 + (y - y_c)^2 \leq R^2 \\ Q_1 = 0m & \text{if } (x - x_c)^2 + (y - y_c)^2 > R^2. \end{cases}$$

The initial conditions were also set up to use wall boundary conditions and the first-order Euler time integrator. At $t = 0$, the dam instantly collapses, creating an outgoing circular wave that propagates until $t = 80$. More initialization parameters are described specifically for each benchmark.

Both cases are benchmarked on different type of systems. First of all, I benchmark on two different multi-GPU systems: *setup 1* and *setup 2*. The first system *setup 1* consists of a 2.67 GHz Intel Core i7 920 CPU, 6 GB system memory and two equal graphics cards, the GeForce GTX 480, each with 1.5 GB memory. The second system *setup 2* consist of 3.4 GHz Intel Core i7 2600K CPU, 8 GB system memory and two graphics cards, the GeForce GTX 580 with 1.5 GB memory and the GeForce GTX 285 with 1 GB memory. I use two different multi-GPU setups to test the impact of auto-tuning when utilizing equal graphics cards, as well as when using two different graphics cards. Finally, I also perform the benchmarks on a cluster system. The cluster consists of the two nodes, *setup 1* and *setup 2*, as described above. This means that it uses two GeForce GTX 480s on the first node and a GeForce GTX 580 and GeForce GTX 285 on the second node.

For each benchmark, I run three different simulations: *Static*, *Static EE*, and *Dynamic*. *Static* defines a simulation where the whole computational domain is distributed equally between the GPUs. Also, early exit is disabled, which means it computes on all cells. *Static EE* executes a similar simulation, but with early exit enabled, which means it calculates fluxes only for wet cells. The last type *Dynamic* defines a simulation running with full auto-tuning of both dynamic domain decomposition and early exit.

This means that it auto-tunes and load-balances the workload as necessary. The auto-tuning interval is specified closer for each benchmark.

Multi-GPU

I first show the benchmarks using two multi-GPU setups *setup 1* and *setup 2*, as described above.

Setup 1: The first benchmark uses the idealised circular dam break case, as seen in figure 4.9. It was run with two different domain resolutions, 2000×2000 (left) and 4000×4000 (right). The primary purpose of this is too see how *Dynamic* performs when the resolution is increased. The benchmarks are run decomposing a domain into two subdomains, each initialized on a different graphics card. The auto-tune interval was set to 1000.

As expected, *Static* has the worst performance. First, both resolutions show the same behavior. The main difference is that the highest resolution gives much better performance. This is because the increased amount of cells to compute on occupies the graphics cards better. At peak performance, *Static EE* has about 60% better performance than *Dynamic*. However, as the water propagates, early exit decreases exponentially. As seen at $t = 30$, *Dynamic* outperforms *Static EE*. I expect this to be related to the amount of wet cells. When the majority of the domain is dry, *Static EE* has a significantly better performance than *Dynamic*. *Static EE* has equal workload, which means that at the start one GPU will compute on the majority or all of the wet cells and the other GPU will only have dry cells. *Dynamic* however, performs worse because here both GPUs compute on equal amount of wet cells. However, the numbers of wet cells are low and do therefore not occupy the graphics cards, which implies that using a single GPU is better. The large amount of dry cells therefore gives a large overhead which negatively impacts *Dynamic*. However, when the water has propagated far enough, *Dynamic* performs better. This is related to that the amount of wet cells have significantly increased, allowing for better occupation of both graphics cards. However, *Static* EE does not sufficiently balance the load between the GPUs. This is because one of the cards performs all the work, while the other computes on far less wet cells.

Then, I show the Malpasset dam break benchmark, as seen in figure 4.10. It was run with two different domain resolutions, 879×2199 cells (left), equally spaced by 7.5m i.e., $\Delta x = \Delta y = 7.5m$ and 1759×4399 cells (right), equally spaced by 3.75m i.e., $\Delta x = \Delta y = 3.75m$. The primary purpose of this is too see how *Dynamic* performs when the resolution is increased. The benchmarks are run decomposing a domain into two subdomains, each initialized on a different graphics card. The auto-tune interval was set to 5000 for the lowest resolution and 4000 for the highest resolution.

Here, *Static EE* outperforms *Dynamic* for the whole simulation. I reason that, as concluded from figure 4.9, that this is mainly due to large amount of dry cells. This especially applies to the Malpasset dam break, because

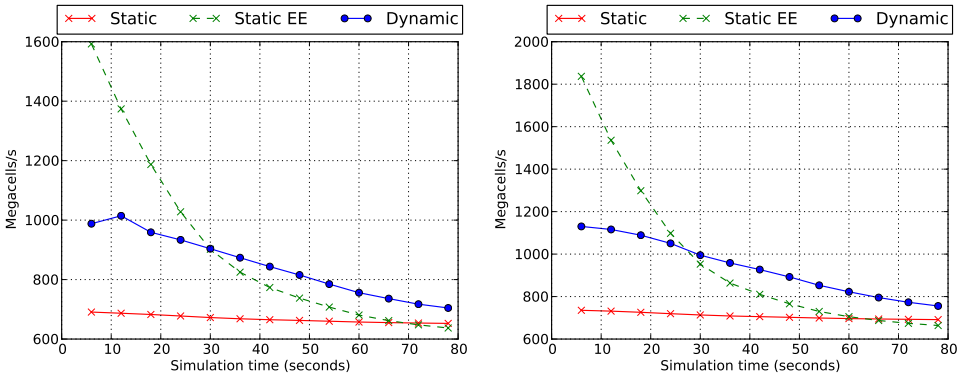


Figure 4.9: Auto-tuning benchmark on an idealised circular dam break, showing the performance in megacells per second throughout the simulation. It is performed with two domain resolutions, 2000 $\times$ 2000 (left) and 4000 $\times$ 4000 (right). The benchmarks are run using two GeForce GTX 480 graphics cards. The results shows that *Dynamic* gives better load-balancing on both graphics cards when there are large amount of wet cells. Also, notice that the highest resolution gives much better performance than the lowest resolution.

for this case there are large amount of dry cells throughout the whole simulation. This can easily be seen in figure 4.7. Here, the water follows the valley after the breach and does therefore not cover the whole domain. As such, *Static EE* gives best performance because it applies the early exit algorithm and has a better utilization of the GPUs. *Dynamic* computes on small amount of wet cells for both GPUs, which is an ineffective solution. Finally, the highest resolution gives much better performance than the lowest resolution, which is due to better occupation of the graphics cards.

Execution times	Circular dam break	Malpasset dam break
Simulation	1.1E+3	1.3E+3
Auto-tune (total)	5.6E+1	2.6E+1
Auto-tune EE	3.6E+0	1.3E+0
Auto-tune BB	4.4E+1	2.1E+1
Auto-tune DD	9.3E+0	4.3E+0

Table 4.2: The execution times (seconds) for the idealised circular dam break (4000 $\times$ 4000) and the Malpasset dam break (1759 $\times$ 4399) using two GeForce GTX 480 graphics cards. The table shows the execution time for the simulation and different parts of the auto-tuning technique. As seen, there is no significant overhead related to the auto-tuning.

Also, I show the execution times related to auto-tuning both benchmarks for the highest resolution in table 4.2. I split the different auto-tuning algorithms into different parts to easily identify the bottleneck of the auto-tuning. Here *Auto-tune (EE)* is the total execution time for auto-tuning early exit, while *Auto-tune (BB)* is the total execution time for *runtime auto-tuning*,

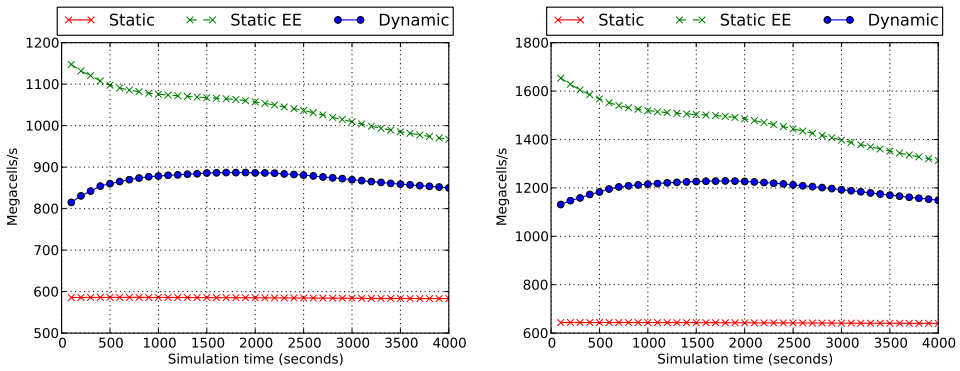


Figure 4.10: Auto-tuning benchmark on the Malpasset dam break, showing the performance in megacells per second throughout the simulation. It is performed with two domain resolutions, 879×2199 (left) and 1759×4399 (right). The benchmarks are run using two GeForce GTX 480 graphics cards. The results shows that *Static EE* gives better performance than *Dynamic* through the whole simulation. Also, notice that the highest resolution gives much better performance than the lowest resolution.

more precisely the calculation of the bounding box and related auto-tuning. *Auto-tune (DD)* measures the total execution time for dynamic domain decomposition. There are summed into the total execution time for the auto-tuning *Auto-tune (total)*. Lastly, the simulation execution time *Simulation* is shown. The table shows that the total auto-tuning execution time is 5% of the circular dam break execution time. For the Malpasset dam break, the total auto-tuning execution time is 2% of the simulation execution time. This means that there is a low overhead of running the auto-tuning. The execution times related to auto-tuning does therefore not impact the simulation significantly. Of the different parts, the bounding box auto-tuning is the bottleneck.

Setup 2: Then, I run benchmarks for both the idealised circular dam break and Malpasset dam break *setup 2*. The benchmarks are completely identical to the ones performed for the first setup *setup 1*. They are shown in figure 4.11, the idealised circular dam break (top) and the Malpasset dam break (bottom). Here, the auto-tune interval is set to 1000. For the first case, the resolutions are 2000×2000 (left) and 4000×4000 (right), and for the last case, 879×2199 cells (left), equally spaced by 7.5m i.e., $\Delta x = \Delta y = 7.5m$ and 1759×4399 cells (right), equally spaced by 3.75m i.e., $\Delta x = \Delta y = 3.75m$. Here, the auto-tune interval was set to 5000 for the lowest resolution and 4000 for the highest resolution.

First of all, the benchmark for the idealised circular dam break shows a similar behavior to that of *setup1* in figure 4.9. However, here *Static EE* shows even better performance than *Dynamic*. At $t = 50$, *Dynamic* performs better than *Static EE*. This is due to the increased amount of wet cells, giving better load-balancing for the *Dynamic* simulation. Then, I compare the Malpasset dam break benchmark with the one for *setup 1*,

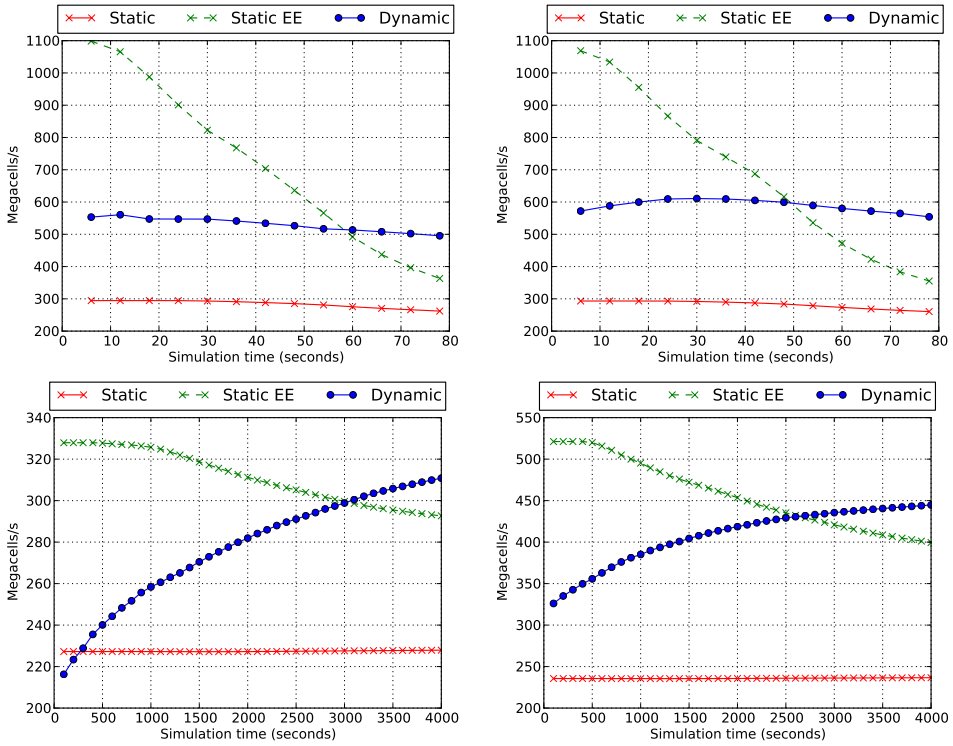


Figure 4.11: Auto-tuning benchmark on an idealised circular dam break (top) and the Malpasset dam break (bottom), showing the performance in megacells per second throughout the simulation. It is performed with two domain resolutions, 2000×2000 (left) and 4000×4000 (right) for the idealised circular dam break and 879×2199 (left) and 1759×4399 (right) for the Malpasset dam break. The benchmarks are run using two graphics cards, the GeForce GTX 580 and the GeForce GTX 285. The results show that *Static EE* has the best performance for the majority of the simulation. Also, notice that the highest resolutions give slightly better performance than the lowest resolutions. Increasing the resolution also increases the performance of *Dynamic* slightly compared to *Static EE*.

as shown in figure 4.10. First of all, for the benchmark in figure 4.11 the overall performance is much lower compared to 4.10. This is probably related to the GeForce GTX 285 card which is less powerful than the other cards, therefore being a bottleneck. Also, *Dynamic* shows a slightly better performance than *Static EE* at the end of the simulation. This did not happen for the benchmark performed on *setup 1*. I reason that this is because the GeForce GTX 580 gets a larger workload than the GeForce GTX 285 as the water reaches the bottom of the domain. Therefore, this gives better performance than *Static EE*, because the GeForce GTX 580 is more powerful than the GeForce GTX 285.

Finally, I show the execution times related to auto-tuning both benchmarks for the highest resolution in table 4.3. The table shows that the total auto-tuning execution time is 3.6% of the circular dam break execution time. For the Malpasset dam break, the total auto-tuning execution

time is 0.7% of the simulation execution time. This means that there is a low overhead of running the auto-tuning. The execution times related to auto-tuning does therefore not impact the simulation significantly. Of the different parts, the bounding box auto-tuning is the bottleneck.

Execution times	Circular dam break	Malpasset dam break
Simulation	1.4E+3	3.3E+3
Auto-tune (total)	5.0E+1	2.4E+1
Auto-tune EE	5.0E+0	2.5E+0
Auto-tune BB	3.6E+1	1.7E+1
Auto-tune DD	8.5E+0	4.7E+0

Table 4.3: The execution times (seconds) for the idealised circular dam break (4000×4000) and the Malpasset dam break (1759×4399) using two graphics cards, the GeForce GTX 580 and the GeForce GTX 285. The table shows the execution time for the simulation and different parts of the auto-tuning technique. As seen, there is no significant overhead related to the auto-tuning.

Multi-node

Finally, I show the benchmarks for the multi-node system, using both *setup 1* and *setup 2* as the two nodes. I first run the the idealised circular dam break case, as seen in figure 4.12 (left). It was run with a domain resolution of 4000×4000. The benchmark is run decomposing a domain into four subdomains, each initialized on a different graphics card. The auto-tune interval was set to 2000. The benchmark shows a similar behavior to the multi-GPU cases in figure 4.9 and 4.11. However, here *Static EE* has a much better performance than *Dynamic* for the majority of the simulation. This might be related to that the domain resolution is too low for multi-node setup with four GPUs. I reason that increasing the resolution should give much better occupation of the GPUs, especially for the *Dynamic* case, therefore giving a performance improvement over *Static EE*. Running a higher resolution was difficult because of the memory limitations on the graphics cards used.

Also, I show a benchmark for the Malpasset dam break, as seen in figure 4.12 (right). It was run with a domain resolution of 1759×4399 cells, equally spaced by 3.75m i.e., $\Delta x = \Delta y = 3.75m$. The benchmark is run decomposing a domain into four subdomains, each initialized on a different graphics card. The auto-tune interval was set to 6000. This benchmark also shows a similar behavior to the multi-GPU cases in figure 4.10 and 4.11. Here, *Static EE* also gives the best performance. I reason that this is also related to the occupation of the graphics cards caused by the large amount of wet cells and the resolution. Running simulations on domains with most of the domain covered with water would likely give better results for *Dynamic*. Also, increasing the resolution should give better utilization of the GPUs, giving a positive performance impact for

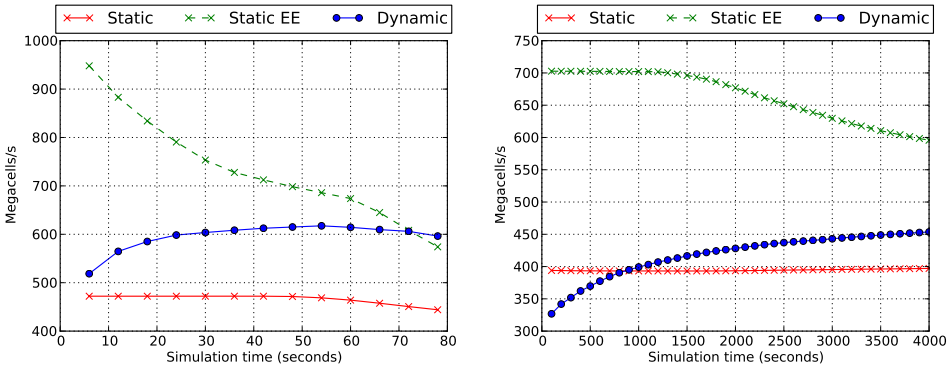


Figure 4.12: Auto-tuning benchmark on an idealised circular dam break (left) with a resolution of 4000 × 4000 and the Malpasset dam break (right) using a resolution of 1759 × 4399. The performance is shown in megacells per second throughout the simulation. The benchmarks are run using two nodes, the first with two GeForce GTX 480 graphics cards and the second with a GeForce GTX 580 and a GeForce GTX 285. The results shows that *Static EE* gives the best performance for both cases.

Dynamic.

Finally, I show the execution times related to auto-tuning both benchmarks in table 4.4. The table shows that the total auto-tuning execution time is 2.4% of the circular dam break execution time. For the Malpasset dam break, the total auto-tuning execution time is 0.6% of the simulation execution time. This means that for multi-node there is also a low overhead of running the auto-tuning. The execution times related to auto-tuning does therefore not impact the simulation significantly. Of the different parts, the dynamic decomposition uses the most time.

Execution times	Circular dam break	Malpasset dam break
Simulation	1.3E+3	3.2E+3
Auto-tune (total)	3.1E+1	1.8E+1
Auto-tune EE	9.0E-1	3.5E-1
Auto-tune BB	1.3E+1	8.1E+0
Auto-tune DD	1.8E+1	9.1E+0

Table 4.4: The execution times (seconds) for the idealised circular dam break (4000 × 4000) and the Malpasset dam break (1759 × 4399) using two nodes, the first with two GeForce GTX 480 graphics cards and the second with a GeForce GTX 580 and a GeForce GTX 285. The table shows the execution time for the simulation and different parts of the auto-tuning technique. As seen, there is no significant overhead related to the auto-tuning.

Auto-tune interval tests

I also performed several tests using a different auto-tune interval for all benchmarks. Because the overhead for the auto-tuning was very low, as shown, I reason that the interval could be increased. This would increase the frequency it changes the workload on, and could potentially increase the performance because the GPUs would compute on more adequate workload. However, experimenting on it showed no added performance improvements. At some benchmarks, it showed a decrease in performance, which probably is because of the increased overhead. I therefore conclude that altering this variable had a negative performance impact.

Chapter 5

Conclusion

In this thesis, I explored load-balancing between multiple GPUs and multiple nodes by investigating dynamic auto-tuning techniques. My results showed that load-balancing had a good effect on domains with a large amount of wet cells for the multi-GPU systems benchmarked. However, for cases with many dry cells, a static workload distribution using early exit showed superior performance.

I extended a single-GPU simulator [7] to an environment of multiple graphics cards and multiple nodes. To decompose the domain across multiple GPUs, the *row domain decomposition* technique was implemented. Also, to enable faster computations, a technique called *Early exit* was implemented. The multi-GPU and multi-node implementation showed good scaling across multiple GPUs and multiple nodes. I also implemented dynamic auto-tuning techniques to load-balance the computational domain between the graphics cards. The technique worked together with the *Early exit* technique to load-balance the computational part of the domain between the cards. The technique was divided into two algorithms. The first algorithm auto-tuned on the computational power of each GPU to determine the optimal workload for that GPU. The second algorithm auto-tuned on the underlying computational domain by computing the bounding box around the wet cells. The bounding box was then distributed between the graphics cards according to the workload determined by the first algorithm.

In this thesis, the dynamic auto-tuned multi-GPU simulator was applied to the shallow water equations. However, the implementation is general in use and should work equally well for any systems of conservation laws, for example for the Euler equations [9], which describes the dynamics of an ideal gas. These equations could also be solved in a computational domain with *wet* and *dry* cells, equal to the shallow water equations. First of all, decomposing a domain into multiple subdomains across multiple GPUs should work equally well as the shallow water equations for any computational domain consisting of other types of fluids or gases. Furthermore, the implemented auto-tuning technique should also work well. For example, computing the bounding box for other types of fluids or gases should work just as well as performing it on water flow for the shallow water equations.

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] AMD. AMD - GPU Association - Targeting GPUs for Load Balancing in OpenGL. http://developer.amd.com/wordpress/media/2012/10/GPU_Association_WhitePaper.pdf, (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, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. In *Scientific Programming*, 2010.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] Khronos Group. OpenGL Shading Language. http://www.opengl.org/documentation/glsl/, (visited on 2014-08-01).

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

[15] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal Discontinuous Galerkin Methods on Graphics Processors. In *Journal of Computational Physics*, 2009.

[16] 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.

[17] 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.

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

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

[20] 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.

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

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

[23] matplotlib. http://matplotlib.org, (visited on 2014-08-01).

[24] MPI. Message Passing Interface Forum. http://www.mpi-forum.org/, (visited on 2014-08-01).

[25] MPICH. http://www.mpich.org/, (visited on 2014-08-01).

[26] NVIDIA. CUDA. http://www.nvidia.com/object/cuda_home_new.html, (visited on 2014-08-01).

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

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

[29] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, (visited on 2014-08-01).

[30] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. In *Proceedings of the IEEE*, 2008.

[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] python. http://www.python.org, (visited on 2014-08-01).

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

[34] 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.

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

[37] 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.

[38] 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.

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

[40] 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.