PARALLELISATION OF SHALLOW WATER SIMULATION FOR HETEROGENEOUS ARCHITECTURES

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2012

By
Michail Emmanouil Pappas
School of Computer Science

Contents

Word Count: 22,269

List of Figures

List of Tables

Abstract

This work presents the parallelisation of a shallow water simulation model. Two parallel implementations are developed. One is for a multi-core NUMA architecture, developed in OpenMP. The other one is for a many-core GPU-accelerated architecture and is developed in OpenCL. The parallelisation process is based on an iterative approach, starting off from a naive implementation. Each iteration involves the identification, analysis and improvement of a particular overhead. The process repeats until no further optimisations can be performed. The evaluation is based on the comparison of the two implementations. The final results show strengths and weaknesses in both implementations, as they are reflected from the differences of the two architectures.

Declaration

No prior of the work referred to in this dissertation has been submitted in support of an application for another degree of qualification of this or any other university or other institute of learning.

Copyright

Dedication

To my grandfather, who started it all...

Acknowledgements

I would like to thank Mr Graham Riley and Prof. John Gurd, for their supervision on this project, as well as for their work on teaching the MSc modules of parallel computing. I would also like to thank my family for their continuous love and support.

Chapter 1

Introduction

During the past few decades, the advances in microprocessor technology provided continuously increasing performance at reduced cost. In the middle of the last decade, a stall in processor frequency and overall system performance was observed. Transistor scaling faced power related issues [1], the difference between memory and processor speed led to the memory wall [2], and the advances of ILP reached diminishing returns [3]. These challenges were overcome by shifting to concurrency, a shift that allowed manufacturers to continue following the trends dictated by Moore's law [4], but also delivered a significant impact to the software development community [5].

The switch to concurrency was expressed through two different directions [6]. One direction followed the *multi-core* approach, with the number of cores doubling approximately with each CMOS generation. The other direction followed the *many-core* approach, mainly expressed by the use of GPUs for general purpose computing, a tactic commonly referred to as GPGPU.

During this process, several heterogeneous architectures emerged, featuring both general-purpose processors and specialised accelerating units [7]. Accelerating units ranged between specialised devices like GPUs and DSPs, custom units (ASICs and FPGAs) or integrated SoCs integrating some of the above on a single die.

Among heterogeneous architectures, GPU-based platforms have been the most successful ones. This is attributed to the ubiquity of GPUs in both home and commercial computers, their relatively low price and their ease of programmability compared to other accelerators like FPGAs. Moreover, their high floating-point throughput, their data-parallel approach and their high memory bandwidth has made them attractive for the acceleration of scientific applications [7], [8].

This work revolves around the parallelisation of a shallow water simulation model. Numerical weather prediction is one of the areas that have significantly benefited from the advent of supercomputers [9]. The shallow water simulation used in this work represents a real atmospheric model, and was originally developed as a benchmark for supercomputers like the CRAY-1 [10]. Similar work for contemporary architectures is presented in [11], [12], [13].

The main part of this work consists of two parallel implementations. The fist is a multi-core implementation developed for a typical NUMA architecture, a 16-core AMD Opteron machine. Parallelisation is performed with the aid of OpenMP [14], a shared-memory multiprocessing framework traditionally used in multi-core systems. The second one is a heterogeneous version developed for a 4-core superscalar Intel Xeon machine, featuring an NVIDIA Quadro 2000 device. Parallelisation is performed with the use of OpenCL [15]. OpenCL is one of the dominant frameworks for development on heterogeneous architectures, being supported by different types of accelerating devices like GPUs, APUs and FPGAs.

The parallelisation process is based on an iterative approach. Each version is naively implemented at first, followed by a sequence of optimisations based on prior analysis. Each iteration involves the identification, analysis and minimisation of a specific overhead. The process continues until it becomes difficult to achieve better performance. Upon completion, each version is compared to a reference implementation. The OpenMP implementation is compared to an MPI version of the same model, while the GPU version is evaluated with respect to the multi-core, OpenMP implementation. The results highlight strengths and weaknesses in both implementations, as they are reflected from the differences of the two architectures.

The rest of this document is organised as follows: Chapter 2 presents the theory behind the shallow water model used, as well as the sequential implementation of the benchmark. Chapter 3 summarises OpenCL, which is the framework of choice for the heterogeneous implementation. A description of modern GPU architectures, focused on the NVIDIA Fermi device family, is presented in Chapter 4. Chapter 5 describes the methodology used for the development of the two parallel implementations, as well as the means and metrics used for the evaluation of the experimental results. Chapters 6 and 7 present the main parts of this work, the multi-core and many-core implementations, along with their associated results. The conclusions of this work are discussed in Chapter 8. Finally, Chapter 9 summarises some possible directions that may be pursued in terms of future work.

Chapter 2

Shallow Water Dynamics

The shallow water dynamics model is a set of hyperbolic partial differential equations describing the flow of fluids in regions where the vertical dimension is considerably smaller than the horizontal one. The application of shallow water equations is wide. They are commonly used to describe various kinds of flows including oceanic, coastal and river hydrodynamics. However, the shallow water dynamics model does not strictly apply to fluids such as water. Certain aspects of atmospheric flow can be described by variants of the shallow water equations, and these models are widely used in numerical weather prediction. Shallow water equations are described in depth in C. D. Vreugdenhil's book “Numerical Methods for Shallow-Water Flow” [16]. The detailed description of these models is outside the scope of this thesis.

This chapter presents the shallow water dynamics simulation used in this work. The first section presents the background theory behind the model used. The sequential implementation as it is derived from the above model follows, together with the achieved performance results when run on a single processor. The chapter closes with some discussion around the model's potential of parallelism, as derived from the data dependencies between computations.

2.1 The Shallow Water Model

The model used in this work is based on the shallow water equations as they were presented in Sadourny's paper in 1975 [17]. Ignoring the Coriolis forces, they can be expressed as

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0 \qquad (9.1)$$

$$\frac{\partial v}{\partial t} - ZU + \frac{\partial H}{\partial y} = 0 \qquad (9.2)$$

$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial x} = 0 \qquad (9.3)$$

where u and v are the wind velocities in the x and y directions, P is the potential

pressure, U and V are the mass fluxes, Z is the potential vorticity and H is the free surface height.

The mass fluxes U and V are further defined as

$$U = Pu \qquad (9.4)$$

$$V = Pv \qquad (9.5)$$

the potential vorticity as

$$Z = \frac{(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y})}{P} \qquad (9.6)$$

and the surface height as

$$H = P + \frac{1}{2}(u^2 + v^2) \qquad (9.7)$$

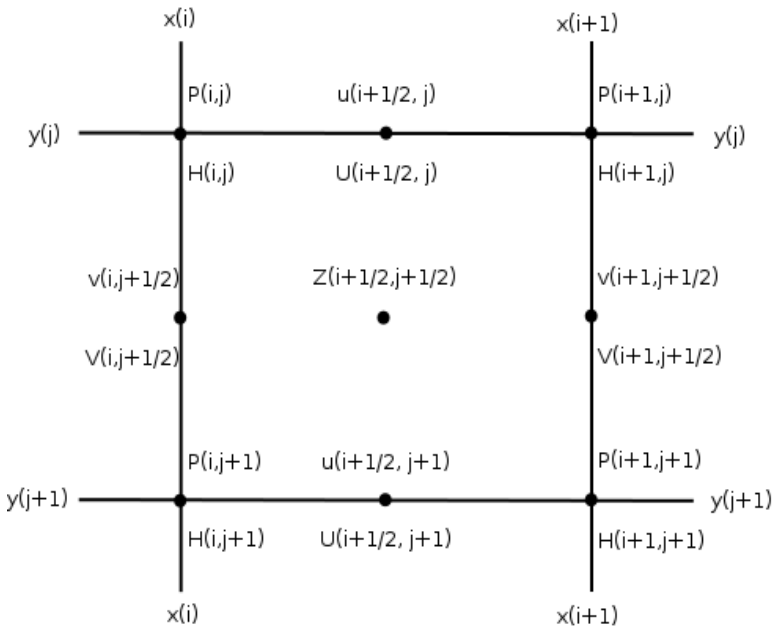


Figure 2.1.1: Mapping of the shallow water variables on an Arakawa-C grid.

The equations are solved using a two-dimensional, finite-difference model that is updated over space and time. The finite-difference model is calculated on an Arakawa C grid [18], as shown in Figure 2.1.1 [10]. The grids dimensions are defined over a rectangle of a < x < b and c < y < d, by selecting M and N so that the grid's elements are defined as:

$$x_i = i\,\Delta x + a \qquad \text{i = 0, ½, 1, … , M+1}$$

$$y_j = y\,\Delta y + b \qquad \text{j = 0, ½, 1, … , N+1}$$

where Δx = (b-a) / (M+1) and Δy = (d-c)/(N+1).

Before computation is performed, velocities are initialised using a stream function, as shown in discrete form below.

$$\Psi_{i+1/2,\,j+1/2} = A \sin\left(\frac{2\pi x}{b-a}\right) \sin\left(\frac{2\pi y}{d-c}\right) \qquad (9.8)$$

Subsequently, initial velocities are derived as

$$u_{i+1/2,\,j} = -\frac{\Psi_{i+1/2,\,j+1/2} - \Psi_{i+1/2,\,j-1/2}}{\Delta y} \qquad (9.9)$$

$$v_{i,\,j+1/2} = \frac{\Psi_{i+1/2,\,j+1/2} - \Psi_{i-1/2,\,j+1/2}}{\Delta x} \qquad (9.10)$$

Pressure is initialised as

$$P_{i,\,j} = \frac{A^2}{4}\left[\left(\frac{2\pi}{d-c}\right)^2 \cos\left(\frac{4\pi i}{b-a}\right) + \left(\frac{2\pi}{b-a}\right)^2 \cos\left(\frac{4\pi j}{d-c}\right)\right] + P_0 \qquad (9.11)$$

with P_0 = 50,000.

Computations of u, v and p are repeated over a large number of cycles. Wind velocities and pressure are computed first, followed by potential vorticity, free surface height and the mass fluxes. The functions calculating each variable, are presented in discrete form below.

$$U_{i+1/2,\,j} = \frac{1}{2}\left(P_{i+1,\,j} + P_{i,\,j}\right) u_{i+1/2,\,j} \qquad (9.12)$$

$$V_{i,\,j+1/2} = \frac{1}{2}\left(P_{i,\,j+1} + P_{i,\,j}\right) v_{i,\,j+1/2} \qquad (9.13)$$

$$Z_{i+1/2,\,j+1/2} = \frac{\left[\frac{\left(v_{i+1,\,j+1/2} - v_{i,\,j+1/2}\right)}{\Delta x} - \frac{\left(u_{i+1/2,\,j+1} - u_{i+1/2,\,j}\right)}{\Delta y}\right]}{\frac{1}{4}\left(P_{i,\,j} + P_{i+1,\,j} + P_{i+1,\,j+1} + P_{i,\,j+1}\right)} \qquad (9.14)$$

$$H_{i,\,j} = P_{i,\,j} + \frac{1}{2}\left[\frac{u^2_{i+1/2,\,j} + u^2_{i-1/2,\,j}}{2} + \frac{v^2_{i,\,j+1/2} + v^2_{i,\,j}}{2}\right] \qquad (9.15)$$

$$u_{i+1/2,\,j}^{(n+1)} = u_{i+1/2,\,j}^{(n-1)} + \left(z_{i+1/2,\,j+1/2} + z_{i+1/2,\,j-1/2}\right) * \left(V_{i+1,\,j+1/2} + V_{i,\,j+1/2} + V_{i+1,\,j-1/2} + V_{i,\,j-1/2}\right) - \left(H_{i+1,\,j} - H_{i,\,j}\right) \quad (9.16)$$

$$v_{i,\,j+1/2}^{(n+1)} = v_{i,\,j+1/2}^{(n-1)} - \left(z_{i+1/2,\,j+1/2} + z_{i-1/2,\,j+1/2}\right) * \left(U_{i-1/2,\,j+1} + U_{i-1/2,\,j} + U_{i+1/2,\,j+1} + U_{i+1/2,\,j}\right) - \left(H_{i,\,j+1} - H_{i,\,j}\right) \quad (9.17)$$

$$p_{i,\,j}^{(n+1)} = p_{i,\,j}^{(n-1)} + \left(U_{i+1/2,\,j} - U_{i-1/2,\,j}\right) - \left(V_{i,\,j+1/2} - V_{i,\,j-1/2}\right) \quad (9.18)$$

In the equations of u, v and p, superscripts indicate the computation's iteration.

Cyclic boundaries are applied to all grids in the form of halos, in both x and y directions. Periodic continuation is applied so that the following equations hold.

$$f(x+b, y) = f(x+a, y) \quad (9.19)$$

$$f(x, y+d) = f(x, y+c) \quad (9.20)$$

Equations (9.16), (9.17), (9.18) are derived using the leapfrog scheme. In order to smooth out the oscillations introduced by this scheme, a time smoothing filter is applied at the end of each computation cycle. The filtering function is defined as

$$F^{(n)} = f^{(n)} + a\left(f^{(n+1)} - 2\mathrm{f}^{(n)} + F^{(n-1)}\right) \quad (9.21)$$

where filtering parameter α being equal to 0.001.

A detailed description of the derivation of the above equations can be found in [10].

2.2 Sequential Implementation

The sequential version of the shallow water benchmark was originally developed in Fortran by Paul Swarztrauber for the US National Center for Atmospheric Modelling (NCAR). An equivalent version in C was provided by Graham Riley and was used as the basis over which the subsequent parallel versions were developed. The complete code is presented in Appendix IV.

The code defines a group of (M+1)x(N+1) sized double precision floating point arrays, each one of which stores the elements of the dependent variables. Array names correspond to the names used in the equations described earlier, with most of them being straightforward to interpret (u, v, p, z, h). Array names cu and cv, standing for “capital u” and “capital v”, correspond to CU and CV, that is the mass fluxes in the horizontal and vertical directions, respectively.

```
for i in 1, 4000:
  //
  // compute cu, cv, z, h
  //
  for i in 0, M; j in 0, N:
      compute_cu(u, p)

  for i in 0, M; j in 0, N:
      compute_cv(v, p)

  for i in 0, M; j in 0, N:
      compute_z(u, v, p)

  for i in 0, M; j in 0, N:
      compute_h(u, v, p)

  //
  // periodic continuation
  //
  pc_cu(cu)
  pc_cv(cv)
  pc_z(z)
  pc_h(h)

  //
  // compute unew, vnew, pnew
  //
  for i in 0, M; j in 0, N:
      compute_unew(uold, cv, z, h)

  for i in 0, M; j in 0, N:
      compute_vnew(vold, cu, z, h)

  for i in 0, M; j in 0, N:
      compute_pnew(pold, cu, cv)
```

```
  //
  // periodic continuation
  //
  pc_unew(unew)
  pc_vnew(vnew)
  pc_pnew(pnew)

  //
  // time smoothing
  //
  for i in 0, M+1; j in 0, N+1:
      compute_uold(u, unew, uold)

  for i in 0, M+1; j in 0, N+1:
      compute_vold(v, vnew, vold)

  for i in 0, M+1; j in 0, N+1:
      compute_pold(p, pnew, pold)

  //
  // update for next cycle
  //
  for i in 0, M+1; j in 0, N+1:
      u[i][j] = unew[i][j]

  for i in 0, M+1; j in 0, N+1:
      v[i][j] = vnew[i][j]

  for i in 0, M+1; j in 0, N+1:
      p[i][j] = pnew[i][j]
```

Listing 2.2.1: The main computation loop of the shallow water benchmark.

The problem is computed for $\Delta t = 90$, over 4,000 iterations. Listing 2.2.1 shows the algorithm of the main computation loop, in the form of pseudocode. The computation of each array is performed in two steps. First, the computations of each element's values are performed over a MxN matrix. Then, the boundary conditions are applied by copying values to the halos. The copying of halos is performed in an anti-diametric fashion, by mirroring border elements across the main diagonal of each array. Two more arrays are introduced, namely *unew* and *uold*. These are helper arrays, that are used to store the values of the previous and next iteration, as they are required for the time smoothing filter described in Section 2.1.

2.3 Data Dependencies and Potential for Parallelism

This section examines the code's potential for parallelism. Obviously, the most computationally intensive part of the code is the main loop, and this will be the target for parallelisation. The potential for parallelism can be extracted by observing the data dependencies between computations. The main loop's iterations are dependent on one another and therefore must be performed sequentially. Within each iteration, though, several computations can be performed in parallel. The structure of the sequential version already reveals a partitioning of computations into three discrete groups. Computations of cu, cv and z and h are performed first, followed by unew, vnew and pnew, and finally by uold, vold and pold.

The computations of cu, cv, z and h are presented in Listing 2.3.1. Elements in cu, cv, z and h depend on u, v and p, but are independent from each other. Furthermore, none of these computations modifies the u, v or p arrays. Parallelism can therefore be performed in two levels: In array level, where each array can be computed independently of another, and in an element level, where the elements of a given array can be computed independently with respect to each other.

The periodic continuation phase for these arrays can also be performed in parallel, since it only involves copying elements within the same array. Nevertheless, its operations depend on the previous computations and therefore they can only be performed once the computation of the corresponding elements are complete.

```
for(i=0; i < M; i++)
  for(j=0; j < N; j++)
    cu[i+1][j] = 0.5 * (p[i+1][j] + p[i][j]) * u[i+1][j];

for(i=0; i < M; i++)
  for(j=0; j < N; j++)
    cv[i][j+1] = 0.5 * (p[i][j+1] + p[i][j]) * v[i][j+1];

for(i=0; i < M; i++)
  for(j=0; j < N; j++)
    z[i+1][j+1] = (fsdx * (v[i+1][j+1] - v[i][j+1]) - fsdy *
                  (u[i+1][j+1] - u[i+1][j])) / (p[i][j] + p[i+1][j] +
                   p[i+1][j+1] + p[i][j+1]);

for(i=0; i < M; i++)
  for(j=0; j < N; j++)
    h[i][j] = p[i][j] + 0.25 * (u[i+1][j] * u[i+1][j] + u[i][j] *
              u[i][j] + v[i][j+1] * v[i][j+1] + v[i][j]*v[i][j]);
```

Listing 2.3.1: Computation of cu, cv, z and h.

```
for (j=0;j<N;j++) {
  cu[0][j] = cu[M][j];
  cv[M][j + 1] = cv[0][j + 1];
  z[0][j + 1] = z[M][j + 1];
  h[M][j] = h[0][j];
}

for (i=0;i<M;i++) {
  cu[i + 1][N] = cu[i + 1][0];
  cv[i][0] = cv[i][N];
  z[i + 1][0] = z[i + 1][N];
  h[i][N] = h[i][0];
}

cu[0][N] = cu[M][0];
cv[M][0] = cv[0][N];
z[0][0] = z[M][N];
h[M][N] = h[0][0];
```

Figure 2.3.1: Periodic continuation of cu, cv, z and h.

Figure 2.3.2 presents a macroscopic view of the computations performed in each iteration, ordered by their capability to be performed concurrently, as it results from their data dependencies.

<table>
<tr><td>cu</td><td>cv</td><td>z</td><td>h</td></tr>
<tr><td>- barrier -</td><td>- barrier -</td><td>- barrier -</td><td>- barrier -</td></tr>
<tr><td>cu
periodic continuation</td><td>cv
periodic continuation</td><td>z
periodic continuation</td><td>h
periodic continuation</td></tr>
<tr><td colspan="4">- barrier -</td></tr>
<tr><td colspan="2">unew</td><td>vnew</td><td>pnew</td></tr>
<tr><td colspan="2">- barrier -</td><td>- barrier -</td><td>- barrier -</td></tr>
<tr><td colspan="2">unew
periodic continuation</td><td>vnew
periodic continuation</td><td>pnew
periodic continuation</td></tr>
<tr><td colspan="4">- barrier -</td></tr>
<tr><td colspan="2">uold</td><td>vold</td><td>pold</td></tr>
<tr><td colspan="2">- barrier -</td><td>- barrier -</td><td>- barrier -</td></tr>
<tr><td colspan="2">uold
periodic continuation</td><td>vold
periodic continuation</td><td>pold
periodic continuation</td></tr>
<tr><td colspan="4">- barrier -</td></tr>
<tr><td colspan="2">copy of u
(update for next cycle)</td><td>copy of v
(update for next cycle)</td><td>copy of p
(update for next cycle)</td></tr>
<tr><td colspan="4">- barrier -</td></tr>
</table>

Figure 2.3.2: Overview of parallelism in array level, for each iteration of the main loop.

2.4 Summary

The shallow-water model used in this work has been presented, along with a description of the sequential implementation of the code. The analysis of code in terms of data dependencies has revealed a significant amount of parallelism. The next chapter presents the fundamental concepts of OpenCL, the framework that is used to parallelise the shallow water code.

Chapter 3

The OpenCL Framework

OpenCL is a framework for writing parallel code on heterogeneous collections of devices. It is an open standard maintained by the Khronos Group [19], a non-profit organisation, that consists of over 100 companies including AMD, Apple, Intel and NVIDIA. They maintain a number of open API specifications including OpenCL, OpenGL and many others.

OpenCL is supported by a number of different devices, a comprehensive list of which is available in [20]. Most of these devices are CPUs and GPUs, some of which being integrated into a single chip. Some work has been performed into compiling OpenCL code to application-specific processors running on FPGAs [21].

The OpenCL API is defined in C, and it includes a C++ wrapping API. There are some third-party bindings for the most popular languages including C#, Java, Python and .NET. The latest specification is OpenCL 1.2 [22], which was announced in November 2011.

Having heterogeneity and portability in mind, OpenCL defines four abstract models that are applicable to a wider set of devices. These models are 1) the Platform Model, 2) the Execution Model, 3) the Memory Model and 4) the Programming Model. Vendors map the four models to the actual hardware by implementing the OpenCL API.

In the following sections, the four models will be described, as they are crucial to understand how OpenCL is implemented on hardware devices, and consequently how OpenCL programs can be optimised by taking into account the underlying device's architecture.

3.1 Platform Model

The platform model defines the fundamental relationships between devices. A *platform* is defined as a collection of devices with discrete roles. In practice, a platform must consist of same-vendor devices. A platform can therefore be thought of as a vendor-specific implementation.

During the execution of an OpenCL *program*, a *host* processor coordinates execution by

assigning work to one or more OpenCL compatible *devices*. Devices consist of one or more *compute units* (CUs), each of which is subdivided into one or more *processing elements* (PEs). Figure 3.1.1 [23] shows a graphical representation of this hierarchy.

Processing elements are capable of executing OpenCL objects called *kernels*. Kernels are described, in the execution model. The platform model further defines a set of APIs the host can use to communicate with the devices through the OpenCL runtime.

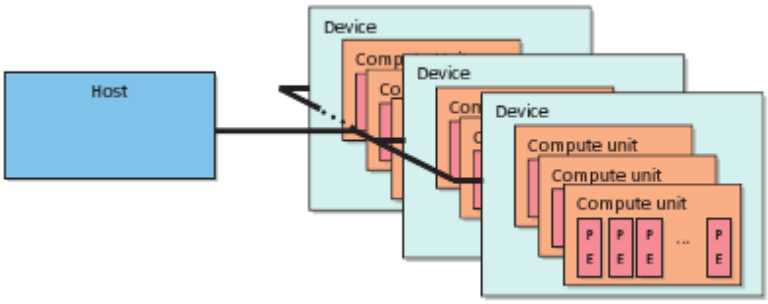


Figure 3.1.1: The OpenCL platform model.

3.2 Execution Model

The execution model defines OpenCL *kernels* and the *host program*. As described earlier, a kernel is a piece of object code that executes on an OpenCL device. The execution of kernels on devices is scheduled by the host program.

Kernels correspond to functions of a larger *program*. Programs are written using the *OpenCL C* language. OpenCL C is based on the ISO/IEC 9899:1999 specification of C, with several additions and restrictions. The standard defines OpenCL C in a separate section. Programs are compiled on the device, using the OpenCL compiler. Once compiled, a program object can be either scheduled directly for execution, or optionally be stored on the filesystem for later use.

In order to execute a kernel, the host needs to define a *context*. A context is a collection of resources, including devices, kernels, program objects and memory objects. In addition to the context, the host defines a *command queue*. The command queue is the means by which a host communicates with the devices; it is a queue from where the host can schedule commands and receive responses. One command queue needs to be created per device. Available commands include commands related to kernel execution, memory transfer, memory mapping, and synchronisation. Commands can be scheduled for in-order or out-of-order execution.

Once a kernel is submitted for execution, several instances of it are created, called work-items. Work-items are mapped into a n-dimensional index space of one, two or three dimensions, called an *NDRange*. The dimensions of the NDRange are defined by the programmer and they usually map to the dimensions of the input or the output data. Each work-item has a unique global ID among the NDRange.

Work-items are organised into larger collections, called *work-groups*. Work-groups are mapped into a more coarse space with the same dimensionality as the work-items' NDRange. Each work-group has a unique ID and work-items belonging to a work-group are assigned an additional local ID, which is unique among the items that belong to the same work-group. Work-items can therefore be either referenced by their global ID, or by the (workgroupID, localID) tuple.

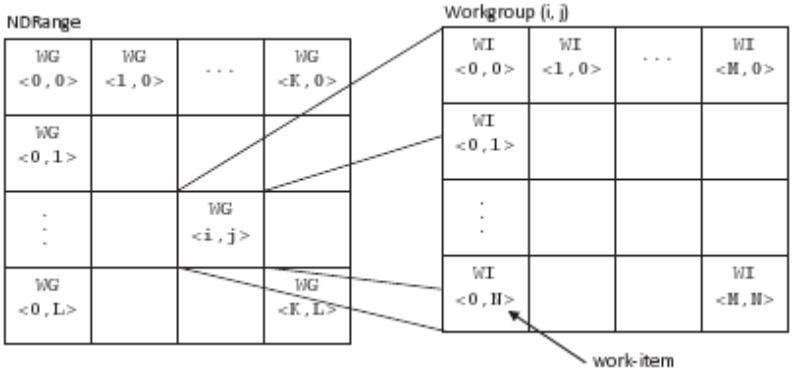


Figure 3.2.1: Partitioning of NDRange into work-groups and work-items.

Each work-group is scheduled for execution on a compute unit, with its work-items being executed concurrently by that compute unit's processing elements. Work-items therefore execute the same code, but can branch to different paths, or operate on different data.

3.3 Memory Model

In OpenCL, the memory space of a device is divided into four distinct regions.

- **Global memory:** This area is accessible by both the host and the devices. Global memory is the area on which the host stores and retrieves data manipulated by the devices. From the device perspective, this area is accessible by all work-items. Global memory is a read/write area and caching may be available depending on the capabilities of a device.

- **Constant memory:** Constant memory is a read-only region of data, initialised by the host. Again, this region is accessible by all work-items.
- **Local memory:** This is a workgroup-local region, that can be viewed as a distinct address space for each compute unit. This region is shared among a work-group's members and therefore is mainly used to allocate variables that are common to all work-items.
- **Private memory:** This region is private to individual work-items. Data in this region are not accessible by other work-items.

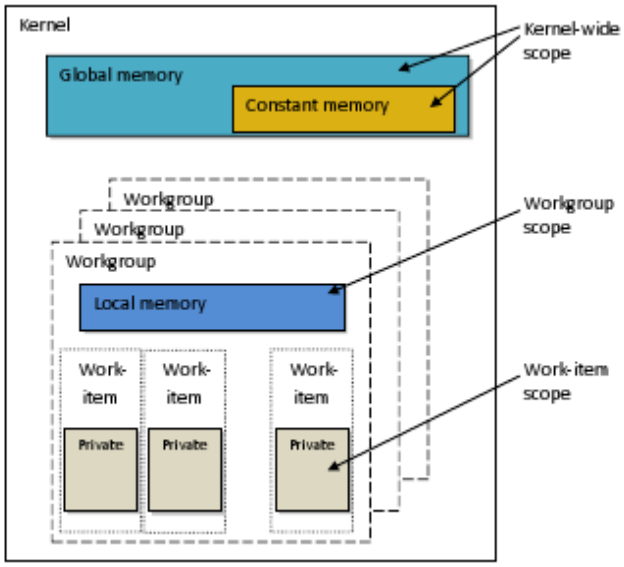


Figure 3.3.1: The OpenCL memory model.

The OpenCL API provides functions for the host and the devices to interact with the memory regions they have access to.

3.4 Programming Model

OpenCL defines two programming models; a data-parallel and a task-parallel model, along with the capability of supporting hybrids of them.

In the data-parallel model, a sequence of instructions is applied over a set of data defined over an NDRange. The data mapping between work-items and data elements can be one-to-one in a strict data parallel model, with OpenCL supporting more relaxed variations.

The task-parallel model describes the capability to execute a single kernel as a single work-item of a single work-group. Under this model, parallelism can be achieved by

using vector types defined on devices, enqueueing multiple tasks, or enqueueing native kernels.

3.5 Summary

This chapter presented an overview of the OpenCL framework. The OpenCL standard defines four models, over which OpenCL implementations are organised, namely the platform model, the execution model, the memory model and the programming model. These are abstract models that allow OpenCL to be implemented on a variety of devices, like CPUs, GPUs, APUs and FPGAs.

The next chapter presents the architecture of the NVIDIA Fermi device family, which is the architecture that the shallow water benchmark is optimised for. Understanding the Fermi architecture allows to investigate how the OpenCL models are implemented on the device, and subsequently how to optimise heterogeneous code in order to achieve higher levels of parallelism.

Chapter 4

GPUs as Hardware Acceleration Units

Although OpenCL aims to provide a portable and unified view of different hardware, to date it has not been able to provide a truly unified solution. Substantial performance improvements can be achieved by exploiting the underlying device's characteristics. The programmer therefore needs to be aware of a device's potential for optimisation, as well as its limitations. Both of them can be derived by being aware of the device's architecture. In this chapter, the details of NVIDIA's CUDA architecture will be explored, with emphasis on the Fermi architecture. Quadro 2000, a Fermi device, is the architecture the GPU version of the shallow water benchmark is optimised for. Detailed specifications of Quadro 2000 are presented in Section 7.1.

4.1 Introduction to CUDA

NVIDIA devices support GPU programming through the Compute Unified Device Architecture, widely known as CUDA. CUDA is a parallel computing software and hardware architecture, that allows C programs to be executed in GPUs through the CUDA API. Bindings are available for the most popular languages including Fortran, Java and Python. CUDA devices support alternative frameworks, like OpenCL and DirectCompute [24].

CUDA made its first appearance in 2007, with the introduction of the Tesla architecture. Since then, several features and improvements have been introduced. NVIDIA collectively refers to these features as *compute capability*. Compute capability is expressed as a version number, with the major digit corresponding to the architecture family and the minor digit to the improvement version. Tesla devices introduced compute capability 1.x. The Fermi family (2007) introduced compute capability 2.x, with the latest generation being the Kepler family (2012), that introduced compute capability 3.x. A full classification of NVIDIA devices by compute capability is provided in [25].

The CUDA model defines a *host* controlling the execution of a *program*, expressed as a group of *kernels*, into one or more *devices*. A CUDA device is organised as an array of s*treaming multiprocessors* (SMs), each one of which containing a number of *CUDA*

cores. Before executing a kernel, the number of threads that will run this kernel is defined by the programmer. Threads are conceptually organised over a *grid*. Upon execution, the grid is partitioned into *thread blocks*. The number of threads per block can also be defined by the programmer. Thread blocks are assigned global IDs within the grid, and individual threads are assigned global IDs within the grid, and local IDs within their block.

Thread blocks are scheduled for execution to the SMs by the main scheduler, referred by NVIDIA as the Giga-Thread Scheduler. Once scheduled to a SM, blocks the execution of blocks leaves the responsibility area of the Giga-Thread Scheduler and is exclusively handled by the SMs.

Multiprocessors handle blocks in *warps*, which are groups of 32 parallel threads. In particular, the multiprocessor breaks a block into warps, that are later scheduled for execution by the multiprocessor's *warp scheduler*.

Upon the execution of a warp, all threads start off from the same instruction and continue by executing one common instruction at a time, in a SIMD [26] manner. Still, the division of the device into individual SMs capable to run independent code, classifies GPUs as SIMT (Single Instruction Multiple Thread) devices [27] [28].

4.2 The Fermi Architecture

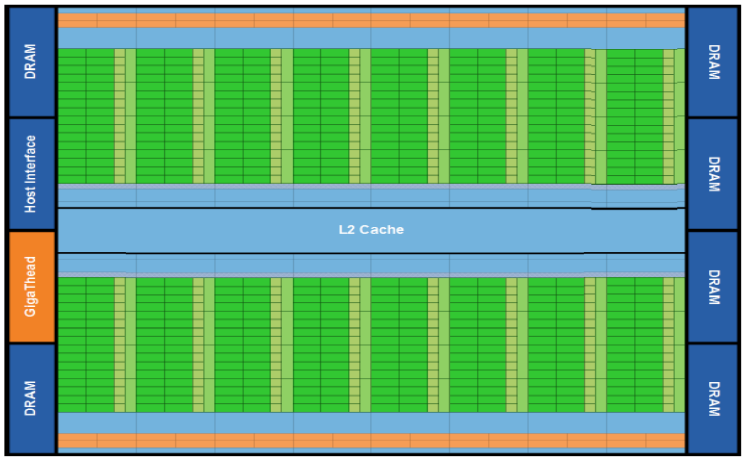


Figure 4.2.1: Diagram of the Fermi architecture.

Figure 4.2.1 [29] shows a diagram of the Fermi architecture. Fermi devices consist of an array of stream multiprocessors that share a global DDR memory and a L2 cache. The device is connected to the host through the *host interface*, which is a PCI-Express bus.

The Giga-Thread scheduler allocates thread blocks to the SMs.

4.2.1 Stream Multiprocessors

Figure 4.2.1 [27] shows the architecture of a stream multiprocessor. Each multiprocessor consists of 32 CUDA cores partitioned in 2 groups of 16. Each core is fully pipelined, that is it can fetch one instruction per clock cycle, and contains one ALU and one FPU. The FPU supports both single and double precision arithmetic.

Apart from the cores, each SM contains four *special function units* (SFUs). The SFUs are designed to execute single-precision transcendental and algebraic functions like trigonometric, reciprocal and square root. Each SFU can execute one instruction per thread per cycle, which means that four units can execute a warp in 32/4 = 8 cycles. Moreover, the SFU pipeline is decoupled from the main dispatch unit, allowing thus the issuing of other instructions while the SFU is occupied.

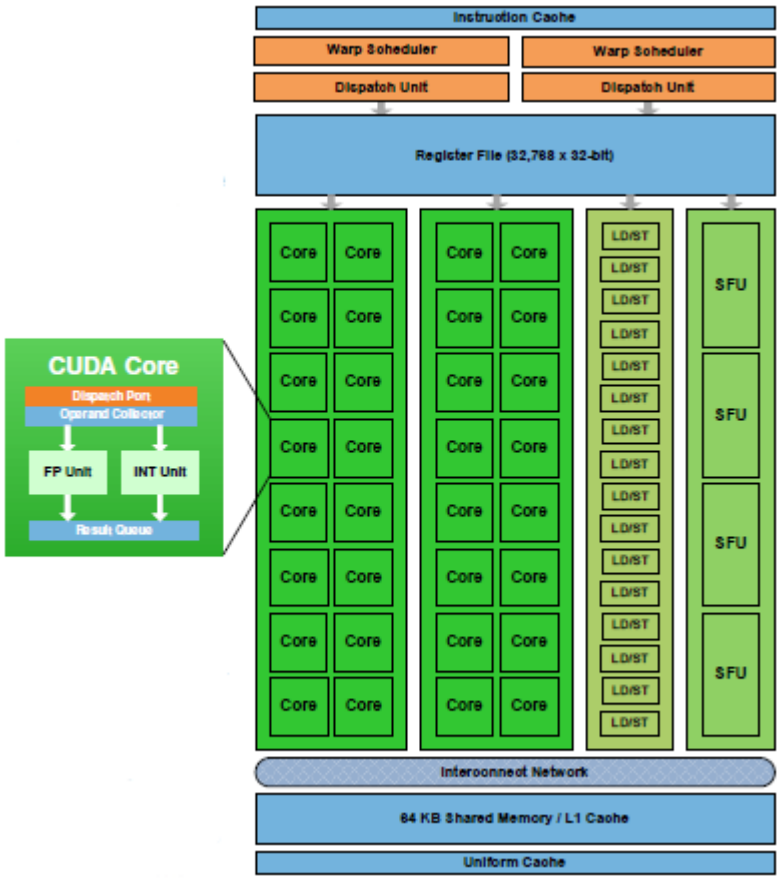


Figure 4.2.2: The architecture of a Fermi Stream Multiprocessor

A user configurable memory unit acting as a shared memory as well as a L1 cache is

shared between all cores within the same multiprocessor. This is a 64 KB on-chip unit that can be configured as a 48 KB shared memory / 16 KB L1 cache, or as a 16 KB shared memory / 48 KB L1 cache. The cache block size both for L1 and L2 caches is 128 B. A common register file of 32 K, 32-bit registers is shared between cores.

Each multiprocessor hosts 16 *load/store units*, each one of which serves addresses from/to the main memory or the cache. The load/store units can calculate an address in one clock cycle and are therefore able to calculate addresses for 16 threads (half-warp) in one cycle.

4.2.2 Memory Hierarchy

4.2.2.1 Host Interface

Communication between the host and the devices is performed through the host interface. In Fermi devices, this is a 16-lane PCI-Express v2.x bus, with a theoretical throughput of 8GB/s. This is considerably lower than the device's global memory throughput, which, depending on the device model, is in the order of tens of GB/s. For this reason, transfers between the host and the device are the most significant factor of overhead in GPGPU programs. Given the high throughput of GPUs, it is therefore preferred to perform a computation on the device rather than to transfer data to the host, even in the cases where a kernel does not perform any better than a CPU-based equivalent program. Moreover, given the overhead associated with each transfer, it is preferable to transfer data in one large batch than to perform many small transfers. Some common techniques to improve transfer performance is to use pinned memory, asynchronous transfers, or overlapping transfers [30].

4.2.2.2 Global Memory

Global memory is the largest and slowest storage within the device, with a latency between 400 and 800 cycles [27]. In Fermi devices, global memory is implemented as a set of DDR5 memory modules. The mapping between memory addresses and the modules is not linear, so that typical memory accesses will not be stalled by falling into the same partition [31]. Access to global memory is coalesced into 128-byte blocks. In compute capability 2.0 devices, memory requests are issued in groups of 32 threads, so optimal performance is achieved when 32 threads attempt to access a continuous area of 128 bytes. It should be mentioned here, that the access pattern of threads within a block,

does not affect coalescing.

4.2.2.2 L2 Cache

Caches were introduced in devices of compute capability 2.0. A L2 cache is fit between the main memory and the SMs. This is a unified cache, that uses a LRU replacement policy and guarantees that threads have a coherent view of data. All loads and stores go through the L2 cache. This includes host accesses, which aside from the obvious benefits, might also have some negative impacts, like cache pollution, when data are transferred asynchronously between the host and the device [31]. The L2 cache and the global memory are the only means of communication among threads of different SMs. Communication between threads residing on different SMs is therefore an inherent drawback of the CUDA architecture.

4.2.2.3 L1 Cache

As described earlier (Section 4.2.1), each SM features a configurable memory chip that hosts both the L1 cache and the shared memory. L1 caches the thread data structures, like the stack. As opposed to the L2 cache, L1 is not designed for temporal locality, meaning that it does not use a LRU policy. Load instructions attempt to find a line in L1, and then in L2. If both fail, a coalesced global memory is issued. When a line is evicted from L1, it is written to L2. From there, it is written to global memory only once invalidated. L1 is read-only, that is writes are performed straight to L2. Therefore, L1 can only enhance read performance [31].

4.2.2.4 Shared Memory

Shared memory is arranged in 32 banks, each one of which is 32-bit wide. The throughput of each bank is 32 bits per two cycles. Successive words are stored in successive banks, so the ideal access pattern is for 32 threads to access 32 different banks. If multiple threads attempt to access different words within the same bank, a bank conflict occurs. In that case, conflicting accesses are serialised and one thread is serviced at a time. If multiple threads, within the same warp, access the same word at the same time, no bank conflict occurs. The word is broadcast to the requesting threads. On the contrary, if multiple threads attempt to write to the same address, only one of them will succeed. Which thread it will be, is undefined [27]. Devices of compute capability 2.0 have been enhanced to support the access of double-precision 64-bit

numbers without serialisation, as long as the accessing threads cause no bank conflicts.

4.2.2.5 Registers and Local Memory

Registers are the fastest and most limited memory of the device. Each SM hosts a register file of 32 K, 32-bit registers, which are shared among the SM's resident threads. The compiler assigns most automatic variables into registers. Some special cases like data structures that would consume too many registers or register spillages are store into local memory. Local memory is only conceptually local to each thread. It is actually stored on the main device memory, and its access is hence slow. Local memory is only accessed for automatic variables.

4.2.2.6 Other Memories

There are a some more memories available in CUDA devices, like constant and texture memory, that are not relevant to this work. For simplicity, they won't be discussed here.

4.2.3 Thread Scheduling

Depending on the number of blocks a grid is partitioned to, the Giga-Thread scheduler allocates one or more blocks to each SM. There is a limited number of blocks and threads each SM can support, and that number depends on the device's compute capability. Once a block is allocated to a SM, it is the multiprocessor's responsibility to handle its execution. In fact, SMs are individual units that handle all their resources independently. This decoupled architecture between the main scheduler and the SMs allows devices to scale to large numbers of multiprocessors and provides current code with the benefit of increased execution speed on future devices.

Stream multiprocessors partition blocks into warps of 32 threads. The partitioning scheme is straightforward, with blocks being serialised by row, and consecutive threads being allocated to each warp into groups of 32.

Fermi multiprocessors feature a *dual-warp scheduler*, that essentially consists of two warp schedulers attached to two separate dispatch units. The first scheduler handles odd-id warps, while the second scheduler handles warps with even-id. At instruction issue time, each scheduler chooses a warp with *active threads*, that is threads that are ready to execute. A thread might not be ready to execute if it is waiting for memory access or earlier issued instructions to complete.

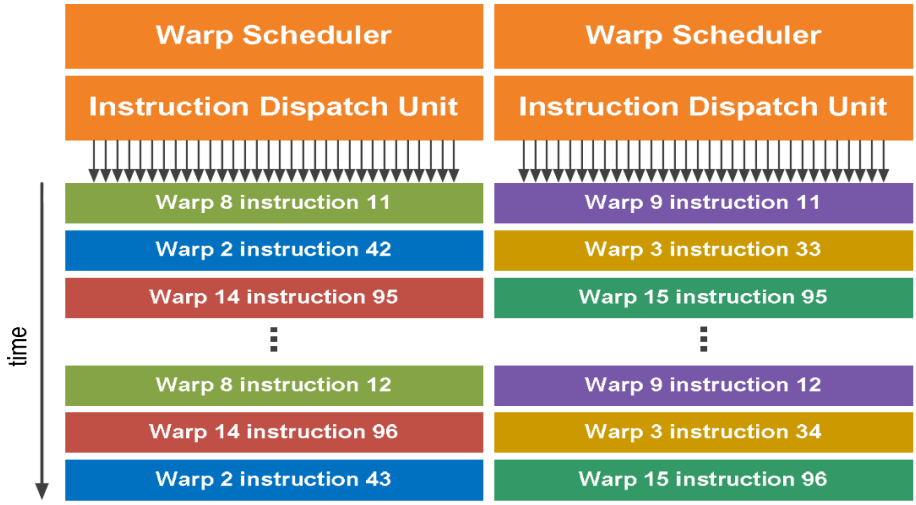


Figure 4.2.3: Scheduling of instructions from different warps, by the dual-warp schedulers.

Each dual-warp scheduler can schedule instructions from different warps, as shown in Figure 4.2.3 [27]. Each scheduler can issue one instruction at a time, to half of the cores of the SM. This means that in order to execute an instruction for the whole warp, the scheduler needs to issue the instruction twice, which gives a peak throughput of 2 warps / 2 cycles. The issued instructions can be of any type (ALU, FP, SFU, load/store) or combinations of them. There is a limitation in double-precision instructions, that can only be issued by one scheduler at a time.

Context switching between different warps comes at no cost, since each warp's context is maintained on-chip. A common register file is partitioned among the warps and a shared memory is partitioned among the different blocks within the same multiprocessor. Context switching occurs on instruction issue time, when the warp-scheduler chooses a warp with active-threads.

4.2.4 Scalability and Levels of Parallelism

In CUDA devices, parallelism is performed by executing multiple threads on SMs. The partitioning of a computation into fixed blocks of threads and the structuring of devices as arrays of multiprocessors, is the key behind the scalability of CUDA devices. Additionally, the decoupled relationship between the GigaThread scheduler and the SMs is highly scalable, given that the GigaThread scheduler only needs to know which SMs are ready to be allocated new blocks.

Within multiprocessors, parallelism is further performed on thread and instruction level. Thread level parallelism (TLP) is achieved by keeping processors busy with active

threads, while other threads are waiting for memory access. As long as there are active threads available, parallelism will therefore be high, and memory latency will be hidden. Occupancy is a metric of TLP, and is defined as the ratio of active warps to the maximum number of resident warps in a multiprocessor:

$$Occupancy = \frac{number\ of\ active\ warps \text{ in } SM}{max.\ number\ of\ resident\ warps \text{ in } SM}$$

Given that a multiprocessor's resources are shared by the resident threads, the maximum possible occupancy depends on the maximum number of blocks that can reside on the SM, and the amount of resources consumed by the resident blocks. In the case where the resources required by blocks exceed the multiprocessor's capabilities, the number of resident blocks is reduced, and so is occupancy.

Instruction level parallelism (ILP) is expressed by the capability of SMs to execute subsequent instructions once the current instruction requires memory access, as long as there are no data dependencies between them. Thread execution will therefore stall only when there are no more independent instructions to available for execution. ILP is further expressed by the decoupled design of SFUs, that can independently execute instructions without stalling the CUDA cores. ILP is another way to hide memory latency, and its effectiveness depends on the number of independent instructions per kernel.

4.3 The Implementation of OpenCL on Fermi Devices

Conceptually speaking, there is a close relationship between the OpenCL and CUDA models. Both OpenCL and CUDA are based on the same platform model fundamentals: A kernel executing a program on one or more devices. The host in both cases is a CPU.

The execution model is essentially the same, with only differences in terminology. More specifically, upon a kernel's execution, an index space of instances (NDRange/Grid) is partitioned into blocks (Work-groups/Thread-blocks), that consist of individual execution units (Work-Items / Threads). Table 4.3.1 summarises the correspondence of two models, in terms of terminology.

From the implementation point of view, CUDA devices implement OpenCL's compute units (CUs) by stream multiprocessors (SMs), and OpenCL's processing elements are implemented by CUDA cores.

OpenCL's conceptual memory hierarchy is implemented by the different memories found in CUDA devices. OpenCL's global memory is implemented by CUDA's device memory, and can be accessed both by the host and the device. In Fermi devices, constant memory is also located in the device memory. There is a conflict in terminology between OpenCL and CUDA when it comes to local memory. OpenCL defines local memory as a region common to all work-items within the same work-group. This is implemented by CUDA's shared memory. On the other hand, what is defined as local memory in CUDA is the region where large data structures and register-spilled variables are hosted. Together with the register file, these two implement OpenCL's private -per thread- memory. Table 4.3.2 summarises theses relationships.

OpenCL	**CUDA**
NDRange	Grid
Work-group	Thread Block
Work-item	Thread

Table 4.3.1: Mapping of OpenCL and CUDA concepts.

OpenCL	**CUDA**
Global Memory	Global Memory
Local Memory	Shared Memory
Constant Memory	Constant Memory
Private Memory	Local Memory / Registers

Table 4.3.2: Mapping of OpenCL and Fermi memories.

4.4 Summary

This chapter provided a concise description of NVIDIA's Fermi architecture. The Fermi family specifics have been covered is some detail, including the scheduling mechanism and the memory hierarchy. The similarities between the OpenCL and the CUDA models, as well how OpenCL is implemented by Fermi devices was discussed. The next chapter presents the methodology used for the parallelisation of the multi-core and many-core implementations, together with the means of evaluation of the experimental results.

Chapter 5

Methodology and Evaluation of Experimental Results

This chapter summarises the methodology used in the parallelisation of the multi-core and many-core implementations and presents the metrics and conventions used when evaluating the experimental results.

5.1 Methodology

The parallelisaton of the multi-core version is driven by the classification and optimisation of a set of overheads, as presented in [32]. Nevertheless, it should be noted that a complete performance modelling analysis should not be expected in this work, as it goes beyond the scope of this thesis. The target architecture is first presented in detail, followed by the description of a naïve implementation and its performance results. The process of optimisation involves the (i) identification, (ii) analysis, and (iii) minimisation of the most significant performance overheads. Each overhead is presented in a different section together with the optimisations performed in order to minimise its effects. The achieved performance of each optimisation is quantified with results. After all optimisations are performed, the overall results are evaluated and discussed.

The parallelisation process of the heterogeneous version follows a different route. Due to the fundamentally different nature of the GPU architectures, the main overhead is known a-priori to be communication between threads. Of course, other overheads of smaller significance are also present. Vendors provide a set of optimisations that may be performed in order to mitigate these overheads. The applicability of these optimisations depends on the algorithm. The optimisation of the many-core implementation is thus based upon the application of best practices in an algorithm-specific way. Similarly to the multi-core version, the target architecture is first presented. A naïve implementation follows, together with the associated results. The main categories of optimisations are then presented, followed by discussion of their applicability to the shallow water algorithm. Subsequent sections describe the effect of different optimisations, both in terms of analysis and achieved performance. Finally, the overall results are evaluated and discussed.

5.2 Evaluation of Results

Both CPU and GPU versions are evaluated with respect to scalability and performance. When comparing performance, effort has been made to present results as fairly as possible, by following the best practices described in [33] and [34]. Common pitfalls that can lead to misleading results are described in [35] and [36].

Execution time, it is expressed as elapsed wall-time. This is preferred to CPU time, as the latter hides various latencies caused by scheduling, paging, locks and I/O. Elapsed time is also what is perceived by people when running a program, and therefore what is ultimately targeted by introducing parallelism.

The execution time of each implementation was measured using an appropriate timer. The performance of the sequential version was timed using gettimeofday(). The host machine was configured to use the timestamp counter (TSC) for this function. TSC is a double-precision, 64-bit register that counts cycles since reset. When it comes to single-processor applications, TSC provides a high-precision, low-overhead solution. Nevertheless, in multi-core environments, issues related to timer synchronisation between cores emerge [37]. Given the above, in the multi-core version, OpenMP's omp_get_wtime() is used instead, which returns the time for the calling thread. The timer precision is shown by omp_get_wticks(). In the GPU implementation, the TSC timer was used for the measurement of the overall execution time, while Quadro's timers were used for various profiling-related measurements. In OpenCL, GPU timers are accessible through a set of API calls that return profiling information for a given kernel. Quadro's timing precision is of the order of nanoseconds.

The multi-core implementation is evaluated with respect of execution time and temporal performance. Execution time is compared relative to a naïve ideal performance curve calculated as Ts/P, where Ts is the execution time of the sequential implementation and P is the number of processors (cores) the benchmark is executed over. Temporal performance is used to graphically represent performance. Temporal performance is expressed as the number of solutions of a problem per second, and is calculated as P/Ts.

The GPU implementation is evaluated through its floating-point performance. Floating point performance is measured in Mflops/s, where one Mflop is equal to 10^6 floating-

point operations [38]. On the presented results, Mflop/s are calculated by dividing the total number of floating point operations performed during the computation of the problem (referred on the tables as Mflop) by the execution time. Mflop/s is a commonly used performance metric, given that many scientific applications involve heavy workloads of this type. Although it is not an absolute metric for performance, since performance depends on many other factors, it is particularly useful in order to expose amounts of overhead when comparing achieved performance to the theoretical computational capabilities of a device.

When evaluating each version's final performance results, a comparison is made with both the single-processor version, as well as with respect to a reference implementation. The multi-core version is compared to an MPI [39] based implementation, that has been developed by Graham Riley and co-workers. The GPU-based version's performance is compared to that of the multi-core implementation, which is where the final conclusions of this work are extracted from.

5.3 Summary

The methodology used in the parallelisation of the multi-core and many-core implementations was described. The multi-core version will be optimised using an overhead-based approach. A set of standard optimisations will be used for the GPU implementation. The means used to evaluate the experimental results as accurately as possible were summarised, in terms of best practices, timers, metrics and the reference versions that will be used for comparison. This chapter completed the background required, before presenting the parallelisation of the shallow water benchmark. The next chapter presents a parallel implementation on a multi-core architecture, that will be used as a reference for comparison between the CPU and GPU based approach.

Chapter 6

Parellelisation on a Multi-core Architecture

6.1 The Target Architecture: AMD Opteron

The target architecture is a typical NUMA architecture, that consists of 4 AMD Opteron 8378 processors. Each processor contains 4 cores running at 2.4 GHz, giving 16 cores in total. Each core features a 64 KB L1 data cache, a 64 KB L1 instruction cache and a 512 KB L2 unified cache. A 6 MB L3 unified cache is shared on-chip, between all cores. The cache block size is 64 B. Processors are linked using AMD's Bi-directional Coherent Hypertransport Technology interface, with a total bandwidth of 8 GB per link (4GB per direction). Each processor is connected to six DDR2 units of 1 GB each, summing up to 24 GB in total. The architecture is shown in Figure 6.1.1 [40].

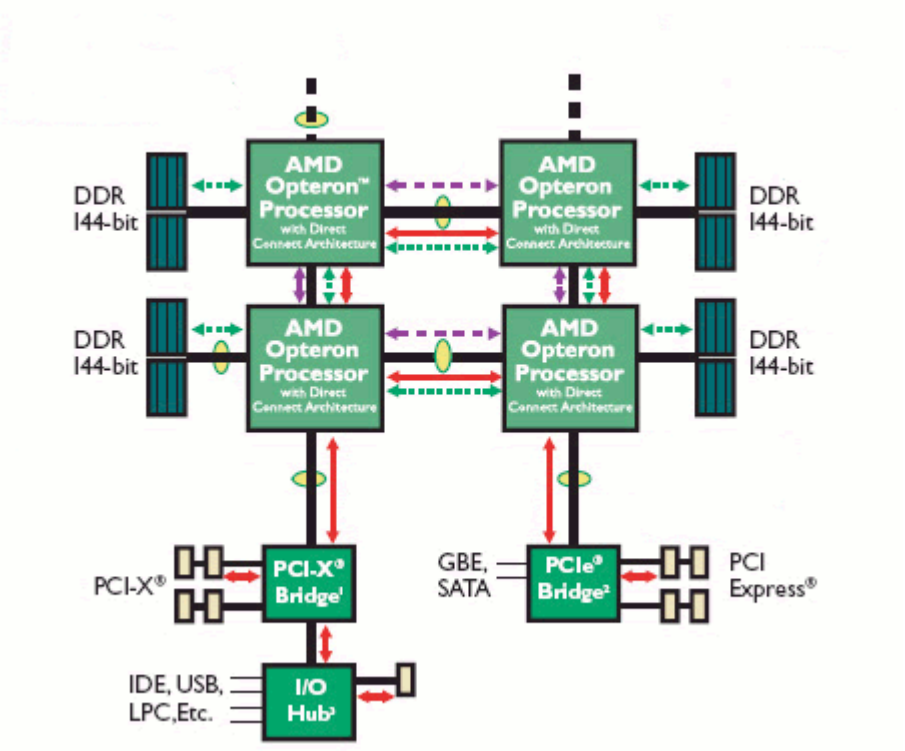


Figure 6.1.1: The multi-core implementation's target architecture.

Upon execution of the shallow water benchmark, core affinity is set so that each thread is assigned to a physical core, with individual threads spread among processors when possible, as shown in Table 6.1.1. This scheme ensures that scheduling, context-

switching and cache coherency overheads are kept to a minimum. Memory hierarchy resources are thus utilised as efficiently as possible. This setup was experimentally verified to provide the best performance among various number of threads, for all different problem sizes.

Number of Threads	Processor 1 (Cores C1-C4)				Processor 2 (Cores C1-C4)				Processor 3 (Cores C1-C4)				Processor 4 (Cores C1-C4)			
	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
1	T1															
2	T1				T2											
4	T1				T2				T3				T4			
8	T1	T2			T3	T4			T5	T6			T7	T8		
16	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T12	T14	T15	T16

Table 6.1.1: The processor affinity scheme used in the multi-core implementation. Cx represents the core number, Tx is the thread number.

Since each thread is assigned to a single core, depending on the problem size and the number of threads, data may fit in different caches. Table 6.1.2 shows the total amount of data involved in each problem size, as deduced by adding up 13 arrays of 8-byte doubles (Section 2.2).

Problem Size	Data Size	Array Size
128x128	1.625 MB	128 KB
256x256	6.5 MB	512 KB
512x512	26 MB	2 MB
1024x1024	104 MB	8 MB

Table 6.1.2: Data involved per problem size.

Keeping in mind the processor affinity scheme described above, each thread will be assigned a subset of the total data. These data may or may not fit into the running core's L1 or L2 cache. Moreover, for the cases where multiple threads are running on the same processor, they will have to share the L3 cache. Table 6.1.3 shows the amount of data allocated to each thread and processor, highlighting the cases where allocated data fit into a cache. Unified caches (L2 and L3) are assumed to be data dominated, given that the entire binary of the shallow water benchmark is less than 30 KB. As expected, the impact of memory access will be the least in small problem sizes where all data fit in some cache.

Problem Size	Number of Threads	Data per Processor	Data per Thread	Comments
128x128	1	1.625 MB	1.625 MB	Processor data fit in L3.
	2	832 KB	832 KB	Processor data fit in L3.
	4	416 KB	416 KB	Processor data fit in L3. Thread data fit in L2.
	8	416 KB	208 KB	Processor data fit in L3. Thread data fit in L2.
	16	416 KB	104 KB	Processor data fit in L3. Thread data fit in L2.
256x256	1	6.5 MB	6.5 MB	
	2	3.25 MB	3.25 MB	Processor data fit in L3.
	4	1.625 MB	1.625 MB	Processor data fit in L3.
	8	1.625 MB	832 KB	Processor data fit in L3.
	16	1.625 MB	416 KB	Processor data fit in L3. Thread data fit in L2.
512x512	1	26 MB	26 MB	-
	2	13MB	13 MB	-
	4	6.5 MB	7.5 MB	-
	8	6.5 MB	3.75 MB	-
	16	6.5 MB	1.875 MB	-
1024x1024	1	104 MB	104 MB	-
	2	52 MB	52 MB	-
	4	26 MB	26 MB	-
	8	26 MB	13 MB	-
	16	26 MB	6.5 MB	-

Table 6.1.3: Partitioning of data per computational unit.

6.2 An Implementation in OpenMP

A naïve parallel version has been developed in OpenMP. This version attempts to exploit the potential for concurrency over the computation of each array's individual elements (Section 19). This has been realised by statically scheduling the iterations of each second-level for-loop[1] among threads. For instance, the loop performing the computation of cu, becomes:

```
#pragma omp parallel for schedule(static)
for(i=0; i < M; i++)
  for(j=0; j < N; j++)
    cu[i+1][j] = 0.5 + (p[i][j+1] + p[i][j]) * u[i+1][j];
```

Listing 6.2.1: OpenMP parallelisation of the computation of cu.

The above directive will cause the iterations of the outer loop to be split into equal groups, each one of which is allocated to one thread. For instance, the execution of a problem with M=N=128 running over 4 threads, will sequentially allocate 32 iterations to each thread, that is 0-31 to the first thread, 32-63 to the second thread etc. Each thread will then iterate over all 128 values of N, computing one element at a time.

Examination of the shallow water algorithm reveals that, in any computation, elements are accessed in one of the following patterns: x[i][j], x[i+1][j], x[i][j+1], x[i+1][j+1], where i, j are sequential iteration indices ranging between 0 to M-1 and 0 to N-1, respectively. This means that each array will be accessed by threads in vertical blocks of elements, with each block being accessed by one thread. This concept is important, since each block can reside on a different processor's memory for the cases where more than one processors are used. Communication will only be required for each block's border elements, as well as during the computation of the periodic continuation.

The performance of the first naïve implementation is shown in Table 6.2.1.

1 Counting from the loop that iterates over the entire set of computations

Problem Size		Number of Threads				
		1	2	4	8	16
128x128	Ideal Time (sec)	3.131	1.565	0.782	0.391	0.196
	Actual Time (sec)	3.789	5.513	6.957	7.967	8.227
	Overhead (sec)	0.658	3.947	6.174	7.575	8.081
256x256	Ideal Time (sec)	16.051	8.025	4.012	2.006	1.003
	Actual Time (sec)	19.185	22.695	31.296	30.322	31.355
	Overhead (sec)	3.134	14.669	27.283	28.315	30.351
512x512	Ideal Time (sec)	107.812	53.096	26.953	13.476	6.738
	Actual Time (sec)	135.674	118.506	117.091	118.323	119.721
	Overhead (sec)	27.862	64.6	90.138	104.846	112.982
1024x1024	Ideal Time (sec)	496.797	248.398	124.199	62.099	31.049
	Actual Time (sec)	541.048	534.889	594.668	658.799	673.872
	Overhead (sec)	44.251	286.49	470.468	596.699	642.822

Table 6.2.1: Performance timings of naïve multi-core implementation.

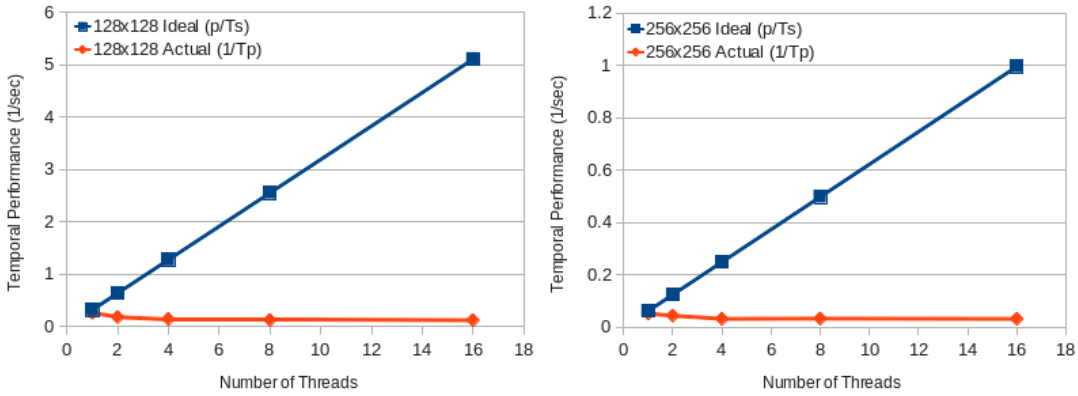


Figure 6.2.1: Performance curves of the naïve implementation.

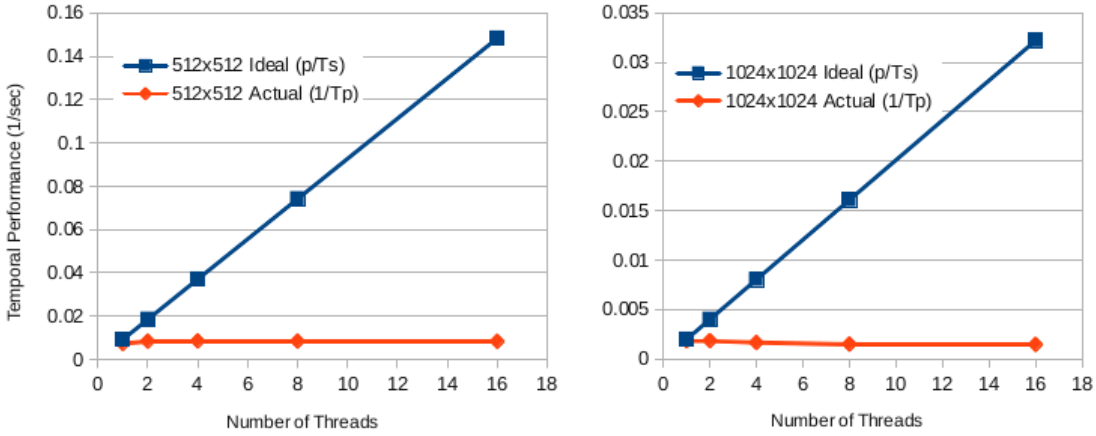


Figure 6.2.2: Performance curves of the naïve implementation.

Figures 6.2.1 and 6.2.2 show the achieved performance with respect to the naïve ideal. Compared to the naive ideal, this implementation does not perform as well as expected. Scalability is poor, and performance deviates around that of the sequential version, in all problem sizes. The effects of overheads are detrimental to this implementation. An observation worth to mention here is that although OpenMP offers important productivity benefits compared to raw thread programming, it cannot substitute understanding of the underlying architecture.

6.3 Optimisation

As described in Section 5.1, the optimisation of this implementation is based on overhead analysis, as presented in [32]. In particular, the overheads investigated are classified into the following categories [41]:

1. **Insufficient Parallelism Overhead:** Introduced when there is insufficient work to keep processors busy.
2. **Algorithmic Overhead:** Occurs when extra work is introduced by the parallel algorithm.
3. **Load Imbalance Overhead:** Occurs when different amount of work allocated to each processor, causing the total execution time to depend on the busiest thread's completion time.
4. **Scheduling Overhead:** This overhead is caused by thread initialisation and termination, as well as with the selection of which computation should be performed by which thread.
5. **Synchronisation Overhead:** Caused by thread synchronisation, barriers etc.
6. **Communication Overhead:** Caused by accessing the main memory and the caches.
7. **Compiler Overhead:** Caused by differences in compiler generated code between the sequential and parallel versions.

In this implementation, the algorithmic and compiler overheads are assumed to be negligible. Given OpenMP's directive-based approach and the simplicity of the shallow water benchmark, no algorithmic changes were introduced on the parallel version. Compiler overhead is dependant upon the OpenMP implementation and its investigation

can be a complex task. For now, it is also expected to be minimal. The validity of these assumptions is experimentally verified in the next section.

6.3.1 Scheduling Overhead

In the naïve implementation, threads are created and terminated on every second-level for-loop[2]. This indicates a possibility that there is significant cost associated with thread management. During each one of the 4,000 main loop's iterations, threads are created, allocated computations, and terminated as many times as the number of second-level loops, that is 19 times. This amounts to 3*19*4,000 = 228,000 thread management operations in total. Even if the kernel performs some kind of optimisation, the possibility of having significant thread scheduling overheads is high.

Problem Size	**Sequential Implementation** (sec)	**Parallel Implementation over 1 Thread** (sec)	**Overhead** (sec)
128x128	3.131	3.789	0.658
256x256	16.051	19.185	3.134
512x512	107.812	135.674	28.754
1024x1024	496.797	541.048	42.645

Table 6.3.1: Overhead comparison between the sequential and the naïve multi-core implementation.

Table 6.3.1 shows a significant difference between the performance of the sequential implementation and the naive multi-core version running over one thread. This difference reflects the cost of parallelism, and should contain the scheduling overhead, as well as any compiler and algorithmic overheads.

Table 6.3.2 shows the achieved performance, after decoupling the thread creation statements from the second-level for loops, and replacing them by a single statement in the beginning of the main loop. Threads are now created before the main loop starts, and are terminated after all computations are complete. This optimisation is expected to reduce the thread management operations, and only leave overheads related to the assignment of computations to threads.

The observed reduction of overhead when executing over one thread, verifies that there was indeed a penalty associated with thread creation and termination. Moreover, the fact that the performance of the single-threaded runs is very close to the ideal, verifies

2 Counting from the loop that iterates over the entire set of computations

the assumption that the impact of algorithmic and compiler overheads are insignificant.

Problem Size		Number of Threads				
		1	**2**	**4**	**8**	**16**
128x128	Ideal Time (sec)	3.131	1.565	0.782	0.391	0.196
	Actual Time (sec)	3.111	1.556	0.922	0.745	1.04
256x256	Ideal Time (sec)	16.051	8.025	4.012	2.006	1.003
	Actual Time (sec)	16.012	6.672	3.479	2.179	2.725
512x512	Ideal Time (sec)	107.812	53.096	26.953	13.476	6.738
	Actual Time (sec)	108.226	62.552	20.26	9.512	6.744
1024x1024	Ideal Time (sec)	496.797	248.398	124.199	62.099	31.049
	Actual Time (sec)	495.548	365.314	247.607	186.234	156.272

Table 6.3.2: Performance timings of the multi-core implementation, after thread scheduling optimisations.

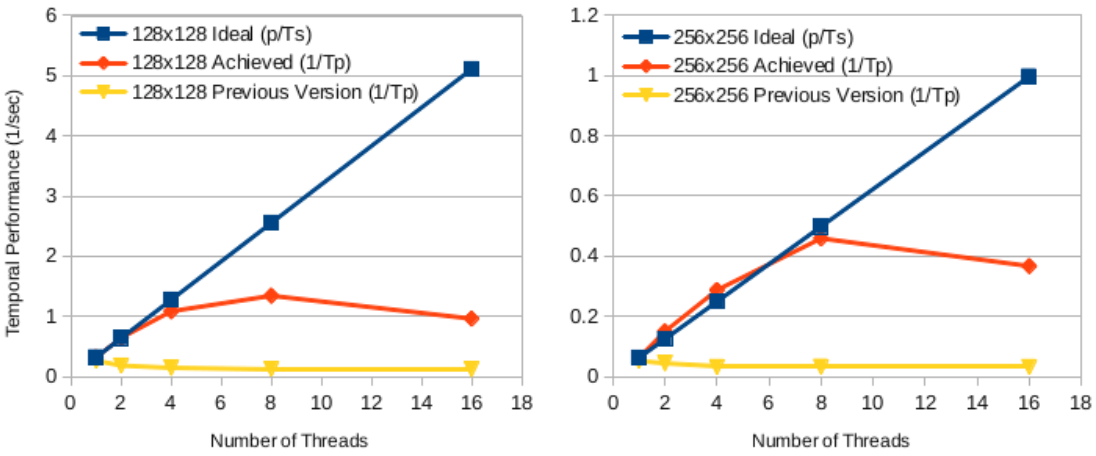


Figure 6.3.1: Performance curves of the multi-core implementation after thread scheduling optimisations.

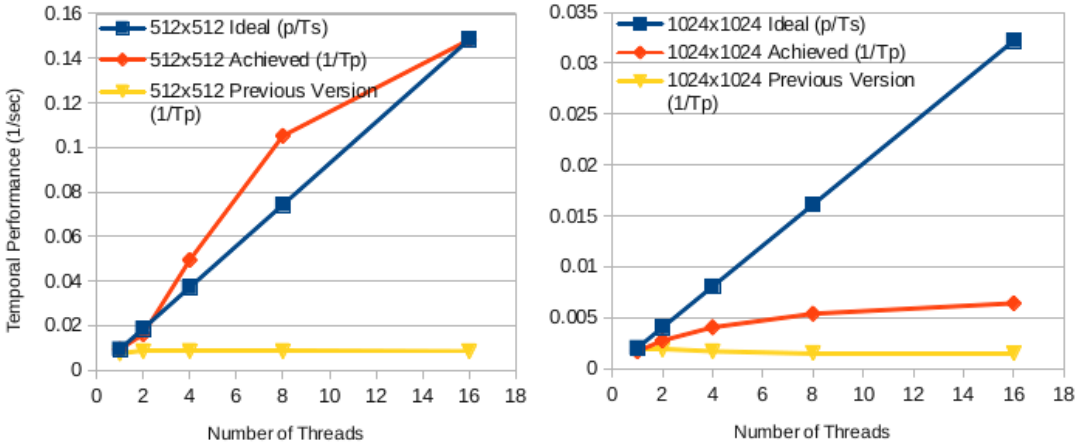


Figure 6.3.2: Performance curves of the multi-core implementation after thread scheduling optimisations.

Figures 6.3.1 and 6.3.2 show the difference in performance between the naïve ideal, the optimised version and the initial, naïve implementation for each problem size. As opposed to the initial implementation, performance now improves, up to a limit, when the problem is run over multiple threads. Single-threaded runs either approach or supersede the ideal performance. The effect of superlinear speedup is apparent in some runs, peaking at the 512x512 problem running over 8 threads. There are still some cases where performance is less than the ideal, with performance over 16 threads being affected the most, in all problem sizes, and performance in 1024x1024 being the lowest, overall. These overheads will be investigated in the following sections.

6.3.2 Synchronisation Overhead

According to the OpenMP standard, *loop* constructs insert an implicit barrier at the end of the loop, in order for threads to synchronise, unless the *nowait* clause is specified. As described in Section 2.3, except from element-level, some computations can also be performed concurrently in array-level, before periodic continuation is applied. Eliminating the implicit barriers within these computations is expected to remove the overhead of the implicit barriers. Table 6.3.3 shows the achieved performance timings, after the elimination of the barriers.

Problem Size		**Number of Threads**				
		1	**2**	**4**	**8**	**16**
128x128	Ideal Time (sec)	3.131	1.565	0.782	0.391	0.196
	Actual Time (sec)	3.049	1.518	0.846	0.667	0.95
256x256	Ideal Time (sec)	16.051	8.025	4.012	2.006	1.003
	Actual Time (sec)	16.703	6.599	3.404	2.064	2.636
512x512	Ideal Time (sec)	107.812	53.096	26.953	13.476	6.738
	Actual Time (sec)	107.756	62.316	20.041	8.905	6.084
1024x1024	Ideal Time (sec)	496.797	248.398	124.199	62.099	31.049
	Actual Time (sec)	496.172	373.352	253.651	187.076	154.141

Table 6.3.3: Performance timings of the multi-core implementation after synchronisation optimisations.

The achieved performance curves with respect to the previous version and the naïve ideal, are shown in figures 6.3.3 and 6.3.4. The effect of barriers was generally small, and the performance benefit varied between different resolutions. The 512x512 resolution benefited the most, while the 1024x1024 being nearly invariant to this

optimisation. This shows that probably there is another overhead currently dominating the execution of the 1024x1024 version, masking-out the potential improvement of the current optimisation. This hypothesis raises questions on whether these overheads would appear more clearly if optimisations were performed in a different order.

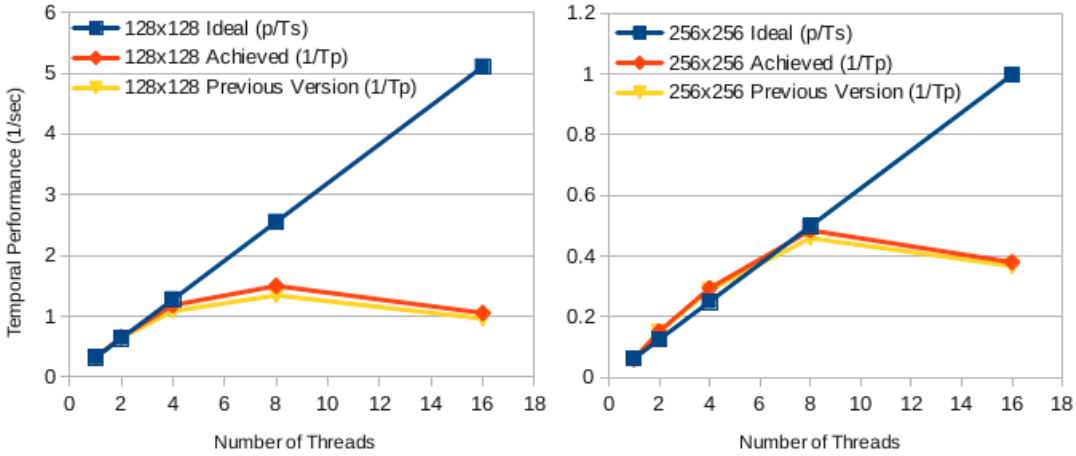


Figure 6.3.3: Performance curves of the multi-core implementation after synchronisation optimisations.

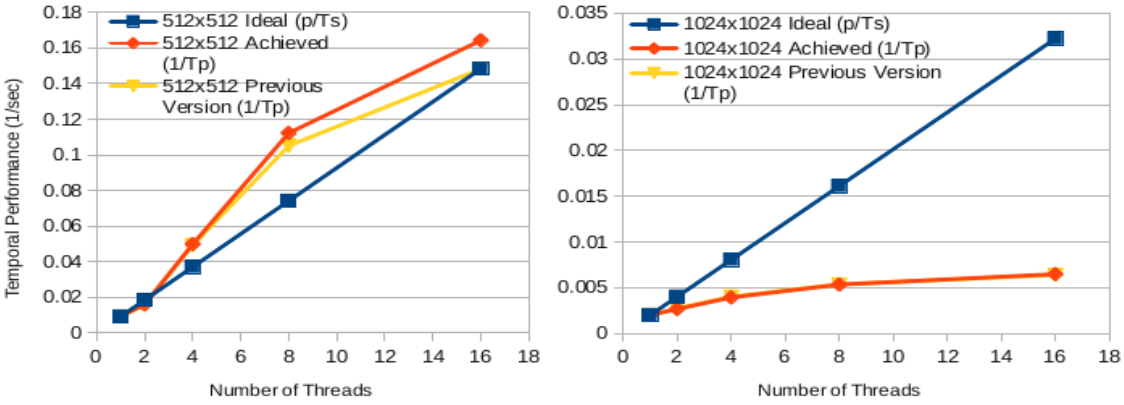


Figure 6.3.4: Performance curves of the multi-core implementation after synchronisation optimisations.

6.3.3 Communication Overhead

Under the current implementation, initialisation of data is performed by the master thread. After the initialisation is complete, all data are stored into the master thread processor's main memory and possibly one or more caches. Threads residing in remote processors will need to fetch data from the remote memory, leading into communication overhead. This overhead is getting larger for those cases where data will not fit in the reading thread processor's L3 cache. In that case, threads will have to perform remote memory accesses in every iteration of the main loop.

Running the initialisation of data in parallel, causes data to reside in each processor's local memory. Communication cost is thus expected to be reduced significantly, especially for those cases where data fit into some cache. The achieved performance of this optimisation are shown in Table 6.3.4.

Problem Size		Number of Threads				
		1	2	4	8	16
128x128	Ideal Time (sec)	3.131	1.565	0.782	0.391	0.196
	Actual Time (sec)	3.091	1.525	0.827	0.593	0.684
256x256	Ideal Time (sec)	16.051	8.025	4.012	2.006	1.003
	Actual Time (sec)	15.968	6.675	3.383	1.95	2.132
512x512	Ideal Time (sec)	107.812	53.096	26.953	13.476	6.738
	Actual Time (sec)	107.852	57.812	18.123	8.602	4.936
1024x1024	Ideal Time (sec)	496.797	248.398	124.199	62.099	31.049
	Actual Time (sec)	495.317	339.481	175.583	111.985	104.096

Table 6.3.4: Performance timings for the multi-core implementation after communication optimisations.

Figures 6.3.5 and 6.3.6 show the achieved performance curves, with respect to the previous version and the naïve ideal. For most problem sizes, the effect of this optimisation is mostly apparent on 16 thread runs. An exception to this is the 1024x1024 resolution, where the effect is apparent for 4 threads and above. This is expected as the 1024x1024 version would be the most affected by communication. The fact that the 128x128 version did not exhibit the expected performance, given data fit into L2 cache, indicates that this version is dominated by some other type of overhead.

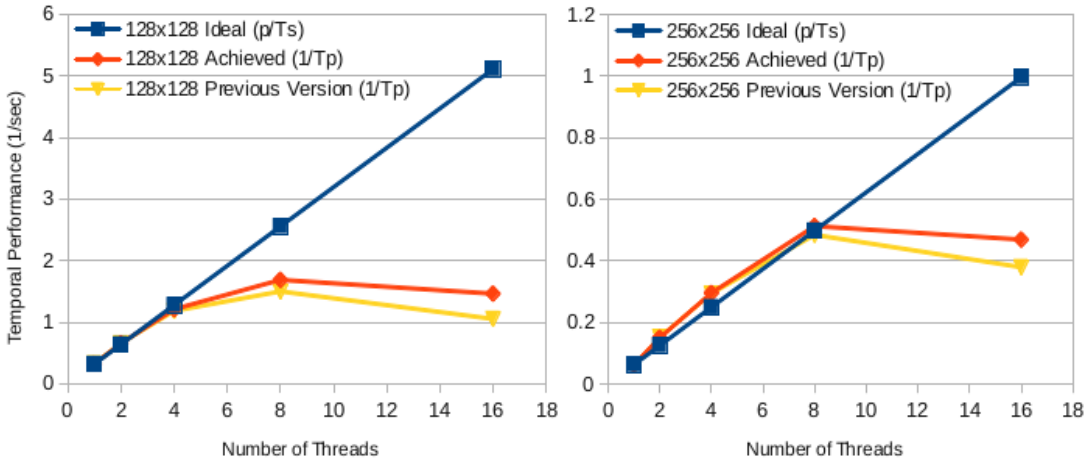


Figure 6.3.5: Performance curves for the multi-core implementation after communication optimisations.

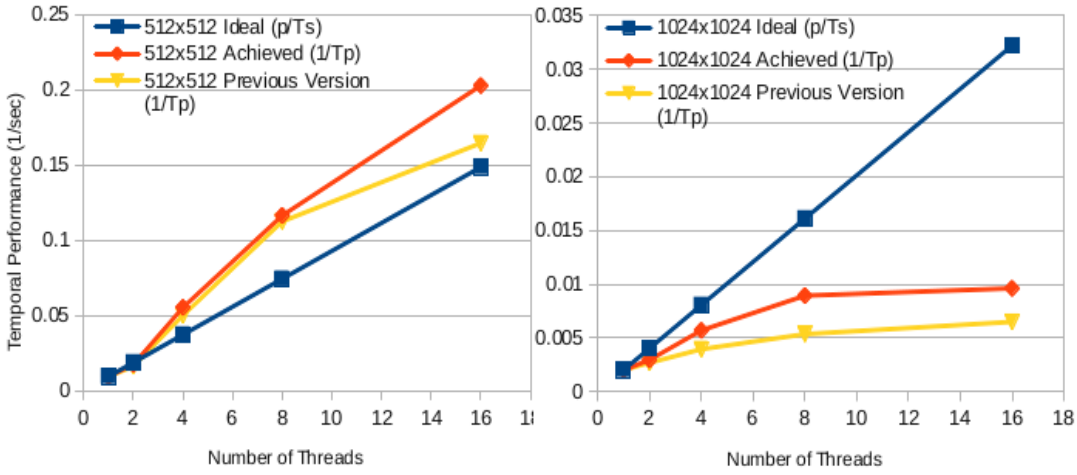


Figure 6.3.6: Performance curves for the multi-core implementation after communication optimisations.

6.3.4 Load Balancing Overhead

So far, the partitioning and assignment of loop iterations to threads has been done automatically by the compiler. In OpenMP, not specifying explicitly the chunk size, can potentially lead to load imbalance. Regarding this issue, the OpenMP standard [42] states the following:

> *"For a team of p threads and a loop of n iterations, let n / p be the integer q that satisfies n = p*q - r, with 0 <= r < p. One compliant implementation of the static schedule (with no specified chunk_size) would behave as though chunk_size had been specified with value q. Another compliant implementation would assign q iterations to the first p-r threads, and q-1 iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation."*

In the case of the shallow water benchmark, there is a potential for imbalance on the partitioning of the second-level loops. Loops computing the problem variables are iterated over M times, while the loop performing time smoothing is iterated over M+1 times. Upon inspection, for a problem size of 128x128 running over 4 threads, the blocks were allocated by the compiler as shown in Table 6.3.5.

Dimension	Block 0	Block 1	Block 2	Block 3
M = 128	0 - 31 (32 elements)	32 - 63 (32 elements)	64 - 95 (32 elements)	96 - 127 (32 elements)
M +1 = 129	0 - 33 (33 elements)	33 - 65 (33 elements)	66 - 98 (33 elements)	99 - 128 (30 elements)

Table 6.3.5: Automatic allocation of blocks by the OpenMP compiler.

Before specifying a fixed chunk size, it is important to realise how the compiler allocates the remaining elements in the case where the n / p division is not perfect. The most expected scenarios would be either the compiler to allocate the remaining elements to the first thread, or to allocate the remaining elements in a round-robin fashion. In order to investigate this issue, three test cases were considered, for statically allocated block sizes of 31, 32 and 33. The allocation of these blocks as performed by the compiler, is shown on the tables 6.3.6, 6.3.7 and 6.3.8.

Dimension	**Block 0**	**Block 1**	**Block 2**	**Block 3**
M = 128	0 – 30 124 – 127 (35 elements)	31 – 61 (31 elements)	62 – 92 (31 elements)	93 – 123 (31 elements)
M +1 = 129	0 – 30 124 - 128 (36 elements)	31 – 61 (31 elements)	62 – 92 (31 elements)	93 – 123 (31 elements)

Table 6.3.6: Automatic allocation of elements for block size = 31.

Dimension	**Block 0**	**Block 1**	**Block 2**	**Block 3**
M = 128	0 - 31 (32 elements)	32 - 63 (32 elements)	64 - 95 (32 elements)	96 – 127 (32 elements)
M +1 = 129	0 – 31, 128 (33 elements)	32 – 63 (32 elements)	64 – 95 (32 elements)	96 – 127 (32 elements)

Table 6.3.7: Automatic allocation of elements for block size = 32.

Dimension	**Block 0**	**Block 1**	**Block 2**	**Block 3**
M = 128	0 - 32 (33 elements)	33 - 65 (33 elements)	66 - 98 (33 elements)	99 - 127 (29 elements)
M +1 = 129	0 - 32 (33 elements)	33 - 65 (33 elements)	66 - 98 (33 elements)	99 - 127 (29 elements)

Table 6.3.8: Automatic allocation of elements for block size = 33.

The results show that the compiler allocates the remaining elements to the first thread. The best load balance is therefore achieved for 32 elements, as shown in Table 6.3.7. As a generic rule, the best load balance is achieved when partitioning in blocks of M/NUM_THREADS, provided that M is exactly divisible by the number of threads. Table 6.3.9 shows the resulting performance after explicitly specifying the chunk size.

Figures 6.3.7 and 6.3.8 show a how performance is shaped with respect to the previous version and the naïve ideal. In all problem sizes, performance is improved proportionally to the number of threads, with the greatest improvement being observed

on 16 thread runs. This is expected, since load imbalance does not only result into delayed completion time based on the performance of the slowest thread, but also into increased communication overhead. When parallelising loops that run over different numbers of iterations (M and M+1 in this case), load imbalance will cause certain blocks being allocated to different threads on each of the above cases. These blocks will be copied from one cache to the other, and that would introduce additional communication overhead. Declaring a fixed chunk size in both loops causes the same data to be allocated to the same threads every time.

Problem Size		**Number of Threads**				
		1	**2**	**4**	**8**	**16**
128x128	Ideal Time (sec)	3.131	1.565	0.782	0.391	0.196
	Actual Time (sec)	3.034	1.503	0.785	0.451	0.294
256x256	Ideal Time (sec)	16.051	8.025	4.012	2.006	1.003
	Actual Time (sec)	16.024	6.61	3.21	1.659	0.894
512x512	Ideal Time (sec)	107.812	53.096	26.953	13.476	6.738
	Actual Time (sec)	108.513	54.288	17.816	8.047	3.684
1024x1024	Ideal Time (sec)	496.797	248.398	124.199	62.099	31.049
	Actual Time (sec)	536.751	308.588	154.158	99.345	71.166

Table 6.3.9: Performance timings for the multi-core implementation after load balancing optimisations.

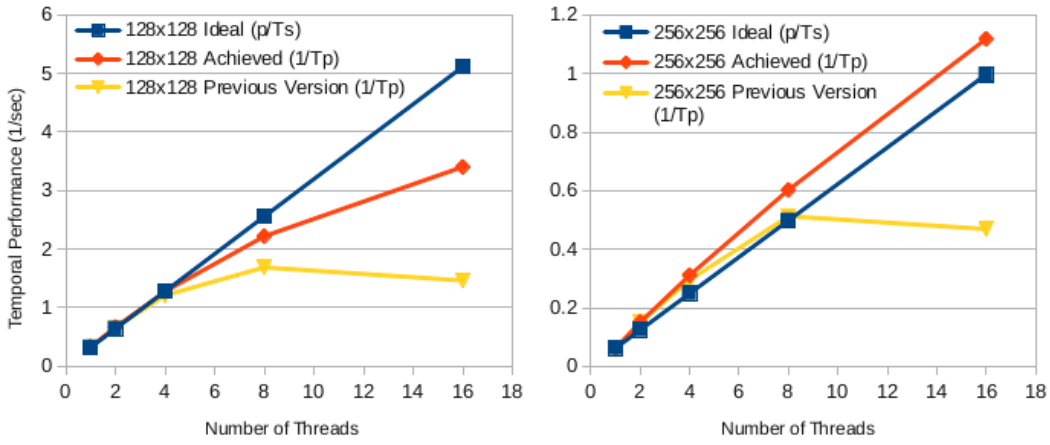


Figure 6.3.7: Performance curves for the multi-core implementation after load balancing optimisations.

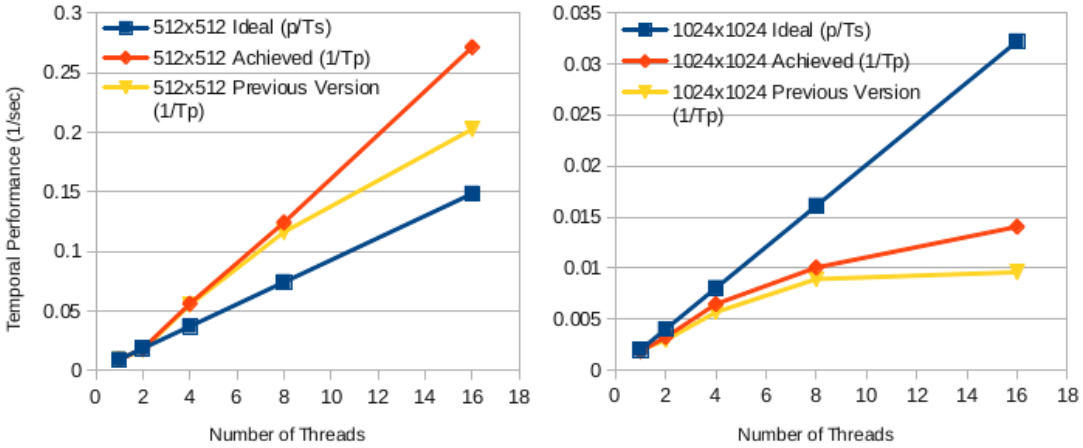


Figure 6.3.8: Performance curves for the multi-core implementation after load balancing optimisations.

6.4 Results

Overall, performance has improved with respect to the original, naïve implementation. Overheads have been in most cases eliminated or even turned to negative (Figure 6.4.1). As shown in Figures 6.3.7 and Error: Reference source not found, peak performance is observed on the 256x256 and 512x512 problem sizes. Both versions scale well with respect to the number of threads, and may potentially scale over more threads.

The worst scalability is observed on the 1024x1024 problem. This is most probably caused by the increased communication cost. The problem is too large to fit any cache. Moreover, computations accessing past the border of their allocated block, involve reading from other processor's memories, leading to increased communication cost. Additional effects are observed when writing the computed elements. Cache lines containing past the border elements need to reside into two different caches, which results into false sharing on write operations. These cost are two times bigger than the 512x512 version, as these accesses are performed for 1024 elements per block. As expected, this cost increases proportionally with the number of threads, since the more the threads the more the communication is required between them. Hence the decreased performance on 16-threaded runs. A possible improvement would be to consider the possibility of modifying the algorithm so that the off-boundary accesses are eliminated. This is left as future work.

On the contrary to the 1024x1024 problem size, it is rather surprising that the 128x128

version does not exceed the ideal performance, given that, in most cases, the entire problem fits into both the L2 and L3 cache (Table 6.1.3). This may be attributed to insufficient parallelism; the problem is too small to exploit the potential of the system. These cases show the two extremes when it comes to memory performance and core utilisation. High memory performance does not yield good results without high core utilisation, and vice-versa.

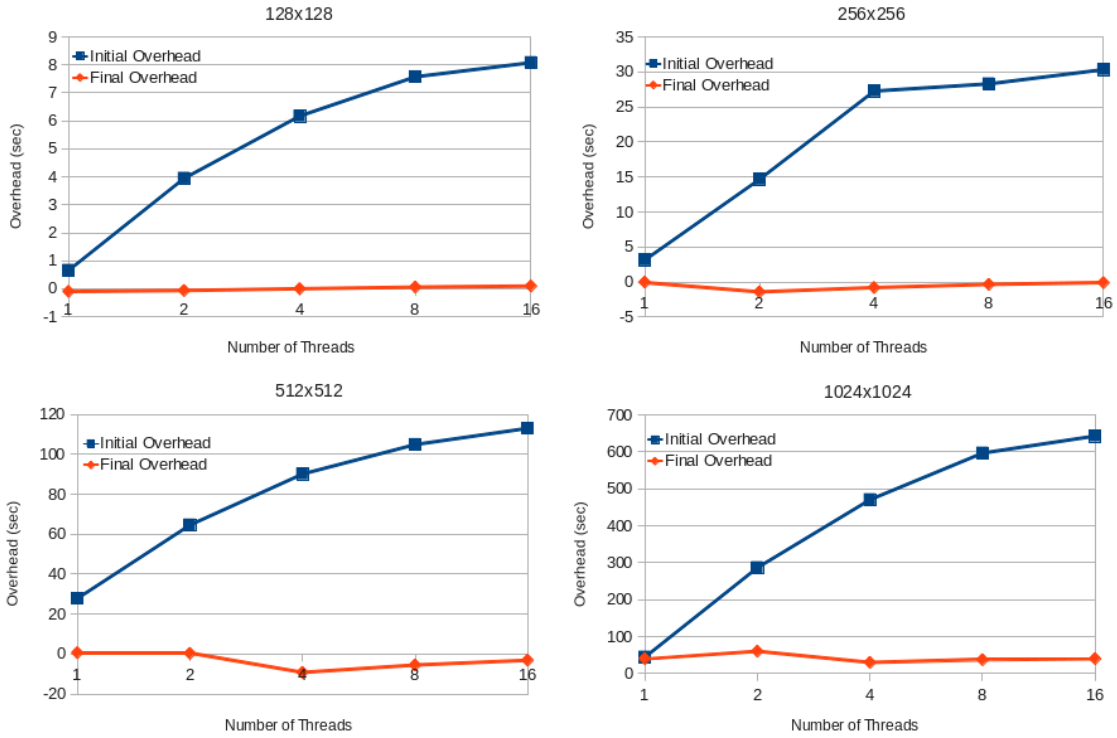


Figure 6.4.1: Overhead comparison between the initial and final versions of the multi-core implementation.

A comparison of the achieved performance compared to the equivalent implementation in MPI is presented in Figure 6.4.2. In most problem sizes, the performance of each implementation is very close to each other, with the OpenMP performing slightly better in all occasions. A performance difference is observed on the 1024x1024 resolution, with OpenMP superseding the MPI implementation significantly. This can be attributed to the cost of message exchange of MPI, which increases proportionally with the resolution and the number of threads.

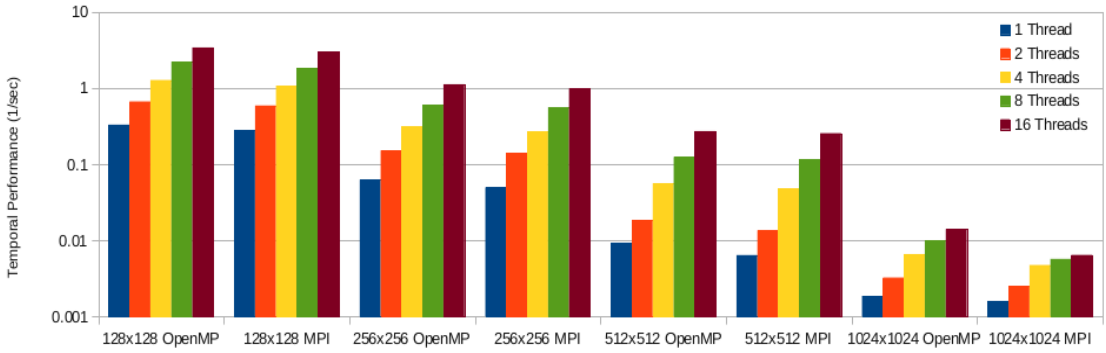


Figure 6.4.2: Performance comparison between the OpenMP and MPI implementations for various problem sizes.

6.5 Summary

This section presented the implementation of the multi-core version. The development process was based on a naïve implementation, that was gradually optimised through the analysis of a series of overheads. Although the results of the naïve implementation were rather poor, the optimisation process eliminated most of them. The highest performance was achieved on the 256x256 and the 512x512 problem sizes, where the effect of superlinear speedup was observed. On the contrary, smaller performance than expected was observed on the 128x128 problem size. This divergence is attributed to insufficient parallelism. The 1024x1024 problem size is the point where this implementation does not scale any further. Compared to the MPI implementation, this version performed very close. Slight differences in performance favouring the OpenMP version are attributed to the inherent communication cost of MPI, an effect that is more apparent on the 1024x1024 problem size. Finally, some interesting remarks arose during the development process. These are discussed on the conclusions chapter.

Chapter 7

Parallelisation on a Heterogeneous Architecture

7.1 The Target Architecture: Quadro 2000

The target architecture is a 4-core superscalar Intel Xeon machine running at 3.30 GHz, with 8 GB RAM, featuring an NVIDIA Quadro 2000 GPU. Quadro 2000 is a mid-range Fermi device of compute capability 2.1, featuring some improvements over devices of compute capability 2.0. The theoretical peak performance of the device for single-precision floating-point calculations is 480 Gflop/s. Official documentation about the double-precision performance was not found, and thus it remains to be an open question. Unofficial sources state it to be up to 1/2 of the single-precision performance.

The device consists of 4 stream multiprocessors (SMs), each one featuring 48 cores, giving 192 cores in total. Main memory is seen as a 128-bit DDR5 unit of 1GB, with theoretical bandwidth of 41.6 GB/s. A L2 cache of 256 KB is shared between SMs. The device interfaces with the host through a 16x PCIExpress 2.0 bus, with theoretical bandwidth of 8 GB/s.

Within each multiprocessor, cores are partitioned into 3 groups of 16 cores. There are 2 dual-warp schedulers, each one of which can issue 2 instructions per half-warp per cycle, giving a maximum throughput of 3 warps / 2 cycles[3]. Each multiprocessor features 8 SFUs, able to execute a warp in 4 cycles. A memory unit is split between shared memory and L1 cache. OpenCL programs see this as a 48 KB shared memory unit and a 16 KB L1 cache. Finally, a 32 KB register file is shared between all resident blocks.

7.2 An Implementation in OpenCL

Following the multi-core implementation's approach, a naïve implementation is developed in OpenCL, that forms the basis over which subsequent optimisations are applied. The idea behind this implementation is to transfer expensive computations to the GPU. The following kernels are defined:

3 The way the three 16-core groups are partitioned between the two schedulers is an open question, as no documentation found on that subject.

- **kernel_l100:** computation of cu, cv, z and h.
- **kernel_l100_pc:** periodic continuation for cu, cv, z and h.
- **kernel_l200:** computation of unew, vnew, pnew.
- **kernel_l200_pc:** periodic continuation for unew, vnew, pnew.
- **kernel_l300:** time smoothing of uold, vold, pold and updating of u, v, p for the next iteration.

These kernels correspond to data-parallel tasks, that can be executed concurrently before synchronisation is required, as was described in Section 2.3 (Figure 2.3.2). Alternative initial approaches were considered, namely breaking down individual computations into smaller kernels to be executed either synchronously or asynchronously. Both approaches have led to lower performance. This is caused by the fact that in larger kernels, some elements are being reused among calculations, and thus are benefited by being cached or stored into registers.

Under the current implementation, kernels are designed so that each thread computes one element of the problem variables described above. In each case, the NDRange is thus equal to the problem size. Listing 7.2.1 shows the source code of kernel_l100.

```
int x = get_global_id(0);
int y = get_global_id(1);

cu[(y+1)*M_LEN+x] = 0.5 * (p[(y+1)*M_LEN+x] + p[y*M_LEN+x]) *
                    u[(y+1)*M_LEN+x];


cv[y*M_LEN+x+1] = 0.5 * (p[y*M_LEN+x+1] + p[y*M_LEN+x]) *
                  v[y*M_LEN+x+1];

z[(y+1)*M_LEN+x+1] = (fsdx * (v[(y+1)*M_LEN+x+1] – v[y*M_LEN+x+1])-
                      fsdy * (u[(y+1)*M_LEN+x+1] – u[(y+1)*M_LEN+x]))
                      / (p[y*M_LEN+x] + p[(y+1)*M_LEN+x] +
                      p[(y+1)*M_LEN+x+1] + p[y*M_LEN+x+1]);

h[y*M_LEN+x] = p[y*M_LEN+x] + 0.25 * (u[(y+1)*M_LEN+x] *
               u[(y+1)*M_LEN+x] + u[y*M_LEN+x] * u[y*M_LEN+x] +
               v[y*M_LEN+x+1] * v[y*M_LEN+x+1] + v[y*M_LEN+x] *
               v[y*M_LEN+x]);
```

Listing 7.2.1: Source code of kernel_l100.

In order to minimise transfers between the host and the device, arrays are initialised on the device and no transfers occur back to the host during the computation of the problem. Data are transferred back to the host once computation is complete. Kernels

created for the initialisation of data are not described here, since their performance is not important, given that it is not part of the measured execution time. In a multi-GPU implementation, the initialisation of data would be more interesting as data would have to be spread on the global memories of different devices.

After data initialisation is complete, the host iterates through the main loop and queues up the execution of the above kernels. The host does not wait for each kernel to complete execution before it queues up the next one. The ordered execution of kernels is handled by the queue. Specifically, the associated queue is an in-order queue, which means that kernels are executed in the order they were queued. Synchronisation is implicitly performed at the end of each kernel's execution.

Performance results of this implementation are displayed on Table 7.2.1. Similarly to the previous implementations, performance is measured after data initialisation is complete. The time taken to transfer data from the device back to the host is included in measurements.

Problem Size	**Time** (sec)	**Mflop** (4000 cycles)	**Mflop/s** (4000 cycles)
128x128	1.036	4,260	4,111
256x256	3.937	17,040	4.328
512x512	15.475	68,158	4,404
1024x1024	58.379	272,629	4,669

Table 7.2.1: Performance results for the naïve version of the heterogeneous implementation.

The obtained results show performance being in the order of 4 GFlop/s. This is considerably lower then the theoretical performance of the device, but also expected, since the computations of each kernel involve a large amount of memory accesses. This argument may be best illustrated by example. Considering the computation of the velocity over the x axis:

$$cu[\ i+1][j] = 0.5 + (p[i+1][j] + p[i][j]) * u[i+1][j]$$

This computation involves 3 floating point operations and four memory accesses. Assuming double-precision floats and the specifications of Quadro 2000 (theoretical peak performance of 240 Gflop/s, main memory bandwidth ~ 40 GB/s), this rate would lower down performance to 3.75 GFlop/s, that is 1.5% of the theoretical peak performance. In order to completely hide memory latency in this device, a computation to communication ratio of 8/1 is required. Of course, this is an experimental method

that ignores many factors that affect the perceived throughput, but it still provides an idea about the expected latency. Table 7.2.2 shows the ratio of computations to global memory accesses, for each kernel.

Kernel	Computations	Memory Accesses	Ratio
l100	100	27	3.70
l100_pc	35	24	1.45
l200	100	26	3.84
l200_pc	22	18	1.22
l300	57	21	2.71

Table 7.2.2: Computation to global memory access ratio for each kernel of the naïve heterogeneous implementation.

The computation to communication ratio is rather low, with kernels featuring many data transfers (periodic continuation kernels and l300), exhibiting the lowest rate.

Table 7.2.3 shows the amount of data involved in each problem size. It is clear that the 256 KB L2 cache is too small for the size of data involved in the shallow water benchmark. The entire problem does does not fit into L2, in any problem size. Additionally, in most problem sizes L2 is too small to fit even a single array. Therefore, the benefit of caching is expected to be small. While some values might be cached from time to time, it is hard to predict which accesses will be cached without knowing the specifics of the cache implementation.

Problem Size	Array Size	Total Size
128x128	128 KB	1.625 MB
256x256	512 KB	6.5 MB
512x512	2 MB	26 MB
1024x1024	8 MB	104 MB

Table 7.2.3: Data involved per problem size.

From the above it is clear that the limiting factor of this implementation is the global memory access.

7.3 Optimisation

As discussed in Section 5.1, the optimisation of this implementation will be based on standard optimisations targeted to the CUDA architecture. These optimisations are documented in [30] and are grouped into the following main categories:

- **Memory Optimisations:** Aiming maximising memory throughput, by minimising access to global memory.
- **NDRange Optimisations:** Aiming hiding memory latency by increasing thread-level parallelism.
- **Instruction Optimisations:** Targeting minimising the cost of cycle consuming instructions, by using faster alternatives implemented on the SFU, or by trading-off precision for performance.
- **Control Flow Optimisations:** Related to how GPUs natively handle branching, these optimisations target on minimising idling threads, due to divergence.

From the above categories, only the first two are applicable for the shallow water benchmark. Control flow optimisations are not relevant as all conditionals are eventually eliminated during optimisation. The execution of conditionals by Fermi devices is described in Appendix I.

Instruction optimisations are also not relevant, since the accelerated kernels only consist of conventional computations like multiplications and additions. No computations are therefore executed by the SFUs. Additionally, all instruction related optimisations, including those related to the execution of common multiply-add (MAD) operations, are only implemented for single-precision floating point numbers, and thus are not applicable on the shallow water code. Experimentation of the effect of converting the shallow water benchmark to use single-precision floats is presented in Appendix II.

In many GPU based implementations, optimisations like the ones described above, are not necessarily applied directly. Sometimes, modifications on the structure of the code are required before these optimisations are applied, as patterns applicable on CPU-based implementations are not necessarily the optimal ones for GPU architectures [43] [44]. The simple structure of the current algorithm's computations provides limited space for this type of modifications.

7.3.1 Optimisation of Occupancy

Maximising occupancy allows the hiding of global memory access. Before anything else, occupancy depends on the number of blocks that reside on a multiprocessor. Specifying the block size explicitly allows to calculate the maximum number of blocks that will fit in the SM. In practice, each SM will host as many blocks as it is allowed by

its resources, which depends on the resources used by threads. Calculation of occupancy depends therefore on balancing three factors: 1) The block size, 2) The amount of shared memory allocated to each block, and 3) The number of registers used by each thread. The optimal balance between those three can be obtained by experimentation. An example of calculating occupancy for kernel l100, is shown in Appendix III.

Given the amount of calculations that need to be performed, NVIDIA provides an occupancy calculator tool, in the form of a spreadsheet. This tool can be used to quickly calculate the effects of altering each of the relevant parameters. Figures 7.3.2, 7.3.1 and 7.3.3, provided by the occupancy calculator, show the impact of varying the relevant parameters, that is the block-size, the number of registers and the amount of shared memory used.

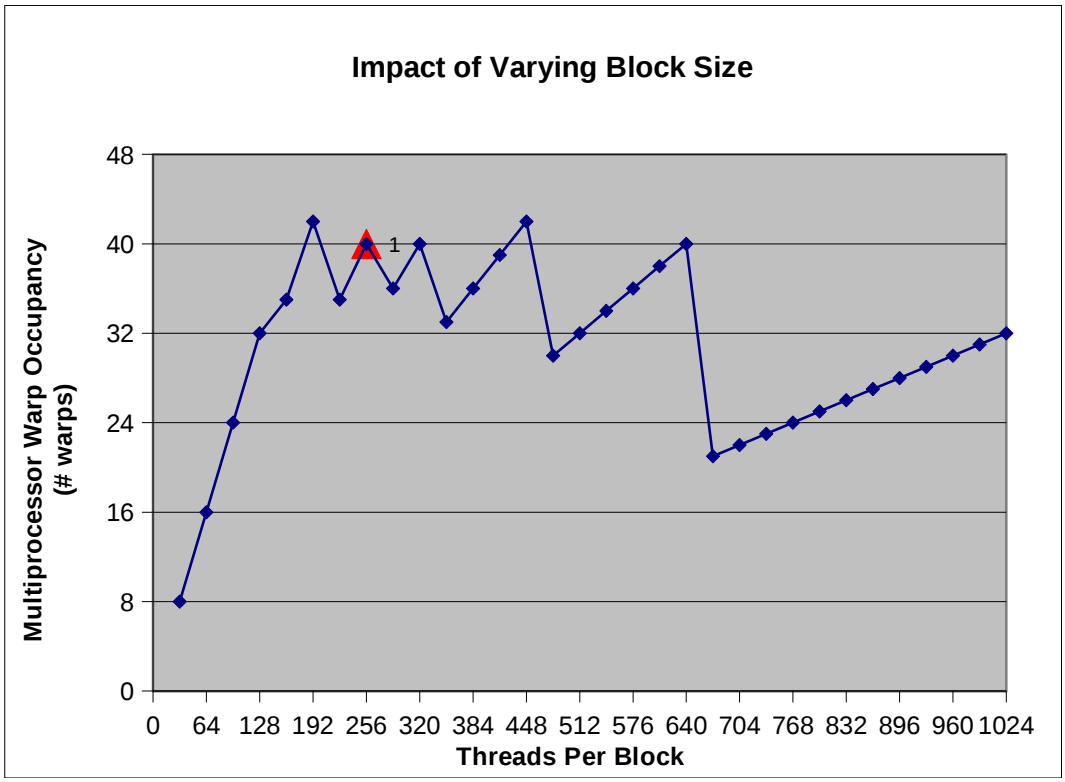


Figure 7.3.1: Occupancy impact of varying block size.

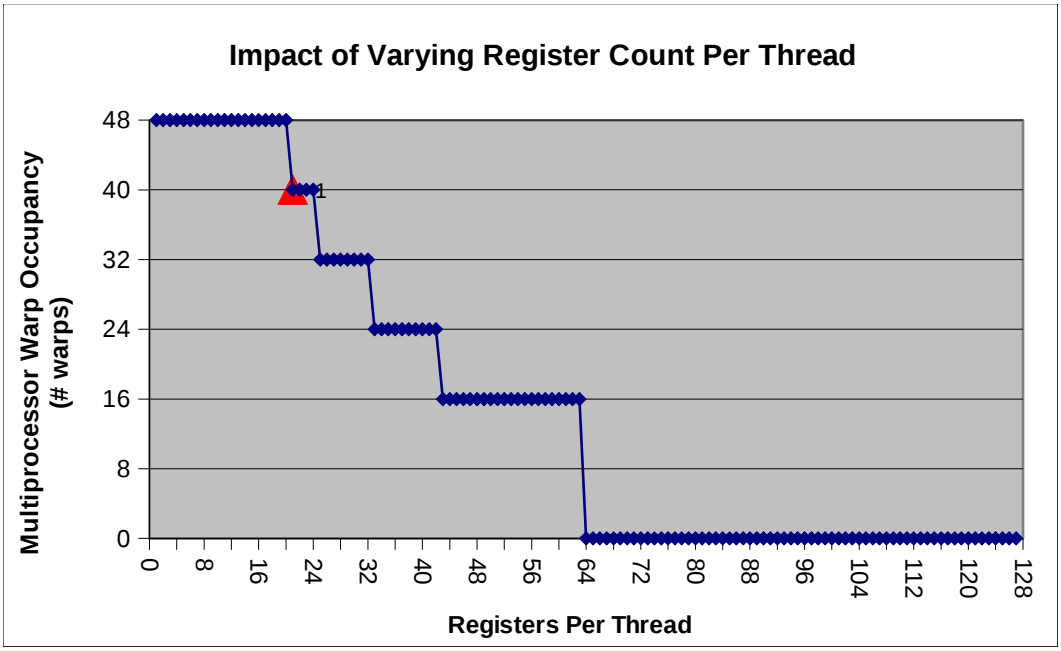


Figure 7.3.2: Occupancy impact of varying register usage per thread.

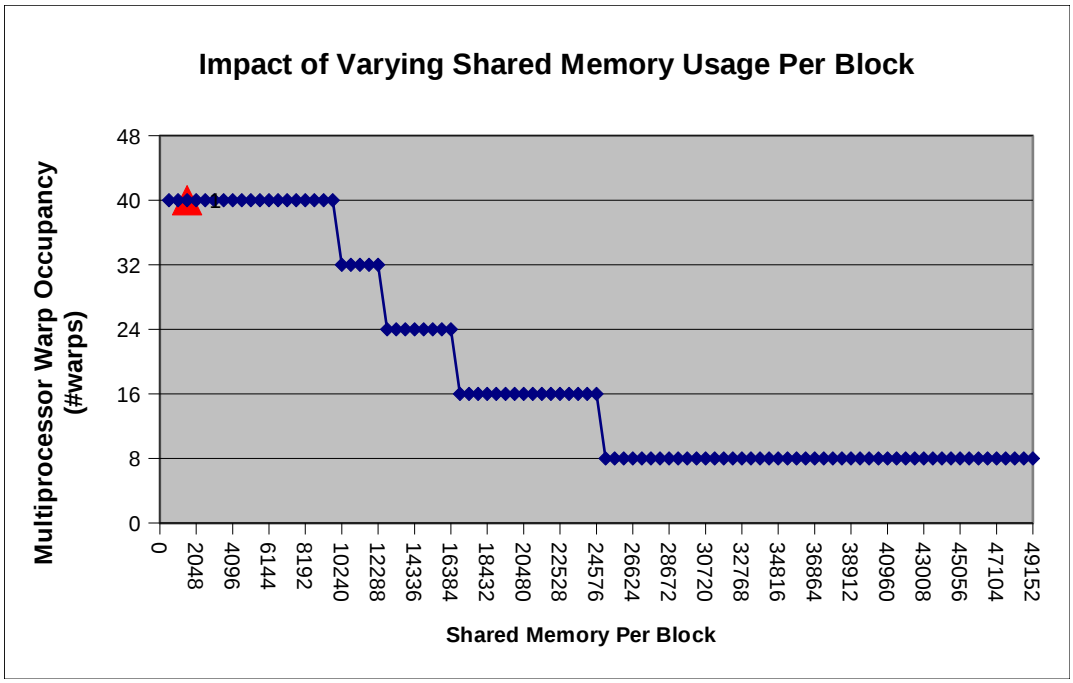


Figure 7.3.3: Occupancy impact of varying shared memory usage per block.

It should be noted here that not all block size values shown in Figure 7.3.1 are valid for this implementation. Specifically, none of the values that achieve higher occupancy can be used, since these block sizes do not divide exactly the dimensions of the NDRange. The remaining options (64, 128, 512 and 1024) result into smaller occupancy. Therefore, partitioning the index space into 256 thread blocks gives the best possible occupancy, given the current usage of resources. The maximum theoretical occupancy for each kernel, together with the achieved occupancy, is shown in Table 7.3.1.

Kernel	**Block Size** (threads)	**Registers**	**Shared Memory**	**Theoretical Occupancy**	**Achieved Occupancy**			
					128x128	**256x256**	**512x512**	**1024x1024**
l100	256	21	0 B	83%	46%	46%	46%	46%
l100_pc	256	10	0 B	100%	89%	93%	96%	97%
l200	256	21	0 B	83%	63%	62%	62%	62%
l200_pc	256	10	0 B	100%	88%	92%	94%	94%
l300	256	19	0 B	100%	72%	83%	65%	87%

Table 7.3.1: Theoretical and achieved occupancy of the heterogeneous implementation.

Before commenting on the above results, it should be noted that improving occupancy above a certain point will not improve performance any further. According to [30], this point is around 50%. From the above table it is observed that theoretical occupancy is generally high, while achieved occupancy is lower. This is caused by low computation to global memory access ratio, that limits the number of active-threads per multiprocessor. The kernels with the lowest ratio, namely l100 and l200 are the most affected. Still, the achieved occupancy is close, or above the 50% threshold.

7.3.2 Minimisation of Redundant Threads

In the initial implementation, the kernels that implement periodic continuation (l100_pc and l200_pc) are executed over a space of MxN threads. Populating the halos by iterating over MxN elements is a convenient way for CPU-based implementations, but on GPU architectures, this pattern causes unnecessary overheads, as explained below. Listing 7.3.1 shows the source code for kernel l100_pc.

```
int x = get_global_id(0);
int y = get_global_id(1);

if(x < N) {
  cu[x] = cu[M*M_LEN + x];
  cv[M*M_LEN + x + 1] = cv[x + 1];
  z[x + 1] = z[M*M_LEN + x + 1];
  h[M*M_LEN + x] = h[x];
}

if(y < M) {
  cu[(y + 1)*M_LEN + N] = cu[(y + 1)*M_LEN];
  cv[y*M_LEN] = cv[y*M_LEN + N];
  z[(y + 1)*M_LEN] = z[(y + 1)*M_LEN + N];
  h[y*M_LEN + N] = h[y*M_LEN];
}

cu[N] = cu[M*M_LEN];
cv[M*M_LEN] = cv[N];
z[0] = z[M*M_LEN + N];
h[M*M_LEN + N] = h[0];
```

Listing 7.3.1: Source code of kernel_l100_pc.

During the execution of this kernel, most threads will perform the same task, by reading and writing to the same elements. This involves many unnecessary memory accesses. Since all threads will be reading the same element, most probably loads will be served from L1. Given that L1 is the same as the shared memory, it is also possible that loads of the same address will be broadcast to all threads (Section 4.2). On the other hand, given that stores are performed directly to L2, they will probably be serialised, since they will also be performed on the same memory address. These are all hypotheses, since no exact details are known about the operation of these components. The effect of conditionals is expected to be compensated by branch predication (Appendix I). The lack of branching cost was verified through the profiler.

Given that in the shallow water algorithm, matrices are square (Section 2.1), the same task can be performed more efficiently by a single-dimensional NDRange of size M. The costs associated with the redundant threads will thus be eliminated from these kernels. The optimised code version of l100_pc is shown below.

```
int x = get_global_id(0);

cu[x] = cu[M*M_LEN + x];
cv[M*M_LEN + (x + 1)] = cv[x + 1];
z[x + 1] = z[M*M_LEN + (x + 1)];
h[M*M_LEN + x] = h[x];

cu[(x + 1)*M_LEN + N] = cu[(x + 1)*M_LEN];
cv[x*M_LEN] = cv[x*M_LEN + N];
z[(x + 1)*M_LEN] = z[(x + 1)*M_LEN + N];
h[x*M_LEN + N] = h[x*M_LEN];

cu[N] = cu[M*M_LEN];
cv[M*M_LEN] = cv[N];
z[0] = z[M*M_LEN + N];
h[M*M_LEN + N] = h[0];
```

Listing 7.3.2: Source code of optimised version of kernel_l100_pc.

Similar modifications have been performed for l200_pc. Table 7.3.2 shows the achieved performance.

Problem Size	Time (sec)	Mflop	Mflop/s
128x128	0.868	4,260	4,907
256x256	3.06	17,040	5,568
512x512	11.791	68,158	5,780
1024x1024	54.261	272,629	5,024

Table 7.3.2: Performance results for the heterogeneous implementation after minimising inactive threads.

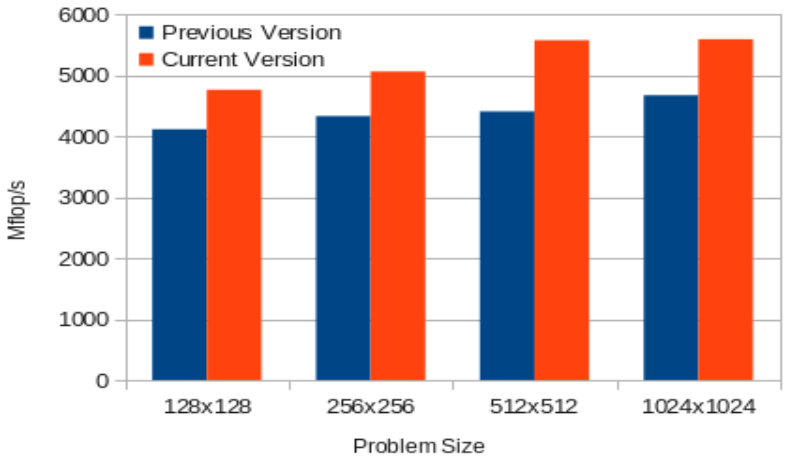


Figure 7.3.4: Performance of heterogeneous version after the elimination of redundant threads.

Figure 7.3.4 shows the improvement in performance with respect to the previous version. The overhead of redundant threads varied roughly from 650 Mflop/s to 1.1 Gflop/s, depending on the problem size.

7.3.3 Data Transfer Between Host and Device

As discussed earlier, all arrays are initialised on the device and no transfers need to be performed back to the host, until the computation of the problem is complete. Upon completion, only three arrays need to be transferred back to the host – velocities and pressure. The performance penalty of this transfer depends on the actual bus bandwidth and the problem size. The throughput obtained for the various problem sizes, as reported by the profiler, is presented below:

Problem Size	Data Transferred	Total Transfer Time	Average Throughput
128x128	390.024 KB	72.512 usec	5.133 GB/s
256x256	1.511 MB	253.759 usec	5.821 GB/s
512x512	6.024 MB	1.610 msec	3.656 GB/s
1024x1024	24.048 MB	6.432 msec	3.653 GB/s

Table 7.3.3: Timings for data transfers between host and device.

From the above data it is clear that the transfer penalty is small, with only 6.5 milliseconds transfer time for the 1024x1024 problem size. The fraction of this overhead compared to total execution time is small, and therefore no further optimisation will be consider at this point.

7.3.4 Optimisation of Memory Throughput Through Coalescing

In the current implementation, kernels are organised in blocks of 256 threads, with each thread computing one array element. These threads are organised as 16x16 sized blocks. Upon execution, each thread selects the element to compute based on its global coordinates on the NDRange, with a one-to-one correspondence between the coordinates of elements and computing threads.

In the global memory each array line is stored in one or more 128B aligned segments, with each block storing 16 doubles. The number of blocks that a line consists of, depends on the problem size. For instance, on a 128x128 problem size, each line will be stored in 8 blocks, and each array will be stored in 1024 blocks. Given that 1) the benchmark uses double-precision floating point numbers, 2) array rows will be aligned

in memory, 3) threads access elements sequentially, then, for any given computation, each thread will access any given element (operand or result), in one of the following patterns:

- **x[i][j]:** In this case, threads read array elements in a linear fashion, with every k^{th} thread accessing the k^{th} element (Figure 7.3.5). Since one segment contains 16 elements, only one coalesced read is needed per half-warp.
- **x[i][j+1]:** This case is the same one as above, but involves a different row. Again, one coalesced read is needed per half-warp.

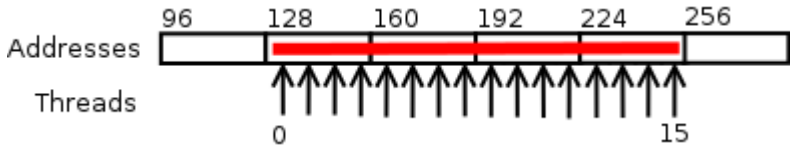


Figure 7.3.5: Aligned access to 16 doubles leading to 1x128B coalesced read.

- **x[i+1][j]:** In this case, every k^{th} thread will read every k^{th}+1 element, leading to a misaligned access pattern (Figure 7.3.6). Moreover, the last thread will read the first element of the next segment, leading to two coalesced reads.
- **x[i+1][j+1]**: Again, this case corresponds to the one as above, involving a different row. Two coalesced reads will be needed per half-warp.

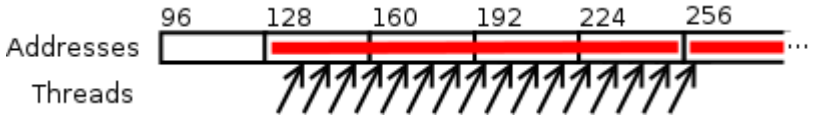


Figure 7.3.6: Unaligned access to 16 doubles leading to 2x128B coalesced reads.

The latter case signifies a possibility that, if kernels are modified so that threads compute sequential elements over a single row, subsequent blocks will be cached, and thus be reused by subsequent half-warps. One possible way to accomplish this, is to change the work-group dimensions to 256x1 (128x2 for the 128x128 problem size). This modification does not affect occupancy, since the number of threads per block does not change.

Figure 7.3.7 verifies this hypothesis by showing a comparison of the L1 hit rate for global memory accesses, before and after this optimisation.

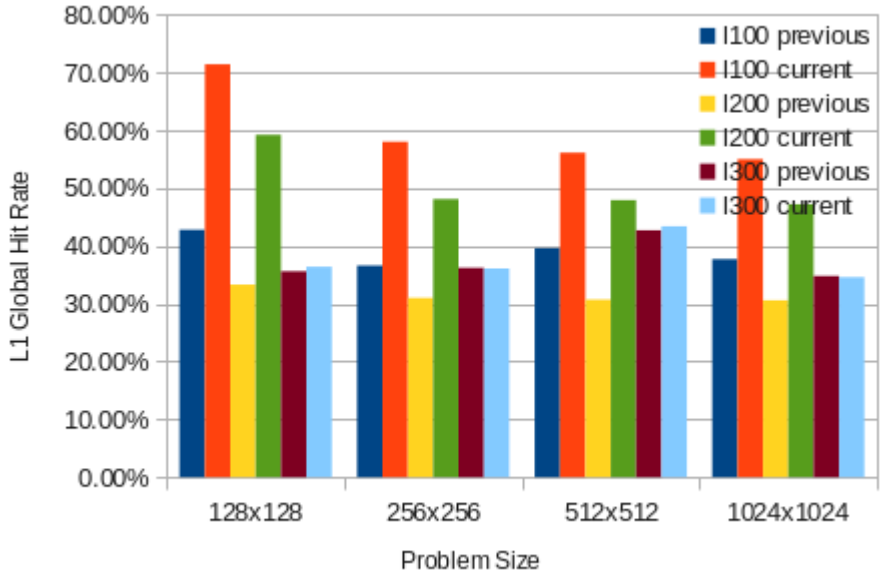


Figure 7.3.7: Comparison of L1 global hit rate for current and previous version.

Table 7.3.4 shows the achieved performance of this opimisation. The variationof performance with respect to the previous version is between 180 Mflop/s and 711 Mflop/s (Figure 7.3.8).

Problem Size	Time (sec)	Mflop	Mflop/s
128x128	0.828	4,260	5,144
256x256	2.953	17,040	5,770
512x512	11.049	68,158	6,168
1024x1024	47.258	272,629	5,768

Table 7.3.4: Performance results for the heterogeneous implementation after optimising for occupancy.

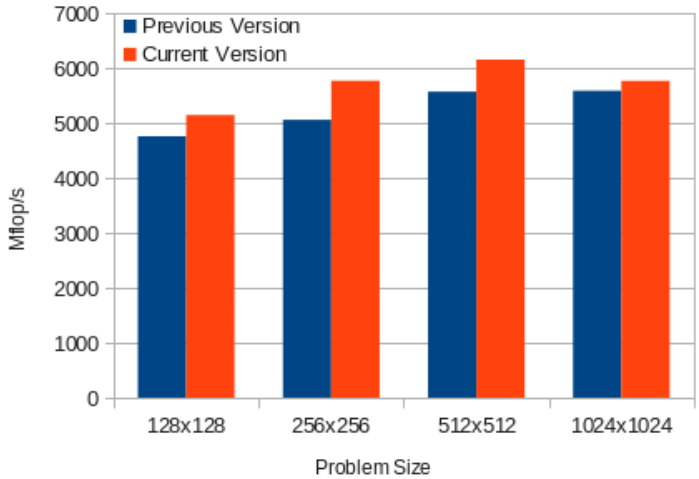


Figure 7.3.8: Performance comparison between current and previous version, after optimising for occupancy.

7.3.5 Improvement of Memory Throughput by Using Registers

The most obvious optimisation for improving memory performance is to move frequently accessed elements into registers. The improvement in memory performance is expected to be proportional to the number of times an element is read from a register instead of the global memory. This argument ignores caching, but given the limited cache available and the higher throughput of registers compared to cache, performance is still expected to improve.

Table 7.3.5 shows the number of elements accessed more than once, for each kernel. The third column shows the number of global memory accesses that can be avoided if all of these elements are stored into registers. Periodic continuation kernels are not included, as they do not include multiply accessed elements.

Kernel	**Elements**	**Potential for Improvement** (number of accesses)
l100	7	13
l200	6	6
l300	3	3

Table 7.3.5: Number of reusable elements per kernel and potential for improvement.

The number of registers used can be increased up to a certain number before occupancy is reduced. Table 7.3.6 shows the number of registers used per thread for each kernel and the number of registers that may be used before occupancy drops.

Kernel	**Registers Used**	**Maximum Registers**
l100	21	24
l200	20	24
l300	19	20

Table 7.3.6: Number of registers available for use per kernel, before occupancy drops.

Given the current levels of occupancy (Table 7.3.1), there is still theoretically room for lowering occupancy before reaching the theoretical 50% where performance is expected to drop. Except from improved memory access time, the use of registers allows to improve performance by increasing ILP, as there will be more instructions available for execution while others are waiting for memory access. The potential benefits of ILP provide additional space for occupancy drop, without performance loss. It has been shown that higher performance values can be reached by moving data into registers and exploiting ILP, even at very low occupancy levels [45].

The optimum number of registers used on each kernel can be found experimentally, by increasing the number of registers progressively and observing the effect on performance. Table 7.3.7 shows how performance varies by moving values of elements into registers, one element at a time, for the l100 kernel.

Elements	Accesses	Registers	Theoretical Occupancy	Performance (sec)			
				128x128	256x256	512x512	1024x1024
1	4	22	83%	0.818	2.908	10.842	44.973
2	4	24	83%	0.811	2.873	10.694	44.387
3	4	25	67%	0.799	2.815	10.527	43.730
4	2	26	83%	0.794	2.805	10.426	43.215
5	2	25	67%	0.795	2.795	10.409	42.774
6	2	26	67%	0.788	2.771	10.311	42.476
7	2	28	67%	0.785	2.756	10.241	42.477

Table 7.3.7: Performance effect of additional register use on kernel l100.

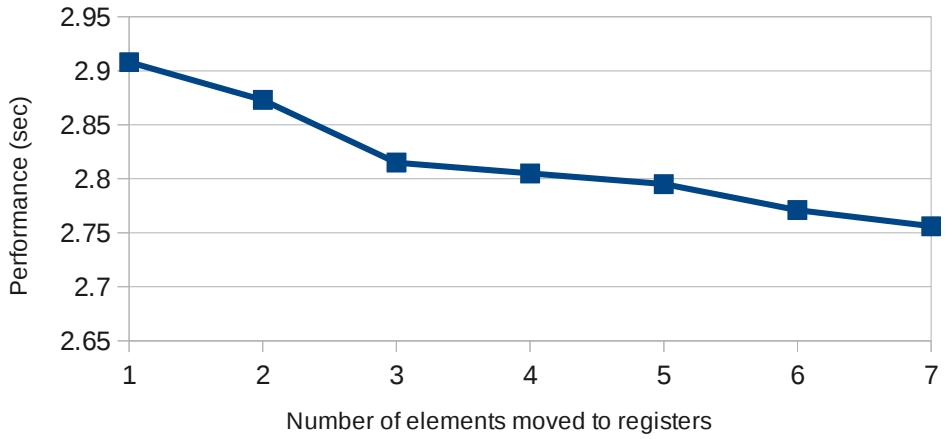


Figure 7.3.9: Variations in performance as the number of elements moved in registers increases.

Figure 7.3.9 shows the performance variation of the 256x256 problem size for kernel_l100. The observed variation is attributed to the effect of 1) changes in occupancy depending on the number of registers used, and 2) compiler optimisations that lead to different amounts of ILP. Although, theoretically, each element should consume two additional registers, Table 7.3.7 shows that the number of registers used by the kernel will vary in an inconsistent manner. Occasionally, moving an element into a register will come at no cost; storing 3 and 5 elements into registers lead into the same

number of registers used by the kernel. This behaviour is the result of how the compiler optimises register usage, and will affect instruction dependencies and consequently ILP. Similar variations are observed for the rest of the kernels, in all problem sizes. Finding the optimum order of instructions that will achieve the maximum level of ILP falls into the scope of micro-optimisation and will not be considered at this point. However, some experimentation was performed by reordering computations, which resulted into higher performance for kernel_l100, with the price of more registers used. No performance increase was achieved for the rest of the kernels by reordering their computations.

Assigning elements to registers, also modifies the computation to communication ratio, both in terms of computations and communications. The reduction in the number of computations is caused by the fact that the array index of an element that is used multiple times, needs only to be computed once, at the time its value is assigned to a register[4]. The number of communications is reduced by the number of times an element is read from a register instead of the global memory. The new ratios for the three kernels are shown in Table 7.3.8.

Kernel	Computations	Memory Accesses	Ratio	Previous Ratio
l100	63	14	4.5	3.70
l200	81	19	4.26	3.84
l300	52	18	2.888	2.71

Table 7.3.8: Modifications of computation to communication ratio, after moving elements registers.

Kernel	Registers Used	Theoretical Occupancy	Achieved Occupancy			
			128x128	256x256	512x512	1024x1024
l100	30	67%	57.7%	60.9%	61.7%	61.5%
l200	25	67%	60.1%	62.2%	62.6%	62.8%
l300	20	100%	69.0%	80.7%	60.8%	86.9%

Table 7.3.9: Comparison between theoretical and achieved occupancy, after moving elements to registers.

A comparison between the theoretical and the achieved occupancy is shown in Table 7.3.9. Compared to that of the naïve implementation (Table 7.3.1), achieved occupancy is now much closer to the theoretical. This is because less memory accesses are performed per kernel, and therefore more active threads are available for scheduling.

4 Recall that array indices in kernels are in the form of [(y + 1)*M_LEN + x + 1]

The performance achieved for each problem size, is shown in Table 7.3.10. Figure 7.3.10 compares the achieved performance relative to the previous version. Performance improvement varies approximately between 700 Mflop/s and 1.5 Gflop/s.

Problem Size	Time (sec)	Mflop	Mflop/s
128x128	0.734	4,260	5,803
256x256	2.501	17,040	6,813
512x512	9.239	68,158	7,377
1024x1024	37.800	272,629	7,212

Table 7.3.10: Performance results for the heterogeneous implementation after moving elements to registers.

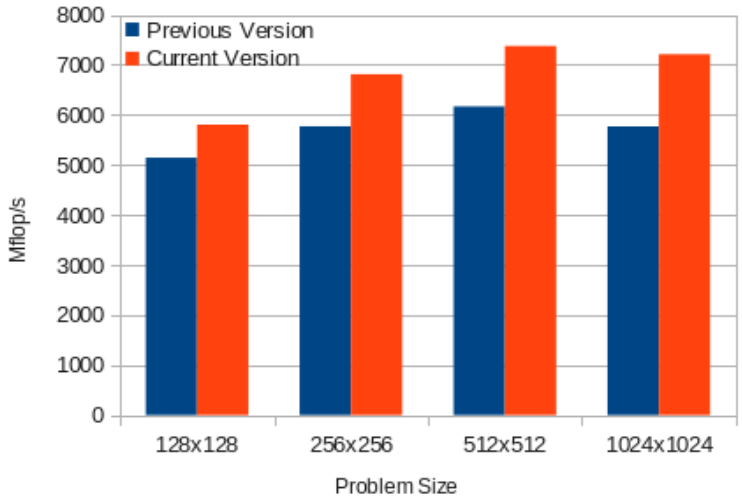


Figure 7.3.10: Performance comparison between current and previous version, after moving elements to registers.

7.4 Results

The final results with respect to those of the naïve implementation are presented in Figure 7.4.1. In order to observe the scalability of the GPU implementation on larger problems, performance was additionally measured for a problem size of 2048x2048.

The final version's performance is improved over the naïve version's in all problem sizes. Performance improvement roughly ranges between 1.7 Gflop/s and 3 Gflop/s, which corresponds to a 40% - 65% improvement over the naïve version. The lowest performance is observed at 5.8 Glop/s, on the 128x128 problem size, and is peaking at 7.2 Gflop/s, on the 512x512 problem. From that point on, as the problem size grows, performance gradually drops, reaching 7 Gflop/s on the 2048x2048 problem.

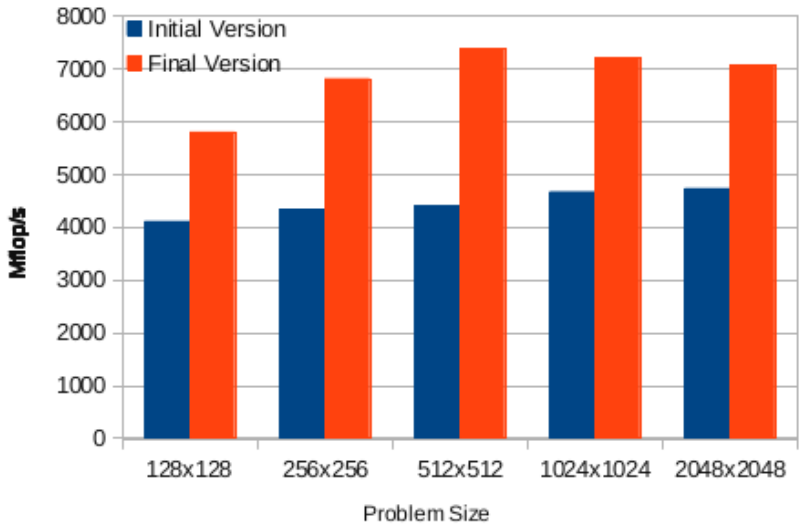


Figure 7.4.1: Performance comparison between the initial and final heterogeneous implementations.

Figure 7.4.2 shows a comparison between the sequential, single-core implementation, the final multi-core implementation running over 16 threads, and the final GPU accelerated version.

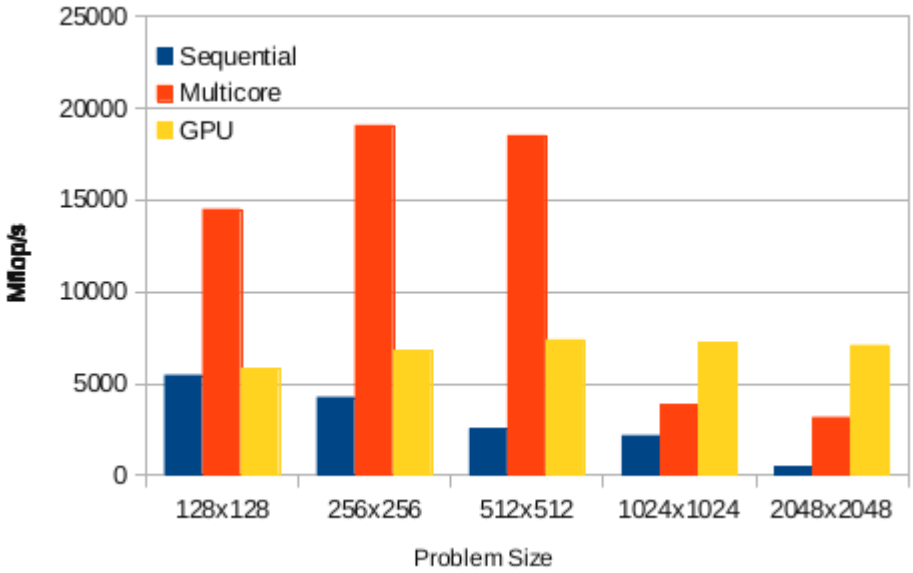


Figure 7.4.2: Performance comparison between the single-core, multi-core and many-core implementations.

From the above results, the following observations are made:

1. Both multi-core and many-core implementations perform better than the sequential implementation. One interesting observation here is the low performance of the GPU implementation, that is close to that of thesequential

implementation. The reasons behind this effect are explained later.

2. The GPU implementation's performance exhibits a much steadier curve compared to that of the sequential and the multi-core implementations.

2. The multi-core implementation performs better in problem sizes up to 512x512 inclusive, exhibiting significantly better performance than the many-core implementation.

3. On large problem sizes, where the multi-core implementation does not scale, the GPU implementation maintains its performance, only exhibiting a small drop.

The observed differences in performance reflect the differences between the two architectures. The NUMA architecture used in the multi-core implementation, features a small number of interconnected cores, able to perform in high speed, as long as communication overheads remain low. Once the problem becomes big enough, performance drops and the implementation fails to scale. On the other hand, GPU architectures, feature a large number of lower speed cores, partitioned into groups of independent computational units. Although all units share the same global memory, communication is abstracted into different layers and, thus, GPUs are able to scale better on larger problem sizes by hiding part of the memory latency. For this reason, small problem sizes are expected to perform poorly, since they don't provide enough computations to keep multiprocessors busy and thus are dominated by low core utilisation.

7.5 Summary

This chapter presented a parallel implementation of the shallow water benchmark on a GPU based architecture. This implementation was bound by memory communication, with achieved performance ranging between 6Gflop/s and 7Gflop/s. Compared to the multi-core implementation it exhibited lower performance up to the 512x512 problem size. However, its performance remained steady on larger problem sizes, superseding that of the multi-core implementation. The next chapters presents overall conclusions and some future work.

Chapter 8

Conclusions

This work presented issues around the implementation of a shallow water dynamics simulation on a GPU accelerated architecture. The background theory behind the shallow water model was presented, together with the sequential implementation and its potential for parallelism. A brief introduction to OpenCL described the basic principles of the framework used on the heterogeneous implementation. The background was completed by a description of the CUDA architecture, focusing the Fermi device family.

An implementation on a traditional NUMA architecture, developed with the aid of OpenMP, using an overhead based optimisation approach, was presented next. This version scaled up to the 512x512 problem size and highlighted the communication issues that limit its scalability towards larger problem sizes.

The development process of the GPU based implementation pursued a heuristics-based approach, with a set of optimisations being evaluated for their applicability on the shallow water algorithm. The GPU accelerated version displayed a relatively steady performance curve, with respect to the different problem sizes, with performance also peaking on the 512x512 problem size. When compared to the multi-core implementation, the GPU version exhibited lower performance on problem sizes up to 512x512. However, its performance was maintained on higher problem sizes, that the multi-core version was unable to follow. This behaviour reflects the differences of the two architectures.

The following sections present some observations made with respect to general differences on the development process on the two architectures, as well as on remarks specific to the parallelisation of the shallow water dynamics model.

8.1 Observations on the Development for Multi-Core and GPU Accelerated Platforms

The most important observation made when comparing the development process between the multi-core and the GPU-based implementations, is that the development

time of the multi-core implementation was significantly shorter. This can be attributed to three factors: 1) Previous familiarity with CPU-based architectures 2) The maturity of multi-core platforms, and 3) The productivity offered by the chosen framework (OpenMP). Having emerged from traditional single processor architectures, multi-core architectures have the advantage of a smooth learning curve. Moreover, being more mature, multi-core architectures are well understood and documented. Longer research has led to the development of more systematic approaches to parallelisation and optimisation, like the one followed on this implementation. On the other hand, GPU architectures are fundamentally different, and more complex. This complexity is reflected by the optimisation process, where the optimisation of one parameter may affect the balance of others. Optimising for these architectures thus requires good understanding of a new architecture, making the learning curve steep. Complexity is a general problem in heterogeneous computing, with its effect being even stronger on more specialised types of accelerators, like FPGAs [8].

Another factor contributing to complexity was the low quality of vendor documentation. Official information regarding interesting parts of the target device's specifications - or other devices of the same family - were not found, (e.g. double-precision performance, bus throughputs and cache latencies) and remained open questions. More importantly, parts of the official documentation were outdated, especially when it comes to architecture-specific optimisations described in the official guides (see [27] [30] with reference to the Fermi architecture). All that led to several dead ends and confusion, before developing a critical mind towards official documentation and expanding research to third-party resources.

When compared to the maturity of multi-core architectures and the development in the field of systematic approaches to optimisation, the relatively short life and rapid changes of GPU architectures has allowed little space for the development of similar formulations. Still, relevant work towards this direction has already been performed [46] [47].

Finally, when it comes to productivity, the directive-based interface provided by OpenMP allowed for rapid development. Still, the results of the naïve multi-core version showed that ease-of-use cannot replace understanding of the underlying architecture (Section 6.2). OpenCL provided a low level framework, with a high degree of flexibility, but higher complexity too. Directive based standards, similar to OpenMP,

have recently appeared in the GPU world, like HMPP [48] and OpenACC [49]. During the research for this dissertation, some work was performed on evaluating the shallow water benchmark's performance on a pre-release version of the CAPS implementation of HMPP [50]. However, the completion of this evaluation was left as future work.

8.2 Observations on the Shallow Water Implementation

During the implementation of the shallow water benchmark, several interesting remarks related to the shallow water dynamics model surfaced, some of which also apply to similar problems.

The multi-core implementation highlighted the need for both sufficient parallelism and memory throughput. The lack of either of them, leads to decreased performance. On GPU architectures this is not necessarily true, as sufficient instruction and thread-level parallelism can hide low memory throughput.

An interesting observation made during the development of the OpenMP version, was that the unbalanced data partitioning of array blocks was accompanied with additional communication overhead, since memory accesses need to be performed on blocks residing on remote memories. This effect may apply not only to the shallow water benchmark, but also to any other problem with similar access patterns that is run on a NUMA architecture.

The OpenMP implementation raised a question regarding the significance of the order in which overhead optimisations are performed, and whether it is possible for high overheads to mask the effect of optimising lower ones. This question is also valid for GPU-based architectures. Some evidence were found that supports the validity of this question; [32] proposes an iterative based approach after the most significant overheads have been identified, and [28] shows variations of the effectiveness of certain optimisations on a simple algorithm, before and after certain overheads have been minimised (p. 118).

The GPU implementation highlighted the simplicity of the shallow water algorithm's structure, and its limited potential for optimisations related to the GPU architecture. The algorithm consisted of simple additions and multiplications, with no loops, conditionals, or transcendental functions that could be easily optimised. Moreover, the requirement of double-precision arithmetic, disallowed the use of device optimisations that apply

exclusively to single-precision computations. It is interesting to see how double-precision support will be developed in future generations of devices, as this feature is highly demanded by the scientific community. On the other hand, the simplicity of the algorithm shifted the focus of optimisation to memory throughput, which is probably the most important, as it applies to all algorithms. Some additional optimisations that were intended to be performed are 1) the investigation of the potential of data prefetching, in order to further exploit instruction-level parallelism, and 2) delegating the periodic continuation computations and the updating of data for the next cycle, to the CPU. These, together with some other future work are described in more detail, in the following section.

Chapter 9

Future Work

9.1 Short Term Objectives

Several issues are still open, in both the multi-core and the many-core implementations. The reasons behind the scaling patterns of the OpenMP implementation can be verified (or disproved) by profiling. A deeper performance analysis would definitely provide additional insight. An interesting approach towards improving communication overheads, would be to investigate the possibility of modifying the algorithm so that off-by-one accesses outside of each thread's block are eliminated. Similar work on the GPU implementation, would ensure that no additional coalesced blocks are transferred because of off-by-one accesses. Most probably, any modifications to the algorithm towards this direction would be very architecture specific, if possible at all.

Some further improvements to the GPU implementation that were mentioned earlier are the investigation of the potential of data prefetching, and the delegation of data-movement operations, to the CPU. Prefetching is a technique described in [28]. It involves the computation of multiple elements by each thread, possibly in the form of tiling, and the fetching of the data of, computation i+1, before performing computation i (Listing 9.1.1). This provides more room for instruction level parallelism, as computation i is performed during the fetching of operands for computation i+1.

```
// load computation 0 operands into registers

for(i=0; i < computed_elements; i++) {
      // move computation i operands from registers to smem
      // synchronize
      // load computation i+1 operands to registers
      // perform computation i using data from shared memory
      // synchronize
}
```

Listing 9.1.1: Prefetching algorithm for increasing instruction-level parallelism.

Delegating data-movement-dominated parts to the CPU involves the periodic continuation kernels, as well as the updating part of kernel_l300. These can be implemented as OpenCL native kernels, that is kernels that are executed by the CPU,

that will execute as event callbacks, triggered upon completion of the kernels that operate on these data. For instance, the periodic continuation of cu will be a native kernel, being executed as a callback to an event signalled by the completion of the kernel computing cu. Data transfers can be enhanced by being transferred asynchronously, during the execution of other kernels that operate on other data. Overall, the performance of this optimisation depends on the relationship between 1) native kernel execution time, 2) host-to-device and device-to-host transfer times and 3) the execution time of kernels that operate on other data. In optimal conditions, the execution time of native kernels can be completely hidden.

9.2 Long Term Objectives

Further work can be performed on the investigation of the GPU computing field. Potential candidates are the study of AMD devices, which provide the additional benefit that multi-core processors can be seen as OpenCL devices [51]. Other architectures of interest are SoC architectures like Accelerated Processing Units (APUs), where an accelerator like a GPU or an FPGA is integrated into the processor chip. These architectures have the benefit of high bandwidth transfers between the accelerating and processing units.

Another possibility is to pursue a large scale direction, with the development of a multi-GPU implementation or the investigation of the challenges of an HPC oriented approach of multiple nodes, running GPU-accelerated OpenMP implementations and communicating each other through MPI, for example.

Some options are available for the improvement of the general development process. An approach to the lack of documentation is to seek alternative sources of research or profiling / benchmarking tools that will provide some further insight on the architecture. Alternatively, the development of low level, custom tools, would be an interesting direction of research.

The development of an overhead-centric methodology for the optimisation of code running on GPUs, similar to the one followed on the multi-core implementation, is another possible direction. That would require the investigation of other major vendor architectures, in order to identify common architectural patterns. Some research performed during this project show that AMD Radeon devices have a similar overall architecture with NVIDIA devices, but use VLIW processors, where CUDA devices use

ALU/FPU cores [52]. OpenCL terminology is also based on the structure of AMD architecture (compute units, processing elements etc.).

Further evaluation of the emerging frameworks for GPU-based computing, like HMPP, is another possibility. A comparison between the productivity offered by different GPU programming paradigms is presented in [53].

Finally, when it comes to the shallow water model, a possible option is the consideration of scientific models that exhibit irregular memory access patterns [54], [55], [56].This direction is already pursued by weather modelling organisations worldwide. Relevant models describe methods to replace the latitude-longitude grid used in traditional models with hexagons. The approach is known to solve problems associated with the distribution at the poles. Tasks involved in this approach require the implementation of a benchmark similar to the one studied in this work, and the application of optimisations focused on the irregular access patterns involved.

References

[1] T. Kuroda, ‘CMOS design challenges to power wall’, in *Microprocesses and Nanotechnology Conference, 2001 International*, 2001, pp. 6 –7.

[2] W. A. Wulf and S. A. McKee, ‘Hitting the memory wall: Implications of the obvious’, *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[3] D. W. Wall, *Limits of instruction-level parallelism*, vol. 19. ACM, 1991.

[4] G. E. Moore and others, ‘Cramming more components onto integrated circuits’, *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[5] H. Sutter, ‘The free lunch is over: A fundamental turn toward concurrency in software’, *Dr. Dobb’s Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[6] W. Hwu, K. Keutzer, and T. G. Mattson, ‘The Concurrency Challenge’, *IEEE Design Test of Computers*, vol. 25, no. 4, pp. 312 –320, Aug. 2008.

[7] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, ‘State-of-the-art in heterogeneous computing’, *Scientific Programming*, vol. 18, no. 1, pp. 1–33, Jan. 2010.

[8] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, ‘Comparing Hardware Accelerators in Scientific Applications: A Case Study’, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 58 –68, Jan. 2011.

[9] P. Lynch, ‘The origins of computer weather prediction and climate modeling’, *Journal of Computational Physics*, vol. 227, no. 7, pp. 3431–3444, 2008.

[10] G. R. Hoffmann, P. N. Swarztrauber, and R. A. Sweet, ‘Aspects of using multiprocessors for meteorological modeling’, *Multiprocessing in meteorological models*, pp. 126–195, 1988.

[11] J. Michalakes and M. Vachharajani, ‘GPU acceleration of numerical weather prediction’, in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –7.

[12] M. de la Asunción, J. Mantas, and M. Castro, ‘Simulation of one-layer shallow water systems on multicore and CUDA architectures’, *The Journal of Supercomputing*, vol. 58, no. 2, pp. 206–214, 2011.

[13] M. Lastra, J. M. Mantas, C. Ureña, M. J. Castro, and J. A. García-Rodríguez, ‘Simulation of shallow-water systems using graphics processing units’, *Mathematics and Computers in Simulation*, vol. 80, no. 3, pp. 598–618, Nov. 2009.

[14] L. Dagum and R. Menon, ‘OpenMP: an industry standard API for shared-memory programming’, *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 – 55, Mar. 1998.

[15] J. E. Stone, D. Gohara, and G. Shi, ‘OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems’, *Comput Sci Eng*, vol. 12, no. 3, pp. 66–72, May 2010.

[16] C. B. Vreugdenhil, *Numerical methods for shallow-water flow*, vol. 13. Springer, 1994.

[17] R. Sadourny, ‘The dynamics of finite-difference models of the shallow-water equations’, 1975.

[18] J. Chang, A. Kasahara, G. A. Corby, A. Gilchrist, P. R. Rowntree, W. M. Washington, D. L. Williamson, A. Arakawa, V. R. Lamb, W. Bourke, and others, ‘Methods in computational physics. Vol. _17: General circulation models of the

atmosphere.’, *Methods in computational physics. Vol. _17: General circulation models of the atmosphere., by Chang, J.; Kasahara, A.; Corby, GA; Gilchrist, A.; Rowntree, PR; Washington, WM; Williamson, DL; Arakawa, A.; Lamb, VR; Bourke, W.; McAvaney, B.; Puri, K.; Thurling, R.. New York, NY (USA): Academic Press, 9+ 337 p.*, vol. 1, 1977.

[19] ‘The Khronos Group Inc.’ [Online]. Available: http://www.khronos.org/. [Accessed: 30-Apr-2012].

[20] ‘OpenCL Conformant Products’. [Online]. Available: http://www.khronos.org/conformance/adopters/conformant-products/. [Accessed: 30-Apr-2012].

[21] P. O. Jaaskelainen, C. S. de La Lama, P. Huerta, and J. H. Takala, ‘OpenCL-based design methodology for application-specific processors’, in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, 2010, pp. 223 –230.

[22] ‘OpenCL 1.2 Specification’. [Online]. Available: http://www.khronos.org/registry/cl/. [Accessed: 21-Apr-2012].

[23] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. Morgan Kaufmann, 2011.

[24] ‘DirectCompute’. [Online]. Available: http://developer.nvidia.com/cuda/directcompute. [Accessed: 05-Sep-2012].

[25] ‘CUDA GPUs | NVIDIA Developer Zone’. [Online]. Available: http://developer.nvidia.com/cuda-gpus. [Accessed: 26-Jun-2012].

[26] M. J. Flynn, ‘Some Computer Organizations and Their Effectiveness’, *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948 –960, Sep. 1972.

[27] ‘OpenCL Programming Guide for the CUDA Architecture Version 4.2’. NVIDIA Corporation, 03-Sep-2012.

[28] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann, 2010.

[29] ‘NVIDIA’s Next Generation CUDA Compute Architecture: Fermi’. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Accessed: 09-May-2012].

[30] ‘OpenCL Best Practices Guide’. NVIDIA Corporation, 14-Feb-2011.

[31] R. Farber, *CUDA Application Design and Development*, 1st ed. Morgan Kaufmann, 2011.

[32] G. D. Riley, J. M. Bull, and J. R. Gurd, ‘Performance improvement through overhead analysis: a case study in molecular dynamics’, in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 36–43.

[33] L. A. Crowl, ‘How to measure, present, and compare parallel performance’, *IEEE Parallel & Distributed Technology: Systems & Technology*, vol. 2, no. 1, pp. 9–25, 1994.

[34] P. J. Fleming and J. J. Wallace, ‘How not to lie with statistics: the correct way to summarize benchmark results’, *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[35] D. H. Bailey, ‘Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers’, May 1991.

[36] J. Dongarra, J. Martin, and J. Vorlton, ‘Computer benchmarking: paths and pitfalls’, *Ieee Spectrum*, vol. 24, no. 7, pp. 38–43, 1987.

[37] Wikipedia contributors, ‘Time Stamp Counter’, *Wikipedia, the free encyclopedia*. Wikimedia Foundation, Inc., 28-Aug-2012.

[38] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2: Architecture*,

Programming and Algorithms, 2 Sub. Taylor & Francis, 1988.
[39] 'Message Passing Interface (MPI) Forum'. [Online]. Available: http://www.mpi-forum.org/. [Accessed: 07-May-2012].
[40] 'Quad-Core AMD Opteron™ Processor'. [Online]. Available: http://www.amd.com/us/products/server/processors/opteron/Pages/opteron-for-server.aspx. [Accessed: 11-May-2012].
[41] J. M. Bull, 'A hierarchical classification of overheads in parallel programs', in *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, 1996, pp. 208–219.
[42] 'OpenMP Specification v2.5'. [Online]. Available: http://openmp.org/wp/openmp-specifications/. [Accessed: 21-Apr-2012].
[43] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, 'Accelerating molecular modeling applications with graphics processors', *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–2640, 2007.
[44] S. S. Stone, J. P. Haldar, S. C. Tsao, W. -m. W. Hwu, B. P. Sutton, and Z.-P. Liang, 'Accelerating advanced MRI reconstructions on GPUs', *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1307–1318, Oct. 2008.
[45] V. Volkov, 'Better performance at lower occupancy', in *Proceedings of the GPU Technology Conference, GTC*, 2010, vol. 10.
[46] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu, 'Program optimization carving for GPU computing', *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.
[47] Y. Zhang and J. D. Owens, 'A quantitative performance analysis model for GPU architectures', in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 382 –393.
[48] R. Dolbeau, S. Bihan, and F. Bodin, 'HMPP: A hybrid multi-core parallel programming environment', in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.
[49] 'OpenACC Home'. [Online]. Available: http://www.openacc-standard.org/. [Accessed: 03-Sep-2012].
[50] 'HMPP Workbench | CAPS entreprise'. [Online]. Available: http://www.caps-entreprise.com/technology/hmpp/. [Accessed: 03-Sep-2012].
[51] 'AMD Accelerated parallel processing OpenCL programming guide'. [Online]. Available: http://developer.amd.com/sdks/amdappsdk/assets/amd_accelerated_parallel_processing_opencl_programming_guide.pdf. [Accessed: 08-May-2012].
[52] 'ATI Radeon™ HD 5870'. [Online]. Available: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#1. [Accessed: 09-May-2012].
[53] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby, 'Productivity of GPUs under different programming paradigms', *Concurrency and Computation: Practice and Experience*, 2011.
[54] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, 'Running unstructured grid-based CFD solvers on modern graphics hardware', *International Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221–229, 2011.
[55] M. Govett, J. Middlecoff, and T. Henderson, 'Running the NIM next-generation weather model on GPUs', in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 792–796.
[56] A. E. MacDonald, J. Middlecoff, T. Henderson, and J. L. Lee, 'A general method

for modeling on irregular grids', *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 392–403, 2011.

Appendix I: Execution of Conditionals in Fermi Devices

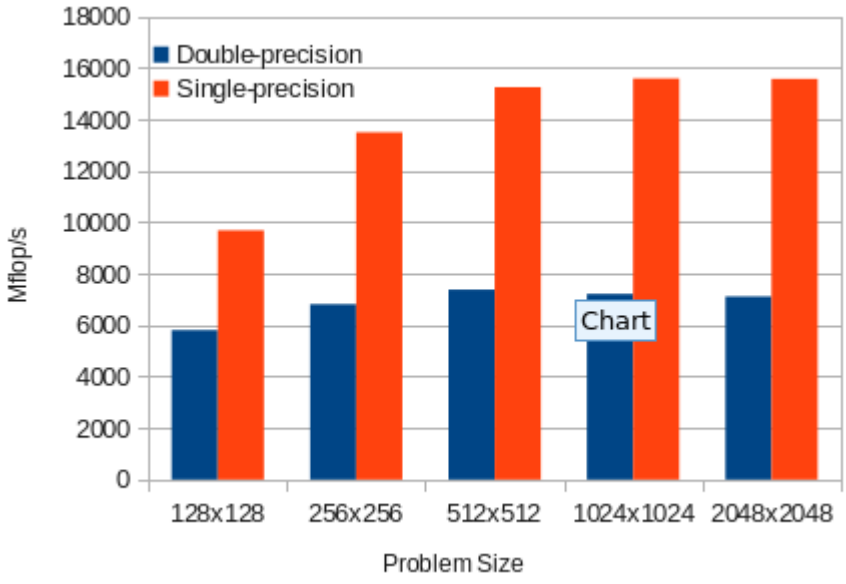
During the execution of a kernel, threads start off from the same instruction and proceed executing one instruction at a time in a SIMD manner. Still, threads maintain individual program counters and are allowed to branch independently. In particular, when a branch condition is met, both branches are evaluated, with conditions being executed sequentially. Depending on the branching structure, conditionals can potentially decrease performance significantly. Branch predication allows short conditionals to be executed without the cost of branching. This procedure substitutes branches with a per-thread conditional flag (a predicate), that is set according to the condition's evaluation for the current thread. During execution of the conditional code, only threads with a true predicate are executed.

Appendix II: Single-Precision Arithmetic Optimisations

Several optimisations are available for kernels performing single-precision computations, like:

- Enabling the use of native multiply-add (FMAD) instructions. This optimisation has the side effect of having the intermediate result of the multiplication truncated, which potentially leads to loss of precision.
- Enabling aggressive compiler optimisations, again with some cost in precision.
- Substituting division and modulo with equivalent bitwise operations, where possible.
- Using native math libraries that map directly to the hardware. Native math functions provide higher performance, than the ones provided by the standard math library, in the price of decreased precision.

In general, NVIDIA encourages the use of single-precision arithmetic where possible. An experiment was performed on the shallow water benchmark, by substituting double-precision with single-precision numbers. The esults are shown below. Performance difference ranged between 3.4 Gflop/s and 8.4 Gflop/s, which roughly corresponds to a 65% to 117% improvement. Optimisation directives like -cl-enable-mad or -cl-fast-relaxed-math had no effect on the performance.



Appendix III: Calculation of Occupancy on Fermi Devices

There are limits imposed on the number of blocks, warps and threads that may reside in a multiprocessor. For devices of compute capability 2.x, the limits are 8 resident blocks or 48 warps per SM. On the initial implementation of the shallow water benchmark, block size is defined to be 256 threads, which translates to 8 warps. Therefore, up to 6 blocks may be allocated to each SM, before the above 48 warps limit is met. Information about the resources used by each thread can be obtained by setting the compiler's verbosity flag. The following snippet is part of the information returned when compiling the unoptimised GPU version for the l100 kernel.

```
ptxas info    : Compiling entry function 'l100' for 'sm_21'
ptxas info    : Function properties for l100
 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 21 registers, 76 bytes cmem[0], 144 bytes cmem[2],
16 bytes cmem[16]
```

According to the above information, this kernel uses 21 registers and no shared memory. In devices of compute capability of 2.x, 32K registers are available per multiprocessor, and registers are shared by all resident threads. Registers are allocated *per warp*, with an allocation granularity of *128 registers*. Similarly, there are 48 KB of shared memory per SM, which are shared by all threads in a SM. The allocation granularity of shared memory is *128 B*, and shared memory is allocated *per block*.

Knowing the block size and the resources used by each thread, occupancy may be calculated as follows: According to the compiler output, each thread uses 21 registers. Each warp will therefore use 21 * 32 = 672 registers. Given that the register allocation granularity is 128 registers, each warp will be allocated ceil(672 / 128) = 6 128-register chunks, that is 6 * 128 = 768 registers / warp. Each block will therefore be allocated 768 registers / warp * 8 warps = 6,144 registers / block. The maximum number of resident blocks per SM will be 32,768 / 6,144 = 5 blocks / SM. The occupancy will therefore be equal to 83%, since ((5 blocks * 256 threads) / 32 threads / warp) / 48 max. warps / SM = 0.8333.

Appendix IV: Source Code of the Shallow Water Benchmark

This is a reduced version of the code. For the sake of brevity, several timing related functions and various printing statements have been removed. Source code follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MIN(x,y) ((x)>(y)?(y):(x))
#define MAX(x,y) ((x)>(y)?(x):(y))

#define TRUE 1
#define FALSE 0
#define M 2048
#define N 2048
#define M_LEN (M + 1)
#define N_LEN (N + 1)
#define ITMAX 4000
#define L_OUT TRUE


int main(int argc, char **argv) {

  // solution arrays
  double u[M_LEN][N_LEN],v[M_LEN][N_LEN],p[M_LEN][N_LEN];
  double unew[M_LEN][N_LEN],vnew[M_LEN][N_LEN],pnew[M_LEN][N_LEN];
  double uold[M_LEN][N_LEN],vold[M_LEN][N_LEN],pold[M_LEN][N_LEN];
  double cu[M_LEN][N_LEN],cv[M_LEN][N_LEN];
  double z[M_LEN][N_LEN],h[M_LEN][N_LEN],psi[M_LEN][N_LEN];

  double dt,tdt,dx,dy,a,alpha,el,pi;
  double tpi,di,dj,pcf;
  double tdts8,tdtsdx,tdtsdy,fsdx,fsdy;

  int mnmin,ncycle;
  int i,j;

  dt = 90.;
  tdt = dt;

  dx = 100000.;
  dy = 100000.;
  fsdx = 4. / dx;
  fsdy = 4. / dy;

  a = 1000000.;
  alpha = .001;

  el = N * dx;
  pi = 4. * atanf(1.);
  tpi = pi + pi;
  di = tpi / M;
  dj = tpi / N;
```

```
pcf = pi * pi * a * a / (el * el);

// Initial values of the stream function and p
for (i=0;i<M_LEN;i++) {
  for (j=0;j<N_LEN;j++) {
    psi[i][j] = a * sin((i + .5) * di) * sin((j + .5) * dj);
    p[i][j] = pcf * (cos(2. * (i) * di) + cos(2. * (j) * dj))
              + 50000.;
  }
}

// Initialize velocities
for (i=0;i<M;i++) {
  for (j=0;j<N;j++) {
    u[i + 1][j] = -(psi[i + 1][j + 1] - psi[i + 1][j]) / dy;
    v[i][j + 1] = (psi[i + 1][j + 1] - psi[i][j + 1]) / dx;
  }
}


// Periodic continuation
for (j=0;j<N;j++) {
  u[0][j] = u[M][j];
  v[M][j + 1] = v[0][j + 1];
}
for (i=0;i<M;i++) {
  u[i + 1][N] = u[i + 1][0];
  v[i][0] = v[i][N];
}
u[0][N] = u[M][0];
v[M][0] = v[0][N];
for (i=0;i<M_LEN;i++) {
  for (j=0;j<N_LEN;j++) {
    uold[i][j] = u[i][j];
    vold[i][j] = v[i][j];
    pold[i][j] = p[i][j];
  }
}

// ** Main loop **

for (ncycle=1;ncycle<=ITMAX;ncycle++) {

  // Compute capital u, capital v, z and h
  c1 = wtime();

  for (i=0;i<M;i++) {
    for (j=0;j<N;j++) {
      cu[i + 1][j] = .5 * (p[i + 1][j] + p[i][j]) * u[i + 1][j];
    }
  }

  for (i=0;i<M;i++) {
    for (j=0;j<N;j++) {
      cv[i][j + 1] = .5 * (p[i][j + 1] + p[i][j]) * v[i][j + 1];
    }
  }

  for (i=0;i<M;i++) {
    for (j=0;j<N;j++) {
```

```
    z[i + 1][j + 1] = (fsdx * (v[i + 1][j + 1] - v[i][j + 1]) -
                       fsdy * (u[i + 1][j + 1] - u[i + 1][j])) /
                      (p[i][j] + p[i + 1][j] + p[i + 1][j + 1] +
                       p[i][j + 1]);
  }
}

for (i=0;i<M;i++) {
  for (j=0;j<N;j++) {
    h[i][j] = p[i][j] + .25 * (u[i + 1][j] * u[i + 1][j] +
              u[i][j] * u[i][j] + v[i][j + 1] * v[i][j + 1] +
              v[i][j] * v[i][j]);
  }
}

// Periodic continuation
for (j=0;j<N;j++) {
  cu[0][j] = cu[M][j];
  cv[M][j + 1] = cv[0][j + 1];
  z[0][j + 1] = z[M][j + 1];
  h[M][j] = h[0][j];
}

for (i=0;i<M;i++) {
  cu[i + 1][N] = cu[i + 1][0];
  cv[i][0] = cv[i][N];
  z[i + 1][0] = z[i + 1][N];
  h[i][N] = h[i][0];
}

cu[0][N] = cu[M][0];
cv[M][0] = cv[0][N];
z[0][0] = z[M][N];
h[M][N] = h[0][0];

// Compute new values u,v and p
tdts8 = tdt / 8.;
tdtsdx = tdt / dx;
tdtsdy = tdt / dy;

for (i=0;i<M;i++) {
  for (j=0;j<N;j++) {
    unew[i + 1][j] = uold[i + 1][j] + tdts8 * (z[i + 1][j + 1] +
                     z[i + 1][j]) * (cv[i + 1][j + 1] +
                     cv[i][j + 1] + cv[i][j] + cv[i + 1][j]) -
                     tdtsdx * (h[i + 1][j] - h[i][j]);
  }
}

for (i=0;i<M;i++) {
  for (j=0;j<N;j++) {
    vnew[i][j + 1] = vold[i][j + 1] - tdts8 * (z[i + 1][j + 1] +
                     z[i][j + 1]) * (cu[i + 1][j + 1] +
                     cu[i][j + 1] + cu[i][j] + cu[i + 1][j]) -
                     tdtsdy * (h[i][j + 1] - h[i][j]);
  }
}

for (i=0;i<M;i++) {
  for (j=0;j<N;j++) {
```

```
    pnew[i][j] = pold[i][j] - tdtsdx * (cu[i + 1][j] - cu[i][j]) -
                 tdtsdy * (cv[i][j + 1] - cv[i][j]);
  }
}


// Periodic continuation
for (j=0;j<N;j++) {
  unew[0][j] = unew[M][j];
  vnew[M][j + 1] = vnew[0][j + 1];
  pnew[M][j] = pnew[0][j];
}

for (i=0;i<M;i++) {
  unew[i + 1][N] = unew[i + 1][0];
  vnew[i][0] = vnew[i][N];
  pnew[i][N] = pnew[i][0];
}

unew[0][N] = unew[M][0];
vnew[M][0] = vnew[0][N];
pnew[M][N] = pnew[0][0];

time = time + dt;

// Time smoothing and update for next cycle
if ( ncycle > 1 ) {

 for (i=0;i<M_LEN;i++) {
    for (j=0;j<N_LEN;j++) {
      uold[i][j] = u[i][j] + alpha * (unew[i][j] - 2. * u[i][j] +
                   uold[i][j]);
    }
  }

  for (i=0;i<M_LEN;i++) {
    for (j=0;j<N_LEN;j++) {
      vold[i][j] = v[i][j] + alpha * (vnew[i][j] - 2. * v[i][j] +
                   vold[i][j]);
    }
  }

  for (i=0;i<M_LEN;i++) {
    for (j=0;j<N_LEN;j++) {
      pold[i][j] = p[i][j] + alpha * (pnew[i][j] - 2. * p[i][j] +
                   pold[i][j]);
    }
  }

  for (i=0;i<M_LEN;i++) {
    for (j=0;j<N_LEN;j++) {
      u[i][j] = unew[i][j];
    }
  }

  for (i=0;i<M_LEN;i++) {
    for (j=0;j<N_LEN;j++) {
      v[i][j] = vnew[i][j];
    }
  }
```

```
    for (i=0;i<M_LEN;i++) {
      for (j=0;j<N_LEN;j++) {
        p[i][j] = pnew[i][j];
      }
    }

  } else {
    tdt = tdt + tdt;

    for (i=0;i<M_LEN;i++) {
      for (j=0;j<N_LEN;j++) {
        uold[i][j] = u[i][j];
        vold[i][j] = v[i][j];
        pold[i][j] = p[i][j];
        u[i][j] = unew[i][j];
        v[i][j] = vnew[i][j];
        p[i][j] = pnew[i][j];
      }
    }

  }
}
```