GPU-parallelisation and integration of multiwavelet grid adaptation for fast discontinuous Galerkin modelling of multiscale flooding on LISFLOOD-FP

Alovya Ahmed Chowdhury

Supervisors: Dr Georges Kesserwani, Dr Charles Rougé, Dr Paul Richmond

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Department of Civil and Structural Engineering
Faculty of Engineering
The University of Sheffield

2 September 2024

Declaration

I, Alovya Ahmed Chowdhury, confirm that this thesis is my own work. I am aware of the University's Guidance on the Use of Unfair Means (`www.sheffield.ac.uk/ssid/unfair-means`). This work has not previously been presented for an award at this, or any other, university.

Alovya Ahmed Chowdhury

2 September 2024

Abstract

The second-order discontinuous Galerkin (DG2) solver in LISFLOOD-FP 8.0 incurs an enormous computational effort in simulating multiscale rapid flow phenomena like tsunamis due to using a uniform mesh, even with parallelisation on a graphics processing unit (GPU). To reduce the computational effort – while still achieving similiar levels of DG2 accuracy as on a uniform mesh or grid – integrating multiwavelet (MW) grid adaptation into the GPU-parallelised DG2 solver (GPU-DG2) is a compelling research aim for this thesis. Nonetheless, implementing a grid adaptation algorithm on a GPU is challenging due to major obstacles in achieving coalesced memory access (aligned memory access by threads) and avoiding warp divergence (different execution paths within a warp). To overcome these obstacles, initially for a Haar wavelet (HW) grid adaptation algorithm, three computational ingredients are proposed – a Z-order space-filling curve, a parallel tree traversal algorithm and an array data structure – to redesign its algorithmic structure and thereby enable its implementation on the GPU. After assessing the extent to which the proposed computational ingredients indeed allow for an efficient GPU implementation of the HW grid adaptation algorithm, their use is extended to now implement a MW grid adaptation algorithm on the GPU and thereby develop a GPU-parallelised MW adaptive DG2 model (GPU-MWDG2), which is then integrated into LISFLOOD-FP. To finally analyse the impact of GPU-MWDG2's grid adaptation capability on the computational effort and accuracy of DG2 modelling, GPU-MWDG2 is compared against GPU-DG2 by simulating four realistic test cases of tsunami-induced flooding. The analyses suggest that GPU-MWDG2 is a better choice than GPU-DG2 for simulating tsunami-induced flooding when considering test cases that require setting a parameter called the maximum refinement level L to ten or higher, and where the tsunami complexity is low. For such test cases, GPU-MWDG2 is up to four times faster than GPU-DG2 while achieving similar levels of DG2 accuracy.

Acknowledgements

I would like to thank my supervisors Dr Georges Kesserwani, Dr Charles Rougé and Dr Paul Richmond for their support throughout my PhD. Thank you very much Georges for the massive amount of time and effort you have put over the past four years into teaching me how to think and write rigorously, even with all the hardships that have gone on in your life: it is a very valuable skill that is hard to acquire without mentorship, and I look forward to using it in many of my future endeavours. Thank you Charles for our broader conversations about academia, "managing up" and for pushing me to break out of the writing-feedback loop. Thank you Paul for the discussions on GPU computing, research software engineering and career planning.

Aside from my family, there are so many people I am incredibly grateful to have had as friends throughout this absolutely brutal and horrific journey that is a PhD. Arranged in no particular order: Xitong, George, Waranya, Casey, Harrison, Kiesar, Jon, Samyuktha, Prem, Alfrd, Leo, Nestor, Gabriel, Oswaldo, Ali, Jay, Riju, Callum, Araf, Opu bhai, Shaila apu, Saadi, Sadiah, Mara, Andre, Kazem, Mohammed, Umair, and Shreeharan. I apologise for not writing something special for each of you, but I have simply had too many wonderful experiences with you all over the lengthy time span of the PhD. Some more wild, some more tame, and some just getting through daily life. Thank you all so much.

To my girlfriend, Zhou Min, I am very fortunate to have met you when I did. I love you and I adore the side of life you have shown me that I would not have been able to see without you.

To my late friend Jacob, I am so sad that we will never be able to play Valheim or Minecraft together ever again. I hope you are well wherever you are.

Table of Contents

Declaration **i**

Abstract **i**

Acknowledgements **ii**

Contents **iii**

List of Figures **vi**

List of Tables **xiii**

List of Algorithms **xv**

Nomenclature **xvi**

Publications **xvi**

1 Introduction **1**

1.1 Research background 1

1.2 Research aim and objectives 4

1.3 Thesis structure 4

2 Literature review **6**

2.1 Modelling floods caused by rapid flow phenomena 6

2.2 GPU computing using CUDA 12

2.2.1 GPU hardware architecture 13

2.2.2 Warp divergence 15
2.2.3 Coalesced memory access 15
2.2.4 Using CUDA to parallelise SWE models on the GPU 18
2.3 Challenges in efficiently implementing grid adaptation on GPUs 20
2.3.1 Solver computations on the GPU over different types of non-uniform grids . . 20
2.3.2 Difficulties in efficiently generating a non-uniform grid on the GPU 23

3 GPU-parallelisation of HW grid adaptation 28
3.1 New GPU-HWFV1 model 30
3.1.1 Sequential CPU-HWFV1 model 30
3.1.2 GPU-parallelisation of CPU-HWFV1 38
3.2 Analysing GPU-HWFV1's performance 47
3.2.1 Verifying accuracy 50
3.2.2 Speedup analyses using simulations of synthetic dam-break flows 53
3.2.3 Further investigations into speedups: realistic flow simulations 60
3.3 Summary 64

4 Integrating GPU-MWDG2 into LISFLOOD-FP 69
4.1 Implementing GPU-MWDG2 70
4.1.1 Overview 70
4.1.2 MRA process of the MW and DG2 update on the GPU 72
4.2 Integration of GPU-MWDG2 into LISFLOOD-FP 8.2 75
4.2.1 Using GPU-MWDG2 within LISFLOOD-FP 8.2 75
4.2.2 Largest allowable test case area due to GPU memory consumption 79
4.2.3 New metrics for analysing GPU-MWDG2's grid adaptation capability 81
4.3 Summary 85

5 Comparing GPU-MWDG2 against GPU-DG2 87
5.1 Selection of tsunami test cases 88
5.2 Monai valley 90
5.3 Seaside, Oregon 95

5.4 Tauranga harbour . . . 100
5.5 Hilo harbour . . . 104
5.6 Summary . . . 109

6 Conclusions 111

6.1 Thesis summary . . . 111
6.2 Significant research findings, limitations and future work . . . 116

References 117

Appendices 150

A GPU computing 151

A.1 GPGPU computing on NVIDIA GPUs using the CUDA programming model . . . 152
A.1.1 GPU hardware architecture . . . 153
A.1.2 Maximising instruction throughput . . . 155
A.1.3 Maximising memory throughput . . . 158

B Filter bank matrices 163

List of Figures

2.1 GPU hardware architecture; (a) the GPU is made up of several "streaming multiprocessors" (SM) which are responsible for computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which SMs access via an L2 cache; (b) each SM is made up of hardware for supporting computation: registers, shared/L1/texture cache, constant cache and "warp schedulers", as well as several "threads" for computation; (c) each thread is made up of functional units for executing various types of instructions. 14

2.2 Warp divergence; a warp is a group of 32 threads with consecutive thread IDs that operate in lockstep. 15

2.3 Memory hierarchy involved in the CUDA programming model. 16

2.4 Optimal and non-optimal access patterns by threads in a warp when accessing global memory via the L2 and "L1 caches" (a) an optimal access leading to coalesced memory access; (b) a non-optimal access pattern leading to uncoalesced memory access. 17

2.5 Uniform-resolution computational grid made up of $9 \times 9 = 81$ cells. The numbers $0, 1, \ldots, 9, 10, \ldots$ indicate row-major numbering or indexing of the cells. 19

2.6 The three different types of non-uniform grids and the tree data structure used to index the coloured part of the cell-based non-uniform grid. 21

2.7 Grid resolution adaptation algorithm: (a) a non-uniform grid generated by a grid adaptation algorithm; the orange lines indicate cells flagged for refinement, whereas the green lines indicate cells flagged for coarsening; (b) the newly generated non-uniform grid after the flagged cells have been refined/coarsened. 24

3.1 Multiresolution analysis (MRA). The left panel shows a hierarchy of grids involved in the MRA process with maximum refinement level $L = 2$. The right panel shows how four cells at refinement level $n + 1$, called the "children", are positioned relative to a single cell at refinement level n, called the "parent". Also shown are the coefficients s and details d involved in the MRA process. 33

3.2 Generating a non-uniform grid via the MRA process. The left panel shows the tree-like structure embedded within the hierarchy of grids after flagging significant details during the encoding step: the cells where the tree terminates are called "leaf" cells (highlighted in green and blue). The right panel shows the non-uniform grid generated from assembling the leaf cells. 36

3.3 Finding the neighbours of a cell in a non-uniform grid to retrieve s_{west}, s_{east}, s_{north}, and s_{south} and compute $\mathbf{L}_c$. 38

3.4 Indexing the hierarchy of grids in row-major order (left panel) and the corresponding positions of the children $s^{(n+1)}_{[0]}$, $s^{(n+1)}_{[1]}$, $s^{(n+1)}_{[2]}$, and $s^{(n+1)}_{[3]}$ in non-adjacent memory locations (right panel). 40

3.5 Creation of a Z-order curve for a $2^2 \times 2^2$ grid. The left panel shows how the binary forms of the i and j indices of each cell making up a $2^2 \times 2^2$ grid are bit interleaved (alternating red and black digits) to yield so-called Morton codes (also shown in binary). The right panel shows how these Morton codes (now shown in decimal form) are followed in ascending order to create a Z-order curve and enforce Z-order indexing of the cells in the grid. 41

3.6 Indexing the hierarchy of grids using Z-order curves (left panel) and the corresponding positions of the children $s^{(n+1)}_{[0]}$, $s^{(n+1)}_{[1]}$, $s^{(n+1)}_{[2]}$, and $s^{(n+1)}_{[3]}$ in adjacent memory locations (right panel). 42

3.7 Parallel tree traversal. The left panel shows the tree of significant details, with different parallel traversal paths indicated in yellow, cyan, magenta and grey. The middle panel shows the traversal paths of each thread during the parallel tree traversal in terms of the Z-order indices of the cells they traverse. The right panel shows the Z-order indices recorded by each thread after parallel tree traversal is complete. . . . 45

3.8 Parallel FV1 update. The upper panel shows the Z-order indices of the leaf cells and their neighbours stored in memory after parallel tree traversal and neighbour finding, which includes duplicate indices. The lower panel shows the Z-order indices of the leaf cells and their neighbours without any duplicates, which are used to retrieve s_c, s_{west}, s_{east}, s_{north}, and s_{south} and compute $\mathbf{L}_c$ for the FV1 update in parallel. 47

3.9 Relatively higher (left panel) and lower (right panel) amounts of grid coarsening in GPU-HWFV1's non-uniform grid from the "Tsunami wave propagation over a complex beach" test case (Section 3.2.3). 50

3.10 Verifying GPU-HWFV1's C-property in the presence of wet-dry fronts with different levels of steepness in the topography and different wetting conditions: smooth humps (left panels), steeper humps (middle panels) and rectangular humps (right panels). The upper panels show the shapes of the humps, the middle panels show the time histories of the maximum discharge errors (solid lines - GPU-FV1, dashed lines - GPU-HWFV1), where $q_x = hu$ and $q_y = hv$, and the bottom panels show the time histories of the number of grid cells. 52

3.11 Verifying GPU-HWFV1's accuracy by simulating a frictional dam-break flow with moving wet-dry fronts over a non-trivial topography. Flood maps predicted by GPU-HWFV1 and GPU-FV1 on the left and right panels, respectively. At 0 s, the dam break wave emerges (upper panels). At 6 s, the wave has submerged the small humps (middle panels). At 12 s, the wave starts to surround the large hump (lower panels). 54

3.12 Circular 2D dam-break flow. Verifying GPU-HWFV1's accuracy using the same choice of ε and L as in Section 3.2.1 (($\varepsilon = 10^{-3}$ and $L = 8$): water depth centrelines at $t = 3.5\,\text{s}$ predicted by GPU-HWFV1 and GPU-FV1 compared to the benchmark solution. The benchmark solution was produced using the FV1 numerical solution of the 1D radial form of the 2D shallow water equations using 256×256 cells, following Toro (2001). 55

3.13 Circular 2D dam-break flow. GPU-HWFV1's speedup over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel). 56

3.14 Pseudo-2D dam-break flow. Verifying GPU-HWFV1 using the same choice of ε and L as in Section 3.2.1 ($\varepsilon = 10^{-3}$ and $L = 8$) : water depths centerlines predicted by GPU-HWFV1 and GPU-FV1 at $t = 2.5$ s compared with the exact solution. 57

3.15 Pseudo-2D dam-break flow. Time histories of the number of cells in the non-uniform grid (first row of panels), the timestep Δt (second row of panels), GPU-HWFV1's speedup over CPU-HWFV1 (third row of row panels) and GPU-FV1 (fourth row of panels) for different values of ε and L. 58

3.16 Dam-break wave interaction with an urban district. Top-down view of the test case area, with the gate indicated in green and topographic blocks colored in yellow. Experimental depth and velocity data are available along $y = 0.2\,\text{m}$, indicated in red. 60

3.17 Dam-break wave interaction with an urban district. Depth (left panel) and velocity (right panel) profiles predicted by GPU-HWFV1 and GPU-FV1 along $y = 0.2\,\text{m}$ in Figure 3.16 at 6 s. 61

3.18 Dam-break wave interaction with an urban district. Time histories of GPU-HWFV1's non-uniform grid cell count (top left panel), Δt (top right panel), speedup over CPU-HWFV1 (bottom left panel) and over GPU-FV1 (bottom right panel). 62

3.19 Tsunami wave propagation over a complex beach. The left panel shows a top-down view of the test case area including the gauge point (red dot). The right panel shows water surface elevation predicted by GPU-HWFV1 and GPU-FV1 at the gauge point. 64

3.20 Tsunami wave propagation over a complex beach. Time histories of GPU-HWFV1's non-uniform grid cell count (top left panel), Δt (top right panel), speedup over CPU-HWFV1 (bottom left panel) and over GPU-FV1 (bottom right panel).. 65

4.1 Performing the MRA process of the MW on the GPU. The left panel shows a hierarchy of grids involved in the MRA process with a maximum refinement level $L = 2$ indexed according to Z-order curves. The right panel shows how four cells in a grid at refinement level $n + 1$, i.e. the "children," are positioned relative to a single cell in a grid at refinement level n, i.e. the "parent" – as well as the locations of the children in memory. 73

4.2 Example of a parameter file showing the parameters needed to run a simulation of the Monai valley test case (Section 5.2) using GPU-MWDG2. The extra parameters needed to correctly use GPU-MWDG2's grid adaptation capability within LISFLOOD-FP's framework for running real-world flood simulations are highlighted in bold. 76

4.3 Initial non-uniform grid generated by GPU-MWDG2 based on the details of the bathymetry and initial flow conditions of the Monai valley test case (Section 5.2). . . 79

4.4 GPU memory consumed by GPU-MWDG2. The left panel shows the percentage breakdown of the memory consumed by the objects in GPU-MWDG2. The right panel shows the amount of GPU memory allocated by GPU-MWDG2 against the maximum refinement level L. The numbers on top of the bars show the largest test case area allowed for a given value of L. The horizontal lines indicate the memory capacities of four GPUs. 80

4.5 Non-uniform grids generated by GPU-MWDG2 for the Monai valley test case (Section 5.2). The left panel shows the initial grid generated by GPU-MWDG2 at the start of the simulation. The right panel shows the intermediate grid generated by GPU-MWDG2 after the simulation has progressed by 17 s, which has a lower amount of coarsening than the initial grid due to flow features produced by the tsunami as it enters and travels through the test case area. 84

5.1 Monai valley test case. Top-down view of bathymetry (top left panel), where the red arrows indicate the direction and distance travelled by the tsunami entering from the left boundary; time history of the free surface elevation of the tsunami at the left boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels). 91

5.2 Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Monai valley test case. 93

5.3 Time series of the water surface elevation $(h + z)$ predicted by GPU-DG2 and GPU-MWDG2 at the three points shown in Figure 5.1 for the Monai valley test case. . . . 94

5.4 Seaside, Oregon test case. Top-down view of bathymetry (top left panel); tsunami time history at the left boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels). 96

5.5 Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Seaside, Oregon test case. 97

5.6 Time series of the water surface elevation $(h + z)$, x-directional velocity (u) and momentum (M_x) produced by GPU-DG2 and GPU-MWDG2 at points A1, B6 and D4 in Figure 5.4. 99

5.7 Tauranga harbour test case. Top-down view of bathymetry (top left panel); tsunami time history at the top boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels). 101

5.8 Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Tauranga harbour test case. 102

5.9 Time series of the water surface elevation $(h+z)$ produced by GPU-DG2 and GPU-MWDG2 at the points labelled A Beacon, Tug Harbour, Sulphur Point and Moturiki and of the speed at the point labelled “ADCP” in Figure 5.7. 103

5.10 Hilo harbour test case. Top-down view of bathymetry (top left panel); tsunami time history at the top boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels). 105

5.11 Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Hilo harbour test case. 107

5.12 Time series simulated by GPU-DG2 and GPU-MDG2 at the labelled points in Figure 5.10. 108

A.1 "Host-device" framework of a CUDA program representing a scientific computing program. In a CUDA program, all operations are handled by the "host", the CPU, except for heavy-duty parallel computation, which is *offloaded* by the host to the "device", the GPU. 153

A.2 GPU hardware architecture; (a) the GPU is made up of several "streaming multiprocessors" (SM) which are responsible for computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which SMs access via an L2 cache; (b) each SM is made up of hardware for supporting computation: registers, shared/L1/texture cache, constant cache and "warp schedulers", as well as several "threads" for computation; (c) each thread is made up of functional units for executing various types of instructions. 154

A.3 Warp divergence; a warp is a group of 32 threads with consecutive thread IDs that operate in lockstep. 156

A.4 Example of the behaviour of a warp scheduler with eight warp slots over four clock cycles. 157

A.5 Memory hierarchy involved in the CUDA programming model. 159

A.6 Optimal and non-optimal access patterns by threads in a warp when accessing global memory via the L2 and "L1 caches" (a) an optimal access leading to coalesced memory access; (b) a non-optimal access pattern leading to uncoalesced memory access. 160

A.7 Shared memory bank conflicts for threads in a warp; (a) no bank conflicts; (b) two-way bank conflicts. 162

List of Tables

3.1 Test cases to compare GPU-HWFV1 against GPU-FV1 and CPU-HWFV1 48

3.2 Aspects ε, L, and flow type against which GPU-HWFV1's speedup over CPU-HWFV1 and GPU-FV1 is systematically analysed. 49

3.3 Zero-velocity initial conditions for generating an unmoving water surface flow for the three hump shapes shown in Figure 3.10. 53

4.1 Functionalities of the extra parameters that must be specified in the parameter file to correctly use GPU-MWDG2's grid adaptation capability (highlighted in bold in Figure 4.2). 78

4.2 Description of the quantities in the `.cumu` output file. GPU-MWDG2 produces this file at the end of a simulation if the `cumulative` keyword has been typed in the parameter file. 82

4.3 Time-dependent metrics for evaluating the potential speedup afforded by GPU-MWDG2's grid adaptation capability. 83

5.1 Summary of four tsunami test cases each with a unique combination of DEM size and tsunami complexity. Parameters needed to run the simulations per test case: maximum refinement level L, simulation end time, denoted by t_{end}, and Manning's friction coefficient, denoted by n_{M}. 89

5.2 RMSE and r values of the water surface elevation predicted by GPU-MWDG2 with respect to the GPU-DG2 predictions for the Monai valley test case. 94

5.3 RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Seaside, Oregon test case 100

5.4 RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Tauranga harbour test case 104

5.5 RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Hilo harbour test case 109

5.6 Summary of key set-up parameters and results per test case. The runtime is the time taken to complete a simulation of a test case. The numbers 10^{-3} and 10^{-4} refer to the value of ε chosen for GPU-MWDG2. 110

List of Algorithms

1 Pseudocode for the process of computing $s^{(n)}$, $d_\alpha^{(n)}$, $d_\beta^{(n)}$, and $d_\gamma^{(n)}$ at the lower refinement levels, called 'encoding'. 35

2 Pseudocode for performing a depth-first traversal of the tree of significant details (obtained after encoding) while applying Equations 3.11a - 3.11d, called "decoding". 37

3 Pseudocode summarising the steps involved in running CPU-HWFV1. 39

4 Pseudocode for performing the decoding step in parallel. 42

5 Pseudocode for the parallel tree traversal algorithm. The algorithm features an iterative procedure (lines 4 to 16) instead of the recursive procedure in the depth-first traversal in Algorithm 2. 46

Publications

- Mohammed Kazem Sharifian, Georges Kesserwani, **Alovya Ahmed Chowdhury**, Jeffrey Neal, Paul Bates 2023. *LISFLOOD-FP 8.1: new GPU-accelerated solvers for faster fluvial/-pluvial flood simulations*, Geoscientific Model Development, 16, 2391–2413, doi: https://doi.org/10.5194/gmd-16-2391-2023.
- **Alovya Ahmed Chowdhury**, Georges Kesserwani, Charles Rougé, Paul Richmond 2023. *GPU-parallelisation of Haar wavelet-based grid resolution adaptation for fast finite volume modelling: application to shallow water flows*, Journal of Hydroinformatics, 25 (4): 1210–1234. doi: https://doi.org/10.2166/hydro.2023.154.
- **Alovya Ahmed Chowdhury**, Georges Kesserwani 2024. *LISFLOOD-FP 8.2: GPU-accelerated multiwavelet discontinuous Galerkin solver with dynamic resolution adaptivity for rapid, multiscale flood simulation*, Geoscientific Model Development. *In review.*

Chapter 1

Introduction

1.1 Research background

Flooding caused by rapid flow phenomena, i.e. impact events such as dam breaks and tsunamis, has severely negative effects on human communities. To help mitigate these negative effects, the extent and severity of flooding caused by such rapid flows are often predicted by running simulations of real-world flooding scenarios using hydrodynamic modelling frameworks. Yet, these modelling frameworks must find an appropriate compromise between the accuracy of the flood simulations – which must be good enough to sufficiently predict real-world flood risk – and the computational effort required to run the simulations – which must be fast enough for practical use: generally, the higher the level of accuracy that is desired, the higher the computational effort that must be dealt with (Teng et al., 2017; Nkwunonwo et al., 2020; Guo et al., 2021; Tom et al., 2022; Kumar et al., 2023).

When simulating flooding caused by rapid flows, notably in tsunami-induced flooding, hydrodynamic modelling frameworks based on numerically solving the two-dimensional (2D) shallow water equations (SWE) impose a relatively low level of computational effort (compared to more complicated equations such as the three-dimensional (3D) Navier-Stokes equations) while still maintaining an acceptable level of accuracy (Brocchini and Dodd, 2008; Synolakis et al., 2008; Apotsos et al., 2011; Gayathri et al., 2017). Still, to maintain an acceptable level of accuracy when simulating *multiscale* rapid flows, i.e. rapid flows that occur over multiple spatial and temporal scales, the SWE should be solved using numerical schemes of second-order accuracy or higher to alleviate the numerical diffusion that would otherwise be introduced by using first-order finite volume (FV1) schemes (George, 2006; LeVeque et al., 2011; Macías et al., 2020a; Ayog et al., 2021).

To achieve second-order accuracy, second-order finite volume (FV2) schemes have been proposed by several authors (Yoon and Kang, 2004; Begnudelli et al., 2008; Aureli et al., 2008; Castro Díaz et al., 2009; Hou et al., 2015; Reis et al., 2018; Zhao et al., 2019; Guerrero Fernández et al., 2020). An interesting alternative to FV2 schemes however for achieving second-order accuracy are discontinuous Galerkin (DG) schemes (Aizinger et al., 2017; Kesserwani et al., 2018; Ghostine et al., 2019; Li and Zhang, 2021). Open-source DG-based hydrodynamic modelling frameworks are relatively uncommon compared to FV-based ones – see Thetis (Kärnä et al., 2018), SLIM (Delandmeter et al., 2018), DGSWEM (Dawson et al., 2011), or LISFLOOD-FP 8.0 (Shaw et al., 2021) as examples. LISFLOOD-FP in particular has a rich set of features for running real-world flood simulations (Bates and De Roo, 2000) and includes a variety of numerical schemes for solving the SWE, including a second-order DG (DG2) scheme – which achieves the highest level of accuracy out of all the numerical schemes in LISFLOOD-FP 8.0, but also imposes the highest computational effort (Shaw et al., 2021). To handle the high computational effort imposed by the DG2 model in LISFLOOD-FP 8.0, it has been parallelised on the graphics processing unit (GPU) to enable higher speedups – referred to hereafter in this thesis as "GPU-DG2". Nevertheless, even with the speedups afforded by GPU-parallelisation, using GPU-DG2 to simulate multiscale rapid flows on a uniform mesh or grid leads to an excessively high computational effort due to the enormous number of cells from using a uniform grid to cover a real-world test case area (Popinet, 2012; Lee, 2017; Kesserwani and Sharifian, 2023).

To reduce the computational effort imposed by the cell count of a uniform grid, FV-based tsunami models typically employ grid resolution adaptation to generate a non-uniform grid that uses fewer cells than a uniform grid (Popinet and Rickard, 2007; LeVeque et al., 2011; Berger et al., 2011; Popinet, 2011, 2012; Liang et al., 2015a; Lee, 2017; de la Asunción and Castro, 2017; Ferreira and Bader, 2017; Park et al., 2019; Qin et al., 2019). Comparatively though, the use of grid adaptation in DG-based tsunami models seems to be less common: most works focus on simulating global-scale tsunami propagation (Blaise and St-Cyr, 2012; Blaise et al., 2013; Bonev et al., 2018; Hajihassanpour et al., 2019; Arpaia et al., 2022), while a limited number of works have focussed on simulating tsunami-induced flooding (Gandham et al., 2015; Castro et al., 2016; Rannabauer et al., 2018). Overall, it seems that there are no DG-based hydrodynamic modelling frameworks that have combined both GPU-parallelisation and grid adaptation to thereby substantially reduce the computational effort of DG2 modelling when simulating multiscale rapid flows.

The tsunami models cited above have implemented grid adaptation using approaches that are similar to a classic grid adaptation algorithm called adaptive mesh refinement (AMR) (Berger and Colella, 1989). However, AMR cannot guarantee a similar level of accuracy as delivered by a SWE model running on a uniform grid without AMR due to issues in preserving robustness (Zhou et al., 2013; Donat et al., 2014; Mandli and Dawson, 2014; Liang et al., 2015b; Pons et al., 2017; Hou et al., 2018; Ghazizadeh et al., 2020; Zhang et al., 2021). To avoid these issues, a promising substitute for AMR is *wavelet-based* grid adaptation, which is purposefully designed to readily deliver a similar level of accuracy as a SWE model running on a uniform grid (Gerhard et al., 2015a; Caviedes-Voullième and Kesserwani, 2015; Kesserwani et al., 2015a; Haleem et al., 2015; Caviedes-Voullième et al., 2020). Wavelet-based adaptive SWE models have been shown to achieve a similar level of accuracy as their reference uniform-grid counterparts while seeing up to 20-fold speedups when implemented on the central processing unit (CPU) (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020).

The speedups and accuracy achieved by wavelet-based grid adaptation suggest that it is a good choice to parallelise on the GPU and thereby integrate a grid adaptation capability into GPU-DG2. Nonetheless, wavelet-based grid adaptation algorithms have til now mainly been parallelised on CPU-based platforms rather than on GPUs (Domingues et al., 2019; Soni et al., 2019; Deiterding et al., 2020; Semakin and Rastigejev, 2020; Engels et al., 2021; Zeidan et al., 2022). The prevalence of parallelising wavelet-based grid adaptation algorithms on CPUs rather than on GPUs is largely because grid adaptation algorithms have an algorithmic structure that is incompatible with GPU computing (Wang et al., 2010; Burstedde et al., 2010; Ji et al., 2015; Schive et al., 2018b; Qin et al., 2019).

As such, redesigning the algorithmic structure of a wavelet-based grid adaptation algorithm in order to make it suitable for efficient implementation on the GPU and thereby integrate a wavelet-based grid adaptation capability into GPU-DG2 that reduces the computational effort of DG2 modelling when simulating multiscale rapid flows – while still achieving a similar level of accuracy as GPU-DG2 when run on a uniform grid – is a compelling research aim for this thesis.

1.2 Research aim and objectives

The research aim of this thesis is to integrate a wavelet-based grid adaptation capability into the GPU-DG2 model in LISFLOOD-FP 8.0 in order to reduce the computational effort of DG2 modelling when simulating multiscale rapid flow phenomena – while still achieving a similar level of accuracy as GPU-DG2 when run on a uniform grid. To achieve the thesis aim, the following three Objectives are defined:

1. Propose computational ingredients for redesigning the algorithmic structure of a wavelet-based grid adaptation algorithm based on "Haar" wavelets (HW) in order to make it suitable for efficient implementation on the GPU;

2. Extend the use of the computational ingredients proposed in Objective 1 in order to efficiently implement a wavelet-based grid adaptation algorithm based on "multiwavelets" (MW) on the GPU and integrate it into the GPU-DG2 model in LISFLOOD-FP;

3. Analyse the impact of the GPU-parallelised MW-based grid adaptation algorithm on the accuracy and computational effort of DG2 modelling when simulating multiscale rapid flow phenomena.

These Objectives are addressed in the remaining chapters of the thesis, which is structured as follows.

1.3 Thesis structure

This thesis is structured as follows. Chapter 2 conducts a literature review, providing more background in Section 2.1 on why combining DG2 modelling, GPU computing and grid adaptation in order to efficiently simulate multiscale rapid flow phenomena is a compelling yet uninvestigated research area, arguing that the lack of investigation is because grid adaptation algorithms are incompatible with GPU computing. To expand on this argument, an overview of GPU computing is given in Section 2.2, and thereafter, informed by this overview, the specific challenges in efficiently implementing grid adaptation algorithms on the GPU are discussed in Section 2.3.

Bearing in mind the challenges in efficiently implementing grid adaptation algorithms on the GPU, Chapter 3 addresses Objective 1 by proposing three computational ingredients to redesign the algorithmic structure of a HW-based grid adaptation algorithm and make it suitable for efficient implementation on the GPU, thereby allowing to develop a GPU-parallelised HW adaptive FV1 shallow water model – referred to hereafter in this thesis as "GPU-HWFV1": the HW cannot be used to integrate a wavelet-based grid adaptation capability into GPU-DG2 – for which the MW are applicable instead – but the HW are less complicated than the MW and therefore its GPU implementation is explored first, as an easier intermediate step. To confirm that the proposed computational ingredients indeed allow for an efficient implementation of the HW-based grid adaptation algorithm on the GPU, the developed GPU-HWFV1 model should be at least as fast and accurate as a GPU-parallelised FV1 shallow water model running on a uniform grid: Chapter 3 therefore analyses the efficiency of GPU-HWFV1's HW-based grid adaptation capability, namely by assessing GPU-HWFV1's speedup over its GPU-parallelised uniform-grid FV1 counterpart.

Having analysed the efficiency of GPU-HWFV1's HW-based grid adaptation capability, Chapter 4 addresses Objective 2 by extending the use of the computational ingredients proposed in Chapter 3 to efficiently implement a MW-based grid adaptation algorithm on the GPU and thereby develop a GPU-parallelised MW adaptive DG2 shallow water model, referred to hereafter in this thesis as "GPU-MWDG2". Chapter 4 then describes how GPU-MWDG2 is integrated into a new version of LISFLOOD-FP (version 8.2) in order to take advantage of LISFLOOD-FP's existing framework for running real-world flood simulations and thereby properly compare GPU-MWDG2 against GPU-DG2 – as the latter is already integrated into LISFLOOD-FP.

After GPU-MWDG2 is developed and integrated into LISFLOOD-FP, Chapter 5 addresses Objective 3 by comparing GPU-MWDG2 against GPU-DG2 over a series of test cases involving tsunami-induced flooding in order to analyse the impact of GPU-MWDG2's MW-based grid adaptation capability on the accuracy and computational effort of DG2 modelling when simulating multiscale rapid flow phenomena.

Chapter 6 provides conclusions obtained from exploring the research aim and addressing the Objectives, ending with a summary of significant research findings, limitations and potential future work for addressing the limitations.

Chapter 2

Literature review

2.1 Modelling floods caused by rapid flow phenomena

Numerous hydrodynamic modelling frameworks are available for simulating floods caused by rapid flow phenomena such as dam breaks and tsunamis – especially for simulating tsunami-induced flooding, owing to the damage that tsunamis can often cause. Many of these modelling frameworks are based on numerically solving a set of partial differential equations that govern hydrodynamic flow over a computational mesh or grid, such as 1) the 3D Navier-Stokes equations, e.g. Tsunami3D (Horrillo et al., 2013), 2) 3D hydrostatic equations, e.g. Thetis (Kärnä et al., 2018), 3) 2D depth-averaged non-hydrostatic or Boussinesq-type equations, e.g. FUNWAVE (Bruno et al., 2009) or NEOWAVE (Lay et al., 2011), or 4) the 2D SWE, e.g. MOST (Titov et al., 2016), Tsunami-HySEA (Macías et al., 2020b), GeoClaw (Berger et al., 2011), COMCOT (Liu et al., 1998), TsunAWI (Rakowsky et al., 2013), NAMIDANCE (Zaytsev et al., 2019) and TUNAMI (Imamura et al., 2006), among others.

When simulating tsunami-induced flooding, hydrodynamic models based on solving the SWE impose the least computational effort out of all the aforementioned equations while still maintaining an acceptable level of accuracy (Brocchini and Dodd, 2008; Synolakis et al., 2008; Apotsos et al., 2011; Gayathri et al., 2017). Nonetheless, many SWE models are rather developed for simulating floods caused by relatively slow flows such as in fluvial and pluvial flooding, in which regions of interest are flooded rather slowly compared to in rapid flows, e.g. in flood events driven by slow flows branching out from rivers in a floodplain in a fractal pattern. When simulating such slow flows, SWE models based on FV1 schemes parallelised on GPUs are the *de facto* standard class of models for running real-world flood simulations (Xia et al., 2019; Echeverribar et al., 2019; Dazzi et al., 2020; Gordillo et al., 2020; Carlotto et al., 2021; Morales-Hernández et al., 2021; Buttinger-Kreuzhuber et al., 2022; Delmas and Soulaïmani, 2022; Caviedes-Voullième et al., 2023; Sanz-Ramos et al., 2023). In contrast, when simulating rapid flows, where regions of interest are quickly and completely flooded by impact events travelling rapidly through the bathymetric area, e.g. tsunamis or dam-break waves, and in particular when modelling *multiscale* rapid flows that occur over multiple spatial scales (e.g. when metre-scale flow features propagate over kilometre-scale domains) and temporal scales (e.g. when flow features propagate in second-scale timesteps over long-duration simulations of many days), numerical schemes of at least second-order accuracy are required to alleviate the excessive numerical diffusion – i.e. the smearing of the numerical solution in a way that resembles the physical process of diffusion – that is imposed by FV1 schemes which would otherwise smear the wavefronts of impact events that drive rapid flows in order to maintain an acceptable level of accuracy (George, 2006; LeVeque et al., 2011; Macías et al., 2020b; Ayog et al., 2021).

To achieve second-order accuracy, FV2 schemes have been proposed which reconstruct a linear polynomial solution over each cell in the mesh using data belonging to a cell's neighbours: due to the reconstruction process, each cell ultimately needs access to its neighbours' neighbours, i.e. to its *non-local* neighbours (Yoon and Kang, 2004; Begnudelli et al., 2008; Aureli et al., 2008; Castro Díaz et al., 2009; Hou et al., 2015; Reis et al., 2018; Zhao et al., 2019; Guerrero Fernández et al., 2020). Despite achieving the desired second-order accuracy, the non-locality of FV2 schemes makes it difficult to add robustness features (e.g. wetting-and-drying or friction effects) and/or efficiency features (e.g. parallelisation or grid resolution adaptation) to SWE models compared to FV1 schemes (Yoon and Kang, 2004; Begnudelli and Sanders, 2007; Hou et al., 2013; Kesserwani et al., 2015a).

To avoid non-locality while retaining second-order accuracy, a DG scheme can be adopted that *locally* constructs a linear (or higher-order) polynomial solution over each cell in the computational grid by using data belonging to a cell itself rather than by using data belonging to a cell's neighbours (Aizinger et al., 2017; Kesserwani et al., 2018; Ghostine et al., 2019; Li and Zhang, 2021). The locality of DG schemes simplifies the implementation of robustness features that are essential for

SWE models to achieve an acceptable level of accuracy in real-world flood simulations, such as well-balancing (Xing and Shu, 2006a,b; Kesserwani et al., 2010), wetting-and-drying (Bokhove, 2005; Ern et al., 2008; Bunya et al., 2009; Xing et al., 2010; Kärnä et al., 2011; Kesserwani and Liang, 2012a; Zhang et al., 2021), slope limiting (Krivodonova et al., 2004; Krivodonova, 2007; Kesserwani and Liang, 2012b; Vater et al., 2015; Aizinger et al., 2017; Vater et al., 2019) and friction effects (Kesserwani and Liang, 2010; Kesserwani et al., 2010; Yang et al., 2021). The locality of DG schemes also makes them readily open to being enhanced with efficiency features that are essential for dealing with the enormous computational effort of running real-world flood simulations on fine-resolution and/or large computational grids, in particular parallelisation (Castro et al., 2016; Gandham et al., 2015; Brus et al., 2017; Rannabauer et al., 2018; Wintermeyer et al., 2018; Shaw et al., 2021) and grid resolution adaptation (Aizinger and Dawson, 2002; Remacle et al., 2006; Bernard et al., 2007; Kubatko et al., 2006; Eskilsson, 2011; Bader et al., 2010; Kesserwani and Liang, 2012c, 2015; Beisiegel et al., 2020). Finally, compared to FV2 schemes, DG schemes have been shown to deliver equally-accurate results on coarse-resolution grids (Kesserwani, 2013; Kesserwani and Wang, 2014; Vater et al., 2017; Shaw et al., 2021) and to achieve higher quality velocity predictions (Ayog et al., 2021; Sun et al., 2023; Kesserwani et al., 2023). All of the aforementioned features suggest that a DG-based SWE model is well-suited for accurately and efficiently simulating multiscale rapid flow phenomena.

Compared to open-source FV-based hydrodynamic modelling frameworks, DG-based hydrodynamic modelling frameworks are less common; see for example Thetis for ocean circulation modelling (Kärnä et al., 2018), SLIM for ocean-ice interaction modelling (Delandmeter et al., 2018), DGSWEM for storm surge modelling (Dawson et al., 2011), and LISFLOOD-FP 8.0 for floodplain inundation modelling (Shaw et al., 2021). LISFLOOD-FP in particular has a rich set of features for running real-world flood simulations on 2D raster grids (Bates and De Roo, 2000), such as (1) the ability to use raster grid digital elevation models (DEM) of real-world topographies obtained from LiDAR satellite data, (2) easy configuration of model and simulation parameters via a simple text file, and (3) input/output of flood maps and velocity fields in a standard Esri geographic information system (GIS) file format for convenient processing in GIS software. LISFLOOD-FP 8.0 has a variety of numerical schemes for solving the SWE including a second-order DG (DG2) scheme, which achieves the highest level of accuracy out of all the numerical schemes in LISFLOOD-FP 8.0

– but also imposes the highest computational effort (Shaw et al., 2021).

To handle the computational effort imposed by the DG2 model in LISFLOOD-FP 8.0, it has been parallelised on the GPU to enable higher speedups, referred to as GPU-DG2. However, even with the speedups afforded by GPU-parallelisation, DG2 modelling of multiscale rapid flows using GPU-DG2 on a uniform grid may lead to an unnecessarily high computational effort (Kesserwani and Sharifian, 2023): this high computational effort may arise when simulating a multiscale rapid flow phenomenon like a tsunami because a fine metre-scale grid resolution is only needed around complex bathymetric details and flow features like tsunami wavefronts, whereas elsewhere in the test case area, a coarse kilometre-scale grid resolution may be sufficient (Popinet, 2012; Lee, 2017). Thus, using a uniform grid with a metre-scale grid resolution to cover the entire test case area leads to an unnecessarily high number of cells in the grid and a correspondingly high computational effort.

To reduce the computational effort imposed by the high cell count of a uniform grid, dynamic-in-time grid resolution adaptation is widely used in FV-based tsunami modelling (Popinet and Rickard, 2007; LeVeque et al., 2011; Berger et al., 2011; Popinet, 2011, 2012; Liang et al., 2015a; Lee, 2017; de la Asunción and Castro, 2017; Ferreira and Bader, 2017; Park et al., 2019; Qin et al., 2019): grid adaptation is a process for generating a *non-uniform* grid that uses finer-resolution (i.e. smaller) cells in the grid only when and where the flow and bathymetric details are significant and uses coarser-resolution (i.e. larger) cells otherwise, thereby reducing the cell count of the grid and thus the computational effort without compromising the accuracy of the grid.

Compared to FV-based tsunami modelling, the use of grid adaptation in DG-based tsunami modelling – and more broadly, the use of DG methods in tsunami modelling in general – seems to be less common (Blaise and St-Cyr, 2012; Blaise et al., 2013; Bonev et al., 2018; Hajihassanpour et al., 2019; Arpaia et al., 2022; Gandham et al., 2015; Castro et al., 2016; Rannabauer et al., 2018). Many of these works are aimed at simulating global-scale tsunami propagation by solving the SWE over a spherical mesh: Blaise and St-Cyr (2012) developed a DG model called MUSE that combined parallel computing CPUs with both mesh adaptation and polynomial solution order adaptation, for which wetting-and-drying was not considered. Blaise et al. (2013) added an adjoint method to MUSE to achieve efficient tsunami source reconstruction and optimisation. Arpaia et al. (2022) presented a mixed 2D/3D DG scheme with modifications for conveniently achieving well-balancing, although they used reflective land boundaries to avoid the issue of wetting-and-drying.

Bonev et al. (2018) developed a DG model using a well-balanced DG scheme with wetting-and-drying, and Hajihassanpour et al. (2019) extended this model by adding and testing static and dynamic earthquake tsunami source generation mechanisms.

A relatively smaller number of works have also investigated DG-based modelling of tsunami-induced flooding: Rannabauer et al. (2018) used a well-balanced ADER-DG scheme with wetting-and-drying to simulate tsunami-induced flooding and combined grid adaptation with parallel computing on CPUs. Castro et al. (2016) presented a GPU-parallelised ADER-DG scheme for simulating tsunami-induced flooding and noted the effectiveness of parallelising DG schemes on the GPU. Gandham et al. (2015) presented a SWE model with a well-balanced DG scheme including wetting-and-drying, and combined local time stepping with GPU-parallelisation. Overall however, it appears that there are no DG-based hydrodynamic modelling frameworks that have adopted both GPU-parallelisation and grid adaptation to substantially reduce the computational effort of DG2 modelling when simulating multiscale rapid flows.

The works cited above on DG-based tsunami modelling have implemented grid adaptation using approaches that are similar to a classic grid adaptation algorithm called AMR (Berger and Colella, 1989). However, using classic AMR to enhance a SWE model with grid adaptation cannot guarantee a similar level of accuracy as delivered by the same SWE model running on a uniform grid, because AMR uses heuristic formulae based on averaging and interpolation to scale flow and topographic information across different refinement levels that were not derived with shallow water modelling in mind, therefore leading issues in robustness properties such as well-balancing (Donat et al., 2014; Mandli and Dawson, 2014; Ghazizadeh et al., 2020; Zhang et al., 2021), wetting-and-drying over complex topography (Zhou et al., 2013; Hou et al., 2018) and mass conservation (Liang et al., 2015b; Pons et al., 2017). To avoid these issues, a promising alternative to AMR is *wavelet-based* grid adaptation: unlike AMR, wavelet-based grid adaptation uses rigorously derived formulae based on wavelets that explicitly keep in mind and incorporate robustness treatments originally devised for SWE models running on uniform grids, thereby delivering a similar level of accuracy as a uniform-grid SWE model and readily preserving the robustness properties of the *reference* numerical scheme that the uniform-grid SWE model is built on (Gerhard et al., 2015a; Caviedes-Voullième and Kesserwani, 2015; Kesserwani et al., 2015a; Haleem et al., 2015; Caviedes-Voullième et al., 2020). Namely, wavelet-based grid adaptation is based on enriching an existing numerical scheme designed to run on a uniform grid – i.e. the so-called the reference scheme – with the “multiresolution analysis” (MRA) of wavelets, such as HW for enriching a reference FV1 scheme (Haleem et al., 2015; Chowdhury et al., 2023) or MW for enriching a reference DG2 scheme (Kesserwani and Sharifian, 2020; Caviedes-Voullième et al., 2020). Enriching the reference scheme with wavelet-based grid adaptation allows to develop a wavelet-based adaptive SWE model, in which the deviation or error allowed to occur from the reference uniform-grid counterpart can be rigorously controlled by a single user-specified error threshold ε (Gerhard et al., 2015b; Hovhannisyan et al., 2014; Gerhard et al., 2015a). Subject to appropriate choices for ε, wavelet-based adaptive SWE models have been shown to readily deliver a similar level of accuracy as the reference uniform-grid counterpart while seeing up to 20-fold speedups when implemented on the CPU (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020).

The speedups and high level of accuracy achieved by wavelet-based grid adaptation suggest that the MRA process of the MW is a good choice to parallelise on the GPU and thereby integrate a grid adaptation capability to GPU-DG2 in order to reduce the computational effort of DG2 modelling while still achieving a similar level of accuracy as GPU-DG2 when run on a uniform grid. However, the parallelisation of wavelet-based grid adaptation algorithms on the GPU remains unreported: rather, all reports about parallelising wavelet-based grid adaptation algorithms have been focussed on CPU platforms (Domingues et al., 2019; Soni et al., 2019; Deiterding et al., 2020; Semakin and Rastigejev, 2020; Engels et al., 2021; Zeidan et al., 2022). Parallelisation on CPU platforms involves using (1) OpenMP (OpenMP, 1998) for achieving parallelisation on a single CPU and/or (2) MPI (MPIForum, 2011) for achieving parallelisation across multiple CPUs: Domingues et al. (2019) developed a wavelet-based magnetohydrodynamics model parallelised using MPI via the AMROC framework. Deiterding et al. (2020) used the same AMROC framework to compare wavelet-based grid adaptation against gradient-based coarsening and refinement methods when applied to the Euler equations using MPI and found that 20-40% fewer cells were required on the grid. Soni et al. (2019) parallelised a wavelet-based solver of the Euler equations using OpenMP, MPI, and a combination of OpenMP and MPI by introducing a data structure called a “multiresolution forest structure”. Semakin and Rastigejev (2020) parallelised a wavelet-based chemical transport model using MPI by devising a special graph data structure. Engels et al. (2021) developed a parallelised wavelet-based model using MPI to simulate flow caused by flapping insects called WABBIT, in which the grid hierarchy was stored using a tree-like data structure. Gillis and van Rees (2022) presented a wavelet grid adaptation framework with MPI parallelisation called MURPHY, which was applied to simulate the advection of a scalar quantity. Zeidan et al. (2022) parallelised a wavelet-based model on a single CPU using OpenMP to analyze two-phase flow.

Most of the works cited above focussed on developing techniques to efficiently parallelise wavelet-based grid adaptation across multiple CPUs on distributed memory architectures using MPI, which are not readily applicable to GPU-parallelisation. Some of the works explored techniques for CPU-based shared memory architectures, which require a level of programming flexibility that is not available on GPUs. Other works did not sufficiently explore techniques such as memory coalescing that are merely beneficial to include on CPUs, but which are essential to include on GPUs. Lastly, there are programming considerations for GPUs that are simply not applicable for CPUs, such as warp divergence, occupancy, scheduling, etc. As such, none of the works cited above were aimed at implementing wavelet-based grid adaptation on the GPU. This is likely because implementing a wavelet-based grid adaptation algorithm on the GPU is difficult (and may even be counterproductive) because grid adaptation algorithms have an algorithmic structure that is incompatible with GPU computing, severely degrading the potential efficiency of a GPU-parallelised grid adaptation algorithm (Wang et al., 2010; Burstedde et al., 2010; Ji et al., 2015; Raghavan and Vadhiyar, 2015; Schive et al., 2018b; Qin et al., 2019). To understand the incompatibility of grid adaptation algorithms with GPU computing, it is instructive to review GPU computing first (Section 2.2), and thereafter, informed by this review, the challenges associated with implementing grid adaptation algorithms on the GPU can be discussed (Section 2.3).

2.2 GPU computing using CUDA

Referring to the CUDA programming guide (NVIDIA, 2023), this section reviews the essential concepts in GPU computing that are necessary for understanding the challenges associated with efficiently implementing a grid adaptation algorithm on the GPU using CUDA (Section 2.3). More in-depth guidance about GPU computing has been included in the Appendix.

The CUDA programming model – developed by NVIDIA exclusively for use with NVIDIA GPUs – remains one of the most popular GPU programming models to date. The CUDA programming model adopts a “host-device” framework in which the CPU, i.e. the “host”, is responsible for everything in a CUDA program except for heavy-duty parallel computation, which is offloaded by the host to the “device”, i.e. the GPU. The GPU performs the parallel computation by executing a program called a “kernel”, which is made up of many parallel “threads”.

An appropriate number of threads should be launched for handling the computation described by a kernel. For example, a kernel that describes adding two vectors of length 1024 should launch 1024 threads. In CUDA, threads are grouped into “blocks”, and blocks are grouped into a “grid” (note that the term “grid” in the context of CUDA programming should not be confused with the computational grid over which the SWE are solved). The block size and grid size should be chosen to launch a kernel with an appropriate number of threads: for example, to launch a kernel with 1024 threads, a block size of 256 and a grid size of 4 could be chosen, since 256 threads per block $\times$ 4 blocks in the grid $=$ 1024 threads in the grid. The block size and/or grid size are usually chosen empirically to maximise the speed of a kernel.

The speed of a kernel should be maximised in order to complete the parallel computation described by the kernel as quickly as possible. To improve the speed of a kernel, the kernel should be designed – i.e. programmed – to use GPU hardware efficiently. However, efficiently using the GPU hardware is difficult due to the complexity of GPU hardware architecture, as shown next.

2.2.1 GPU hardware architecture

GPU hardware architecture is complex as it involves many different hardware components whose behaviour must be considered when designing a kernel to efficiently use the hardware. Figure 2.1 shows the hardware architecture of a GPU. As shown in Figure Figure 2.1a, a GPU is made up of several “streaming multiprocessors” (SM) that handle computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which the SMs access via an “L2 cache” – a hardware component for storing small amounts of data for faster access (compared to e.g. GPU DRAM). The number of SMs and the amount of DRAM depend on the specific GPU being used. Each SM has several threads for directly performing computations, as well as other hardware for supporting computation. As mentioned, the more efficiently the GPU hardware is used by a kernel, the higher the speed of the kernel. To design a kernel that uses the GPU hardware efficiently, it is necessary to understand how the execution of the kernel is coupled with the GPU hardware.

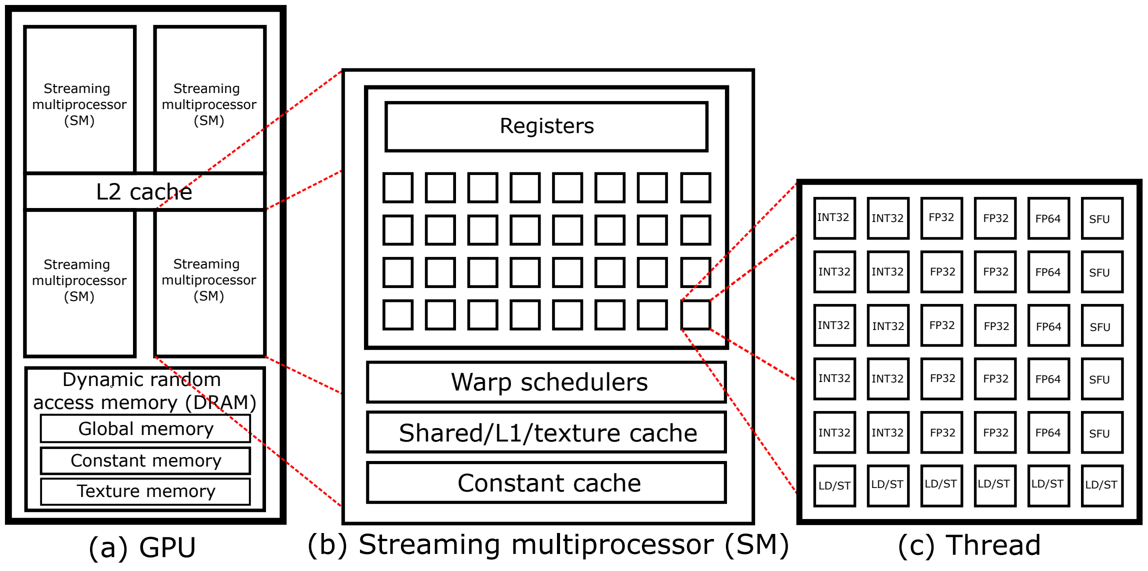
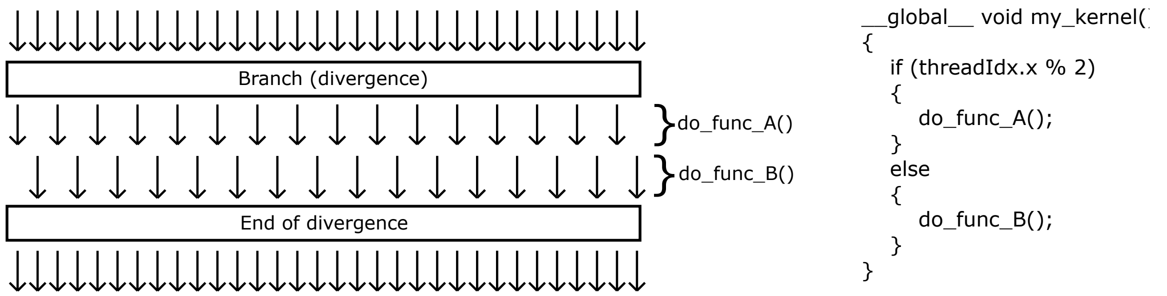

Streaming multiprocessor (SM)
Streaming multiprocessor (SM)
L2 cache
Streaming multiprocessor (SM)
Streaming multiprocessor (SM)
Dynamic random access memory (DRAM)
Global memory
Constant memory
Texture memory
Registers
Warp schedulers
Shared/L1/texture cache
Constant cache
INT32
FP32
FP64
SFU
LD/ST
(a) GPU
(b) Streaming multiprocessor (SM)
(c) Thread

Figure 2.1: *GPU hardware architecture; (a) the GPU is made up of several "streaming multiprocessors" (SM) which are responsible for computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which SMs access via an L2 cache; (b) each SM is made up of hardware for supporting computation: registers, shared/L1/texture cache, constant cache and "warp schedulers", as well as several "threads" for computation; (c) each thread is made up of functional units for executing various types of instructions.*

Recall that during kernel execution, the kernel launches many threads that are grouped into blocks, and the blocks are organised into a grid. The blocks are distributed among the SMs for execution: for example, if a grid is made up of 128 blocks and there are 16 SMs, there can be up to 128 blocks per 16 SMs = 8 blocks per SM. Blocks are partitioned into "warps", which are groups of 32 consecutive threads: for example, a block made up of 512 threads is made up of 512 threads / 32 threads per warp = 16 warps. Warps execute instructions following a single-instruction multiple-thread (SIMT) computing paradigm (NVIDIA, 2023), in which all threads in the warp execute the same instruction in lockstep, i.e. in parallel. Familiarity with the notion of a warp and how the execution of a kernel is coupled with the GPU hardware allows to explain two important factors that should be considered when designing a fast kernel: coalesced memory access and warp divergence.

2.2.2 Warp divergence

Warp divergence occurs when threads in a warp perform differing instructions and should be minimised when designing a kernel (NVIDIA, 2023). This is because when warp divergence occurs, the differing instructions cannot be performed by the warp in parallel and must be performed in serial instead, reducing the speed of the warp. To clarify, an example of warp divergence is shown in Figure 2.2, in which the arrows represent 32 threads in a warp; in CUDA code, `threadIdx.x` gives the index of a thread within a block: in the CUDA code shwon in Figure 2.2, the threads with an odd thread index execute the instructions in the function `func_A()`, whereas the threads with an even thread index execute the instructions in the function `func_B()`. The execution of these functions get serialised: the odd-indexed threads execute `func_A()` while the even-indexed threads wait; then, the even-indexed threads execute `func_B()` while the odd-indexed threads wait. Due to the serialised execution of the functions, the speed of the warp is reduced because it cannot finish executing until both the odd- and even-indexed threads have executed their respective instructions. The warp divergence shown in Figure 2.2 is an example of "two-way" divergence because there are two branches in execution: in general, N number of branches in execution lead to "N-way" divergence. The larger the number of branches in execution, the more divergence there is and the slower the speed of the warp becomes. As mentioned, warp divergence should be minimised when designing a kernel.



```
__global__ void my_kernel(
{
   if (threadIdx.x % 2)
   {
      do_func_A();
   }
   else
   {
      do_func_B();
   }
}
```

Figure 2.2: *Warp divergence; a warp is a group of 32 threads with consecutive thread IDs that operate in lockstep.*

2.2.3 Coalesced memory access

Coalesced memory access occurs when threads in a warp make full use of the data fetched from the L2 cache lines and should be maximised when designing a kernel (NVIDIA, 2023). To clarify what coalesced memory access is, some familiarity with the complicated memory hierarchy involved in the CUDA programming model is needed. Figure 2.3 shows the memory hierarchy. As shown by the labelled boxes in Figure 2.3, the memory hierarchy is made up of several memory spaces in which data can be stored and accessed by a kernel: CPU DRAM, GPU DRAM, L2 cache, local caches per SM, and registers per thread. Data in CPU DRAM cannot be directly accessed by a kernel and must be transferred to GPU DRAM first, which is the main memory space for storing large amounts of data on the GPU. Whenever data in GPU DRAM is accessed by a kernel, the data must pass through some or all of the memory spaces in order, as shown using the small arrows in Figure 2.3.

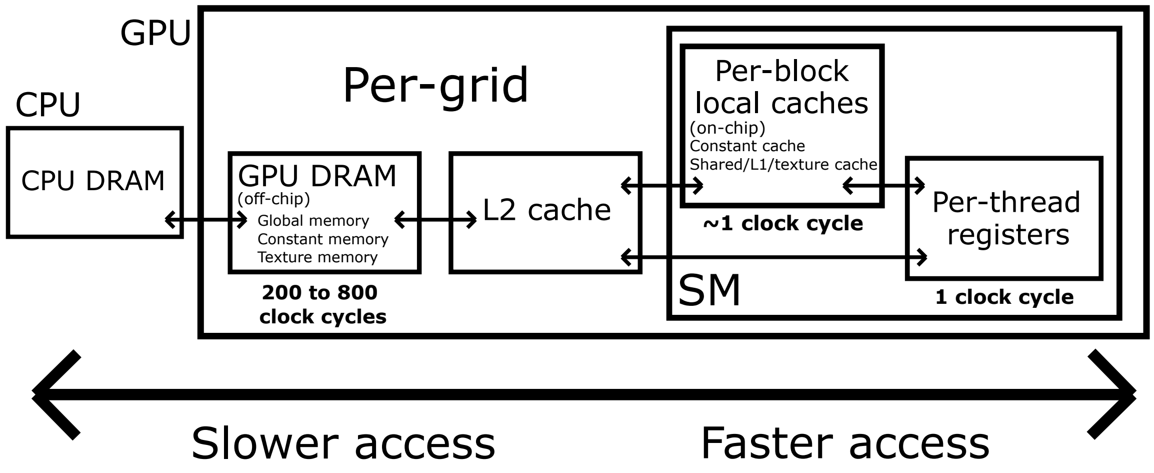


Figure 2.3: *Memory hierarchy involved in the CUDA programming model.*

GPU DRAM is made up of global memory, constant memory and texture memory. Global memory is the most commonly used when storing data in GPU DRAM (see Figure 2.1a or Figure 2.2): for example, in a SWE model, the computational grid is very often stored in global memory. Global memory should be accessed by a kernel in a way that maximises coalesced memory access, explained as follows.

Data in global memory is accessed via the L2 cache as chunks of 32 bytes of data, i.e. as 32-byte *cache lines*. This must be considered when devising an optimal access pattern for global memory. Figures 2.4a and 2.4b show optimal and non-optimal access patterns for threads in a warp to access global memory, respectively. In Figure 2.4a, the upper label around the grey boxes marks off a 32-byte cache line in the L2 cache, and the smaller four green segments represent four L2 cache lines. The arrows indicate 32 threads in a warp. The warp accesses global memory via the L2 cache using an optimal access pattern. In this optimal access pattern, the threads access global memory locations that are aligned with the 32-byte boundaries of each cache line (indicated by the numbers 0, 32, . . . , 128 in Figure 2.4a) fetched from the L2 cache. This alignment leads to coalesced memory access because the threads make use of all the data in the L2 cache lines. Namely, in Figure 2.4a, the threads require access to 128 bytes of data, meaning that 128 bytes required / 32 bytes per cache line = 4 cache lines need to be fetched at minimum. Since the warp indeed fetches only four cache lines – as shown by the smaller four green segments in Figure 2.4a – this results in 4 cache lines minimum / 4 cache lines fetched = 100% cache line usage.

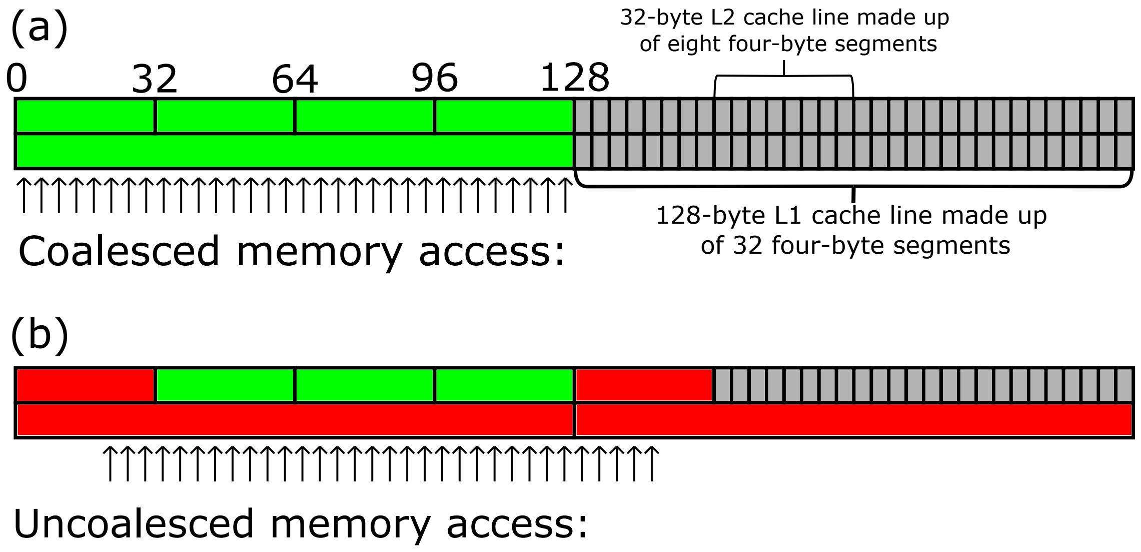

(a)
0
32
64
96
128
32-byte L2 cache line made up of eight four-byte segments
128-byte L1 cache line made up of 32 four-byte segments
Coalesced memory access:
(b)
Uncoalesced memory access:

Figure 2.4: *Optimal and non-optimal access patterns by threads in a warp when accessing global memory via the L2 and "L1 caches" (a) an optimal access leading to coalesced memory access; (b) a non-optimal access pattern leading to uncoalesced memory access.*

Less than 100% cache line usage means that the threads in a warp do not make use of all the data moved by the cache lines and implies that the warp is fetching an excess number of cache lines (i.e. uncoalesced memory access occurs), which reduces the speed of a warp. An example of uncoalesced memory access is shown in Figure 2.4b, in which the access pattern of the warp is shifted compared to the access pattern of the warp in Figure 2.4a. Due to the shifted access pattern, the warp accesses global memory locations that are no longer aligned with the cache line boundaries and five cache lines are fetched, as shown using the small red and green segments in Figure 2.4b. However, as the warp still requires only 128 bytes, i.e. 128 bytes required / 32 bytes per cache line = 4 cache lines need to be fetched at minimum, this results in 4 cache lines minimum

/ 5 cache lines fetched = 80% cache line usage.

In summary, when designing a kernel, coalesced memory access should be maximised and warp divergence should be minimised. Next, in Section 2.2.4, it is shown that maximising coalesced memory access and minimising warp divergence is relatively straightforward when parallelising a SWE model that runs on a uniform computational grid.

2.2.4 Using CUDA to parallelise SWE models on the GPU

SWE models running on uniform computational grids are ideal candidates for GPU-parallelisation because maximising coalesced memory access and minimising warp divergence is readily achievable when parallelising a uniform-grid SWE model, particularly for the calculations associated with applying the numerical scheme used by the model to numerically solve the SWE – e.g. an FV1 or DG2 scheme (Vacondio et al., 2014; Morales-Hernández et al., 2020).

The calculations associated with applying a numerical scheme over a computational grid – e.g. flux calculations, time integration, etc – are referred to hereafter in this thesis as “solver computations”. To demonstrate why maximising coalesced memory access and minimising warp divergence is readily achievable when parallelising the solver computations over a uniform grid, an example of a GPU-parallelised SWE model running on a uniform grid is considered.

2.2.4.1 Solver computations on the GPU over a uniform grid

SWE models can run simulations over 2D uniform raster grids made up of $N \times M$ cells. Figure 2.5 shows a 9×9 uniform grid over which a hypothetical GPU-parallelised SWE model performs solver computations. To perform the solver computations on the GPU, the model launches a kernel with one thread per cell in the grid, i.e. it launches a kernel with $9 \times 9 = 81$ threads, where each thread corresponds to a cell in the grid and all 81 threads perform the same set of solver computations. When executing the kernel, the GPU-parallelised SWE model can readily maximise coalesced memory access and minimise warp divergence by numbering the threads in the same order as the cells in the grid. To minimise warp divergence, recall that all threads in a warp should perform the same instructions. Here, no warp divergence is incurred by the SWE model because all 81 threads perform the same set of solver computations.

To ensure coalesced memory access, recall that threads in a warp should make full use of the

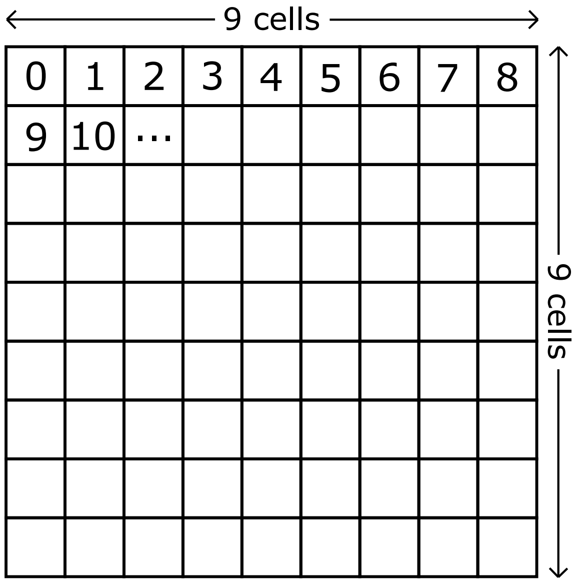


Figure 2.5: *Uniform-resolution computational grid made up of 9 × 9 = 81 cells. The numbers 0, 1, . . . , 9, 10, . . . indicate row-major numbering or indexing of the cells.*

data fetched from the L2 cache lines, in particular by aligning the access pattern of a warp with the cache line boundaries. This alignment can be readily ensured by the SWE model as follows: in Figure 2.5, as shown by the numbers 0, 1, . . . , 9, 10 . . . , the cells in the grid are numbered or indexed from left to right, i.e. the cells are indexed in *row-major* order. Since the cells are indexed in row-major order, data belonging to adjacent cells are stored in adjacent memory locations. When these adjacent memory locations are accessed by threads in a warp, it is relatively straightforward to align the access pattern of the warp with the L2 cache line boundaries by appropriately shifting the access pattern, with the neighbours being accessed contiguously from row to row.

Thus, a GPU-parallelised SWE model can readily achieve coalesced memory access and avoid warp divergence if running on a uniform grid. However, if the SWE model instead runs on a non-uniform grid – e.g. due to using a grid adaptation algorithm – then achieving coalesced memory access and/or avoiding warp divergence becomes much more challenging (Zabelok et al., 2015; Ji et al., 2015; Schive et al., 2018b; Giuliani and Krivodonova, 2019; Menshov and Pavlukhin, 2020). These challenges are reviewed in the next section (Section 2.3).

2.3 Challenges in efficiently implementing grid adaptation on GPUs

Grid adaptation algorithms really involve two distinct categories of operations that hinder coalesced memory access and/or incur warp divergence when implemented on the GPU: (1) the operations for generating a non-uniform grid, and (2) the operations for performing the solver computations (as defined in Section 2.2.4) over the generated non-uniform grid. For the first category of operations – i.e. the operations for generating a non-uniform grid – efficiently implementing these operations on the GPU is not feasible without fundamentally redesigning the algorithmic structure of the grid adaptation algorithm (Luo et al., 2016; Giuliani and Krivodonova, 2019; Dubey et al., 2021; Jude et al., 2022), which will be discussed in more detail later (Section 2.3.2). Conversely, when implementing the second category of operations on the GPU – i.e. the operations for performing the solver computations over a non-uniform grid – achieving coalesced memory access and/or avoiding warp divergence may yet be attainable depending on the type of grid adaptation algorithm being used (Dubey et al., 2014; Sætra et al., 2015; Schive et al., 2018b; Qin et al., 2019; Dunning et al., 2020), as discussed next (Section 2.3.1).

2.3.1 Solver computations on the GPU over different types of non-uniform grids

Grid adaptation algorithms can be broadly categorised into three types: block-based, patch-based and cell-based (de la Asunción and Castro, 2017), which are shown in Figure 2.6. In a block-based grid adaptation algorithm (Berger and Colella, 1989; Wang et al., 2010; Beckingsale et al., 2015; Raghavan and Vadhiyar, 2015; Qin et al., 2019), the non-uniform grid consists of a hierarchy of uniform, non-overlapping groups of cells called *blocks* that can cover arbitrary rectangular zones of the grid albeit being hierarchically *nested*, as shown in Figure 2.6a: nested means that blocks of finer (smaller) cells must be overlaid on blocks of coarser (larger) cells. Since the blocks are uniform, the cells in each block can be indexed in row-major order, thereby allowing to achieve coalesced memory access and avoid warp divergence when performing the solver computations – just like for the uniform grid discussed in Section 2.2.4.1.

In a patch-based grid adaptation algorithm (MacNeice et al., 2000; Fryxell et al., 2000; Schive et al., 2018a; Ji et al., 2015), the non-uniform grid is made up of a hierarchy of uniform, non-overlapping groups of cells called patches that can cover predefined tessellations of the grid, e.g.

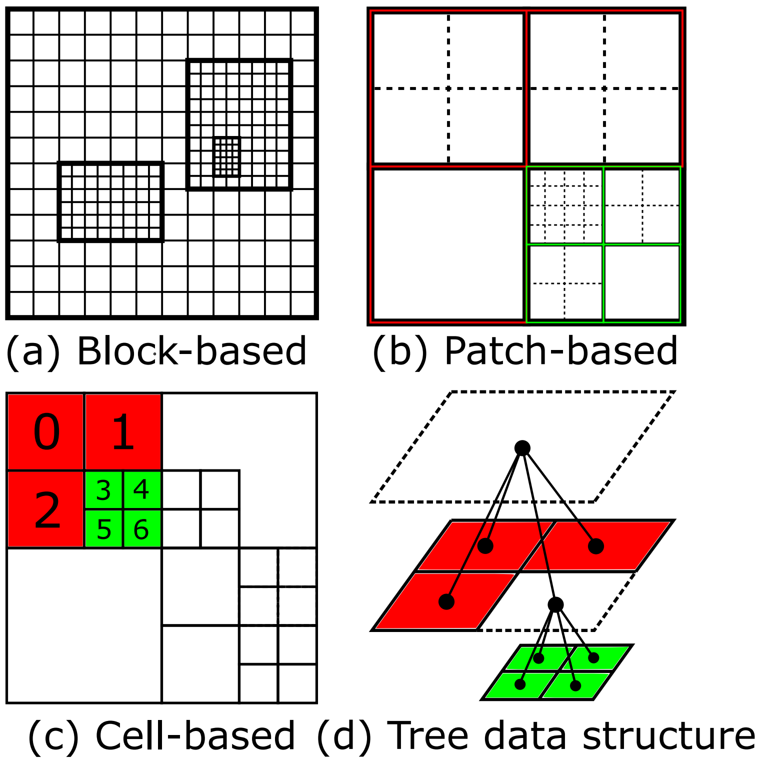


Figure 2.6: *The three different types of non-uniform grids and the tree data structure used to index the coloured part of the cell-based non-uniform grid.*

$2^n \times 2^n$ tessellations as shown by the red and green borders in Figure 2.6b; the cells in each patch can then be *refined* (split into smaller, finer cells) or *coarsened* (grouped into larger, coarser cells) collectively according to these predefined tessellations. For example, in Figure 2.6b, there are seven patches, three highlighted in red and four highlighted in green: two of the red patches are 2×2 cells large (top left and right) and one is 1×1 cell large (bottom right); for the green patches, one is 4×4 (top left), two are 2×2 (top right and bottom left), and one is 1×1 (bottom right). If all the green patches were 2×2 cells large, then they would be grouped into a single 4×4 red patch. Similar to the blocks in a block-based non-uniform grid, the patches in a patch-based non-uniform grid are uniform and therefore the cells in each patch can be indexed in row-major order, thereby allowing to achieve coalesced memory access and avoid warp divergence when performing the solver computations.

In a cell-based grid adaptation algorithm, individual cells are refined or coarsened rather than refining/coarsening entire blocks or patches like in a block- or patch-based grid adaptation algorithm. The advantage of a cell-based grid adaptation algorithm is that since individual cells are refined or coarsened, the refinement and coarsening process tracks the local variation in the numerical solution more closely and sensibly, thereby preventing unnecessary refinement and minimizing the number of fine cells in the non-uniform grid – and thus the computational effort (Khokhlov, 1998; Teyssier, 2002; Trujillo et al., 2011; Luo et al., 2016; Giuliani and Krivodonova, 2019). Conversely, in block- or patch-based grid adaptation algorithms, since entire blocks or patches are refined/coarsened, there is often an excessive number of refined cells and therefore a higher computational effort (Berger and Colella, 1989; George, 2011; Raghavan and Vadhiyar, 2015; Ferreira and Bader, 2017). Nevertheless, the disadvantage of a cell-based grid adaptation algorithm is that performing the solver computations over a cell-based non-uniform grid typically hinders coalesced memory access because the grid cannot be indexed in row-major order, and is instead more easily indexed using a *tree* data structure (Khokhlov, 1998; MacNeice et al., 2000; Agbaglah et al., 2011; Ji et al., 2015; Pavlukhin and Menshov, 2021), which does not readily facilitate coalesced memory access, as clarified next.

Figure 2.6c shows the tree data structure used to index the (cell-based) non-uniform grid in Figure 2.6b. For simplicity, the tree is only shown for the coloured portion of the grid. A tree data structure is made up of "nodes" (indicated by the black dots in Figure 2.6c), starting with a "root" node at the top, whereby each node may have "children" below. The nodes where the tree terminates are termed "leaf" nodes. These leaf nodes represent the individual cells making up the non-uniform grid. For example, the tree in Figure 2.6c starts at the "root" node, shown using the black dot in the top-most dashed outline in Figure 2.6c. The root node's children are the cells numbered 0, 1 and 2 in Figure 2.6b – which are leaf nodes – and another ghost cell, which is a node. This node's children are the cells numbered 3, 4, 5, and 6, which are all leaf nodes.

Using the tree data structure in Figure 2.6c, which indexes the cell-based non-uniform grid in Figure 2.6b, suppose the solver computations were performed over the grid by launching one thread per cell in the grid, i.e. by launching a kernel with one thread per leaf node (similar to launching a kernel with one thread per cell in the uniform grid described in Section 2.2.4.1). When the threads access the leaf nodes, coalesced memory access is unlikely to occur: for example, consider the thread corresponding to cell 3: to perform the solver computations, this thread needs to access the neighbouring cells, i.e. the cells numbered 1, 4, 5, 2. This numbering corresponds to locations in global memory that are non-adjacent, so coalesced memory access is hard to achieve.

Tree data structures are also used to index block- or patch-based non-uniform grids (Wang et al., 2010; Qin et al., 2019; Shimokawabe and Onodera, 2019; Korneev et al., 2021; Jude et al., 2022): when a block- or patch-based non-uniform grid is indexed using a tree data structure, the tree's nodes represent entire blocks or patches rather than individual cells. Thus, when a node is accessed, an entire block or patch is returned; thereafter, to access a specific cell in a block or patch, row-major indexing is used. In other words, when a block- or patch-based non-uniform grid is indexed using a tree data structure, the non-uniform grid hierarchy consisting of all the blocks or patches in the grid is indexed using the tree's nodes, but the cells in each individual block or patch are indexed in row-major order.

It therefore seems that instead of using a cell-based grid adaptation algorithm, using a block- or patch-based grid adaptation algorithm may be better for achieving coalesced memory access and avoiding warp divergence – at least when performing the solver computations. Nonetheless, recall that a grid adaptation algorithm also involves performing operations for generating the actual non-uniform grid over which the solver computations are performed. Efficiently implementing these operations on the GPU is difficult regardless of the type of grid adaptation algorithm being used, as discussed next (Section 2.3.2).

2.3.2 Difficulties in efficiently generating a non-uniform grid on the GPU

The operations for generating a non-uniform grid are difficult to implement on the GPU efficiently because they are highly irregular and unpredictable, which ultimately hinders coalesced memory access and incurs warp divergence (Zabelok et al., 2015; Ji et al., 2015; Schive et al., 2018b; Giuliani and Krivodonova, 2019). To help illustrate the irregular and unpredictable operations involved in grid adaptation algorithm, Figure 2.7 shows an example thereof. The functionality of a grid adaptation algorithm is to generate a non-uniform grid by flagging certain cells for refinement or coarsening. For example, Figure 2.7a shows a non-uniform grid generated by a grid adaptation algorithm: the orange lines indicate cells flagged for refinement, whereas the green lines indicate cells flagged for coarsening. Cells can be flagged for refinement or coarsening by measuring the local variation in the numerical solution. For example, in a SWE model, cells can be flagged for refinement or coarsening by measuring gradients in the water depth, velocity, and topography (Remacle et al., 2006; Kesserwani and Liang, 2012c; Garcia and Popiolek, 2014): cells that show significant variation are flagged for refinement, whereas cells that show less variation are flagged for coarsening. Figure 2.7b shows the new non-uniform grid generated by the grid adaptation algorithm after the flagged cells have been refined or coarsened: evidently, the number of finer or coarser cells – and therefore the number of operations for refinement and coarsening – are completely arbitrary and random because they depend on the local variation in the numerical solution, which is unknown in advance, so the refinement and coarsening operations are irregular and unpredictable. This can be clarified further by considering that non-uniform grids are indexed using tree data structures.

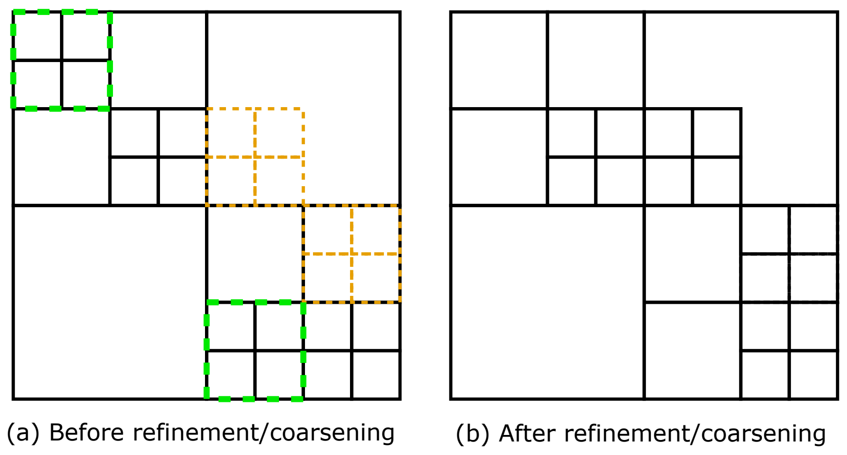

(a) Before refinement/coarsening
(b) After refinement/coarsening

Figure 2.7: *Grid resolution adaptation algorithm: (a) a non-uniform grid generated by a grid adaptation algorithm; the orange lines indicate cells flagged for refinement, whereas the green lines indicate cells flagged for coarsening; (b) the newly generated non-uniform grid after the flagged cells have been refined/coarsened.*

Generally speaking, generating a non-uniform grid involves three main steps (MacNeice et al., 2000; Beckingsale et al., 2015; Luo et al., 2016; de la Asunción and Castro, 2017): (1) flagging potential cells for refinement or coarsening by traversing the non-uniform grid hierarchy, (2) performing the necessary refinement and coarsening operations on any flagged cells, and (3) accounting for any changes in the layout of the non-uniform grid due to cell refinement/coarsening by modifying the non-uniform grid hierarchy. All of these steps involve traversing the non-uniform grid hierarchy, which in turn entails traversing the tree data structure used to index the hierarchy. Unfortunately, traversing a tree data structure efficiently on the GPU is not trivial: tree traversal is typically solved with recursive algorithms like depth-first traversal (Sedgewick and Wayne, 2011), which are inherently sequential and cannot be implemented on the GPU (Khokhlov, 1998; Bédorf et al., 2012; Karras, 2012; Goldfarb et al., 2013; Chitalu et al., 2018). For example, one approach for traversing a tree data structure that indexes a non-uniform grid on the GPU would be to launch one thread per cell to be flagged for refinement or coarsening. However, the number of cells to be flagged is unknown in advance, making it challenging to launch one thread per cell. Even if the number of cells to be flagged were known in advance and one thread per cell were launched, some threads would perform refinement operations while others would perform coarsening operations, leading to warp divergence. Furthermore, for threads to perform the refinement or coarsening operations, each thread would need to access a leaf node in the tree, but the indices of the leaf nodes are likely to place them in non-adjacent memory locations, hindering coalesced memory access (similar to the issue of uncoalesced memory access that occurs when performing solver computations over a cell-based non-uniform grid, as discussed in Section 2.3.1).

Since it is difficult to efficiently generate a non-uniform grid on the GPU, many models simply avoid this difficulty altogether by generating the grid on the CPU instead and then transferring the grid to the GPU for the solver computations (Wang et al., 2010; Burstedde et al., 2010; Ji et al., 2015; Raghavan and Vadhiyar, 2015; Schive et al., 2018b; Qin et al., 2019). Wang et al. (2010) implemented a hydrodynamic model in which the solver computations were mapped onto the GPU while the grid generation operations were performed on the CPU. Ji et al. (2015) applied a strategy to an immersed boundary model in which the GPU handled the solver computations whereas the CPU handled the grid generation operations. Burstedde et al. (2010) presented a wave propagation model called dGea in which the numerical scheme ran on the GPU while the grid generation operations were performed on the CPU using the p4est AMR framework (Burstedde et al., 2011). Qin et al. (2019) implemented a shallow water model in which the Riemann solvers, wave limiter, and timestep calculation ran on the GPU while the CPU performed the operations related to grid generation. Raghavan and Vadhiyar (2015) used a patch-based grid adaptation algorithm in which they mapped individual *patches* to CUDA blocks to perform solver computations while overlapping the computations with CPU-GPU data transfers to hide data transfer costs. Schive et al. (2018b) overlapped CPU and GPU computations in a patched-based grid adaptation framework with a rich set of features for astrophysics-related simulations. All of the aforementioned models adopted a hybrid CPU-GPU approach in which the solver computations are performed on the GPU and the grid generation is performed on the CPU. However, this hybrid CPU-GPU approach imposes a data transfer overhead because data is repeatedly transferred between the CPU and the GPU, which may slow down the model (Sætra et al., 2015; Beckingsale et al., 2015; Luo et al., 2016; Qin et al., 2019; Menshov and Pavlukhin, 2020). Thus, CPU-GPU data transfers remain a bottleneck in implementing a hybrid CPU-GPU grid generation approach even as GPU hardware vendors strive to increase CPU-GPU data transfer bandwidth and to integrate CPU and GPU memory spaces more tightly.

To avoid the CPU-GPU data transfer bottleneck, many models have implemented the grid generation operations on the GPU (Sætra et al., 2015; Beckingsale et al., 2015; Wahib et al., 2016; Luo et al., 2016; Lung et al., 2016; Giuliani and Krivodonova, 2019; Dunning et al., 2020). Luo et al. (2016) implemented a cell-based grid adaptation algorithm entirely on the GPU for unstructured quadrilateral grids to reduce the CPU-GPU data transfer overhead for an inviscid flow model. Giuliani and Krivodonova (2019) developed novel algorithms to efficiently implement cell-based grid adaptation on the GPU for unstructured triangular meshes in a gas dynamics model. Lung et al. (2016) briefly presented a grid adaptation algorithm running entirely on the GPU based on a wavelet collocation method for a combustion model. Sætra et al. (2015) presented the GPU implementation of a block-based grid adaptation algorithm and paired it with a second-order Kurganov-Petrova scheme for solving the SWE (Kurganov and Petrova, 2007). Beckingsale et al. (2015) developed a block-based grid adaptation library called CLAMR that runs entirely on the GPU and applied it to SWE simulations. Dunning et al. (2020) presented a framework based on the same CLAMR library in which the grid generation operations are performed on the GPU in a way that decouples the grid generation operations from the solver computations. Wahib et al. (2016) developed a compiler-based framework that automatically transforms uniform-grid code provided by the user into parallel adaptive-grid code optimized for GPU-accelerated computing clusters; they explored a problem about solving the SWE using the Kurganov-Petrova scheme.

All of the models cited above – except for one (Lung et al. (2016)) – have implemented grid adaptation algorithms on the GPU that are similar to classical AMR (Berger and Colella, 1989). Rather than implementing AMR on the GPU though, recall that the thesis' research aim is instead to implement a wavelet-based grid adaptation algorithm based on the MRA process of the MW (Kesserwani and Sharifian, 2020) on the GPU. Nonetheless, the MRA process is difficult to implement on the GPU efficiently due to a number of obstacles in achieving coalesced memory access and avoiding warp divergence – similar to those discussed in Sections 2.3.1 and 2.3.2. To overcome these obstacles, the algorithmic structure of the MRA process should be redesigned: this is explored in the next chapter (Chapter 3), in which computational ingredients are proposed to redesign the algorithmic structure of a wavelet-based grid adaptation algorithm based on the MRA process of the HW and thereby make it suitable for efficient implementation on the GPU.

Chapter 3

GPU-parallelisation of HW grid adaptation

Contents used to prepare for this chapter have been published in the following publications: *Alovya Ahmed Chowdhury, Georges Kesserwani, Charles Rougé, Paul Richmond 2023. GPU-parallelisation of Haar wavelet-based grid resolution adaptation for fast finite volume modelling: application to shallow water flows, Journal of Hydroinformatics, 25 (4): 1210–1234. doi: https://doi.org/10.2166/hydro.2023.154.*

In this chapter, Objective 1 is addressed by proposing three computational ingredients to re-design the algorithmic structure of a wavelet-based grid adaptation algorithm to make it suitable for efficient implementation on the GPU. In particular, the wavelet-based grid adaptation algorithm based on the MRA process of the HW is considered, which allows to enhance a reference FV1 model that ordinarily runs on a uniform grid with grid adaptation. Note that the MRA process of the HW cannot be used to integrate a wavelet-based grid adaptation capability to GPU-DG2– for which the MRA process of the MW is applicable instead (Kesserwani et al., 2015b; Keinert, 2003) – but the HW are less complicated than the MW and their implementation on the GPU is therefore considered first as an easier, intermediate step.

The MRA process of the HW is difficult to implement on the GPU efficiently due to three obstacles. The first obstacle is that the MRA process generates a cell-based non-uniform grid by "encoding" (coarsening), "decoding" (refining), analysing and traversing modelled data across a deep hierarchy of nested, uniform grids of increasingly coarser resolution: the modelled data in the hierarchy of grids may be close together in physical space, but they are not guaranteed to be close together in GPU global memory space unless the hierarchy is indexed in a deliberate way, hindering coalesced memory access when accessing the modelled data in the hierarchy of grids. The second obstacle is that traversing the hierarchy of grids boils down to a tree traversal problem, which – as discussed in Section 2.3.2 – is an inherently sequential task usually completed with recursive algorithms such as depth-first traversal that cannot be parallelised on the GPU. The third obstacle is that the MRA process generates a cell-based non-uniform grid, which – as discussed in Section 2.3.1 – is usually indexed using a tree data structure that hinders coalesced memory access.

To overcome these three obstacles and implement the MRA process on the GPU efficiently, this chapter proposes three computational ingredients. The first ingredient is using Z-order space-filling curves (SFC) (Sagan, 1994; Bader, 2013) to index the hierarchy of grids. The second ingredient is adopting a parallel tree traversal algorithm (Karras, 2012) to replace the sequential depth-first traversal algorithm. The third ingredient is using an array data structure instead of a tree data structure to index the cell-based non-uniform grid.

The rest of this chapter is organised as follows. Section 3.1 presents how the three proposed computational ingredients are used to implement a new GPU-parallelised HW adaptive FV1 shallow water model, referred to as GPU-HWFV1. To assess the extent to which using the three proposed ingredients indeed allow for efficient GPU implementation of the HW-based grid adaptation algorithm, the new GPU-HWFV1 model should be at least as fast and accurate as a GPU-parallelised FV1 shallow water model running on a uniform grid – i.e. the *de facto* standard class of SWE models for running real-world flood simulations (see Section 2.1) (Xia et al., 2019; Echeverribar et al., 2019; Dazzi et al., 2020; Gordillo et al., 2020; Carlotto et al., 2021; Morales-Hernández et al., 2021; Buttinger-Kreuzhuber et al., 2022; Delmas and Soulaïmani, 2022; Caviedes-Voullième et al., 2023; Sanz-Ramos et al., 2023). Section 3.2 therefore analyses the efficiency of GPU-HWFV1's grid adaptation capability by comparing GPU-HWFV1 against the GPU-parallelised FV1 model running on a uniform grid in LISFLOOD-FP 8.0 (Shaw et al., 2021) – referred to hereafter in this thesis as GPU-FV1 . For completeness, GPU-HWFV1 is also compared against its sequential (i.e. non-parallelised) version running on the CPU, referred to hereafter in this thesis as CPU-HWFV1

(Kesserwani et al., 2019; Kesserwani and Sharifian, 2020). Section 3.3 presents conclusions on the efficiency of GPU-HWFV1's grid adaptation capability.

3.1 New GPU-HWFV1 model

GPU-HWFV1 is implemented by parallelising the sequential CPU version of the HWFV1 model presented in Kesserwani and Sharifian (2020) – i.e. CPU-HWFV1. CPU-HWFV1 adopts a wavelet-based grid adaptation algorithm based on the MRA process of the HW which, as mentioned, is difficult to efficiently parallelise on the GPU due to three obstacles. To understand why these three obstacles arise, Section 3.1.1 gives an overview of CPU-HWFV1. Section 3.1.2 then discusses how the obstacles are overcome using the three proposed computational ingredients: a Z-order space-filling curve, a parallel tree traversal algorithm and an array data structure.

3.1.1 Sequential CPU-HWFV1 model

3.1.1.1 Overview

CPU-HWFV1 is a wavelet-based adaptive SWE model that runs hydrodynamic simulations by solving the 2D SWE over a non-uniform grid. CPU-HWFV1 is made up of two mechanisms: a wavelet-based grid adaptation algorithm for generating a non-uniform grid, and an FV1 scheme for updating the modelled data on this non-uniform grid – i.e. for performing the solver computations (as defined in Section 2.2.4) over the non-uniform grid. CPU-HWFV1 is dynamically adaptive, meaning it generates the non-uniform grid every timestep before applying the FV1 scheme to perform the solver computations.

Wavelet-based grid adaptation algorithm. CPU-HWFV1's wavelet-based grid adaptation algorithm is based on the MRA of the HW (Haleem et al., 2015; Kesserwani and Sharifian, 2020). The MRA process starts on a reference uniform grid made up of $2^L \times 2^L$ cells where L is a user-specified integer denoting the maximum refinement level that controls the number of cells in the reference uniform grid. The SWE in conservative form are initially discretised over the reference $2^L \times 2^L$ grid, which can be written as:

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) + \partial_y \mathbf{G}(\mathbf{U}) = \mathbf{S}_b(\mathbf{U}) + \mathbf{S}_f(\mathbf{U}) \tag{3.1}$$

where ∂_t, ∂_x, and ∂_y represent partial derivatives with respect to t, x, and y, respectively. The vector $\mathbf{U} = [h, hu, hv]^{\mathrm{T}}$ contains the flow variables, $\mathbf{F} = [hu, (hu^2/h + 1/2gh^2), huv]^{\mathrm{T}}$, $\mathbf{G} = [hv, huv, (hv^2/h + 1/2gh^2)]^{\mathrm{T}}$ are the components of the flux vector, and $\mathbf{S}_b = [0, -gh\partial_x z, -gh\partial_y z]^{\mathrm{T}}$ and $\mathbf{S}_f = [0, -C_f u\sqrt{u^2+v^2}, -C_f v\sqrt{u^2+v^2}]^{\mathrm{T}}$ are source term vectors. The variable $h(x, y, t)$ is the water depth (m), $u(x, y, t)$ and $v(x, y, t)$ are the x- and y-components of the velocity (m/s), respectively, and g is the gravitational acceleration constant (m/s^2). In $\mathbf{S}_b$, $z(x, y)$ is the topographic elevation (m), and in $\mathbf{S}_f$, $C_f = \frac{gn_M^2}{h^{1/3}}$ where n_M is Manning's coefficient.

FV1 scheme. CPU-HWFV1's FV1 scheme locally approximates the variables h, hu, hv, and z as piecewise-constant solutions over each cell Q_c in the grid, whereby the solution for each quantity is controlled by scalar coefficients h_c, $(hu)_c$, $(hv)_c$, and z_c per cell, respectively. The vector of flow coefficients $\mathbf{U}_c = [h_c, (hu)_c, (hv)_c]^{\mathrm{T}}$ are updated in time – i.e., from timestep p to timestep $p + 1$ – by performing an "FV1 update". The FV1 update consists of a forward-Euler scheme and a spatial operator $\mathbf{L}_c$, in which $\mathbf{U}_c$ is updated from timestep p (denoted by a superscript) to timestep $p + 1$ as follows:

$$\mathbf{U}_c^{p+1} = \mathbf{U}_c^p + \Delta t \mathbf{L}_c(\mathbf{U}_c^p) \tag{3.2}$$

where Δt is a timestep computed based on the Courant-Friedrichs-Lewy (CFL) condition using a CFL number of 0.5 to ensure numerical stability (Kesserwani and Sharifian, 2020). The expression for $\mathbf{L}_c$ is:

$$\mathbf{L}_c = -\frac{1}{\Delta x}\left(\tilde{\mathbf{F}}_e - \tilde{\mathbf{F}}_w\right) - \frac{1}{\Delta y}\left(\tilde{\mathbf{G}}_n - \tilde{\mathbf{G}}_s\right) + \begin{bmatrix} 0 \\ -\frac{2g\sqrt{3}}{\Delta x}\bar{h}_c^{0x}\bar{z}_c^{1x} \\ -\frac{2g\sqrt{3}}{\Delta y}\bar{h}_c^{0x}\bar{z}_c^{1y} \end{bmatrix} \tag{3.3}$$

where $\tilde{\mathbf{F}}_e$, $\tilde{\mathbf{F}}_w$, $\tilde{\mathbf{G}}_n$, and $\tilde{\mathbf{G}}_s$ are numerical fluxes at the eastern, western, northern, and southern interfaces between cells, respectively. These fluxes are computed using two-argument numerical flux functions $\tilde{\mathbf{F}}(\cdot, \cdot)$ and $\tilde{\mathbf{G}}(\cdot, \cdot)$ based on the Harten-Lax-van Leer (HLL) approximate Riemann solver (Toro 2001). The two arguments of the flux functions are the limits of the piecewise-constant solutions at the left and right sides of a cell interface after temporary revision of the solution limits according to a wetting-and-drying condition to ensure well-balancedness and non-negative water depths (Kesserwani et al. 2018). As an example, consider the computation of $\tilde{\mathbf{F}}_e$ at the eastern interface: for the flow solution, the limit at the left side of the eastern interface is $\mathbf{U}_e^L = \mathbf{U}_c$, while the limit at the right side is $\mathbf{U}_e^R = \mathbf{U}_{\text{east}}$, where the east subscript is the index of the eastern neighbor. For the left and right limits of the topography solution, $z_e^L = z_c$ and $z_e^R = z_{\text{east}}$, respectively. These limits are revised as follows and are denoted with a star superscript: first set $z_e^* = \max(z_e^L, z_e^R)$, and then obtain $h_e^{L,R,*} = \max(0, (h_e^{L,R} - z_e^{L,R}) - z_e^*)$, $(hu)_e^{L,R,*} = h_e^{L,R,*} u_e^{L,R}$, $(hv)_e^{L,R,*} = h_e^{L,R,*} v_e^{L,R}$, and $z_e^* = z_e^* - \max(0, -(h_e^L - z_e^L) - z_e^*)$. The revised limits of the flow solution $\mathbf{U}_e^{L,R,*} = [h_e^{L,R,*}, (hu)_e^{L,R,*}, (hv)_e^{L,R,*}]$ are used to compute $\tilde{\mathbf{F}}_e = \tilde{\mathbf{F}}(\mathbf{U}_e^{L*}, \mathbf{U}_e^{R*})$. To compute $\tilde{\mathbf{F}}_w$, $\tilde{\mathbf{G}}_n$, and $\tilde{\mathbf{G}}_s$, the revised limits $\mathbf{U}_w^{L,R,*}$, $\mathbf{U}_n^{L,R,*}$, and $\mathbf{U}_s^{L,R,*}$ are obtained by following a similar set of steps as for obtaining $\mathbf{U}_e^{L,R,*}$.

In addition to computing $\tilde{\mathbf{F}}_e$, $\tilde{\mathbf{F}}_w$, $\tilde{\mathbf{G}}_n$, and $\tilde{\mathbf{G}}_s$ using the revised limits, the bed source terms are also computed using the revised limits, namely by computing $\overline{h}_c^{0x} = \frac{1}{2}\left(h_e^{L,*} + h_w^{R,*}\right)$, $\overline{z}_c^{1x} = \frac{1}{2\sqrt{3}}\left(z_e^* - z_w^*\right)$, $\overline{h}_c^{0y} = \frac{1}{2}\left(h_n^{L,*} + h_s^{R,*}\right)$, and $\overline{z}_c^{1y} = \frac{1}{2\sqrt{3}}\left(z_n^* - z_s^*\right)$.

Before applying $\mathbf{L}_c$, friction effects are integrated separately using a split implicit scheme (Liang and Marche, 2009). Define the x-directional discharge $q_x = hu$ and a discharge decelerated by friction q_{fx} as:

$$q_{fx}(\mathbf{U}) = q_x + \Delta t \frac{S_{fx}}{D_x}, \tag{3.4}$$

where D_x is

$$D_x = 1 + \frac{\Delta t C_f}{h} \frac{2u^2 + v^2}{\sqrt{u^2 + v^2}}. \tag{3.5}$$

The discharge coefficient $q_{x,c}^p$ is then updated in time as follows:

$$q_{x,c}^{p+1} = q_{fx}(\mathbf{U}_c^p). \tag{3.6}$$

Similarly, in the y-direction:

$$q_{fy}(\mathbf{U}) = q_y + \Delta t \frac{S_{fy}}{D_y}, \tag{3.7}$$

where

$$D_y = 1 + \frac{\Delta t C_f}{h} \frac{u^2 + 2v^2}{\sqrt{u^2 + v^2}}, \tag{3.8}$$

and

$$q_{y,c}^{p+1} = q_{fy}(\mathbf{U}_c^p). \tag{3.9}$$

Ordinarily, the FV1 update is performed over a uniform grid. However, as mentioned, CPU-

HWFV1 performs the FV1 update over a non-uniform grid – a grid whose spatial resolution is locally coarsened according to the details of the flow and topography. CPU-HWFV1 generates the non-uniform grid every timestep by performing the MRA process of the HW, described next.

3.1.1.2 MRA process of the HW

The MRA process of the HW involves a hierarchy of uniform nested grids of increasingly coarser resolution relative to the reference $2^L \times 2^L$ grid at the maximum refinement level L. Figure 3.1a shows a hierarchy of grids with $L = 2$. The refinement level of each grid in the hierarchy is denoted by n: the finest grid in the hierarchy is at refinement level $n = L = 2$, the second-finest grid is at refinement level $n = 1$ and the coarsest grid is at $n = 0$. For ease of presentation of the MRA process, let s_c be a dummy variable standing for any of h_c, $(hu)_c$, $(hv)_c$ or z_c , and let $s_c^{(n)}$ denote the coefficients s_c at refinement level n.

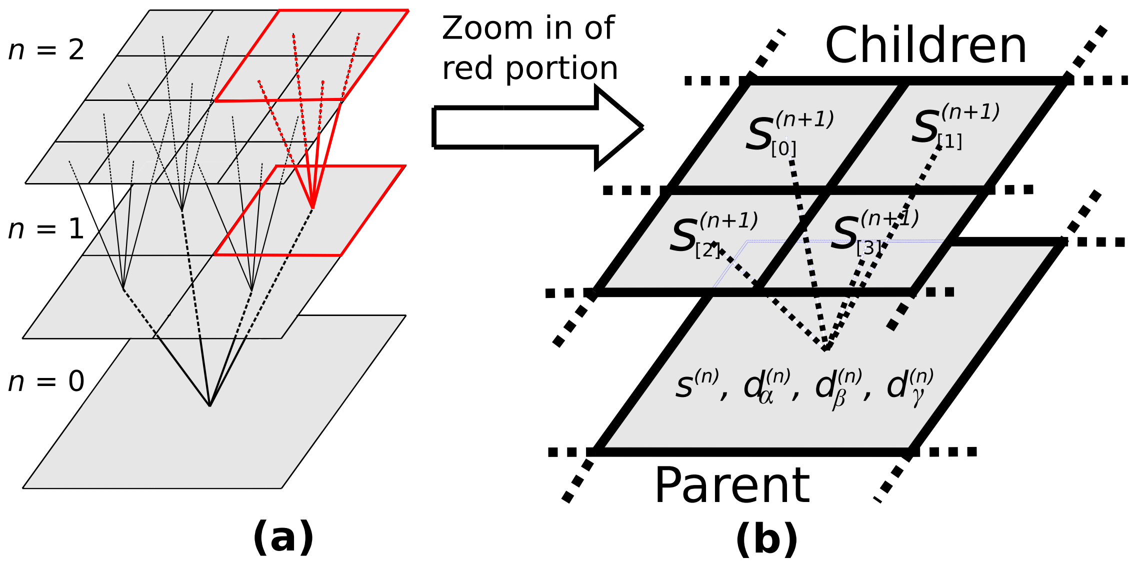


Figure 3.1: *Multiresolution analysis (MRA). The left panel shows a hierarchy of grids involved in the MRA process with maximum refinement level $L = 2$. The right panel shows how four cells at refinement level $n+1$, called the "children", are positioned relative to a single cell at refinement level n, called the "parent". Also shown are the coefficients s and details d involved in the MRA process.*

$$s^{(n)} = H^0\left(H^0 s_{[0]}^{(n+1)} + H^1 s_{[2]}^{(n+1)}\right) + H^1\left(H^0 s_{[1]}^{(n+1)} + H^1 s_{[3]}^{(n+1)}\right) \tag{3.10a}$$

$$d_\alpha^{(n)} = H^0\left(G^0 s_{[0]}^{(n+1)} + G^1 s_{[2]}^{(n+1)}\right) + H^1\left(G^0 s_{[1]}^{(n+1)} + G^1 s_{[3]}^{(n+1)}\right) \tag{3.10b}$$

$$d_\beta^{(n)} = G^0\left(H^0 s_{[0]}^{(n+1)} + H^1 s_{[2]}^{(n+1)}\right) + G^1\left(H^0 s_{[1]}^{(n+1)} + H^1 s_{[3]}^{(n+1)}\right) \tag{3.10c}$$

$$d_\gamma^{(n)} = G^0\left(G^0 s_{[0]}^{(n+1)} + G^1 s_{[2]}^{(n+1)}\right) + G^1\left(G^0 s_{[1]}^{(n+1)} + G^1 s_{[3]}^{(n+1)}\right) \tag{3.10d}$$

Encoding. At the start of the MRA process, only $s^{(L)}$ are available, corresponding to the coefficients obtained from initially discretising the SWE over the reference $2^L \times 2^L$ grid. The MRA process continues by producing $s_c^{(n)}$ at the lower refinement levels, i.e. at $n = L-1, L-2, \ldots, 1, 0$. The coefficient $s_c^{(n)}$ of a cell at refinement level n (called the "parent") is produced using the coefficients $s_{[0]}^{(n+1)}$, $s_{[1]}^{(n+1)}$, $s_{[2]}^{(n+1)}$, and $s_{[3]}^{(n+1)}$ of four cells at refinement level $n+1$ (called the "children"), in particular by applying Equation 3.10a. Figure 3.1b shows the parent-children stencil that dictates how the children $s_{[0]}^{(n+1)}$, $s_{[1]}^{(n+1)}$, $s_{[2]}^{(n+1)}$, and $s_{[3]}^{(n+1)}$ at level $n+1$ are positioned in the hierarchy relative to the parent $s_c^{(n)}$ at level n. Equations 3.10b - 3.10d are also applied to produce so-called "details," denoted by $d_\alpha^{(n)}$, $d_\beta^{(n)}$, and $d_\gamma^{(n)}$. These details represent the encoded difference in flow and topographic complexity across two subsequent levels of the hierarchy - i.e. the difference in complexity between the children and the parent, or how much complexity would be lost if only a single parent were used to represent information instead of the four children - and are used to decide which cells to include in the non-uniform grid based on a normalised detail $d_{\text{norm}}^{(n)} = \max(d_\alpha^{(n)}, d_\beta^{(n)}, d_\gamma^{(n)})/s_{\max}$, where $s_{\max}$ is the largest coefficient for all $s^{(L)}$. Cells whose $d_{\text{norm}}^{(n)}$ is greater than $2^{n-L}\varepsilon$ are deemed to have significant details, where ε is a user-specified error threshold. In Equations 3.10a - 3.10d, $H_0 = 1/\sqrt{2}$, $H_1 = 1/\sqrt{2}$, $G_0 = 1/\sqrt{2}$, and $G_1 = -1/\sqrt{2}$, which are filter coefficients derived from the HW (Keinert, 2003), resulting in very simple algebraic relations that allow to scale (i.e. refine and coarsen) flow and topographic information across the different refinement levels in the hierarchy. The process of computing $s_c^{(n)}$, $d_\alpha^{(n)}$, $d_\beta^{(n)}$, and $d_\gamma^{(n)}$ at the lower refinement levels is called "encoding."

Algorithm 1 shows pseudocode summarising the process of encoding. The pseudocode has an outer loop (lines 2 to 10) and an inner loop (lines 3 to 9). The outer loop iterates over each grid in the hierarchy, starting from the grid at the second-highest refinement level, $n = L - 1$, and ending at the grid at the lowest refinement level, $n = 0$. The inner loop iterates through each cell in the grid while applying Equations 3.10a - 3.10d (lines 5 and 6) and flagging significant details (line 8).

Algorithm 1 Pseudocode for the process of computing $s^{(n)}$, $d_\alpha^{(n)}$, $d_\beta^{(n)}$, and $d_\gamma^{(n)}$ at the lower refinement levels, called 'encoding'.

```
1: procedure ENCODING($L, \varepsilon$)
2: for each grid in hierarchy from $L - 1$ to 0 do
3:     for each cell in the grid do
4:         load children $s_{[0]}^{(n+1)}, s_{[1]}^{(n+1)}, s_{[2]}^{(n+1)}, s_{[3]}^{(n+1)}$ for Equations 3.10a - 3.10d
5:         compute parent coefficient $s_c^{(n)}$ using Equation 3.10a
6:         compute details $d_\alpha, d_\beta, d_\gamma$ using Equations 3.10b - 3.10d
7:         compute normalised detail $d_{\text{norm}}$
8:         flag as significant if $d_{\text{norm}} \geq 2^{n-L}\varepsilon$
9:     end for
10: end for
11: end procedure
```

$$s_{[0]}^{(n+1)} = H^0 \left(H^0 s^{(n)} + G^0 d_\alpha^{(n)} \right) + G^0 \left(H^0 d_\beta^{(n)} + G^0 d_\gamma^{(n)} \right) \tag{3.11a}$$

$$s_{[2]}^{(n+1)} = H^0 \left(H^1 s^{(n)} + G^1 d_\alpha^{(n)} \right) + G^0 \left(H^1 d_\beta^{(n)} + G^1 d_\gamma^{(n)} \right) \tag{3.11b}$$

$$s_{[1]}^{(n+1)} = H^1 \left(H^0 s^{(n)} + G^0 d_\alpha^{(n)} \right) + G^1 \left(H^0 d_\beta^{(n)} + G^0 d_\gamma^{(n)} \right) \tag{3.11c}$$

$$s_{[3]}^{(n+1)} = H^1 \left(H^1 s^{(n)} + G^1 d_\alpha^{(n)} \right) + G^1 \left(H^1 d_\beta^{(n)} + G^1 d_\gamma^{(n)} \right) \tag{3.11d}$$

Decoding. Flagging significant details during the encoding step results in a tree-like structure of significant details embedded within the hierarchy of grids (see example in Figure 3.2a): this tree is essential for generating a non-uniform grid. To generate the non-uniform grid, CPU-HWFV1 identifies and assembles the cells that make up the non-uniform grid by climbing the tree of significant details while applying Equations 3.11a to 3.11d. These equations exploit the details that were produced in the encoding process – which, recall, represent and store the difference in flow and topograhic complexity between the children and the parent – by "adding back" the complexity stored in the details to the parents in order to produce the children. The climbing starts from the coarsest cell and restarts whenever it reaches either (1) a cell on the finest grid (e.g. the green cells in Figure 3.2a), or (2) a cell with a detail that is not significant (e.g. the blue cells in Figure 3.2a):

these cells are termed “leaf cells” and are assembled into a non-uniform grid. The non-uniform grid is indexed using a tree data structure – hence the term leaf cells as the cells making up the non-uniform grid are analogous to the leaf nodes of a tree data structure for indexing a cell-based non-uniform grid (see Section 2.3.1). Figure 3.2b shows the non-uniform grid generated by CPU-HWFV1 after assembling the leaf cells in Figure 3.2a. Climbing the tree in the way described above corresponds to a depth-first traversal of a tree (Sedgewick and Wayne, 2011). The path of the depth-first traversal of the tree in Figure 3.2a can be traced by following the cells labelled from 0 to 12 in ascending order, and the process of performing a depth-first traversal while applying Equations 3.11a - 3.11d and identifying leaf cells is called “decoding”.

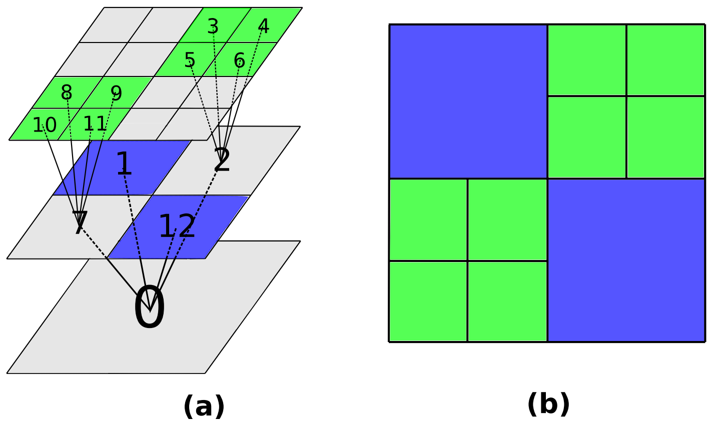


Figure 3.2: *Generating a non-uniform grid via the MRA process. The left panel shows the tree-like structure embedded within the hierarchy of grids after flagging significant details during the encoding step: the cells where the tree terminates are called “leaf” cells (highlighted in green and blue). The right panel shows the non-uniform grid generated from assembling the leaf cells.*

Algorithm 2 shows pseudocode describing the decoding step. The algorithm starts at the coarsest cell in the tree, i.e. at the coefficient $s^{(0)}_{[0]}$. The children of this cell, $s^{(1)}_{[0]}$, $s^{(1)}_{[1]}$, $s^{(1)}_{[2]}$, and $s^{(1)}_{[3]}$, are computed using Equations 3.11a - 3.11d (line 6) and then the algorithm is recursively launched at the children at one refinement level higher (lines 7 to 10). The algorithm is recursively launched until a cell with a detail that is not significant is reached or a cell on the finest grid is reached (line 2), at which point the cell is identified as a leaf cell (line 3). After identifying the leaf cells and assembling them into a non-uniform grid, CPU-HWFV1 can perform the FV1 update (i.e. perform the solver computations) over the assembled non-uniform grid. The FV1 update is described next.

Algorithm 2 Pseudocode for performing a depth-first traversal of the tree of significant details (obtained after encoding) while applying Equations 3.11a - 3.11d, called "decoding".

```
1: procedure DECODING($s^{(n)}, n$)
2: if detail is not significant or reached finest grid then
3:     identify cell as leaf cell
4:     stop decoding
5: else
6:     compute children $s_{[0]}^{(n+1)}, s_{[1]}^{(n+1)}, s_{[2]}^{(n+1)}, s_{[3]}^{(n+1)}$ with Equations 3.11a - 3.11d
7:     DECODING($s_{[0]}^{(n+1)}, n+1$)
8:     DECODING($s_{[1]}^{(n+1)}, n+1$)
9:     DECODING($s_{[2]}^{(n+1)}, n+1$)
10:    DECODING($s_{[3]}^{(n+1)}, n+1$)
11: end if
12: end procedure
```

3.1.1.3 Neighbour finding to perform the FV1 update over the non-uniform grid

To perform the FV1 update over the non-uniform grid, CPU-HWFV1 needs to compute $\mathbf{L}_c$ for each cell in the grid. To compute $\mathbf{L}_c$ for each cell, CPU-HWFV1 needs to retrieve s_{west}, s_{east}, s_{north}, and s_{south} – i.e. the coefficients of the neighbours of each cell. Finding the neighbours is trivial on a uniform grid because a cell can look left, right, up, and down to find its neighbours, but finding the neighbours is not trivial on a non-uniform grid because a cell can have multiple neighbours in a given direction. For instance, Figure 3.3 shows an example of a cell and its neighbours in a non-uniform grid. In this example, finding s_{west} (blue cell), s_{south}, and s_{north} (grey cells) is straightforward. However, it is not clear what s_{east} should be because the eastern neighbours (red cells) are at a higher refinement level, and there are multiple neighbours. CPU-HWFV1 avoids this problem by taking s_{east} to be the set of coefficients of the eastern neighbor at the same refinement level (yellow cell in Figure 3.3), whose coefficient is readily available since the encoding and decoding steps have produced $s_c^{(n)}$ at all refinement levels. Namely, when computing the flux between the cell and the red eastern neighbours at a higher refinement level, the information of the yellow eastern neighbour at the same refinement level, i.e. s_{east}, is used instead of the red eastern neighbours, which information is readily available because the MRA process has already produced flow and topographic information in all refinement levels of the hierarchy. This strategy for finding the neighbours makes it possible for CPU-HWFV1 to compute $\mathbf{L}_c$ and perform the FV1 update over the non-uniform grid efficiently – although less accurately – as choosing to compute only a single flux per neighbour per cell and therefore always computing only four fluxes in total per cell completely eliminates the possibility of warp divergence, whereas the alternative strategy of computing and accumulating the four fluxes of the four red eastern neighbours at a higher refinement level would generally lead to computing a different number of fluxes per cell in the grid, all but certainly leading to warp divergence.

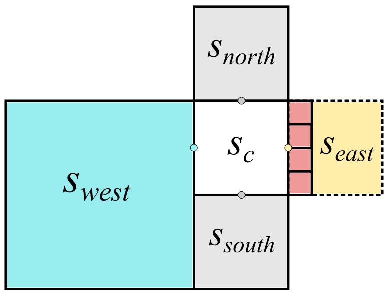


Figure 3.3: *Finding the neighbours of a cell in a non-uniform grid to retrieve s_{west}, s_{east}, s_{north}, and s_{south} and compute $\mathbf{L}_c$.*

The overview of CPU-HWFV1 is here completed, and Algorithm 3 shows pseudocode summarising the steps involved in running CPU-HWFV1. To run CPU-HWFV1, the user needs to specify the maximum refinement level (L), the error threshold (ε), and the simulation end time (t_{end}) (line 1). CPU-HWFV1 runs in a loop until t_{end} is reached (lines 3 to 10). Every iteration of the loop, a non-uniform grid is generated, i.e. a non-uniform grid is generated every timestep (lines 4 to 6). Note that after the first timestep, the details are zeroed first before redoing the encoding step, whereby encoding is performed only along the tree of significant details. CPU-HWFV1 performs an FV1 update over the generated non-uniform grid (line 7), after which the simulation time is incremented by the current timestep (line 8), and a new timestep is recomputed based on the CFL condition (line 9).

3.1.2 GPU-parallelisation of CPU-HWFV1

Having given an overview of CPU-HWFV1, the three obstacles that make it difficult to parallelise CPU-HWFV1 can be stated precisely. The first obstacle is that coalesced memory access is hindered when accessing the children $s_{[0]}^{(n+1)}$, $s_{[1]}^{(n+1)}$, $s_{[2]}^{(n+1)}$, and $s_{[3]}^{(n+1)}$ in the encoding and decoding steps unless the hierarchy of grids is indexed in a deliberate way. The second obstacle is that a

Algorithm 3 Pseudocode summarising the steps involved in running CPU-HWFV1.

```
1: procedure HWFV1($L, \varepsilon, t_{\text{end}}$)
2:     Get $s^{(L)}$ from initial discretisation of SWE on the finest grid
3:     while current time < $t_{\text{end}}$ do
4:         ENCODING($L, \varepsilon$)
5:         DECODING($s^{(0)}, 0$)                    ▷ Start decoding from coarsest cell
6:         Find neighbours to compute $\mathbf{L}_c$
7:         Use $\mathbf{L}_c$ to perform FV1 update
8:         Increment current time by timestep
9:         Compute new timestep based on CFL condition
10:    end while
11: end procedure
```

recursive depth-first traversal algorithm is used in the decoding step, which fundamentally cannot be parallelised on the GPU. The third obstacle is that the non-uniform grid generated by the MRA process is indexed using a tree data structure that hinders coalesced memory access. These three obstacles are overcome using a Z-order space-filling curve (Section 3.1.2.1), a parallel tree traversal algorithm (Section 3.1.2.2) and an array data structure (Section 3.1.2.3), respectively.

3.1.2.1 Z-order curves to achieve coalesced memory access in encoding and decoding

Encoding. Consider parallelising the encoding step assuming the cells in the hierarchy of grids are indexed in row-major order. Figure 3.4 shows the hierarchy of grids in Figure 3.1a indexed in row-major order (left panel) and the corresponding locations of the children $s_{[0]}^{(n+1)}$, $s_{[1]}^{(n+1)}$, $s_{[2]}^{(n+1)}$, and $s_{[3]}^{(n+1)}$ in memory (right panel). Using row-major indexing leads to uncoalesced memory access when loading the children (see line 4 of Algorithm 1) because the children reside in non-adjacent memory locations, as shown in Figure 3.4 to the right. To ensure the children reside in adjacent memory locations and thereby ensure coalesced memory access, an indexing scheme other than row-major order is needed.

A different indexing scheme is needed that ensures that coefficients that are nearby in the hierarchy are also nearby in memory. This kind of indexing is allowed by an SFC by definition because an SFC maps spatial data to a one-dimensional line such that data close together in space tend to be close together on the line (Sagan, 1994; Bader, 2013). There are a few different types of SFCs such as the Sierpinski curve, the Peano curve, the Hilbert curve and the Z-order curve (also known as Lebesgue or Morton curve): all of these SFCs have previously been used in the

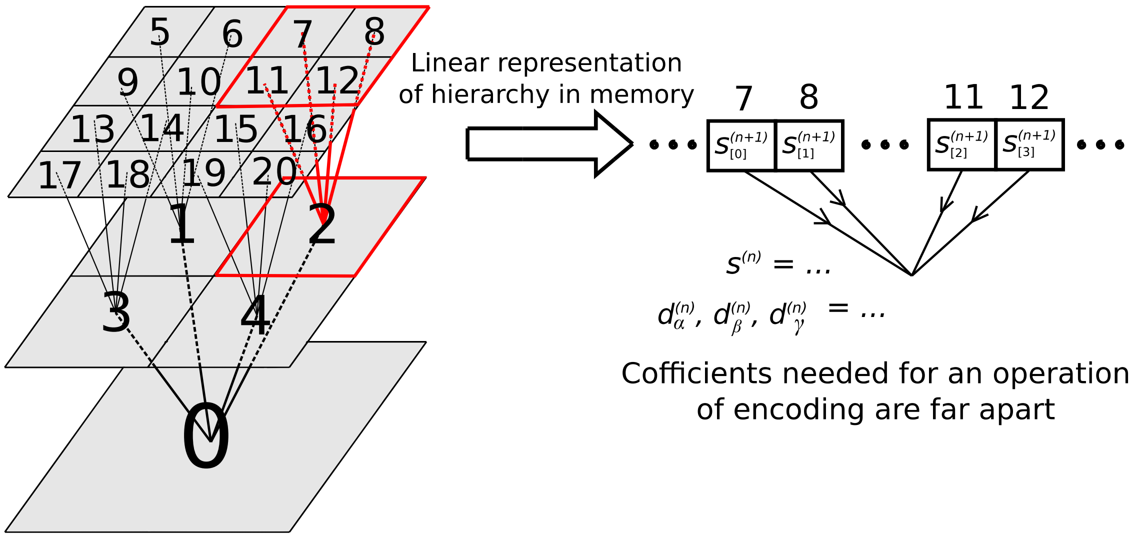

5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
1
2
3
4
0
Linear representation of hierarchy in memory
7
8
11
12
$s^{(n+1)}_{[0]}$
$s^{(n+1)}_{[1]}$
$s^{(n+1)}_{[2]}$
$s^{(n+1)}_{[3]}$
$s^{(n)} = \dots$
$d^{(n)}_{\alpha}, d^{(n)}_{\beta}, d^{(n)}_{\gamma} = \dots$
Cofficients needed for an operation of encoding are far apart

Figure 3.4: *Indexing the hierarchy of grids in row-major order (left panel) and the corresponding positions of the children* $s^{(n+1)}_{[0]}$*,* $s^{(n+1)}_{[1]}$*,* $s^{(n+1)}_{[2]}$*, and* $s^{(n+1)}_{[3]}$ *in non-adjacent memory locations (right panel).*

context of grid adaptation (Brix et al., 2009; Burstedde et al., 2011; Meister et al., 2016; Weinzierl and Mehl, 2011). Brix et al. (2009) used the Hilbert curve in a wavelet-based grid adaptation algorithm parallelised over multiple CPUs using MPI. Burstedde et al. (2011) presented the p4est AMR framework in which the Z-order curve was used to achieve load balancing. Meister et al. (2016) presented the sam(oa)^2 AMR framework in which the Sierpinski curve was applied to 2D triangular meshes. Weinzierl and Mehl (2011) applied a Peano curve to index 3D non-uniform Cartesian grids. However, none of these works were aimed at addressing the GPU-parallelisation of wavelet-based grid adaptation.

In this work, the Z-order curve is chosen because its pattern matches exactly with the parent-children square stencil shown in Figure 3.1b. A Z-order curve can be created for a square $2^n \times 2^n$ grid by following the so-called Morton codes of each cell in the grid in order. The Morton code of a cell is obtained by interleaving the bit representation of its i and j positional indices in the grid. For example, Figure 3.5 shows the creation of a Z-order curve for a $2^2 \times 2^2$ grid: the left panel shows the i (black) and j (red) indices of each cell in binary form and how the bits are interleaved (alternating red and black) to yield Morton codes (in binary form). The right panel shows how these Morton codes (in decimal form) are followed in ascending order to create the Z-order curve. The curve allows to enforce Z-order indexing of the grid because each cell in the grid can be identified on the curve (and also in memory) via its (unique) Morton code.

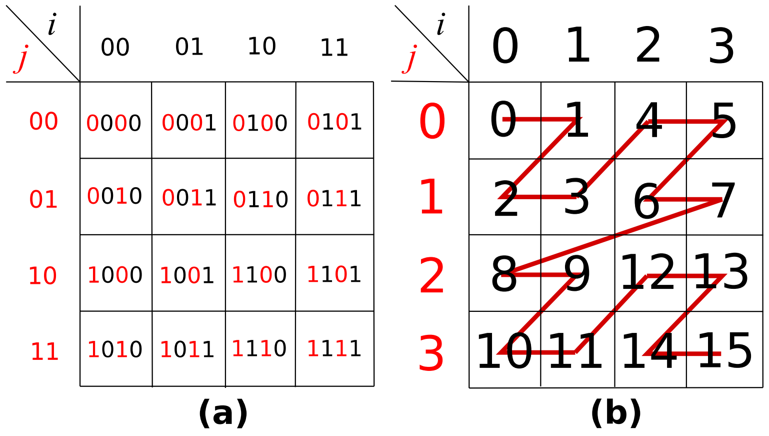


Figure 3.5: *Creation of a Z-order curve for a $2^2 \times 2^2$ grid. The left panel shows how the binary forms of the i and j indices of each cell making up a $2^2 \times 2^2$ grid are bit interleaved (alternating red and black digits) to yield so-called Morton codes (also shown in binary). The right panel shows how these Morton codes (now shown in decimal form) are followed in ascending order to create a Z-order curve and enforce Z-order indexing of the cells in the grid.*

In GPU-HWFV1, Z-order curves are created for each grid in the hierarchy while enforcing continuity in the indexing of the curves of subsequent grids, resulting in a unique Z-order index for each cell in the hierarchy. Figure 3.6 shows the hierarchy of grids in Figure 3.1a once it has been indexed using Z-order curves, alongside the corresponding locations of the children $s_{[0]}^{(n+1)}$, $s_{[1]}^{(n+1)}$, $s_{[2]}^{(n+1)}$, and $s_{[3]}^{(n+1)}$ in memory. The indexing of the Z-order curves allows for coalesced memory access during encoding because the children reside in adjacent memory locations, as seen in the right panel of Figure 3.6.

Decoding. In contrast to the encoding step, the decoding step is more difficult to parallelise because it involves applying Equations 3.11a - 3.11d while using an inherently sequential depth-first traversal algorithm to traverse the tree of significant details and identify the leaf cells (Sedgewick and Wayne, 2011), which cannot be parallelised on the GPU. To overcome this difficulty, decoding is broken down into two parts that are parallelised separately: the first part is applying Equations 3.11a - 3.11d, and the second part is traversing the tree of significant details. *Hereafter, decoding refers only to applying Equations 3.11a - 3.11d but without traversing the tree.*

Decoding – now taken to be the application of Equations 3.11a - 3.11d without traversing the tree of significant details – can be parallelised relatively easily because it can be performed using

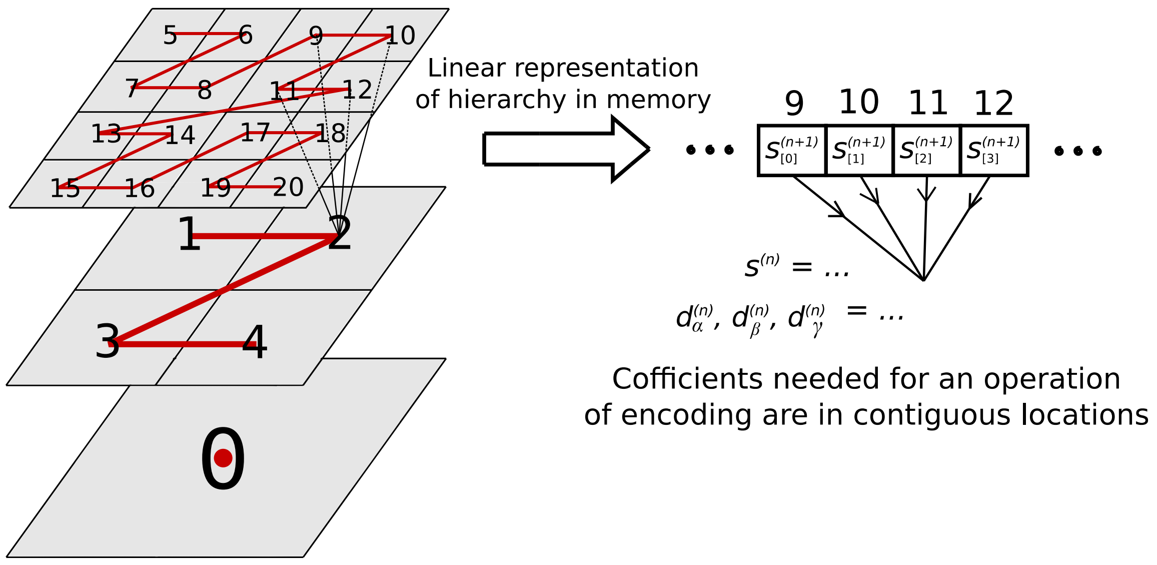

5
6
9
10
7
8
11
12
13
14
17
18
15
16
19
20
1
2
3
4
0
Linear representation of hierarchy in memory
9 10 11 12
$s^{(n+1)}_{[0]}$ $s^{(n+1)}_{[1]}$ $s^{(n+1)}_{[2]}$ $s^{(n+1)}_{[3]}$
$s^{(n)} = \ldots$
$d^{(n)}_{\alpha}, d^{(n)}_{\beta}, d^{(n)}_{\gamma} = \ldots$
Cofficients needed for an operation of encoding are in contiguous locations

Figure 3.6: *Indexing the hierarchy of grids using Z-order curves (left panel) and the corresponding positions of the children $s^{(n+1)}_{[0]}$, $s^{(n+1)}_{[1]}$, $s^{(n+1)}_{[2]}$, and $s^{(n+1)}_{[3]}$ in adjacent memory locations (right panel).*

for-loops, much like in Algorithm 1. Algorithm 4 shows pseudocode describing how to perform decoding in parallel. The pseudocode involves an outer loop and a parallelised inner loop. The outer loop iterates over the grids in the hierarchy starting from the coarsest grid up to the second-finest grid (lines 2 to 9), while the inner loop iterates through each cell in the grid in parallel (lines 3 to 8). The inner loop checks if the detail is significant (line 4), loads the parent $s^{(n)}_c$ and the details $d^{(n)}_{\alpha}$, $d^{(n)}_{\beta}$, and $d^{(n)}_{\gamma}$ (line 5), and computes the children $s^{(n+1)}_{[0]}$, $s^{(n+1)}_{[1]}$, $s^{(n+1)}_{[2]}$, and $s^{(n+1)}_{[3]}$ (line 6). This parallel inner loop achieves coalesced memory access because of the indexing of the Z-order curves, particularly when loading the parents in line 5: this can be seen by interpreting the arrows in Figure 3.6 in the reverse direction.

Algorithm 4 Pseudocode for performing the decoding step in parallel.

```
1: procedure PARALLEL_DECODING($L$)
2:     for each grid in hierarchy from 0 to $L-1$ do
3:         for each cell in the grid in parallel do
4:             if detail is significant then
5:                 Load parent $s^{(n)}_c$ and details $d^{(n)}_{\alpha}$, $d^{(n)}_{\beta}$, $d^{(n)}_{\gamma}$
6:                 Compute children $s^{(n+1)}_{[0]}$, $s^{(n+1)}_{[1]}$, $s^{(n+1)}_{[2]}$, $s^{(n+1)}_{[3]}$ with Equations 3.11a - 3.11d
7:             end if
8:         end for
9:     end for
10: end procedure
```

3.1.2.2 Parallel tree traversal algorithm

Unlike decoding (defined in Section 3.1.2.1 to be the application of Equations 3.11a - 3.11d without traversing the tree of significant details), traversing the tree of significant details fundamentally boils down to a tree traversal problem (Sedgewick and Wayne 2011). The parallelisation of tree traversal algorithms on the GPU has been investigated before (Lohr, 2009; Bédorf et al., 2012; Goldfarb et al., 2013; Karras, 2012; Zola et al., 2014; Nam et al., 2016; Chitalu et al., 2018): Bédorf et al. (2012) and Zola et al. (2014) presented parallel algorithms for construction and traversal of trees for solving gravitational N-body problems. Lohr (2009), Karras (2012), and Chitalu et al. (2018) implemented parallel algorithms for traversing tree-based bounding volume hierarchies in computer graphics applications. Goldfarb et al. (2013) demonstrated tree traversal techniques on the GPU for general-purpose rather than application-specific use. Nam et al. (2016) evaluated a parallel tree traversal algorithm in the context of nearest neighbor querying. These works suggest that a parallel tree traversal algorithm can be adopted to traverse the tree of significant details on the GPU.

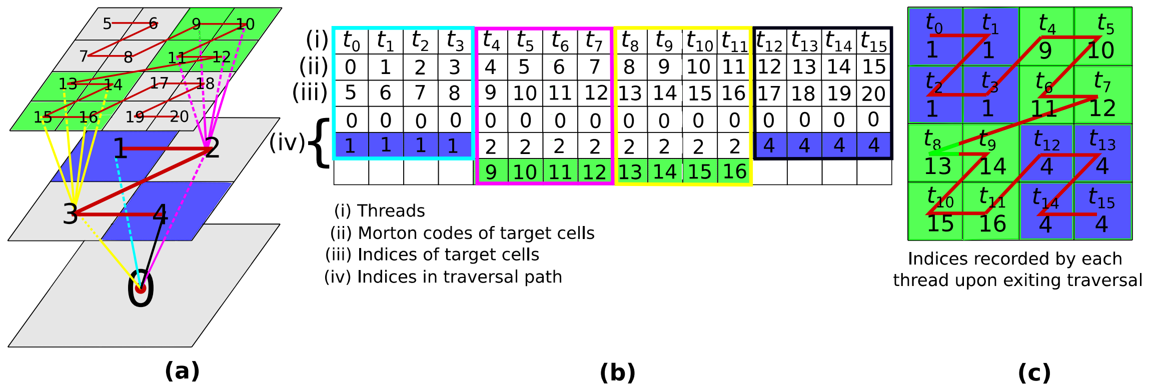
In this thesis, the parallel tree traversal algorithm developed by Karras (2012) is adopted because it can be easily modified to work with the indexing of Z-order curves (see Section 3.1.2.1). The parallel tree traversal algorithm's functionality and efficiency is explained by example by considering how it is used to traverse the tree of significant details in Figure 3.2a in parallel on the GPU (without loss of generality).

Figure 3.7a shows the tree of significant details embedded within the hierarchy of grids in Figure 3.2a after the hierarchy has been indexed using Z-order curves. To traverse this tree in parallel on the GPU, the parallel tree traversal algorithm starts by launching a CUDA kernel with one thread per cell in the finest grid in the hierarchy – for example, in Figure 3.7a, there are $2^2 \times 2^2$ = 16 cells in the finest grid, so the kernel is launched with 16 threads – whereby each thread corresponds to a "target cell" in the finest grid. Namely, let t_m denote a thread with thread index m (here, m = 0, 1, . . . , 15): t_m corresponds to a target cell in the finest grid with a Morton code m, e.g. t_6's target cell is the cell in the finest grid with Morton code 6. Each thread traverses the tree in parallel by starting at the coarsest cell in the tree (i.e. the cell with Z-order index 0; see Figure 3.7a) and climbing up progressively finer cells while aiming to reach its target cell.

Note that the target cell may not necessarily be part of the tree. For example, t_6 tries to climb up to its target cell with Morton code 6 in the finest grid. A thread stops climbing the tree if it encounters either (1) its target cell (which would be in the finest grid), or (2) a cell with a detail that is not significant (which would not be in the finest grid). For example, t_6 climbs up the cells with Z-order indices (0, 2, 11), i.e. t_6 has a *traversal path* of (0, 2, 11): it starts at the cell with Z-order index 0, climbs up to the cell with Z-order index 2, climbs up again to the cell with Z-order index 11, and then stops climbing the tree as it has reached its target cell. As another example, t_3 has a traversal path of (0, 1): it starts at the cell with Z-order index 0, climbs up to the cell with Z-order index 1, and then stops climbing the tree as it has reached a cell with a detail that is not significant. Figure 3.7b shows the traversal paths of all 16 threads launched by the parallel tree traversal algorithm. The parallel tree traversal algorithm incurs minimal warp divergence because threads in a warp are likely to have similar traversal paths. For example, t_0 to t_3 follow the same cyan path in Figure 3.7a. Similarly, t_4 to t_7 follow the magenta path, t_8 to t_{11} follow the yellow path and t_{12} to t_{15} follow the grey path.

When a thread reaches the end of its traversal path, i.e. when it stops climbing the tree of significant details, it records the Z-order index of the cell at which it stops climbing the tree: this amounts to identifying a leaf cell and recording its Z-order index. The Z-order indices of the leaf cells are recorded in an array data structure, whereby t_0 records the Z-order index of its leaf cell in the 0^{th} slot of the array, t_1 records the Z-order index in the 1^{st} slot, t_m records the Z-order index in the m^{th} slot, etc. Figure 3.7c shows the Z-order indices of the leaf cells recorded by all 16 threads after the parallel tree traversal algorithm is completed. Some of the recorded Z-order indices are duplicates because some threads (e.g. t_0 to t_3 and t_{12} to t_{15}) stop climbing the tree at the same leaf cell and therefore record the same Z-order indices. Crucial to reducing warp divergence, duplicates are more likely to emerge with increasing L as a higher value of L is more likely to introduce a higher number of cells in the non-uniform grid that are not at the highest refinement level, which reduces warp divergence because duplicate traversal paths, by definition, incur no warp divergence.

The duplicate Z-order indices are exploited to find the neighbours of each leaf cell (i.e. the neighbour finding described before in Section 3.1.1.3) in order to perform the FV1 update. The neighbours are found by making each thread look left, right, up and down from the perspective of the thread's target cell in the finest grid (see Figure 3.7c) and recording the Z-order indices of


5
6
9
10
7
8
11
12
13
14
17
18
15
16
19
20
1
2
3
4
0
(a)
(b)
(c)
(i) Threads
(ii) Morton codes of target cells
(iii) Indices of target cells
(iv) Indices in traversal path
Indices recorded by each thread upon exiting traversal

(i)	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
(ii)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(iii)	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
(iv)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	2	2	2	2	2	2	2	2	4	4	4	4
					9	10	11	12	13	14	15	16				

t_0 1	t_1 1	t_4 9	t_5 10
t_2 1	t_3 1	t_6 11	t_7 12
t_8 13	t_9 14	t_{12} 4	t_{13} 4
t_{10} 15	t_{11} 16	t_{14} 4	t_{15} 4

Figure 3.7: *Parallel tree traversal. The left panel shows the tree of significant details, with different parallel traversal paths indicated in yellow, cyan, magenta and grey. The middle panel shows the traversal paths of each thread during the parallel tree traversal in terms of the Z-order indices of the cells they traverse. The right panel shows the Z-order indices recorded by each thread after parallel tree traversal is complete.*

the neighbours. The Z-order indices of the neighbours are recorded in four array data structures – one array for each of the western, eastern, northern and southern neighbours of all the leaf cells. Figure 3.8a shows the five arrays used to store the Z-order indices of the leaf cells and their neighbours in global memory – one for the leaf cells and four for the neighbours. Duplicate sets of indices are removed – as indicated by the double black lines – via parallel stream compaction using the CUB library (Merrill, 2022). Figure 3.8b shows the arrays used to store the Z-order indices of the unique leaf cells and their neighbours without any duplicates. The unique leaf cells make up a non-uniform grid, and the arrays effectively allow to generate and index a non-uniform grid using an array data structure instead of a tree data structure. This array data structure is used to ensure coalesced memory access when performing the FV1 update over the non-uniform grid, as explained next.

3.1.2.3 Array data structure to ensure coalesced memory access in the FV1 update

Due to using an array to index the non-uniform grid, ensuring coalesced memory access when performing the FV1 update over the non-uniform grid is relatively simple. Recall from Section 3.1.1.3 that to perform the FV1 update over the non-uniform grid, $\mathbf{L}_c$ needs to be computed for each cell in the grid, and to compute $\mathbf{L}_c$ for each cell, s_c, s_{west}, s_{east}, s_{north}, and s_{south} need to be retrieved. To retrieve s_c, s_{west}, s_{east}, s_{north}, and s_{south}, a kernel is launched with one thread per leaf

Algorithm 5 Pseudocode for the parallel tree traversal algorithm. The algorithm features an iterative procedure (lines 4 to 16) instead of the recursive procedure in the depth-first traversal in Algorithm 2.

```
1:  procedure PARALLEL_TREE_TRAVERSAL(finest grid)
2:      for all cell in finest grid in parallel do
3:          start at coarsest cell
4:          while try to reach finer cell do
5:              if detail is not significant then
6:                  record z-order index of cell
7:                  stop traversing
8:              else
9:                  if reached finest cell then
10:                     record z-order index of cell
11:                     stop traversing
12:                 else
13:                     try to reach finer cell
14:                 end if
15:             end if
16:         end while
17:     end for
18: end procedure
```

cell. For example, in Figure 3.8b, there are ten leaf cells, so ten threads t_0 to t_9 are launched. Each thread uses the arrays of the Z-order indices of the leaf cells and neighbours (see Figure 3.8b) to retrieve s_c, s_{west}, s_{east}, s_{north}, and s_{south} from the hierarchy and store them in five arrays, one for each of s_c, s_{west}, s_{east}, s_{north}, and s_{south}. Note that when retrieving s_c, s_{west}, s_{east}, s_{north}, and s_{south} from the hierarchy, some uncoalesced memory access may occur. For example, consider retrieving s_c using the Z-order indices of the leaf cells in Figure 3.8b: the indices are 1, 9, ..., 16, and 4. These Z-order indices correspond to memory locations that are mostly adjacent (e.g., the indices 9, ..., 16) but not completely (e.g., the indices 1, 9, and 16, 4), thus introducing some uncoalesced memory access.

After retrieving s_c, s_{west}, s_{east}, s_{north}, and s_{south}, ensuring coalesced memory access when performing the FV1 update is trivial. The FV1 update is performed by launching a kernel with one thread per leaf cell. For example, in Figure 3.8b, there are ten leaf cells, so ten threads t_0 to t_9 are launched. Now suppose the arrays in Figure 3.8b are storing s_c, s_{west}, s_{east}, s_{north}, and s_{south} instead of the Z-order indices of the leaf cells and neighbours, i.e. they explicitly store flow and topographic information instead of indices or pointers: to perform the FV1 update in parallel, each

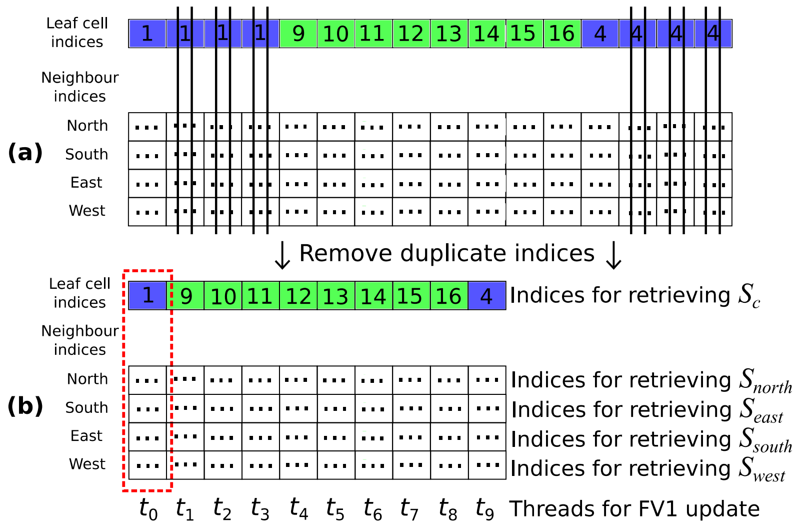

Leaf cell indices
1 1 1 1 9 10 11 12 13 14 15 16 4 4 4 4
Neighbour indices
North
(a)
South
East
West
↓ Remove duplicate indices ↓
Leaf cell indices
1 9 10 11 12 13 14 15 16 4
Indices for retrieving S_c
Neighbour indices
North
(b)
South
East
West
Indices for retrieving S_{north}
Indices for retrieving S_{east}
Indices for retrieving S_{south}
Indices for retrieving S_{west}
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 Threads for FV1 update

Figure 3.8: *Parallel FV1 update. The upper panel shows the Z-order indices of the leaf cells and their neighbours stored in memory after parallel tree traversal and neighbour finding, which includes duplicate indices. The lower panel shows the Z-order indices of the leaf cells and their neighbours without any duplicates, which are used to retrieve s_c, s_{west}, s_{east}, s_{north}, and s_{south} and compute $\mathbf{L}_c$ for the FV1 update in parallel.*

thread retrieves s_c, s_{west}, s_{east}, s_{north}, and s_{south} from the arrays to compute $\mathbf{L}_c$. For example, t_0 would use the first column of indices (red box in Figure 3.8b), t_1 would use the second column, and so on. Each thread t_0 to t_9 retrieves s_c, s_{west}, s_{east}, s_{north}, and s_{south} from adjacent memory locations, so coalesced memory access is ensured. After performing the FV1 update, incrementing the current simulation time by the timestep is trivial, and the new minimum timestep based on the CFL condition is computed by performing a so-called parallel minimum reduction using the CUB library (Merrill, 2022).

The three obstacles described at the start of Section 3.1.2 are thus overcome, and all the steps involved in running CPU-HWFV1 (lines 4 to 9 of Algorithm 3) are parallelised, so implementing GPU-HWFV1 is complete. Having implemented GPU-HWFV1, the efficiency of its GPU-parallelised HW-based grid adaptation capability can be analysed.

3.2 Analysing GPU-HWFV1's performance

The primary aim of this section is to analyse the efficiency of GPU-HWFV1's GPU-parallelised HW-based grid adaptation capability. The efficiency analysis is performed quantifying GPU-HWFV1's potential speedup over two existing and validated SWE models, namely the GPU-parallelised FV1

Table 3.1: *Test cases to compare GPU-HWFV1 against GPU-FV1 and CPU-HWFV1*

Test name	**Test type**	**Use**	**Previously used in**
Quiescent flow over irregular topographies with different steepness. Dam-break flow over realistic topography with friction effects (Section 3.2.1)	Synthetic	Verifying accuracy	Song et al. (2011); Huang et al. (2013); Kesserwani et al. (2018); Kesserwani and Sharifian (2020); Shirvani et al. (2021)
Circular 2D dam-break flow (Section 3.2.2)	Synthetic	Assessing speedup	Kesserwani and Sharifian (2020); Wang et al. (2010)
Pseudo-2D dam-break flow (Section 3.2.2)	Synthetic	Assessing speedup	Kesserwani et al. (2019); Kesserwani and Sharifian (2020)
Dam-break wave interaction with an urban district (Section 3.2.3)	Realistic	Assessing speedup	Jeong et al. (2012); Caviedes-Voullième et al. (2020); Kesserwani and Sharifian (2020)
Tsunami wave propagation over a complex beach (Section 3.2.3)	Realistic	Assessing speedup	Hou et al. (2015); Arpaia and Ricchiuto (2018); Caviedes-Voullième et al. (2020); Kesserwani and Sharifian (2020)

model running on a uniform grid in LISFLOOD-FP 8.0 (Shaw et al., 2021), i.e. GPU-FV1, and the serial CPU version of the HWFV1 model (Kesserwani and Sharifian, 2020), i.e. CPU-HWFV1. A secondary aim of this section is to verify GPU-HWFV1's accuracy, namely by confirming that GPU-HWFV1 preserves a similar level of accuracy as GPU-FV1. This verification is mainly carried out by checking that GPU-HWFV1 obtains flow predictions that are close to those obtained by GPU-FV1.

GPU-HWFV1's accuracy and speedup is analysed by running simulations of five test cases that feature a wide range of topographies and flow types (see Table 3.1). The first test case (Section 3.2.1) involves simulating both quiescent and dam-break flows over irregular topographies with frictional effects and is used solely to verify GPU-HWFV1's accuracy.

The second and third test cases (Section 3.2.2) involve simulating synthetic dam-break flows over flat topographies and are mainly used to systematically analyse the speedup achieved by GPU-

Table 3.2: *Aspects ε, L, and flow type against which GPU-HWFV1's speedup over CPU-HWFV1 and GPU-FV1 is systematically analysed.*

Aspect	**Description**	**Effect on amount of grid coarsening**
L	User-specified parameter for controlling the grid resolution of the grid at the maximum refinement level	Finer maximum allowable grid resolution with a higher value of L
ε	User-specified parameter for controlling the amount of grid coarsening	Higher amount of coarsening introduced into GPU-HWFV1's non-uniform grid with a larger value of ε
Flow type	Test-case dependent: ranges from rapid (e.g. shocks, wave reflections, and diffractions) to smooth (e.g. very shallow or nearly flat waves)	Rapid flows allow for less grid coarsening

HWFV1 with respect to three aspects, namely the error threshold ε, the maximum refinement level L, and the flow type: these three aspects are considered because they affect the amount of coarsening in GPU-HWFV1's non-uniform grid (see Table 3.2), which is hypothesised to directly affect GPU-HWFV1's speedup over CPU-HWFV1 and GPU-FV1.

As an example, Figure 3.9 shows relatively higher and lower amounts of coarsening in GPU-HWFV1's non-uniform grid on the left and right, respectively. GPU-HWFV1's speedup over CPU-HWFV1 may increase with a lower amount of grid coarsening: with a lower amount of grid coarsening, the number of cells in the non-uniform grid is higher and therefore so is the computational throughput or workload – and the speedup afforded by efficient GPU-parallelisation should scale with the computational throughput in a phenomenon known as weak scaling (Gustafson, 1988; Shaw et al., 2021). Conversely, GPU-HWFV1's speedup over GPU-FV1 may increase with a higher amount of grid coarsening, as this reduces the cell count of GPU-HWFV1's non-uniform grid compared to GPU-FV1's uniform grid, thus reducing the computational effort (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020).

The systematic analysis of the speedup is therefore performed by simulating the second and third test cases using GPU-HWFV1 and CPU-HWFV1 with different pairs of ε and L, with $\varepsilon = \{10^{-4}, 10^{-3}, 10^{-2}\}$ to cover a range around the recommended value of 10^{-3} for maintaining a fair compromise between accuracy and speedup (Kesserwani et al., 2019; Kesserwani and Sharifian,

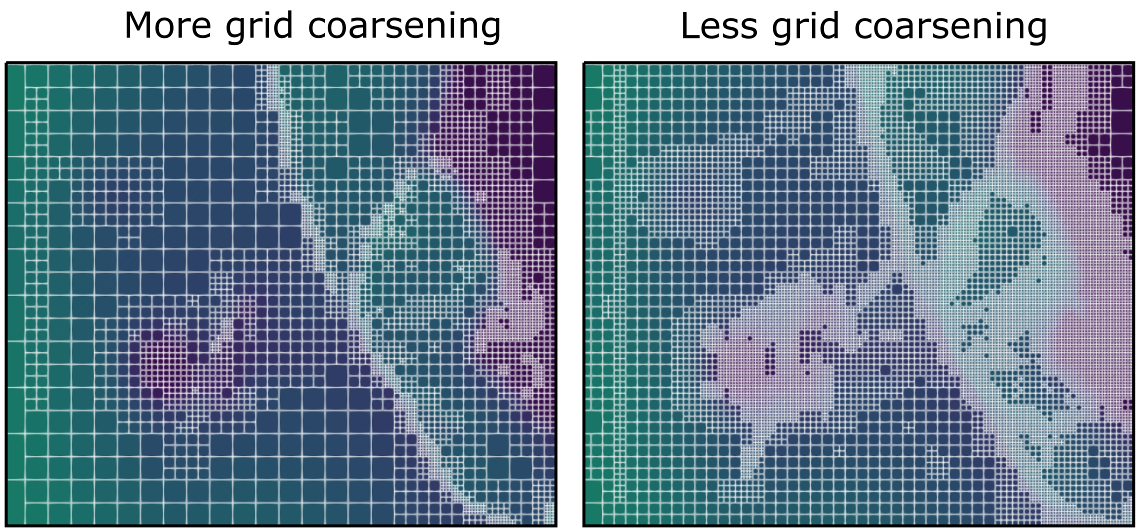


Figure 3.9: *Relatively higher (left panel) and lower (right panel) amounts of grid coarsening in GPU-HWFV1's non-uniform grid from the "Tsunami wave propagation over a complex beach" test case (Section 3.2.3).*

2020), and with $L = \{8, 9, 10, 11\}$ – as no speedups were identified for $L \leq 7$ and using $L \geq 12$ exceeded the memory capacity of the GPU card used (RTX 2070). Simulations are also run using GPU-FV1 – always on a uniform grid at the finest grid resolution accessible to GPU-HWFV1 at a given value of L.

Informed by the systematic analysis of the speedup achieved by GPU-HWFV1 when simulating synthetic dam-break flows over flat topographies, GPU-HWFV1 is finally analysed in the fourth and fifth test cases (Section 3.2.3), which involve simulating more realistic flows over complex topographies represented by raster grid DEMs. Due to the involvement of the DEMs, GPU-HWFV1 must be run with fixed values of L to accommodate the size of the DEM raster grid in each test case (unlike in the synthetic test cases where no DEMs were present and L could be freely chosen). Furthermore, the topography limits the amount of coarsening allowed in the non-uniform grid because GPU-HWFV1's grid adaptation capability cannot introduce any coarsening in its non-uniform grid beyond that allowed by the topography.

3.2.1 Verifying accuracy

The first test case is used solely to verify GPU-HWFV1's accuracy by considering two simulation scenarios. The first scenario is to verify GPU-HWFV1's ability to preserve a quiescent (i.e. unmoving) flow state in the presence of wet-dry fronts with different levels of steepness in topography and different wetting conditions. This is done by checking its C-property, which is the ability to balance the fluxes with the source terms under steady-state conditions (Greenberg and Leroux, 1996). If GPU-HWFV1 is C-property, then it should not disturb an initially quiescent flow state as the fluxes should exactly balance with the source terms. The second simulation scenario is to verify GPU-HWFV1's ability to reproduce a realistic dam-break flow with friction effects and moving wet-dry fronts. For both scenarios, the test case area is 70 m $\times$ 30 m with closed wall boundaries, which includes humps to represent an irregular topography. Three hump shapes are considered with increasingly steeper topographic slopes to verify the C-property for realistic topographies, as shown in the upper panels of Figure 3.10 (smooth on the left, steeper in the middle and rectangular on the right). For each hump shape, appropriate initial conditions are applied (see Table 3.3) with zero velocities to generate an unmoving free-surface elevation that leads to different wetting conditions around and/or at the humps.

Simulations are run with GPU-FV1 and GPU-HWFV1 up to 100 s with $\varepsilon = 10^{-3}$ and $L = 8$ (requiring around 3,000 timesteps to complete) while measuring the discharges. For GPU-HWFV1 to be deemed C-property, the free-surface elevation should stay undisturbed during the simulation, meaning that the discharges should not grow too much from zero and the number of cells in the adaptive grid should be constant over time. The middle and bottom row of panels of Figure 3.10 show the time histories of the maximum discharge errors (i.e. the maximum deviation from zero) and the number of grid cells for the three hump profiles. The errors are seen to become increasingly higher with increased irregularity in the hump profile, but nonetheless remain bounded as also observed for the CPU model counterparts (Kesserwani and Sharifian, 2020); meanwhile, the number of cells remains constant over time. This demonstrates that GPU-HWFV1 is C-property irrespective of the steepness of the bed slope and the presence of wet-dry fronts in the domain area.

Next, GPU-HWFV1 is used to run a simulation of a frictional dam-break flow ($n_{\mathrm{M}} = 0.018$ $\mathrm{m}^{1/3}$/s) over a non-trivial topography, namely the smooth hump profile (Figure 3.10, top left panel). The dam-break flow involves an initial condition where a water body of 1.875 m upstream of the humps is blocked by a hypothetical dam located at $x = 16$ m. The dam-break flow is simulated up to 12 s using GPU-HWFV1 with the same choice of ε and L as above. Another simulation is run with GPU-FV1 to allow for like-for-like comparisons of flood maps at outputs times reported in previous

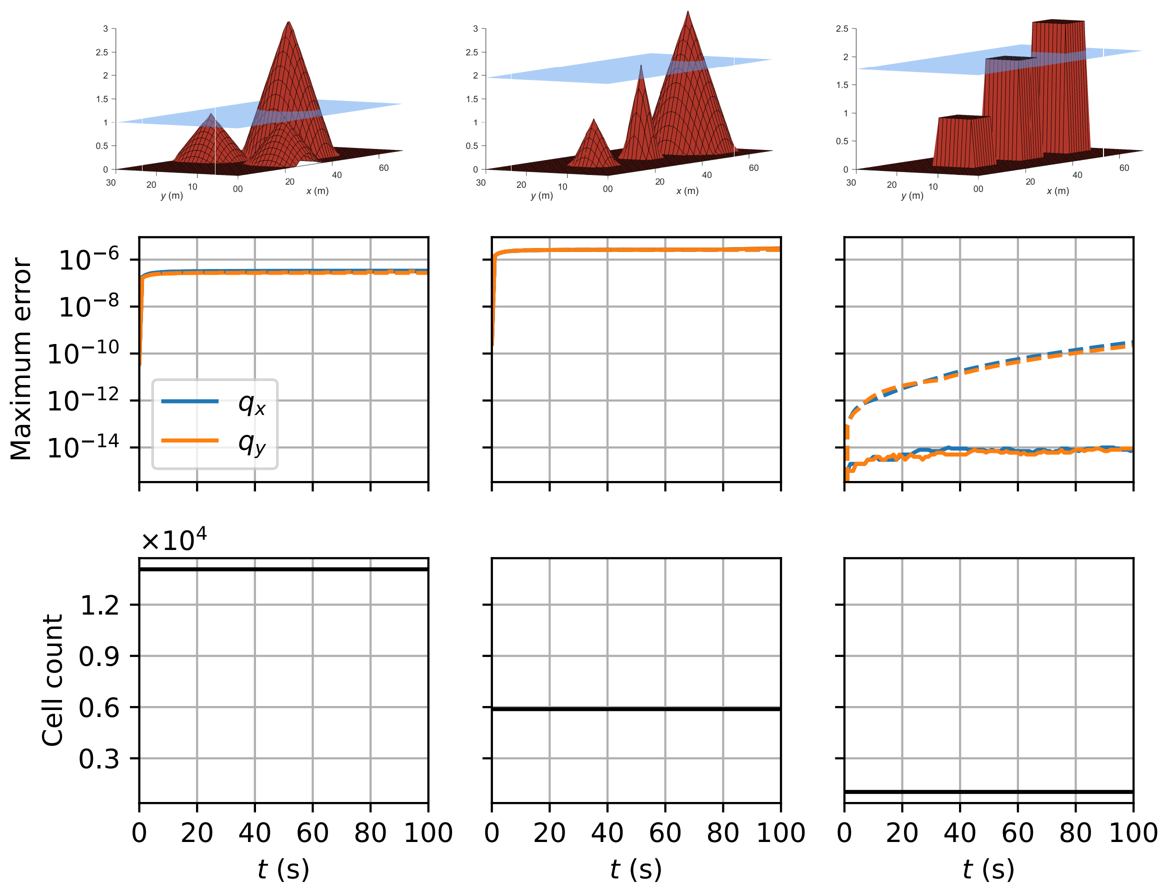


Figure 3.10: *Verifying GPU-HWFV1's C-property in the presence of wet-dry fronts with different levels of steepness in the topography and different wetting conditions: smooth humps (left panels), steeper humps (middle panels) and rectangular humps (right panels). The upper panels show the shapes of the humps, the middle panels show the time histories of the maximum discharge errors (solid lines - GPU-FV1, dashed lines - GPU-HWFV1), where $q_x = hu$ and $q_y = hv$, and the bottom panels show the time histories of the number of grid cells.*

studies (Shirvani et al., 2021; Song et al., 2011). Figure 3.11 includes the flood maps predicted by GPU-HWFV1 (left panel) compared to those predicted by GPU-FV1 (right panel) at 0, 6, and 12 s. At 0 s (upper panel), both models are seen to start from the same initial flow conditions. At 6 s (middle panel), both models predict that the small humps are completely submerged and that the dam-break wave has reached the large hump, where the L^1 error between flood maps predicted by GPU-HWFV1 and GPU-FV1 is 4.6×10^{-4}. There are similar wave patterns surrounding the large hump at 12 s (lower panel), where the L^1 error is 9.2×10^{-4}. In all the predictions, GPU-HWFV1 shows symmetrical flood maps that are similar to those reproduced by GPU-FV1 and other hydrodynamic profiles reported in previous works (Shirvani et al., 2021; Song et al., 2011).

Table 3.3: *Zero-velocity initial conditions for generating an unmoving water surface flow for the three hump shapes shown in Figure 3.10.*

Hump profile	$h + z$ **(m)**	**Wetting conditions**	**Reference**
Smooth	0.875	Dry around the highest hump, critical ($h = 0$ m) over the two small humps	Song et al. (2011); Huang et al. (2013); Shirvani et al. (2021)
Steeper	1.78	Dry around the highest hump, critical ($h = 0$ m) at the peak of the medium hump, wet above the shortest hump ($h > 0$ m)	Kesserwani et al. (2018)
Rectangular	1.95	Dry around the highest hump, critical ($h = 0$ m) at the peak of the medium hump, wet above the shortest hump ($h > 0$ m)	Kesserwani et al. (2018)

These results suggest that GPU-HWFV1 preserves a similar level of accuracy as well-established models in simulations of frictional dam-break flows over non-trivial topographies.

In the next test cases, in addition to verifying GPU-HWFV1's accuracy, analyses of the speedups are also performed.

3.2.2 Speedup analyses using simulations of synthetic dam-break flows

Circular 2D dam-break flow. This test case has often been used to verify the accuracy of new SWE models by assessing their ability to simulate the symmetric propagation of shocks and rarefaction waves in a closed $[-20\,\text{m}, 20\,\text{m}]^2$ test case area (Toro, 2001). The test case involves an initial water depth of 2.5 m inside a hypothetical cylindrical dam that is separated from a water depth of 0.5 m outside the dam. The dam is removed at $t = 0\,\text{s}$, causing a dam-break flow over a flat and frictionless topography in which a shock wave moves radially outwards and a rarefaction wave moves radially inwards, which eventually collapses to form a secondary shock. Due to the presence of the shock and rarefaction wave in the test case area, the flow in this test case is always rapid.

The test case is first used to verify GPU-HWFV1 by comparing the water depths predicted by GPU-HWFV1 to those predicted by GPU-FV1. Simulations of this test case are run up to $t = 3.5\,\text{s}$ using GPU-HWFV1 – with the same choice of ε and L as in Section 3.2.1 – and GPU-

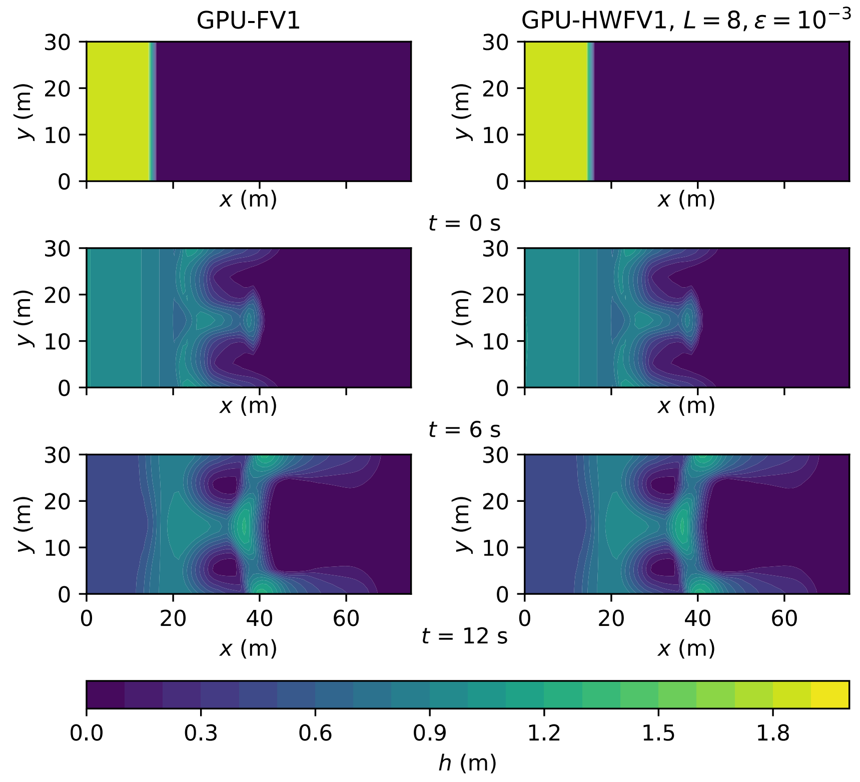


Figure 3.11: *Verifying GPU-HWFV1's accuracy by simulating a frictional dam-break flow with moving wet-dry fronts over a non-trivial topography. Flood maps predicted by GPU-HWFV1 and GPU-FV1 on the left and right panels, respectively. At 0 s, the dam break wave emerges (upper panels). At 6 s, the wave has submerged the small humps (middle panels). At 12 s, the wave starts to surround the large hump (lower panels).*

FV1. Figure 3.12 shows the water depth centrelines predicted by the two models. GPU-HWFV1 predicts water depths that are visually identical to those predicted by GPU-FV1. The benchmark solution was produced using the FV1 numerical solution of the 1D radial form of the 2D shallow water equations using 256×256 cells, following Toro (2001). Next, more simulations of this test case are run using GPU-HWFV1, GPU-FV1, and CPU-HWFV1 to analyse GPU-HWFV1's speedup.

To analyse the speedup, simulations of this test case are now run using different pairs of ε and L as described at the start of Section 3.2. Runtimes of the simulations are recorded to calculate GPU-HWFV1's speedup over CPU-HWFV1 and GPU-FV1. Figure 3.13 shows GPU-HWFV1's speedup over CPU-HWFV1 and GPU-HWFV1 against the maximum refinement level L in the left and right panels, respectively. The black lines indicate the average speedup ratios over the three values of ε, and the dash-dotted lines indicate the breakeven point above which GPU-HWFV1 is faster than GPU-FV1 or CPU-HWFV1 (which is used in all subsequent speedup figures). As seen

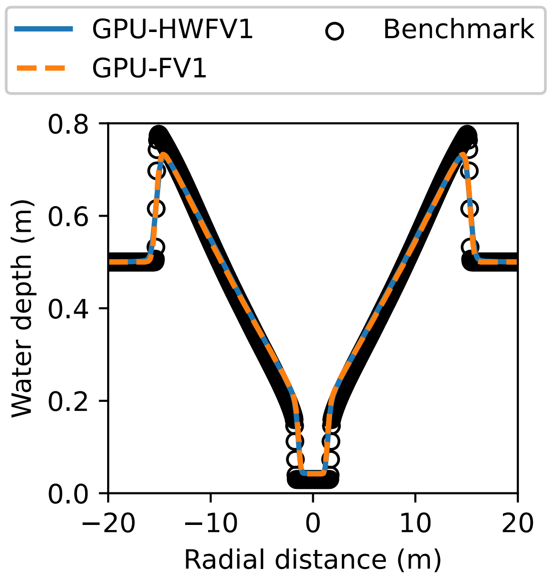


Figure 3.12: *Circular 2D dam-break flow. Verifying GPU-HWFV1's accuracy using the same choice of ε and L as in Section 3.2.1 (($\varepsilon = 10^{-3}$ and $L = 8$): water depth centrelines at $t = 3.5\,s$ predicted by GPU-HWFV1 and GPU-FV1 compared to the benchmark solution. The benchmark solution was produced using the FV1 numerical solution of the 1D radial form of the 2D shallow water equations using 256×256 cells, following Toro (2001).*

in the left panel of Figure 3.13, GPU-HWFV1 is 5 to 46× faster than CPU-HWFV1. Furthermore, the speedup is proportional to the increase in L and the decrease in ε. This observation suggests that GPU-HWFV1's grid adaptation capability is much more efficient than CPU-HWFV1 due to being implemented on the GPU and becomes more efficient as the choice of L is increased and/or the choice of ε is decreased. This finding seems reasonable because as L increases or ε decreases, the cell count of the non-uniform grid increases, leading to a higher computational throughput for which GPU-parallelisation is expected to yield a higher speedup due to weak scaling. Next, GPU-HWFV1's potential speedup over GPU-FV1 is analysed.

Compared to GPU-FV1, GPU-HWFV1 does not become faster until $L \geq 9$, as seen in the right panel of Figure 3.13. Furthermore, GPU-HWFV1 achieves a maximum speedup of 3× for the largest $\varepsilon = 10^{-2}$, and around 2× for the smaller $\varepsilon = 10^{-3}$ and 10^{-4}. These observations suggest that despite the rapid flow present in this test case – which reduces the amount of coarsening allowed in GPU-HWFV1's non-uniform grid and thus does not reduce the computational effort much – GPU-HWFV1 is still faster than GPU-FV1. This finding suggests that GPU-HWFV1 is likely to achieve higher speedups over GPU-FV1 the lower the value of ε – which increases the

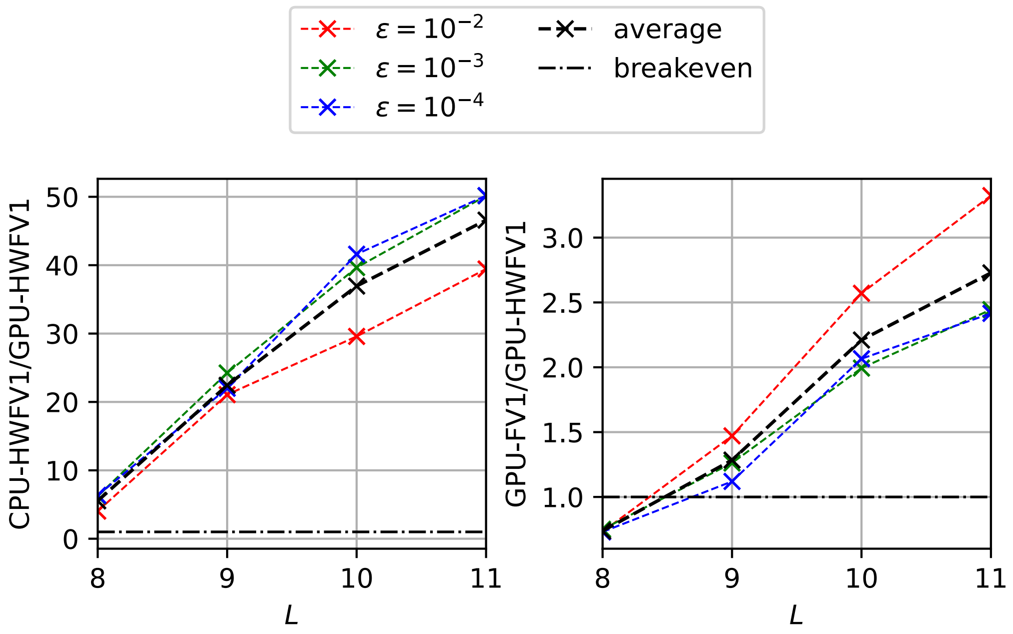


Figure 3.13: *Circular 2D dam-break flow. GPU-HWFV1's speedup over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel).*

amount of coarsening in GPU-HWFV1's non-uniform grid and thereby lowers the computational effort more – and/or the higher the value of L – which may lead to a very fine uniform grid for GPU-FV1 and thus a very high computational effort (for GPU-FV1).

In the next test case, the speedup achieved by GPU-HWFV1 is analysed *over time*, by running simulations in which the flow gradually transitions from rapid to very smooth and therefore features a much wider range of flow types, unlike in this test case where only rapid flows were present.

Pseudo-2D dam-break flow. This test case about 1D dam-break flow is conventionally used to verify the accuracy of SWE models using a relatively short 2.5 s simulation that involves modelling the propagation of a shock and rarefaction wave moving in opposite directions. However, it was recently used in much longer 40 s simulation to analyse the speedup of CPU-based wavelet adaptive SWE models when modelling a flow that gradually transitions from rapid to very smooth (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020). The test case area is 50 m × 25 m and is assumed to be flat and frictionless with open boundary conditions. Within this test case area, a hypothetical dam is located at $x = 10$ m, which initially separates an upstream water depth of 6 m from a downstream water depth of 2 m. The dam is removed at $t = 0$ s, introducing a shock and a rarefaction wave that move in opposite directions. Both the shock and the rarefaction wave remain in the test case area up to 3 s. After 3 s, the shock wave exits the test case area from the downstream boundary, and the flow is driven only by the rarefaction wave, up to 10 s. After 10 s, the rarefaction wave exits the test case area from the upstream boundary and thus, the flow gradually becomes smoother and smoother up to 40 s.

The test case is first used to verify GPU-HWFV1 by running a short 2.5 s simulation and comparing the water depths predicted by GPU-HWFV1 to those predicted by GPU-FV1. Simulations are run using GPU-HWFV1 with the same choice of ε and L as in Section 3.2.1 ($\varepsilon = 10^{-3}$ and $L = 8$), and also using GPU-FV1. Figure 3.14 shows water depth centrelines predicted by GPU-HWFV1 and GPU-FV1, all showing good agreement with the exact solution (Delestre et al., 2011), although exhibiting a smoothening of the shock wave due to the numerical diffusion from using an FV1 scheme that is first-order accurate.

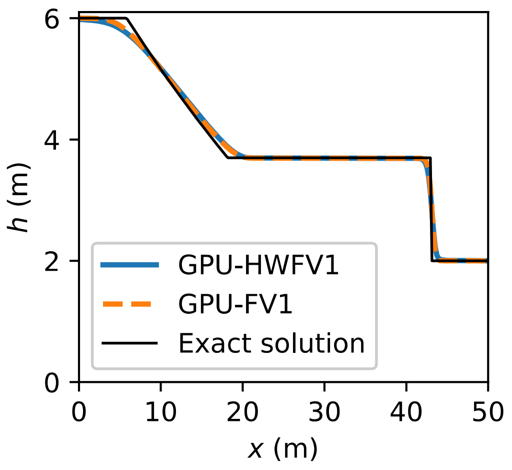


Figure 3.14: *Pseudo-2D dam-break flow. Verifying GPU-HWFV1 using the same choice of ε and L as in Section 3.2.1 ($\varepsilon = 10^{-3}$ and $L = 8$) : water depths centerlines predicted by GPU-HWFV1 and GPU-FV1 at $t = 2.5$ s compared with the exact solution.*

Next, longer 40 s simulations of this test case are run using GPU-HWFV1, CPU-HWFV1, and GPU-FV1 to assess GPU-HWFV1's potential speedup. This is done while utilising the different combinations of ε and L as discussed at the start of Section 3.2. However, unlike the previous test case where the runtimes of the simulations were recorded, in this test case, *time series* of the runtimes are recorded throughout the 40 s simulations. These time series of runtimes are used to calculate the time series of GPU-HWFV1's speedup over CPU-HWFV1 and GPU-FV1, which are shown in Figure 3.15 for the different values of ε and L.

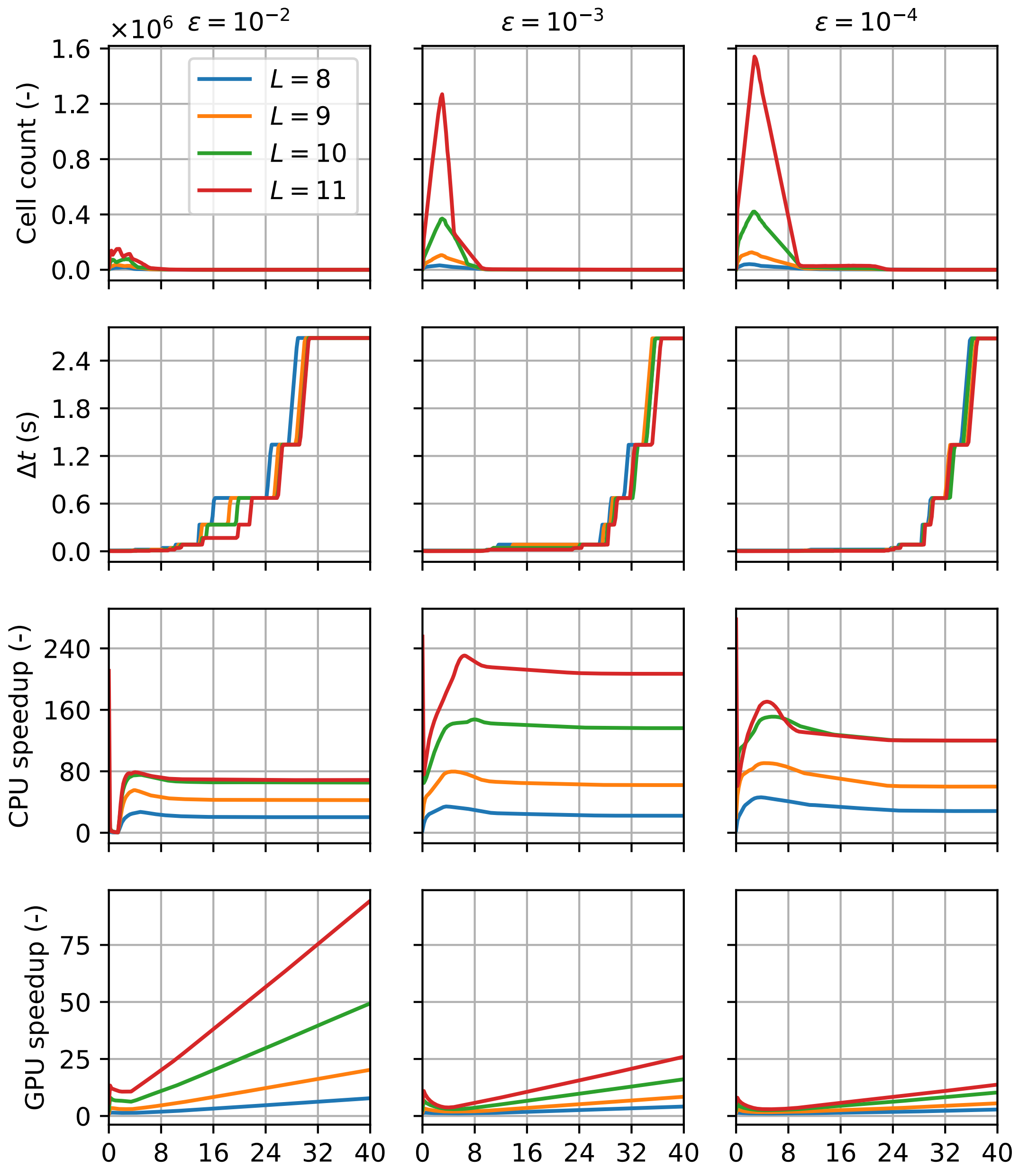


Figure 3.15: *Pseudo-2D dam-break flow. Time histories of the number of cells in the non-uniform grid (first row of panels), the timestep Δt (second row of panels), GPU-HWFV1's speedup over CPU-HWFV1 (third row of row panels) and GPU-FV1 (fourth row of panels) for different values of ε and L.*

GPU-HWFV1 is always faster than CPU-HWFV1 (Figure 3.15, upper panels) except at $\varepsilon = 10^{-2}$ and $L = 8$ up to 2.5 s. This means that GPU-HWFV1 is always faster than CPU-HWFV1 except at the smallest value of L – which leads to a finest allowable grid with a relatively coarse resolution – and the largest value of ε – which leads to a relatively high amount of coarsening in the non-uniform grid: with this combination of ε and L, the cell count of the grid is relatively low as shown in the top left panel, and thus the computational throughput may be too low for GPU-parallelisation to yield a speedup. Nonetheless, even at $\varepsilon = 10^{-2}$ and $L = 8$, GPU-HWFV1 achieves a peak 20× speedup at around 3 s, when the shock and rarefaction wave have produced rapid flow features that refine the non-uniform grid and thus increase the computational throughput, thereby increasing the speedup yielded by GPU-parallelisation. With all the other combinations of ε and L, GPU-HWFV1 always achieves a significant speedup. Namely – in line with the findings from the previous test case – GPU-HWFV1 becomes increasingly faster than CPU-HWFV1 as ε decreases or L increases: at the smallest ε and highest L, GPU-HWFV1 achieves an average speedup of 100× throughout the simulation, and a maximum speedup of 240× at 2.5 s when rapid flow features, i.e. the shock and rarefaction wave, are still present in the test case area. These results suggest that GPU-HWFV1's grid adaptation capability benefits from GPU-parallelisation over the CPU version when modelling all types of flow – whether rapid or smooth – but especially with a higher cell count in the non-uniform grid.

Compared to GPU-FV1 (Figure 3.15, lower panels), at $\varepsilon = 10^{-2}$, GPU-HWFV1 achieves speedups that scale with the value of L, achieving a maximum speedup of 80× at $L = 11$. For $L \geq 9$, GPU-HWFV1 is always faster than GPU-FV1 during the entire simulation, but even at $L = 8$, GPU-HWFV1 achieves an 10× speedup by the end of the simulation. Thus, it appears that GPU-HWFV1 achieves a higher speedup as the simulation progresses because the flow becomes smoother and a higher amount of coarsening is allowed in GPU-HWFV1's non-uniform grid, thereby reducing the computational effort more. At $\varepsilon = 10^{-3}$, GPU-HWFV1 again achieves speedups that scale with the value of L, although achieving a lower maximum speedup of 25× at $L = 11$. For $L \geq 10$, GPU-HWFV1 is always faster than GPU-FV1, while for $L \leq 9$, GPU-HWFV1 achieves a speedup after 10 s, when the rarefaction wave has exited the test case area and the flow has become considerably smoother. At $\varepsilon = 10^{-4}$, the same patterns are seen as for the other values of ε, where GPU-HWFV1 achieves a lower maximum speedup of 12× (compared to 75× and 25× at $\varepsilon = 10^{-2}$ and 10^{-3}, respectively), which is achieved after the flow becomes very smooth. These results seem to confirm the findings from the previous test case that GPU-HWFV1's grid adaptation capability yields a higher speedup over GPU-FV1 either as the choice of ε is increased, e.g. for $\varepsilon > 10^{-4}$ – as this allows for a higher amount of coarsening in GPU-HWFV1's non-uniform grid – or as the choice of L is increased, e.g. for $L \geq 9$ – as this leads to a very fine uniform grid for GPU-FV1. The results also suggest that GPU-HWFV1's grid adaptation capability may yield a higher speedup over GPU-FV1 when the flow is smooth, as this also allows for a higher amount of coarsening in GPU-HWFV1's non-uniform grid.

Informed by the effect of ε, L, and the flow type on the speedup achieved by GPU-HWFV1 when simulating synthetic dam-break flows over flat topographies, the next two test cases are used to investigate the speedup achieved by GPU-HWFV1 when simulating more realistic rapid flows over complex topographies.

3.2.3 Further investigations into speedups: realistic flow simulations

Dam-break wave interaction with an urban district. This test case involves a dam-break flow over a laboratory-scale topography that represents an urban district. The test case has widely been used for verifying SWE model predictions due to the availability of a set of spatial experimental data for the water depth and the velocities from the physical experiment (Soares-Frazão and Zech, 2008). The dam-break flow occurs in a $36\,\text{m} \times 3.6\,\text{m}$ smooth channel ($n_{\text{M}} = 0.01$), in which there is a wall barrier with a gate (green bar in Figure 3.16) that initially separates an upstream water body of $0.4\,\text{m}$ from a downstream water body of $0.011\,\text{m}$. Downstream of the gate, there are twenty-five $0.3\,\text{m} \times 0.3\,\text{m}$ square blocks with $0.1\,\text{m}$ gaps between them; the square blocks and the wall barrier have a height of $2\,\text{m}$. The gate is opened abruptly at $t = 0\,\text{s}$, at which point a dam-break wave forms and flows swiftly to collide with the blocks. The blocks almost entirely impede the shock, creating a backwater zone upstream, while the unimpeded flow cascades through the gaps to form a hydraulic jump downstream as the simulation progresses.

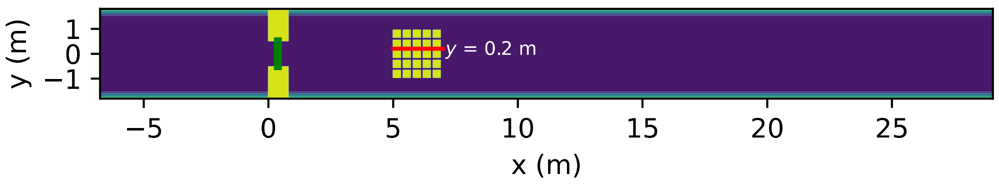


Figure 3.16: *Dam-break wave interaction with an urban district. Top-down view of the test case area, with the gate indicated in green and topographic blocks colored in yellow. Experimental depth and velocity data are available along $y = 0.2\,m$, indicated in red.*

Based on the height and the dimensions of the square blocks and the wall barrier in the topography (as reported in Soares-Frazão and Zech (2008)), a DEM representing the topography can be constructed. The DEM consists of a raster grid with a size of $1790 \times 180 = 322,200$ cells at a resolution of 0.02 m. Note that the raster grid explicitly includes the square blocks and the wall barrier as topographic features, which are therefore included in the well-balanced topography source terms in GPU-HWFV1's FV1 scheme. With this DEM, a 10 s simulation of this test case is run using GPU-HWFV1 – with $L = 11$ to accommodate the DEM size – using two values of $\varepsilon = \{10^{-4}, 10^{-3}\}$, and also using GPU-FV1. Figure 3.17 shows the water depth (left panel) and velocity (right panel) profiles along $y = 0.2$ m (red line in Figure 3.16) at 6 s predicted by GPU-HWFV1 and GPU-FV1, as well as the experimental data. Both GPU-HWFV1 and GPU-FV1 predicted profiles that are within the expected range of agreement with the experimental profiles, with any deviations being similar to as seen in other shallow water models in the literature and also due to the experimental sensors not capturing flow data at the same spatial resolution as the computational grid (Caviedes-Voullième et al., 2020; Kesserwani & Sharifian, 2020). Compared to the prediction made by GPU-FV1, those made by GPU-HWFV1 with $\varepsilon = 10^{-4}$ are closer than with $\varepsilon = 10^{-3}$, though the difference is not significant. Next, the speedups are analysed.

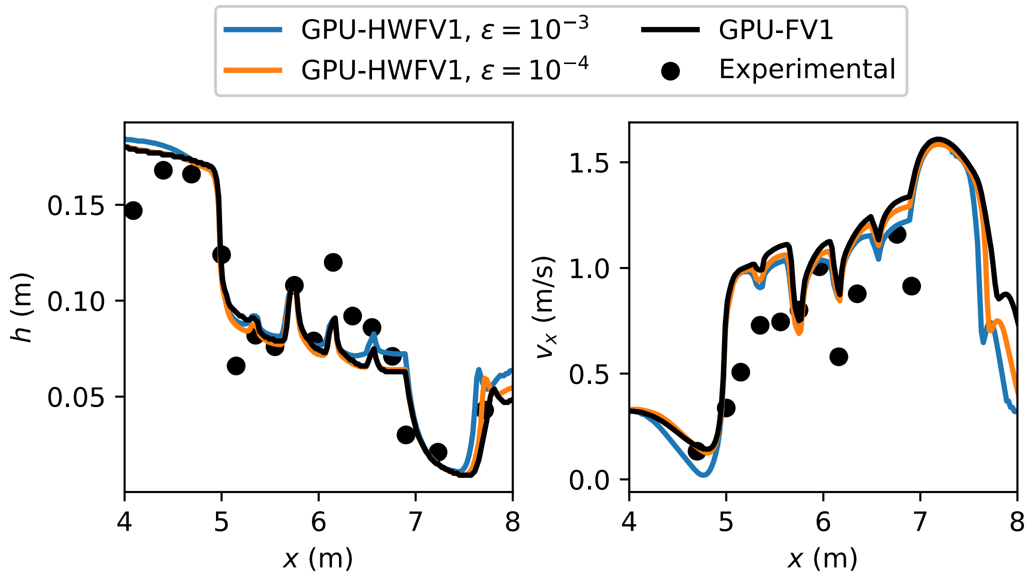


Figure 3.17: *Dam-break wave interaction with an urban district. Depth (left panel) and velocity (right panel) profiles predicted by GPU-HWFV1 and GPU-FV1 along $y = 0.2$ m in Figure 3.16 at 6 s.*

Figure 3.18 shows the time series of GPU-HWFV1's speedup over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel). On average, GPU-HWFV1 achieves a 19× and 25× speedup over

CPU-HWFV1 during the 10 s simulation using $\varepsilon = 10^{-3}$ and 10^{-4} respectively: a higher speedup is achieved using a larger ε, in line with the findings in Section 3.2.2. GPU-HWFV1 is also faster than GPU-FV1, being on average around 2.4× faster for both $\varepsilon = 10^{-3}$ and 10^{-4}, which is expected given the large value of $L = 11$ needed to accommodate the DEM size in this test case. Up to 2 s, GPU-HWFV1 achieves a higher speedup at $\varepsilon = 10^{-3}$ than 10^{-4}, whereas after 2 s, the speedup decreases, likely due to the rapid flow features produced by the dam-break wave as it cascades through the square blocks that reduce the amount of coarsening GPU-HWFV1's non-uniform grid, increasing the computational effort and decreasing the speedup. Overall, GPU-HWFV1 remains faster than both CPU-HWFV1 and GPU-FV1 in this test case, which – supported by the findings in Section 3.2.2 that the speedup achieved by GPU-HWFV1 increases as the value of L increases – can be expected given the large value of $L = 11$ needed to accommodate the DEM size. In the next and final test case investigated in this chapter, a smaller DEM size is considered

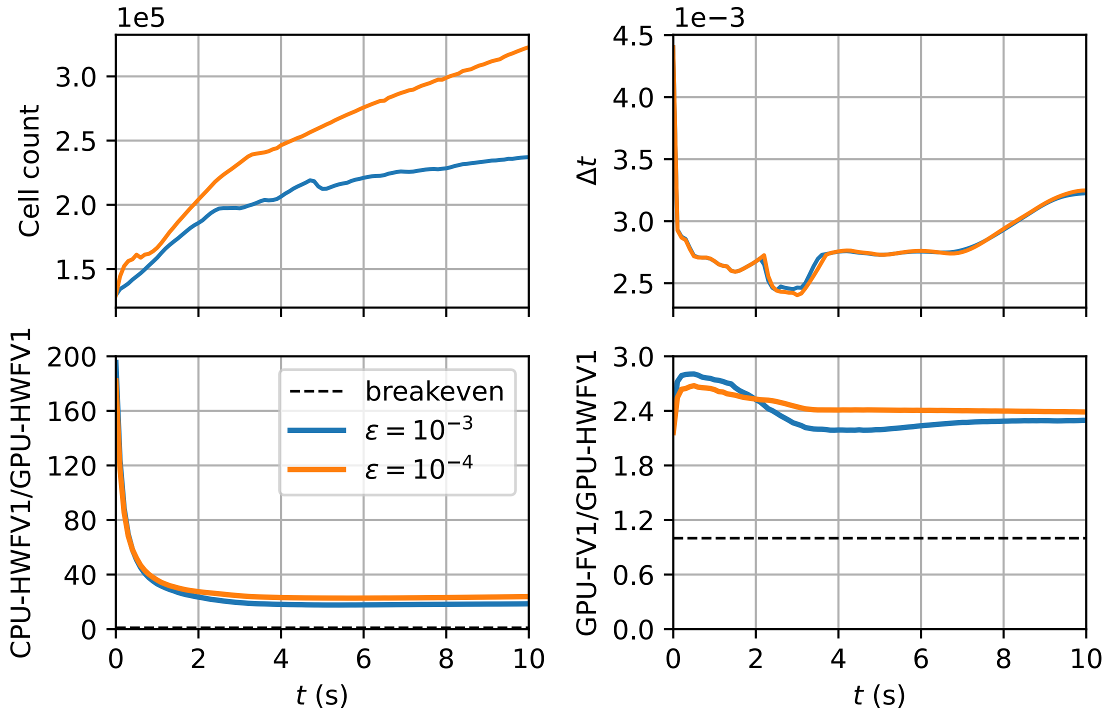


Figure 3.18: *Dam-break wave interaction with an urban district. Time histories of GPU-HWFV1's non-uniform grid cell count (top left panel), Δt (top right panel), speedup over CPU-HWFV1 (bottom left panel) and over GPU-FV1 (bottom right panel).*

Tsunami wave propagation over a complex beach. This test case considers a 1:400 scaled replica of the 1993 tsunami at Okushiri island, Japan (Matsuyama & Tanaka, 2001). It has been used in other works for verifying SWE models, and also for assessing the speedup of wavelet-based SWE models versus their uniform-grid versions when run on the CPU (Caviedes-Voullième et al., 2020; Kesserwani and Sharifian, 2020). The test case involves a complex topography with a smooth surface ($n_{\mathrm{M}} = 0.01$) in a $5.488\,\mathrm{m} \times 3.402\,\mathrm{m}$ test case area with closed boundaries except for the left boundary, through which a tsunami enters. The tsunami propagates from left to right until eventually flooding the coastal area (Figure 3.19, left panel, yellow contours), where there is a gauge point (Figure 3.19, left panel, red dot; $x = 4.521\,\mathrm{m}$, $y = 1.696\,\mathrm{m}$). At this gauge point, experimental time histories of the water surface elevation are available.

The complex topography of this test case is represented by a DEM with a size of $392 \times 243 = 95,256$ cells at a resolution of $0.014\,\mathrm{m}$: as the DEM size is smaller compared to the previous test case, a smaller value of L is needed to accommodate the DEM. Namely, this test case is simulated using GPU-HWFV1 with a smaller value of $L = 9$ instead of 11 like in the previous test case. The right panel of Figure 3.19 shows the time histories of the water surface elevations predicted by GPU-HWFV1 and GPU-FV1 at the gauge point. Both GPU-HWFV1 and GPU-FV1 predict the expected gradual retraction in the water surface elevation between 12 s and 15 s, followed by a sharp increase that peaks at around 17 s. At $\varepsilon = 10^{-4}$, GPU-HWFV1 obtains predictions that are visually indistinguishable from those predicted by GPU-FV1, with the black, blue and orange lines almost completely overlapping each other. With $\varepsilon = 10^{-3}$, the predictions are similar, subject to small, localised discrepancies when there is a sharp flow transition, such as at around 18 and 21 s. Next, the speedups are analysed, for which simulations are also run using CPU-HWFV1.

Figure 3.20 shows the time series of GPU-HWFV1's speedup over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel). GPU-HWFV1 is significantly faster than CPU-HWFV1, achieving speedups of 200× and 400× with $\varepsilon = 10^{-3}$ and 10^{-4}, respectively. This is likely because the topography is more complex compared to the previous test case, thus reducing the amount of coarsening in the non-uniform grid that is further reduced by the rapid flow features produced by the tsunami as it propagates through the test case area. Due to the flow features reducing the amount of coarsening in the non-uniform grid, the cell count of the grid is high and so is the computational throughput, and so GPU-parallelisation yields a higher speedup, and GPU-HWFV1 achieves a remarkable speedup over CPU-HWFV1 that increases as ε is decreased from 10^{-3} to $\varepsilon = 10^{-4}$ – in line with the observations in the previous test case and in Section 3.2.2 – but the

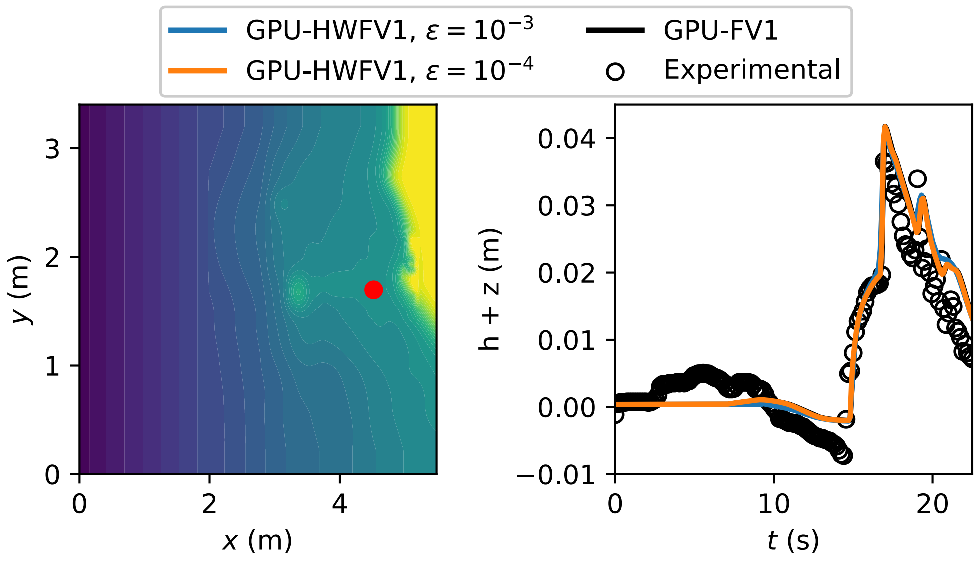


Figure 3.19: *Tsunami wave propagation over a complex beach. The left panel shows a top-down view of the test case area including the gauge point (red dot). The right panel shows water surface elevation predicted by GPU-HWFV1 and GPU-FV1 at the gauge point.*

speedup doubles in this test case.

Compared to GPU-FV1 (Figure 3.20, right panel), GPU-HWFV1 only achieves a speedup in this test case at $\varepsilon = 10^{-3}$ (around 1.25×), whereas at $\varepsilon = 10^{-4}$, the speedup falls slightly below the breakeven line meaning that GPU-HWFV1 becomes slower than GPU-FV1. This is likely because at $\varepsilon = 10^{-4}$, the amount of coarsening in GPU-HWFV1's non-uniform grid is low due to the complex topography and flow features, meaning that the computational effort is not reduced enough to achieve a speedup. Overall, in this test case, GPU-HWFV1 is much faster than CPU-HWFV1, and is faster than GPU-FV1 at $\varepsilon = 10^{-3}$ but not at $\varepsilon = 10^{-4}$. Nonetheless, despite the complexity of the topography and flow features and the small value of $L = 9$ used to accommodate the DEM, GPU-HWFV1 remains faster and a viable choice over GPU-FV1 at $\varepsilon = 10^{-3}$.

3.3 Summary

This chapter addressed Objective 1 by redesigning the algorithmic structure of a HW-based grid adaptation algorithm based on the MRA process of the HW in order and make it suitable for efficient implementation on the GPU, namely by proposing the use of three computational ingredients: a Z-order space-filling curve, a parallel tree traversal algorithm, and an array data structure.

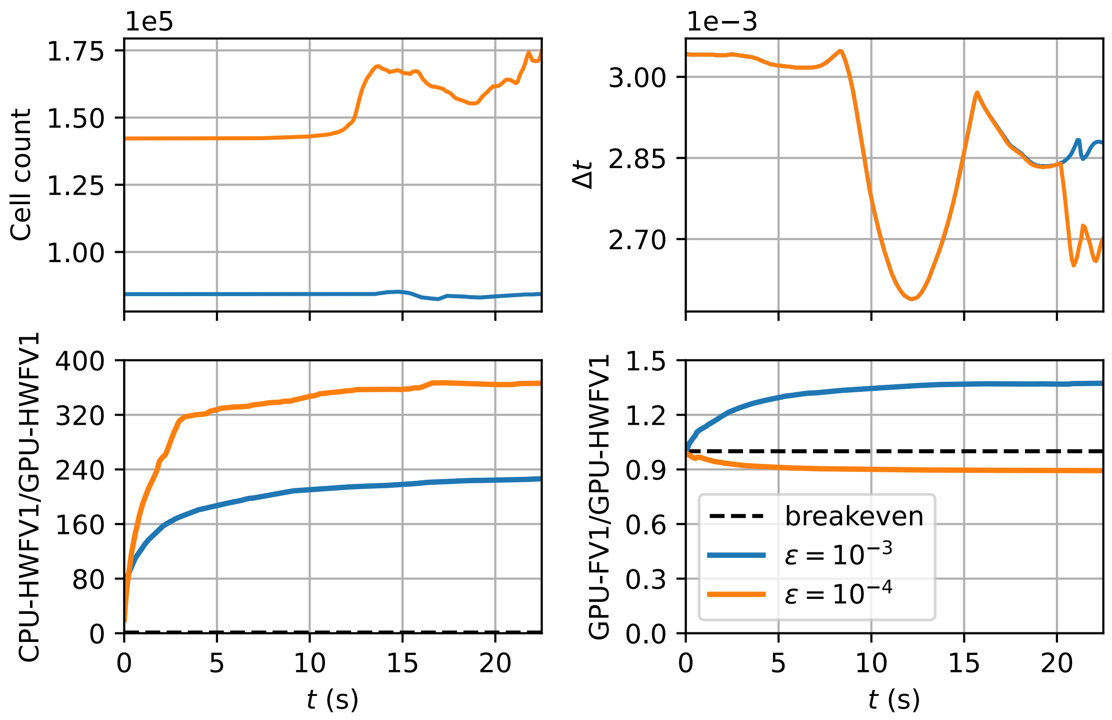


Figure 3.20: *Tsunami wave propagation over a complex beach. Time histories of GPU-HWFV1's non-uniform grid cell count (top left panel), Δt (top right panel), speedup over CPU-HWFV1 (bottom left panel) and over GPU-FV1 (bottom right panel)..*

The MRA process generates a cell-based non-uniform grid by "encoding" (coarsening), "decoding" (refining), analysing, and traversing modelled data across a deep hierarchy of nested, uniform grids of increasingly coarser resolution, in which the coarsest grid in the hierarchy is made up of a single cell, while the finest grid in the hierarchy is made up of $2^L \times 2^L$ cells, where L is a user-specified maximum refinement level. In the encoding step, a user-specified error threshold ε is needed to flag significant "details" to decide which cells to include in the non-uniform grid. The encoding step results in a tree-like structure of significant details that is traversed in the decoding step using a sequential depth-first traversal algorithm to identify "leaf cells" which make up a non-uniform grid. Once the identified leaf cells are assembled into a non-uniform grid, an FV1 scheme is applied to perform the solver computations (as defined in Section 2.2.4) over the non-uniform grid.

The MRA process is difficult to implement on the GPU efficiently due to three obstacles. The first obstacle is that coalesced memory access is hindered when accessing the children in the encoding and decoding steps unless the hierarchy of grids is indexed in a deliberate way. The second obstacle is that a recursive depth-first traversal algorithm is used in the decoding step, which fundamentally cannot be parallelised on the GPU. The third obstacle is that the non-uniform grid generated by the MRA process is indexed using a tree data structure, which hinders coalesced memory access. This chapter overcame these three obstacles using the three computational ingredients proposed above: first, Z-order curves were used to index the hierarchy of grids and thereby ensure coalesced memory access in the encoding and decoding steps. Second, the parallel tree traversal algorithm was used to replace the depth-first traversal algorithm in order to traverse the tree of significant details on the GPU – with minimal warp divergence. Finally, an array data structure was used instead of a tree data structure to index the non-uniform grid and thereby achieve coalesced memory access when performing the solver computations over the non-uniform grid via the FV1 scheme.

Using the three computational ingredients allowed to implement the HW-based grid adaptation algorithm on the GPU and thereby develop a GPU-parallelised HW adaptive FV1 shallow water model (GPU-HWFV1). To assess the extent to which using the three computational ingredients allowed for an efficient GPU implementation of the HW-based grid adaptation algorithm, GPU-HWFV1's accuracy and speedup were analysed against its CPU version (CPU-HWFV1) and against its reference uniform-grid counterpart, i.e. a GPU-parallelised FV1 shallow water model running on a uniform grid in LISFLOOD-FP 8.0 (GPU-FV1) – the latter always run on a uniform grid with the finest grid resolution accessible to GPU-HWFV1. The analysis was performed by running simulations of five test cases featuring a wide range of topographies and flow types. The first test case was used solely to verify GPU-HWFV1's accuracy, namely by checking GPU-HWFV1's well-balancedness over wet-dry fronts using increasingly irregular topographic features, and also by checking GPU-HWFV1's ability to simulate a frictional dam-break flow with moving wet-dry fronts over a non-trivial topography. The second and third test cases were used to systematically analyse GPU-HWFV1's potential speedup over CPU-HWFV1 and GPU-FV1 when simulating synthetic dam-break flows over flat topographies, particularly with respect to three aspects: the error threshold ε, the maximum refinement level L, and the flow type – these aspects were considered because they affect the amount of grid coarsening introduced by GPU-HWFV1's grid adaptation capability, which was hypothesised to directly affect GPU-HWFV1's potential speedup. Therefore, the systematic analysis was performed simulating by the second and third test cases – which together included a range of flow types – using different combinations of ε and L, with $\varepsilon = \{10^{-2}, 10^{-3}, 10^{-4}\}$ to cover a range around the recommended value of $\varepsilon = 10^{-3}$ for obtaining a fair compromise between accuracy and speedup, and with $L = \{8, 9, 10, 11\}$. Informed by the effects of these three aspects on the speedup achieved by GPU-HWFV1 when simulating synthetic dam-break flows over flat topographies, the fourth and fifth test cases were used to analyse GPU-HWFV1's performance (using $\varepsilon = \{10^{-3}, 10^{-4}\}$) when simulating rapid flows over complex topographies represented by DEMs.

GPU-HWFV1's performance in all the test cases provided strong evidence that it delivers a similar level of accuracy as GPU-FV1 in replicating realistic flows including in the presence of irregular topographies, wet-dry fronts, and friction effects. In terms of the speedup over CPU-HWFV1, GPU-HWFV1 yielded significant speedups in all the test cases (20× to 400×), which increased either as ε was decreased or as L was increased: both of these increase the number of cells in the non-uniform grid and thus the computational throughput, so this finding indicates that GPU-HWFV1's grid adaptation capability becomes increasingly faster than its CPU version as the computational throughput is increased, therefore hinting at GPU-HWFV1's weak scaling behaviour.

In terms of the speedup over GPU-FV1, the findings from the synthetic test cases suggested that GPU-HWFV1's grid adaptation capability yields a higher speedup over GPU-FV1 (which was between 1.1 to 75) either as the choice of ε is increased, e.g. for $\varepsilon \geq 10^{-4}$, as this introduces a higher amount of coarsening in GPU-HWFV1's non-uniform grid and thus reduces the computational effort more compared to GPU-FV1's uniform grid, or as the choice of L is increased, e.g. for $L \geq 9$, as this leads to a very fine uniform grid for GPU-FV1 and thus a very high computational effort (for GPU-FV1). The results also suggested that GPU-HWFV1's grid adaptation capability may yield a higher speedup over GPU-FV1 when simulating smooth flows, as this also allows for a higher amount of coarsening in GPU-HWFV1's non-uniform grid. For the realistic test cases, GPU-HWFV1 showed speedup over GPU-FV1 for the test with $L = 11$, and also for the test with $L = 9$, although only for $\varepsilon = 10^{-3}$ – in line with the findings from the synthetic test cases.

The conclusions suggest that the GPU implementation of the MRA process of the HW – which was achieved using the three proposed computational ingredients, i.e. the Z-order curve, the parallel tree traversal algorithm, and the array data structure – is suitably efficient. Thus, extending the use of the three computational ingredients in order to implement the MRA process of the MW on the GPU and thereby enhance GPU-DG2 with grid adaptation appears to be a research direction worth exploring. This direction is explored in the next chapter (Chapter 4).

Chapter 4

Integrating GPU-MWDG2 into LISFLOOD-FP

This chapter addresses Objective 2, in which the three computational ingredients proposed in Chapter 3 for efficiently parallelising the MRA process of the HW – i.e. the Z-order curve, the parallel tree traversal algorithm and the array data structure – are extended to the MRA process of the MW. Extending the use of the three ingredients to the MRA process of the MW allows to implement the MRA process of the MW on the GPU and thereby develop a GPU-parallelised MW adaptive DG2 model, i.e. GPU-MWDG2.

The aim of developing GPU-MWDG2 is to achieve faster DG2 modelling than GPU-DG2 when simulating test cases that involve multiscale rapid flow phenomena – i.e. rapid flow phenomena that occur over multiple spatial and temporal scales – where it is hypothesised that GPU-MWDG2's grid adaptation capability may reduce the computational effort of DG2 modelling while still allowing to achieve a similar level of accuracy as GPU-DG2. To test this hypothesis, GPU-MWDG2 is integrated into a new version of LISFLOOD-FP – version 8.2 – to take advantage of LISFLOOD-FP's existing framework for running real-world flood simulations and thereby properly compare GPU-MWDG2 against GPU-DG2 – as the latter is already integrated into LISFLOOD-FP.

This chapter is organised as follows. Section 4.1 presents the implementation of GPU-MWDG2. Section 4.2 describes the integration of GPU-MWDG2 into LISFLOOD-FP 8.2. Section 4.3 gives a summary of the GPU-MWDG2 implementation and its integration into LISFLOOD-FP 8.2.

4.1 Implementing GPU-MWDG2

4.1.1 Overview

Like the GPU-HWFV1 model (Section 3.1), GPU-MWDG2 is a GPU-parallelised wavelet-based adaptive SWE model that runs hydrodynamic simulations by solving the 2D SWE over a non-uniform grid. GPU-MWDG2 is made up of two mechanisms: a wavelet-based grid adaptation algorithm for generating a non-uniform grid, followed by a numerical scheme performing solver computations over the generated non-uniform grid. GPU-MWDG2 is dynamically adaptive, meaning it applies the wavelet-based grid adaptation algorithm to generate a non-uniform grid every timestep before performing the solver computations. Unlike GPU-HWFV1 however, the wavelet-based grid adaptation algorithm in GPU-MWG2 is based on the MRA process of the MW (Kesserwani and Sharifian, 2020; Keinert, 2003) instead of the HW, while the solver computations are performed using a DG2 scheme (Kesserwani et al., 2018) instead of an FV1 scheme.

DG2 scheme. Unlike the FV1 scheme in GPU-HWFV1 – which locally approximates the variables h, hu, hv, and z as piecewise-constant solutions over each cell Q_c in the grid – the DG2 scheme locally approximates the variables as piecewise-planar solutions. For ease of presentation of the DG2 scheme, let s be a dummy variable standing for any of these variables: the shape of the piecewise-planar solution for s over a cell Q_c is controlled by a "shape vector" made up of three scalars $\mathbf{s}_c = [s_c^0, s_c^{1x}, s_c^{1y}]^{\mathrm{T}}$ – instead of only a single scalar for controlling the piecewise-constant solution in the FV1 scheme – where s_c^0 is an average coefficient, and s_c^{1x} and s_c^{1y} are x- and y-directional slope coefficients. The shape vector $\mathbf{s}_c$ stands for any of $\mathbf{h}_c$, $(\mathbf{hu})_c$, $(\mathbf{hv})_c$ or $\mathbf{z}_c$, which are shape vectors for the variables h, hu, hv, and z, respectively. In the DG2 scheme, the average and slope components of the flow coefficients, i.e. $\mathbf{U}_c^K = [\mathbf{h}_c^K, (\mathbf{hu})_c^K, (\mathbf{hv})_c^K]^{\mathrm{T}}$, $K = 0, 1x, 1y$, are updated in time – from timestep p to timestep $p + 1$ – by performing a "DG2 update". The DG2 update consists of a two-stage Runge-Kutta (RK2) scheme and a spatial operator $\mathbf{L}_c^K$. In the RK2 scheme, $\mathbf{U}_c^K$ is first updated from timestep p (denoted by a superscript) to an intermediate timestep *int*, and then from timestep *int* to the next full timestep $p+1$ as follows:

$$\mathbf{U}_c^{K,int} = \mathbf{U}_c^{K,n} + \Delta t \mathbf{L}_c^K(\mathbf{U}_c^{K,p}), \tag{4.1a}$$

$$\mathbf{U}_c^{K,p+1} = \frac{1}{2}\left[\mathbf{U}_c^{K,p} + \mathbf{U}_c^{K,int} + \Delta t \mathbf{L}_c^K(\mathbf{U}_c^{K,int})\right], \tag{4.1b}$$

where Δt is a timestep computed based on the CFL condition using a CFL number of 0.33 to ensure numerical stability (Kesserwani and Sharifian, 2020). The expressions for $\mathbf{L}_c^K$, $K = 0, 1x, 1y$ are:

$$\mathbf{L}_c^0 = -\frac{1}{\Delta x}\left(\tilde{\mathbf{F}}_e - \tilde{\mathbf{F}}_w\right) - \frac{1}{\Delta y}\left(\tilde{\mathbf{G}}_n - \tilde{\mathbf{G}}_s\right) + \begin{bmatrix} 0 \\ -\frac{2g\sqrt{3}}{\Delta x}\overline{h}_c^0 \overline{z}_c^{1x} \\ -\frac{2g\sqrt{3}}{\Delta y}\overline{h}_c^0 \overline{z}_c^{1y} \end{bmatrix} \tag{4.2a}$$

$$\mathbf{L}_c^{1x} = -\frac{\sqrt{3}}{\Delta x}\left((\tilde{\mathbf{F}}_e + \tilde{\mathbf{F}}_w - \mathbf{F}(\overline{\mathbf{U}}_c^{0x} - \overline{\mathbf{U}}_c^{1x}) - \mathbf{F}(\overline{\mathbf{U}}_c^{0x} + \overline{\mathbf{U}}_c^{1x})\right) + \begin{bmatrix} 0 \\ 2g\overline{h}_c^{1x}\overline{z}_c^{1x} \\ 0 \end{bmatrix}, \tag{4.2b}$$

$$\mathbf{L}_c^{1y} = -\frac{\sqrt{3}}{\Delta y}\left((\tilde{\mathbf{G}}_n + \tilde{\mathbf{G}}_s - \mathbf{G}(\overline{\mathbf{U}}_c^{0y} - \overline{\mathbf{U}}_c^{1y}) - \mathbf{G}(\overline{\mathbf{U}}_c^{0y} + \overline{\mathbf{U}}_c^{1y})\right) + \begin{bmatrix} 0 \\ 0 \\ 2g\overline{h}_c^{1y}\overline{z}_c^{1y} \end{bmatrix}. \tag{4.2c}$$

where – as in the FV1 scheme – $\tilde{\mathbf{F}}_e$, $\tilde{\mathbf{F}}_w$, $\tilde{\mathbf{G}}_n$, and $\tilde{\mathbf{G}}_s$ are numerical fluxes at the eastern, western, northern, and southern cell interfaces, respectively, which are computed using two-argument numerical flux functions $\tilde{\mathbf{F}}(\cdot,\cdot)$ and $\tilde{\mathbf{G}}(\cdot,\cdot)$ based on the HLL approximate Riemann solver (Toro, 2001). Unlike the FV1 scheme, however, the two arguments of the flux functions are now the limits of the piecewise-planar solutions – instead of the limits of piecewise-constant solutions – at the left and right sides of a cell interface after temporary revision of the solution limits according to the same wetting-and-drying condition described in Section 3.1.1.1 to ensure well-balancedness and non-negative water depths (Kesserwani et al., 2018). When applying the wetting-and-drying condition using the limits of piecewise-planar solutions, the solution limits at the northern, eastern, southern, and western interfaces are computed using the shape coefficients in $\mathbf{s}_c$ as $s_{c,\mathrm{lim,n}} = s_c^0 + \sqrt{3}s_c^{1y}$, $s_{c,\mathrm{lim,e}} = s_c^0 + \sqrt{3}s_c^{1x}$, $s_{c,\mathrm{lim,s}} = s_c^0 - \sqrt{3}s_c^{1y}$, and $s_{c,\mathrm{lim,w}} = s_c^0 - \sqrt{3}s_c^{1x}$, respectively. For example, at the eastern interface of a cell Q_c, the left limit is computed using the shape coefficients in $\mathbf{s}_c$ as $s_e^L = s_{c,\mathrm{lim,e}} = s_c^0 + \sqrt{3}s_c^{1x}$, while the right limit is computed using the shape coefficients in $\mathbf{s}_{\mathrm{east}}$ as $s_e^R = s_{\mathrm{east,lim,w}} = s_{\mathrm{east}}^0 - \sqrt{3}s_{\mathrm{east}}^{1x}$, where the *east* subscript is the index of the eastern neighbour cell. The overbars in Equations 4.2a to 4.2c indicate temporary coefficients obtained from the revised limits, namely $\overline{\mathbf{U}}_c^{0x} = \frac{1}{2}[\mathbf{U}_w^{L,*} + \mathbf{U}_e^{R,*}]$, $\overline{\mathbf{U}}_c^{0y} = \frac{1}{2}[\mathbf{U}_n^{L,*} + \mathbf{U}_s^{R,*}]$, $\overline{\mathbf{U}}_c^{1x} = \frac{1}{2}\left[\frac{1}{3}(\mathbf{U}_w^{L,*} - \mathbf{U}_e^{R,*})\right]$, and $\overline{\mathbf{U}}_c^{1y} = \frac{1}{2}[\mathbf{U}_n^{L,*} - \mathbf{U}_s^{R,*}]$.

Before applying $\mathbf{L}_c^K$, friction effects are integrated separately using the same split implicit scheme described in Section 3.1.1.1 (Liang and Marche, 2009). However, in the DG2 scheme, the x- and y-directional slope coefficients of the discharge are also updated:

$$q_{x,c}^{1x,p+1} = \frac{1}{2}\left[q_{fx}(\mathbf{U}_c^{0,p} + \mathbf{U}_c^{1x,p}) - q_{fx}(\mathbf{U}_c^{0,p} - \mathbf{U}_c^{1x,p})\right], \tag{4.3a}$$

$$q_{x,c}^{1y,p+1} = \frac{1}{2}\left[q_{fx}(\mathbf{U}_c^{0,p} + \mathbf{U}_c^{1y,p}) - q_{fx}(\mathbf{U}_c^{0,p} - \mathbf{U}_c^{1y,p})\right], \tag{4.3b}$$

$$q_{y,c}^{1x,p+1} = \frac{1}{2}\left[q_{fy}(\mathbf{U}_c^{0,p} + \mathbf{U}_c^{1x,p}) - q_{fy}(\mathbf{U}_c^{0,p} - \mathbf{U}_c^{1x,p})\right], \tag{4.3c}$$

$$q_{y,c}^{1y,p+1} = \frac{1}{2}\left[q_{fy}(\mathbf{U}_c^{0,p} + \mathbf{U}_c^{1y,p}) - q_{fy}(\mathbf{U}_c^{0,p} - \mathbf{U}_c^{1y,p})\right]. \tag{4.3d}$$

4.1.2 MRA process of the MW and DG2 update on the GPU

The MRA process of the MW is implemented on the GPU by extending the use of the computational ingredients used to parallelise the MRA process of the HW in GPU-HWFV1 (see Section 3.1.2) – i.e. the Z-order curve (Section 3.1.2.1), the parallel tree traversal algorithm (Section 3.1.2.2), and the array data structure (Section 3.1.2.3). The MRA process of the MW is very similar to the MRA process of the HW and also involves a hierarchy of $2^n \times 2^n$ grids of increasingly coarser resolution relative to a reference $2^L \times 2^L$ grid at the highest refinement level. Figure 4.1a shows an example of a hierarchy of grids with maximum refinement level $L = 2$: the finest grid in the hierarchy is at refinement level $n = L = 2$, the second-finest grid is at refinement level $n = 1$, and the coarsest grid is at $n = 0$. Let $\mathbf{s}_c^{(n)}$ denote $\mathbf{s}_c$ at refinement level n: similar to GPU-HWFV1, GPU-MWDG2 stores all $\mathbf{s}_c^{(n)}$ for all refinement levels as an array in GPU memory (one for each of s_c^0, s_c^{1x}, and s_c^{1y}) indexed according to Z-order curves (see Section 3.1.2.1), as shown in Figure 4.1a.

At the start of the MRA process, only $\mathbf{s}_c^{(L)}$ are available, corresponding to the coefficients produced by GPU-MWDG2 from performing a projection over each cell on the reference $2^L \times 2^L$

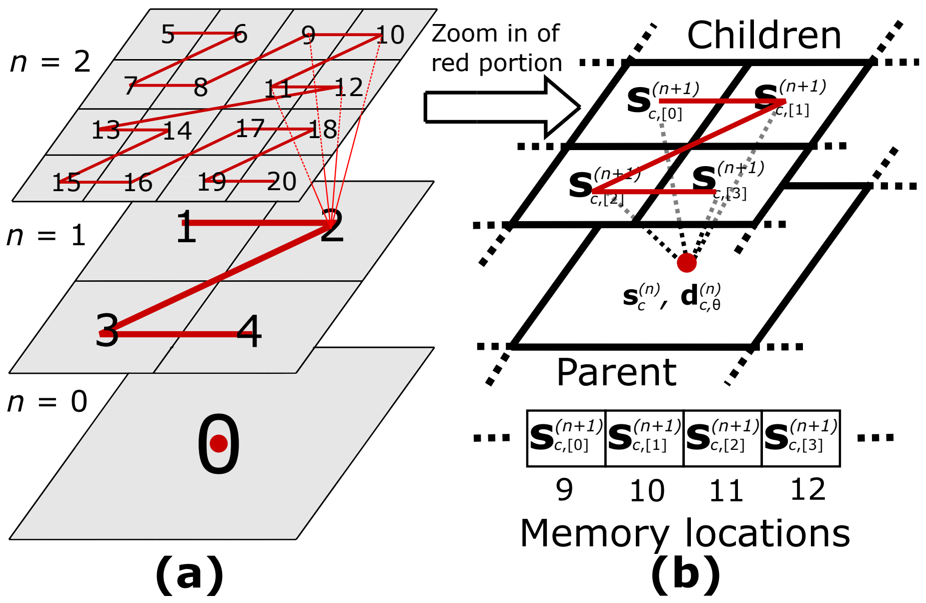


Figure 4.1: *Performing the MRA process of the MW on the GPU. The left panel shows a hierarchy of grids involved in the MRA process with a maximum refinement level $L = 2$ indexed according to Z-order curves. The right panel shows how four cells in a grid at refinement level $n + 1$, i.e. the "children," are positioned relative to a single cell in a grid at refinement level n, i.e. the "parent" – as well as the locations of the children in memory.*

grid (Kesserwani and Sharifian, 2020). The MRA process continues by performing the encoding procedure of the MW to produce $\mathbf{s}_c^{(n)}$ at the lower refinement levels, i.e. at $n = L-1, L-2, \ldots, 0$. In the encoding procedure of the MW, the vector $\mathbf{s}_c^{(n)}$ of a cell at refinement level n (i.e. the parent) is produced using the vectors $\mathbf{s}_{[0]}^{(n+1)}$, $\mathbf{s}_{[1]}^{(n+1)}$, $\mathbf{s}_{[2]}^{(n+1)}$, and $\mathbf{s}_{[3]}^{(n+1)}$ of four cells at refinement level $n+1$ (i.e. the children) by applying Equation 4.4a. Figure 4.1b shows the parent-children stencil that dictates how the children $\mathbf{s}_{[0]}^{(n+1)}$, $\mathbf{s}_{[1]}^{(n+1)}$, $\mathbf{s}_{[2]}^{(n+1)}$, and $\mathbf{s}_{[3]}^{(n+1)}$ at level $n+1$ are positioned relative to the parent $\mathbf{s}_c^{(n)}$ at level n. When applying Equation 4.4a, coalesced memory access occurs due to the indexing of the Z-order curves because the children reside in adjacent memory locations, like for GPU-HWFV1 (see Section 3.1.2.1).

$$\mathbf{s}_c^{(n)} = \mathbf{HH}^0\mathbf{s}_{[0]}^{(n+1)} + \mathbf{HH}^1\mathbf{s}_{[2]}^{(n+1)} + \mathbf{HH}^2\mathbf{s}_{[1]}^{(n+1)} + \mathbf{HH}^3\mathbf{s}_{[3]}^{(n+1)} \tag{4.4a}$$

$$\mathbf{d}_{c,\alpha}^{(n)} = \mathbf{GA}^0\mathbf{s}_{[0]}^{(n+1)} + \mathbf{GA}^1\mathbf{s}_{[2]}^{(n+1)} + \mathbf{GA}^2\mathbf{s}_{[1]}^{(n+1)} + \mathbf{GA}^3\mathbf{s}_{[3]}^{(n+1)} \tag{4.4b}$$

$$\mathbf{d}_{c,\beta}^{(n)} = \mathbf{GB}^0\mathbf{s}_{[0]}^{(n+1)} + \mathbf{GB}^1\mathbf{s}_{[2]}^{(n+1)} + \mathbf{GB}^2\mathbf{s}_{[1]}^{(n+1)} + \mathbf{GB}^3\mathbf{s}_{[3]}^{(n+1)} \tag{4.4c}$$

$$\mathbf{d}_{c,\gamma}^{(n)} = \mathbf{GC}^0\mathbf{s}_{[0]}^{(n+1)} + \mathbf{GC}^1\mathbf{s}_{[2]}^{(n+1)} + \mathbf{GC}^2\mathbf{s}_{[1]}^{(n+1)} + \mathbf{GC}^3\mathbf{s}_{[3]}^{(n+1)} \tag{4.4d}$$

Equations 4.4b - 4.4d are also applied (again in a coalesced manner, using the same reasoning as for the application of Equation 4.4a) to produce the details, denoted by $\mathbf{d}_{c,\Theta}^{(n)} = [d_{c,\Theta}^{(0,n)}, d_{c,\Theta}^{(1x,n)}, d_{c,\Theta}^{(1y,n)}]^{\mathrm{T}}$, $\Theta = \alpha, \beta, \gamma$. The details are used to decide which cells to include in the non-uniform grid based on a normalised detail $d_{c,norm}^{(n)} = \max(\mathbf{d}_{c,\Theta}^{(n)})/s_{\max}$, where $s_{\max}$ is the largest s_c^0 for all $\mathbf{s}_c^{(L)}$. Cells whose $d_{c,norm}^{(n)}$ is greater than $2^{n-L}\varepsilon$ are deemed to have significant details, where ε is the error threshold. In Equations 4.4a - 4.4d, $\mathbf{HH}^J$, $\mathbf{GA}^J$, $\mathbf{GB}^J$, and $\mathbf{GC}^J$, $J = 0, 1, 2, 3$, are filter bank matrices derived from the MW (Keinert, 2003), which have been included in the Appendix for brevity.

After the encoding procedure, the remaining steps of the MRA process are (1) decoding (see Algorithm 4), (2) parallel tree traversal (see Section 3.1.2.2), and (3) neighbour finding (see Section 3.1.2.2). These steps are implemented on the GPU in GPU-MWDG2 almost identically to GPU-HWFV1 except for three differences. The first difference is that unlike GPU-HWFV1, GPU-MWDG2 performs the decoding procedure by loading the parents and applying Equations 4.5a - 4.5d instead of Equations 3.11a - 3.11d, where coalesced memory access occurs due to Z-order indexing because the parents reside in adjacent memory locations. The second difference is that after completing the parallel tree traversal and generating a non-uniform grid, the cells in the non-uniform grid hold modelled data as shape vectors, namely $\mathbf{s}_c^{(n)}$ belonging to the leaf cells, instead of as scalars like the cells in GPU-HWFV1's non-uniform grid. The third difference is that after neighbour finding, GPU-MWDG2 performs the DG2 update differently to the FV1 update in GPU-HWFV1 (see Section 3.1.2.3). This is because the FV1 update is based on a forward-Euler scheme with a single time stage, whereas two time stages are involved in the RK2 scheme in the DG2 update. In particular, the first time stage of the RK2 scheme consists of GPU-MWDG2 advancing the data in the non-uniform grid from timestep p to the intermediate timestep int, while the second time stage consists of advancing the data from timestep int to the next full timestep $p + 1$. Recall however that the data of the non-uniform grid and the neighbours are stored as separate arrays in memory (see Section 3.1.2.3). Therefore, after GPU-MWDG2 completes the first time stage, the arrays holding the neighbour data become invalid because GPU-MWDG2 has only updated the arrays holding the non-uniform grid data to timestep int, while the neighbour arrays still hold data at timestep p. To update the neighbour arrays from timestep p to int, GPU-MWDG2 performs the encoding procedure again to update the data in the hierarchy of grids from timestep p to int. GPU-MWDG2 then re-retrieves the data from the hierarchy – which are now at timestep int – to update the neighbour arrays for completing the second time stage.

$$\mathbf{s}_{[0]}^{(n+1)} = [\mathbf{HH}^0]^{\mathrm{T}}\mathbf{s}_c^{(n)} + [\mathbf{GA}^0]^{\mathrm{T}}\mathbf{d}_{c,\alpha}^{(n)} + [\mathbf{GB}^0]^{\mathrm{T}}\mathbf{d}_{c,\beta}^{(n)} + [\mathbf{GC}^0]^{\mathrm{T}}\mathbf{d}_{c,\gamma}^{(n)} \tag{4.5a}$$

$$\mathbf{s}_{[2]}^{(n+1)} = [\mathbf{HH}^1]^{\mathrm{T}}\mathbf{s}_c^{(n)} + [\mathbf{GA}^1]^{\mathrm{T}}\mathbf{d}_{c,\alpha}^{(n)} + [\mathbf{GB}^1]^{\mathrm{T}}\mathbf{d}_{c,\beta}^{(n)} + [\mathbf{GC}^1]^{\mathrm{T}}\mathbf{d}_{c,\gamma}^{(n)} \tag{4.5b}$$

$$\mathbf{s}_{[1]}^{(n+1)} = [\mathbf{HH}^2]^{\mathrm{T}}\mathbf{s}_c^{(n)} + [\mathbf{GA}^2]^{\mathrm{T}}\mathbf{d}_{c,\alpha}^{(n)} + [\mathbf{GB}^2]^{\mathrm{T}}\mathbf{d}_{c,\beta}^{(n)} + [\mathbf{GC}^2]^{\mathrm{T}}\mathbf{d}_{c,\gamma}^{(n)} \tag{4.5c}$$

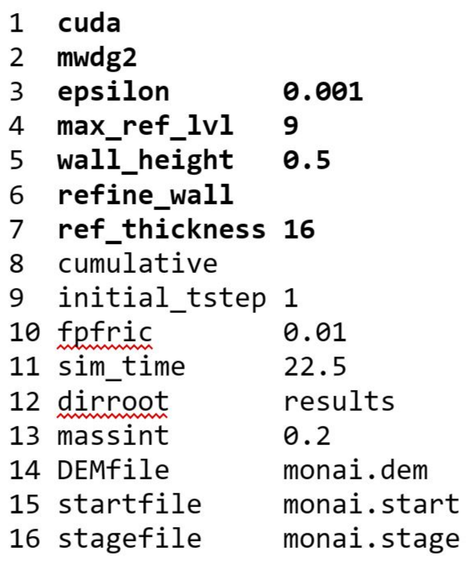
$$\mathbf{s}_{[3]}^{(n+1)} = [\mathbf{HH}^3]^{\mathrm{T}}\mathbf{s}_c^{(n)} + [\mathbf{GA}^3]^{\mathrm{T}}\mathbf{d}_{c,\alpha}^{(n)} + [\mathbf{GB}^3]^{\mathrm{T}}\mathbf{d}_{c,\beta}^{(n)} + [\mathbf{GC}^3]^{\mathrm{T}}\mathbf{d}_{c,\gamma}^{(n)} \tag{4.5d}$$

4.2 Integration of GPU-MWDG2 into LISFLOOD-FP 8.2

In this section, the integration of GPU-MWDG2 into a new version of LISFLOOD-FP – version 8.2 – is discussed. The previous version – LISFLOOD-FP 8.1 – only included static-in-time grid adaptation whereby the non-uniform grid is generated only once, according to the topographic details (Sharifian et al., 2023), whereas LISFLOOD-FP 8.2 includes dynamic-in-time grid adaptation whereby the non-uniform grid is generated every timestep according to both the topographic *and* flow details. Section 4.2.1 describes how to use GPU-MWDG2 within LISFLOOD-FP's existing framework for running real-world flood simulations. Section 4.2.2 discusses the largest allowable simulation area that can be considered using GPU-MWDG2. Section 4.2.3 describes new metrics for analysing the impact of GPU-MWDG2's grid adaptation capability on the computational effort of DG2 modelling.

4.2.1 Using GPU-MWDG2 within LISFLOOD-FP 8.2

The procedure for running a flood simulation of a real-world test case using any model in LISFLOOD-FP – e.g. the diffusive wave model (Hunter et al., 2005), the local inertial model (Bates et al., 2010), or the FV1 and DG2 models (Shaw et al., 2021) – requires the user to provide a number of input files to LISFLOOD-FP, such as a parameter file for specifying various model features (see Figure 4.2), and other files that specify various physical characteristics of the test case. For example, the user should provide "`.dem`" and "`.start`" files in ESRI raster grid format to specify the DEM and initial flow conditions of the test case, respectively, whereby the DEM file represents the bathymetry of the test case. The same procedure applies to GPU-MWDG2, but a few extra parameters must be specified in the parameter file to correctly use GPU-MWDG2's grid adaptation capability within LISFLOOD-FP's framework for running real-world flood simulations.



```
cuda
mwdg2
epsilon       0.001
max_ref_lvl   9
wall_height   0.5
refine_wall
ref_thickness 16
cumulative
initial_tstep 1
fpfric        0.01
sim_time      22.5
dirroot       results
massint       0.2
DEMfile       monai.dem
startfile     monai.start
stagefile     monai.stage
```

Figure 4.2: *Example of a parameter file showing the parameters needed to run a simulation of the Monai valley test case (Section 5.2) using GPU-MWDG2. The extra parameters needed to correctly use GPU-MWDG2's grid adaptation capability within LISFLOOD-FP's framework for running real-world flood simulations are highlighted in bold.*

To correctly use GPU-MWDG2's grid adaptation capability, seven extra parameters must be specified in the parameter file, which are highlighted in bold in Figure 4.2, and which are summarised in Table 4.1. Figure 4.2 is an example of a parameter file, namely the one used to run simulations of the Monai valley test case (investigated in Section 5.2) using GPU-MWDG2. This test case is consistently used throughout this chapter as an example. Table 4.1 describes the functionalities of each of the extra parameters needed to correctly use GPU-MWDG2's grid adaptation capability, which are used as follows: the `cuda` keyword should be typed in the parameter file to access the GPU-parallelised models in LISFLOOD-FP 8.2, e.g. the GPU-DG2 or GPU-MWDG2 models. The `mwdg2` keyword should be typed to choose GPU-MWDG2 out of all the other GPU-parallelised solvers in LISFLOOD-FP 8.2. The `epsilon` keyword followed by a numerical value specifies the error threshold ε which controls the amount of coarsening in GPU-MWDG2's non-uniform grid: the higher the value, the higher the amount of grid coarsening; recommended values for ε are discussed later in Chapter 5. The `max_ref_lvl` keyword followed by an integer specifies the maximum refinement level L, which should be specified to accommodate the dimensions of the test case

DEM. For example, consider the Monai valley test case (Section 5.2): recall that the MRA process in GPU-MWDG2 involves a hierarchy of grids with a square $2^L \times 2^L$ grid at the highest refinement level, which is the finest-resolution grid accessible to GPU-MWDG2. However, LISFLOOD-FP usually considers test cases that involve a DEM in a rectangular raster grid format, suppose made up of $N \times M$ cells, but a rectangular $N \times M$ test case area does not fit exactly into the $2^L \times 2^L$ area of the square grid. To overcome this problem, the user should specify `max_ref_lvl` to be the smallest value of L such that $2^L \geq \max(N, M)$ so that GPU-MWDG2 constructs an artificial $2^L \times 2^L$ grid that can accommodate the $N \times M$ test case area. For example, in the Monai valley test case, $N = 392$ and $M = 243$, so `max_ref_lvl` should be specified as 9 because this is the smallest value of L that accommodates the 293×342 test case area, i.e. $2^9 = 512 \geq \max(392, 243)$.

Figure 4.3 shows the initial non-uniform grid generated by GPU-MWDG2 for the same Monai valley example. Since the non-uniform grid is generated via the MRA process involving the artificial square grid – whose dimensions are larger than the DEM dimensions – two areas emerge in the non-uniform grid: (1) the actual test case area (see red label text in Figure 4.1) which includes the DEM data and the initial flow conditions, and (2) void areas (see red label text as well in Figure 4.1) where no DEM data are available and where no flow should occur. In area (1), GPU-MWDG2 initialises the data in the cells by using the values specified in the `.dem` and `.start` raster files. Meanwhile, in area (2), GPU-MWDG2 initialises the flow data to zero, while the bathymetry data are assigned the numerical value that follows the `wall_height` keyword specified by the user. The numerical value specified for this keyword must be sufficiently high such that a wall is generated between the test case area and the void areas (see the red lines in Figure 4.3) that prevents any water from leaving the test case area (e.g. by choosing a numerical value that is higher than the largest water surface elevation that may appear during the simulation). For the Monai valley test case, the `wall_height` keyword is specified as 0.5 to generate a wall that is high enough to prevent any water from leaving the test case area. The `refine_wall` keyword and the `ref_thickness` keyword – followed by an integer for the latter, typically between 16 and 64 – should also be typed in the parameter file to prevent GPU-MWDG2 from excessively coarsening the non-uniform grid around the walls so as to avoid unrealistic calculations (labeled with the curly braces in Figure 4.1). For the Monai valley test case, the `refine_wall` keyword is specified to trigger refinement around the wall, and `ref_thickness` is specified as 16 to trigger 16 cells at the highest refinement level

Table 4.1: *Functionalities of the extra parameters that must be specified in the parameter file to correctly use GPU-MWDG2's grid adaptation capability (highlighted in bold in Figure 4.2).*

Parameter	**Functionality**
`cuda`	Boolean keyword instructing LISFLOOD-FP 8.2 to use the GPU-parallelised models.
`mwdg2`	Boolean keyword instructing LISFLOOD-FP 8.2 to use GPU-MWDG2.
`epsilon`	Keyword followed by a decimal number. Specifies the error threshold ε used by GPU-MWDG2 to control the amount of coarsening in the non-uniform grid. A larger value allows for a higher amount of coarsening.
`max_ref_lvl`	Keyword followed by an integer. Specifies the maximum refinement level L used by GPU-MWDG2 in the MRA process. For a test case involving a DEM made up of $N \times M$ cells, the user should specify the value of L to be the smallest integer such that $2^L \geq \max(N, M)$.
`wall_height`	Keyword followed by a decimal number. Specifies the height of the wall in metres used by GPU-MWDG2 to fill the void areas beyond the test case area. The user must specify a sufficiently high wall height to prevent any flow from exiting past the walls.
`refine_wall`	Boolean keyword to prevent GPU-MWDG2 from excessively coarsening the non-uniform grid around the wall separating the test case area from the void areas. The user can enable this refinement to ensure that very coarse cells around the wall do not lead to unrealistic solver computations.
`ref_thickness`	Keyword followed by an integer. Specifies the number of cells at the highest refinement level that are to be refined between the wall and the test case area by GPU-MWDG2 when the `refine_wall` keyword is also specified.
`cumulative`	Boolean keyword instructing GPU-MWDG2 to produce a file called `.cumu` at the end of the simulation that contains time series data designed to help analyse the impact of GPU-MWDG2's grid adaptation capability on the computational effort of DG2 modelling during a simulation.

between the wall and the test case, which is enough refinement at the wall to prevent unrealistic calculations.

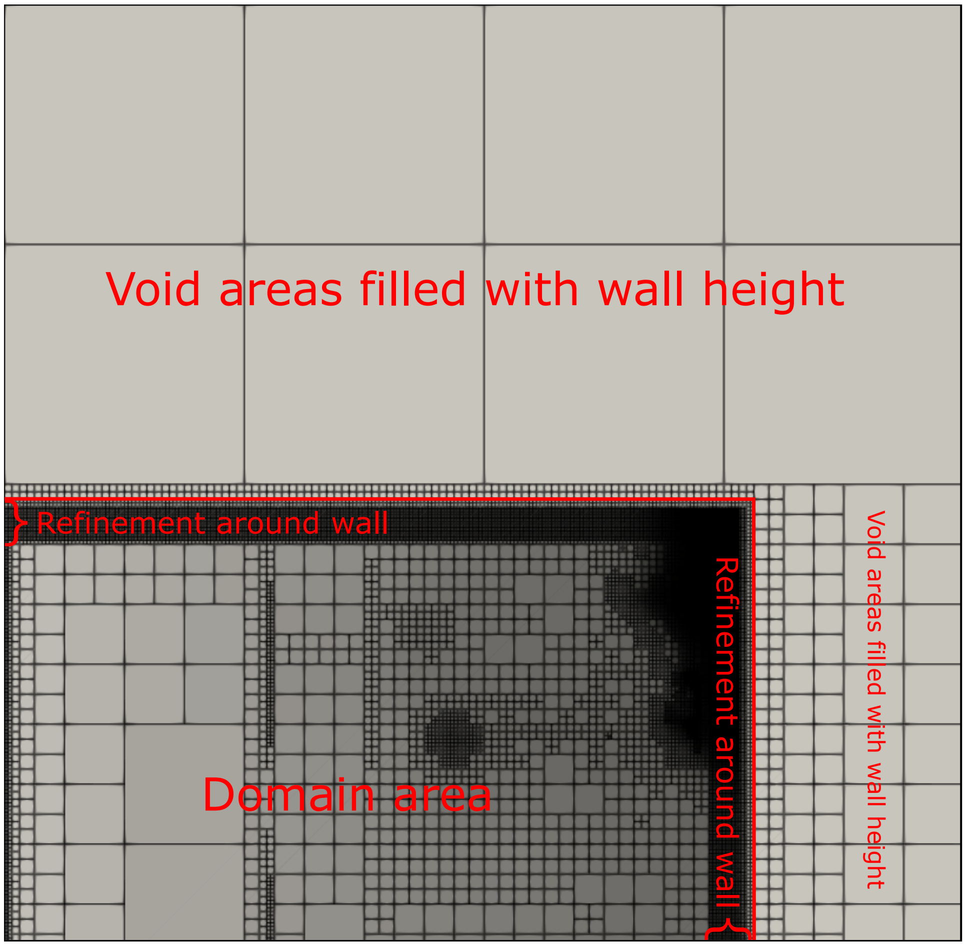

Void areas filled with wall height
Refinement around wall
Refinement around wall
Void areas filled with wall height
Domain area

Figure 4.3: *Initial non-uniform grid generated by GPU-MWDG2 based on the details of the bathymetry and initial flow conditions of the Monai valley test case (Section 5.2).*

4.2.2 Largest allowable test case area due to GPU memory consumption

When running a simulation of a test case using GPU-MWDG2, there is an upper limit on the test case area depending on the memory capacity of the GPU being used due to the large amount of GPU memory consumed by GPU-MWDG2. GPU-MWDG2 consumes a large amount of memory because it stores all the objects involved in the GPU-MWDG2 code – particularly the non-uniform grid, the neighbours, and the hierarchy of grids – in GPU memory. The percentage breakdown of the memory consumed by each of these objects is shown in the left panel of Figure 4.4. In theory, the percentage breakdown should depend on the amount of grid coarsening: the higher the amount of coarsening, the lower the cell count of the non-uniform grid, and the shorter the length of array data structure needed to store the non-uniform grid and the neighbours (see Section 3.1.2.3) – and thus, the lower the amount of memory consumed by GPU-MWDG2. However, the GPU-MWDG2 code has been designed to allocate arrays of fixed length assuming the worst-case scenario where there is no grid coarsening to guarantee that the arrays can store the non-uniform grid and the neighbours in GPU memory – regardless of the amount of grid coarsening. For example, as seen in the left panel of Figure 4.4, the non-uniform grid and the neighbours that hold the flow and topographic information consume nearly 80% of the memory allocated by GPU-MWDG2 – regardless of the amount of coarsening. Allocating arrays of fixed length this way also prevents GPU-MWDG2 from having to repeatedly reallocate memory in response to the amount of coarsening, which is beneficial because memory allocation is slow.

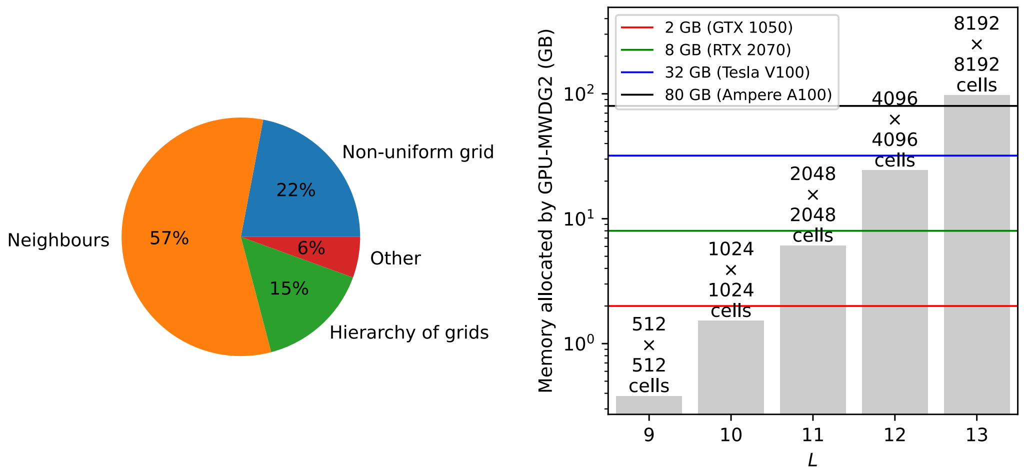


Figure 4.4: *GPU memory consumed by GPU-MWDG2. The left panel shows the percentage breakdown of the memory consumed by the objects in GPU-MWDG2. The right panel shows the amount of GPU memory allocated by GPU-MWDG2 against the maximum refinement level L. The numbers on top of the bars show the largest test case area allowed for a given value of L. The horizontal lines indicate the memory capacities of four GPUs.*

Since the GPU-MWDG2 code allocates arrays of fixed length regardless of the amount of coarsening, the amount of memory consumed by GPU-MWDG2 depends only on the maximum refinement level L. The right panel of Figure 4.4 shows the amount of memory consumed by GPU-MWDG2 with respect to L, where the numbers on top of the bars show the largest test area allowed for a given value of L. The horizontal lines indicate the memory capacities of four GPUs. As seen in the right panel of Figure 4.4, the higher the value of L, the more memory GPU-MWDG2 consumes: at $L = 9$, GPU-MWDG2 consumes only 0.2 GB of memory, but by $L = 13$, GPU-MWDG2 consumes around 100 GB of memory.

GPU-MWDG2 cannot allocate more memory than the memory capacity of a given GPU, which restricts the highest allowable value for L using that GPU, and thus places an upper limit on the test case area that can be considered using that GPU. For example, as seen in Figure 4.4, the GTX 1050 GPU has a memory capacity of 2 GB (red line), and values of L greater than 10 exceed the 2 GB memory capacity of this GPU. Thus, if using this GPU, GPU-MWDG2 cannot use a value of L larger than 10, so the largest test case area that can be considered using this GPU is $210 \times 2^{10} = 1024 \times 1024$ cells. Therefore, as of writing, GPU-MWDG2 cannot use a value of L larger than 12 – meaning the largest test case area that can be considered using GPU-MWDG2 is $2^{12} \times 2^{12} = 4096 \times 4096$ cells – as doing so requires more than 80 GB of GPU memory, which is higher than the memory capacity of most commercially available GPUs. This limitation on the largest allowable test case area is problematic for many real-world test cases as they often require a computational grid size bigger than 4096×4096 cells.

4.2.3 New metrics for analysing GPU-MWDG2's grid adaptation capability

Analysing the impact of GPU-MWDG2's grid adaptation capability on reducing the computational effort of DG2 modelling during a simulation is essential for a user. As noted in other works that have explored wavelet adaptivity, the computational effort and speedup of a GPU-MWDG2 simulation should ideally be exactly correlated with the number of cells in the GPU-MWDG2 non-uniform grid (since the number of cells dictates the number of DG2 solver updates to be performed). However, in practice, this rarely occurs as the ideal speedup is diminished by the additional computational effort spent by GPU-MWDG2 to generate the non-uniform grid every timestep via the MRA process (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020; Kesserwani and Sharifian, 2023; Chowdhury et al., 2023). Thus, to thoroughly assess the potential speedup of a GPU-MWDG2 simulation, the user must consider the interdependent effects of the number of cells in the non-uniform grid, the computational effort of performing the DG2 solver updates, and the computational effort of performing the MRA process.

To this end, starting from LISFLOOD-FP 8.2, the user can include the `cumulative` keyword in the parameter file to produce a "`.cumu`" file that contains the time histories of several quantities for analysing the speedup achieved by GPU-MWDG2's grid adaptation capability, which are described in Table 4.2. These time histories can be postprocessed into several time-dependent metrics for analysing the speedups of GPU-MWDG2 simulations compared to GPU-DG2 simulations, which are described in Table 4.3, and their use in analysing the speedup of a GPU-MWDG2 simulation compared to a GPU-DG2 simulation is explained next by way of an example.

Table 4.2: *Description of the quantities in the* `.cumu` *output file. GPU-MWDG2 produces this file at the end of a simulation if the* `cumulative` *keyword has been typed in the parameter file.*

Heading in file	**Description of time series**
`simtime`	Simulation time in seconds.
`num_timesteps`	Number of timesteps taken by GPU-MWDG2 to reach a given simulation time.
`dt`	Timestep size at a given simulation time.
`num_cells`	Number of cells in GPU-MWDG2's non-uniform grid at a given simulation time.
`inst_time_solver`	Computational effort (measured in seconds) spent by GPU-MWDG2 to perform the DG2 solver update at a given timestep.
`cumu_time_solver`	Cumulative computational effort (measured in seconds) spent by GPU-MWDG2 in performing the DG2 solver updates up to a given simulation time.
`inst_time_mra`	Computational effort (measured in seconds) spent by GPU-MWDG2 to perform the MRA process to generate the non-uniform grid at a given timestep.
`cumu_time_mra`	Cumulative computational effort (measured in seconds) spent by GPU-MWDG2 in performing the MRA process to generate the non-uniform grid up to a given simulation time.
`runtime_total`	Sum of the `cumu_time_mra` and the `cumu_time_solver` time histories; total computational effort (measured in seconds) spent by GPU-MWDG2 to reach a given simulation time.

In a GPU-MWDG2 simulation of an impact event, the computational effort per timestep changes depending on the change in the number of cells in the GPU-MWDG2 non-uniform grid. The number of cells changes over time because finer cells are generated by GPU-MWDG2 to track the flow features produced by the impact event as it enters and travels through the bathymetric area. Using the same Monai Valley example (Section 5.2), the left panel of Figure 4.5 shows the initial non-uniform grid generated by GPU-MWDG2 at the start of the simulation, while the right panel shows an intermediate non-uniform grid generated by GPU-MWDG2 after the simulation has progressed

Table 4.3: *Time-dependent metrics for evaluating the potential speedup afforded by GPU-MWDG2's grid adaptation capability.*

Metric	Description
$N_{\text{cells}}(t)$	Number of cells in GPU-MWDG2's non-uniform grid compared to GPU-DG2's grid (as a percentage) against simulation time.
$R_{\text{DG2}}(t)$	Computational effort spent by GPU-MWDG2 to perform the DG2 solver updates at a given timestep (relative to GPU-DG2 as a percentage) against simulation time.
$R_{\text{MRA}}(t)$	Computational effort spent by GPU-MWDG2 to perform the MRA process and generate the non-uniform grid at a given timestep (relative to GPU-DG2 as a percentage) against simulation time.
$S_{\text{inst}}(t)$	Instantaneous speedup achieved by GPU-MWDG2 over GPU-DG2 at a given timestep against simulation time.
$N_{\Delta t}(t)$	Number of timesteps taken by GPU-MWDG2 to reach a given simulation time.
$C_{\text{DG2}}(t)$	Cumulative computational effort spent by GPU-MWDG2 to perform the DG2 solver updates (quantified in units of wall clock time) up to a given simulation time.
$C_{\text{MRA}}(t)$	Cumulative computational effort spent by GPU-MWDG2 to perform the MRA process (quantified in units of wall clock time) up to a given simulation time.
$C_{\text{tot}}(t)$	Total cumulative computational effort spent by GPU-MWDG2 to complete a simulation (quantified in units of wall clock time) up to a given simulation time.
$S_{\text{acc}}(t)$	Accumulated speedup of GPU-MWDG2 over GPU-DG2 up to a given simulation time.

by 17 s, i.e. after an impact event, here a tsunami, has entered and propagated through the bathymetric area. At the start of the simulation, the initial non-uniform grid is coarsened as much as allowed, based only on the static features of the bathymetric area and initial flow conditions, leading to a minimal number of cells in the grid, which is quantified by N_{cells}. The number of cells determines the number of DG2 solver updates to be performed at a given timestep, leading to a corresponding computational effort per timestep, which is quantified by R_{DG2}. There is also the computational effort of performing the MRA process at a given timestep, which is quantified by R_{MRA}. Based on the combined computational effort of performing both the MRA process and the DG2 solver updates at a given timestep, the instantaneous speedup in completing one timestep of a GPU-MWDG2 simulation can be computed (relative to the GPU-DG2 simulation), which is quantified by S_{inst}. In Figure 4.5, after the simulation has progressed by 17 s, the number of cells in the non-uniform grid has increased due to using finer cells to track the tsunami's wavefronts and wave diffractions, which leads to a higher value of N_{cells} and R_{DG2} (and possibly also to a higher value of R_{MRA}, as a higher number of cells in the non-uniform grid means more cells must be processed during the MRA process); thus, S_{inst} is expected to drop. Generally, the higher the complexity of the impact event, the higher the number of cells in the GPU-MWDG2 non-uniform grid, and the lower the potential speedup.

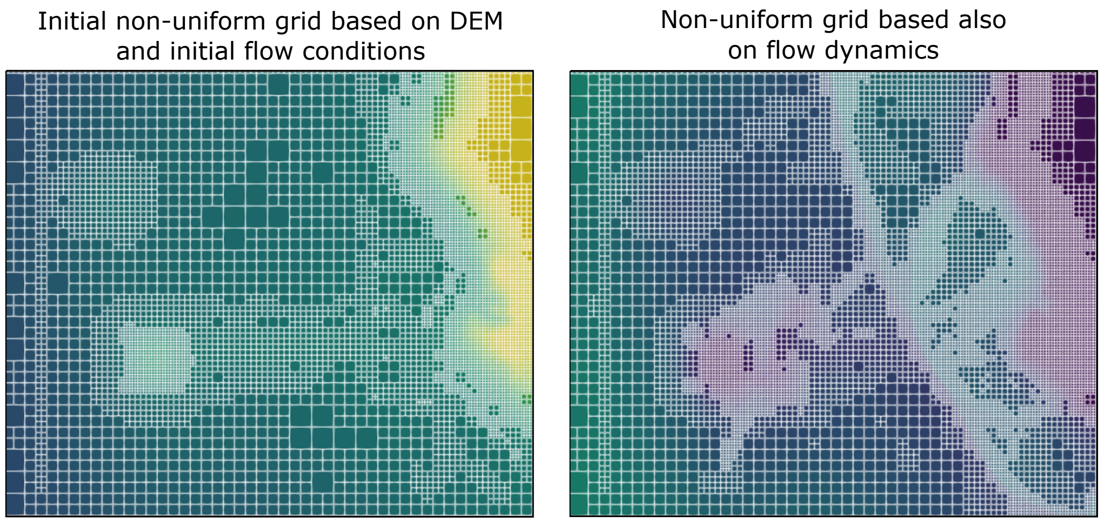

Initial non-uniform grid based on DEM and initial flow conditions
Non-uniform grid based also on flow dynamics

Figure 4.5: *Non-uniform grids generated by GPU-MWDG2 for the Monai valley test case (Section 5.2). The left panel shows the initial grid generated by GPU-MWDG2 at the start of the simulation. The right panel shows the intermediate grid generated by GPU-MWDG2 after the simulation has progressed by 17 s, which has a lower amount of coarsening than the initial grid due to flow features produced by the tsunami as it enters and travels through the test case area.*

The metrics N_{cells}, R_{DG2}, R_{MRA}, and S_{inst} quantify the computational effort and speedup of a GPU-MWDG2 simulation per timestep compared to a GPU-DG2 simulation. However, the overall or *cumulative* computational effort and speedup of a GPU-MWDG2 simulation depends on accumulating the computational effort and speedup per timestep from *all* the timesteps taken by GPU-MWDG2 to reach a given simulation time. The higher the number of timesteps taken by GPU-MWDG2 to reach a given simulation time (quantified by $N_{\Delta t}$), the higher the cumulative computational effort spent by GPU-MWDG2 to reach that simulation time (quantified by C_{tot}). The C_{tot} metric is computed by summing the cumulative computational effort spent by GPU-MWDG2 to perform the DG2 solver updates and the MRA process, which are quantified by C_{DG2} and C_{MRA}, respectively. Using the cumulative metrics, the overall speedup accumulated by a GPU-MWDG2 simulation can be computed, which is quantified by S_{acc}. The metrics in Table 4.3

are extensively used in the next chapter (Chapter 5) to assess GPU-MWDG2's performance in simulating several test cases of tsunami-inducing flooding.

4.3 Summary

This chapter addressed Objective 2, in which the use of the computational ingredients proposed in Chapter 3 for efficiently parallelising the MRA process of the HW on the GPU – i.e. the Z-order curve, the parallel tree traversal algorithm, and the array data structure – was extended to the MRA process of the MW, thereby allowing to implement a GPU-parallelised MW adaptive DG2 model, i.e. GPU-MWDG2. GPU-MWDG2 was then integrated into a new version of LISFLOOD-FP – version 8.2 – in order to take advantage of LISFLOOD-FP's existing framework for running real-world flood simulations and thereby allow for a proper comparison of GPU-MWDG2 against GPU-DG2, as the latter is already integrated into LISFLOOD-FP.

The MRA process of the MW is very similar to the MRA process of the HW and was thus implemented on the GPU in almost exactly the same way as for GPU-HWFV1, with some key differences. First, in the MRA process of the MW, the encoding and decoding steps involve computing "shape vectors" (instead of scalars like in MRA process of the HW in GPU-HWFV1), and at the end of the MRA process, a non-uniform grid is generated whose cells hold modelled data as shape vectors instead of scalars. These shape vectors – each made up of three scalar components – are needed for compatibility with the DG2 scheme in GPU-MWDG2, in which the solution over each cell in the grid is piecewise-planar (instead of piecewise-constant like in the FV1 scheme in GPU-HWFV1). The shape of the planar solution over each cell is controlled by the three scalar components of each shape vector per cell, which are an average coefficient and two x- and y-directional slope coefficients. The three components are updated in time via a "DG2 update," which involves an RK2 scheme with two time stages (instead of only one time stage as in the forward-Euler scheme in the FV1 update of GPU-HWFV1). Due to the involvement of two time stages, the DG2 update was implemented on the GPU in a slightly different way compared to the FV1 update: namely, after completing the first time stage, the encoding step is performed again to update the modelled data of the neighbours to an intermediate timestep, after which the second time stage can be completed.

When using GPU-MWDG2 within LISFLOOD-FP's framework for running real-world flood simulations, seven extra keywords must be specified in the parameter file to correctly use GPU-MWDG2's grid adaptation capability. Notably, to run a simulation of a real-world test case involving a DEM with a raster grid made up of $N \times M$ cells using GPU-MWDG2, the maximum refinement level L should be specified to be the smallest integer such that $2^L \geq \max(N, M)$. This should be done so that GPU-MWDG2 constructs an artificial $2^L \times 2^L$ grid – which is the size of the reference uniform grid at the highest refinement level in the hierarchy of grids in the MRA process – that can accommodate the $N \times M$ raster grid of the DEM. Thus, the larger the DEM raster grid, the higher the value of L that must be specified to accommodate the DEM.

The higher the value of L, the higher the amount of memory consumed by GPU-MWDG2; however, GPU-MWDG2 cannot consume more memory than the memory capacity of a given GPU, so the memory capacity of a GPU places an upper limit on the value of L and thus the maximum test case area that can be considered using GPU-MWDG2 with that GPU. As of writing, GPU-MWDG2 cannot be used with L larger than 12 – equivalent to a maximum test case area of 4096×4096 cells – because doing so requires more than 80 GB of GPU memory, which is higher than the memory capacity of most commercially available GPUs.

GPU-MWDG2 can be instructed to produce a `.cumu` file at the end of a simulation that contains several categories of diagnostic time series data. These time series data allow to compute a set of time-dependent metrics for analysing the impact of GPU-MWDG2's grid adaptation capability on reducing the computational effort of DG2 modeling during a simulation. The metrics are used extensively in the next chapter (Chapter 5) to assess GPU-MWDG2's performance when simulating test cases that feature multiscale rapid flow phenomena.

Chapter 5

Comparing GPU-MWDG2 against GPU-DG2

This chapter addresses Objective 3 and compares the adaptive GPU-MWDG2 model against the uniform-grid GPU-DG2 model to analyse the impact of GPU-MWDG2's grid adaptation capability on the accuracy and computational effort of DG2 modelling when simulating test cases that feature multiscale rapid flow phenomena, particularly tsunami-induced flooding – where grid resolution adaptation is hypothesised to reduce the computational effort of DG2 modelling while still allowing to achieve a similar level of accuracy as GPU-DG2.

This chapter is organised as follows. Section 5.1 selects a series of test cases over which the potential benefit of using GPU-MWDG2's grid adaptation capability when simulating multiscale rapid flow phenomena is quantitatively analysed. Supported by the metrics proposed in Table 4.3 of Chapter 4, Sections 5.2 to 5.5 include results and discussions about the accuracy and computational effort of DG2 modelling obtained from running simulations of the test cases selected in Section 5.1 using GPU-MWDG2 and GPU-DG2. Section 5.6 provides a summary of the key results and concludes with recommendations on when GPU-MWDG2's grid adaptation capability makes it a potentially better choice than GPU-DG2 for simulating multiscale rapid flow phenomena.

5.1 Selection of tsunami test cases

Fundamentally, the impact of GPU-MWDG2's grid adaptation capability on reducing the computational effort of DG2 modelling – and thus GPU-MWDG2's potential speedup over GPU-DG2 during a simulation – depends on the complexity of the DEM and the flow features involved in a test case: the lower the complexity of the DEM and the flow features, the more grid resolution coarsening achieved by GPU-MWDG2's grid adaptation capability and the lower the number of cells in GPU-MWDG2's non-uniform grid. The lower the cell count of the non-uniform grid, the fewer DG2 updates that must be performed and the lower the computational effort – and thus, the higher the speedup of GPU-MWDG2 over GPU-DG2.

In addition to the complexity of the tsunami involved in a test case, the speedup achieved by GPU-MWDG2's adaptation capability may also depend on the DEM size. Namely, recall that when running GPU-MWDG2, the maximum refinement level L must be specified such that GPU-MWDG2's $2^L \times 2^L$ square grid can accommodate the size of the DEM raster grid (see Section 4.2.1): as the DEM size increases, so does the value of L required to accommodate the DEM raster grid. At the same time, the speedup achieved by wavelet-based grid adaptation when simulating test cases that involve mainly rapid flows was shown in Section 3.2.2 to scale with the value of L (see Figure 3.13, right panel), and in Section 3.2.3 to require $L \geq 9$ for realistic test cases involving complex DEMs (see Figure 3.20, right panel): thus, as the DEM size increases, so does the value of L needed to accommodate the DEM raster grid, and so does the potential for GPU-MWDG2 to achieve a speedup – hypothetically for $L \geq 9$ in realistic test cases involving rapid flows over complex DEMs.

Therefore, to systematically assess the effect of the DEM size and the tsunami complexity on the speedup achieved by GPU-MWDG2's grid adaptation capability, GPU-MWDG2 is compared against GPU-DG2 by running simulations of four tsunami test cases that each have a unique combination of DEM size (requiring either $L = 10$ or 12) and tsunami complexity (either simple or complex), as shown in Table 5.1. Simulations run using GPU-DG2 are always performed on a uniform grid at the finest resolution available to GPU-MWDG2 based on the DEM size. All simulations are run on an A100 GPU with 80 GB memory on the Stanage high performance computing cluster at the University of Sheffield in order to have enough GPU memory to accommodate the test case DEMs, which require specifying L up to 12 (see Section 4.2.1).

Table 5.1: *Summary of four tsunami test cases each with a unique combination of DEM size and tsunami complexity. Parameters needed to run the simulations per test case: maximum refinement level L, simulation end time, denoted by t_{end}, and Manning's friction coefficient, denoted by n_M.*

Test case	**DEM size**	L	**Tsunami complexity**	$t_{\mathbf{end}}$	$n_{\mathbf{M}}$
Monai valley (Section 5.2)	784×486 cells	10	A simple tsunami featuring a single peak and trough.	22.5 s	0.01
Seaside, Oregon (Section 5.3)	2181×1091 cells	12	A simple tsunami featuring a single peak and no trough.	40 s	0.025
Tauranga harbour (Section 5.4)	4096×2196 cells	12	A complex tsunami involving three peaks and troughs.	40 hr	0.025
Hilo harbour (Section 5.5)	701×692 cells	10	A complex tsunami with many peaks and troughs.	6 hr	0.025

When running GPU-MWDG2, consideration must also be given to the error threshold ε, which controls the amount of coarsening in GPU-MWDG2's non-uniform grid: a larger ε leads to more grid coarsening – and thus a higher potential speedup – but with the disadvantage of a lower level of DG2 accuracy compared to GPU-DG2. A higher level of accuracy can be obtained by using a smaller ε – but with the disadvantage of less grid coarsening and thus a lower potential speedup. As long as an error threshold $\varepsilon \leq 10^{-3}$ is used, GPU-MWDG2 preserves a similar level of DG2 accuracy as GPU-DG2 – in the sense that for this range of ε, GPU-MWDG2 gives flow predictions that are close to those of GPU-DG2 (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020). Therefore, two values of ε are explored: $\varepsilon = 10^{-3}$, as this is the maximum value recommended for flood modelling (Kesserwani et al., 2019; Kesserwani and Sharifian, 2020; Sharifian et al., 2023), and $\varepsilon = 10^{-4}$, as this smaller value is recommended for tsunami modelling where triggering a finer grid resolution is better for capturing smaller-scale features of dispersive tsunami waves (Sharifian et al., 2019).

The closeness of the GPU-MWDG2 and GPU-DG2 predictions obtained for each test case are shown visually, and are also quantified by computing the root mean squared error (RMSE) and correlation coefficient (r) of the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions:

$$\mathrm{RMSE} = \frac{\sqrt{\sum_i \left(P_{i,\mathrm{MW}} - P_{i,\mathrm{DG2}}\right)^2}}{T} \tag{5.1a}$$

$$r = \frac{\sum_i \left(P_{i,\mathrm{MW}} - \overline{P}_{\mathrm{MW}}\right)\left(P_{i,\mathrm{DG2}} - \overline{P}_{\mathrm{DG2}}\right)}{\sqrt{\sum_i \left(P_{i,\mathrm{MW}} - \overline{P}_{\mathrm{MW}}\right)^2 / T \sum_i \left(P_{i,\mathrm{DG2}} - \overline{P}_{\mathrm{DG2}}\right)^2 / T}} \tag{5.1b}$$

where T is the number of points in the prediction dataset, $P_{i,MW}$ is the i^{th} GPU-MWDG2 prediction, $P_{i,DG2}$ is the i^{th} GPU-DG2 prediction, $\overline{P}_{MW}$ is the mean of the GPU-MWDG2 predictions, and $\overline{P}_{DG2}$ is the mean of the GPU-DG2 predictions. The RMSE quantifies the prediction error between the GPU-MWDG2 and GPU-DG2 predictions, whereas r quantifies the similarity between the two datasets, which, along with the visual plots, together allow to verify the GPU-MWDG2's accuracy more holistically. The lower the value of the RMSE and the closer the value of r to 1, the closer the GPU-MWDG2 predictions are to the GPU-DG2 predictions.

5.2 Monai valley

This test case was used to validate many hydrodynamic solvers (Caviedes-Voullième et al., 2020; Kesserwani & Liang, 2012; Kesserwani & Sharifian, 2020; Matsuyama & Tanaka, 2001). It involves a 1:400 scaled replica of the 1993 tsunami that flooded Okushiri Island after a wave runup of 30 m at the tip of a very narrow gulley in a small cove at Monai Valley (Liu et al., 2008). The scaled DEM has 784×486 cells for which its associated initial square uniform grid is generated with $L = 10$. In Figure 5.1, a top-down view of the bathymetric area is shown (top left panel), which has a small island in the middle and a coastal shoreline to the right, including Okushiri Island and Monai Valley. The coastal shoreline gets flooded by a tsunami that initially enters the bathymetric area from the left boundary and then travels to the right by 4.5 m (indicated by the red arrows), interacting with the small island as it travels through the bathymetric area. This tsunami is simulated for 22.5 s, during which it travels through the bathymetric area in three stages of flow over time: the entry stage (0 to 7 s), the travelling stage (7 to 17 s) and the flooding stage (17 to 22 s). During the entry stage, the tsunami does not enter the bathymetric area, as seen in the hydrograph of the tsunami's water surface elevation (bottom left panel of Figure 5.1). During the travelling stage, the tsunami enters from the left boundary and travels right towards the coastal shoreline. Lastly, during the flooding stage, the tsunami floods the coastal shoreline, due to which many flow dynamics such as wave reflections and diffractions are produced that must be tracked using finer cells, thus increasing the number of cells in the GPU-MWDG2 non-uniform grid. In the right panels of Figure 6, the initial GPU-MWDG2 grids at $\varepsilon = 10^{-3}$ and 10^{-4} are depicted for the portion of the bathymetric area framed by the white box (top left panel of Figure 6). With $\varepsilon = 10^{-3}$ and 10^{-4}, the initial GPU-MWDG2 grid has 6% and 12% of the number of cells as in the GPU-DG2 uniform grid, respectively.

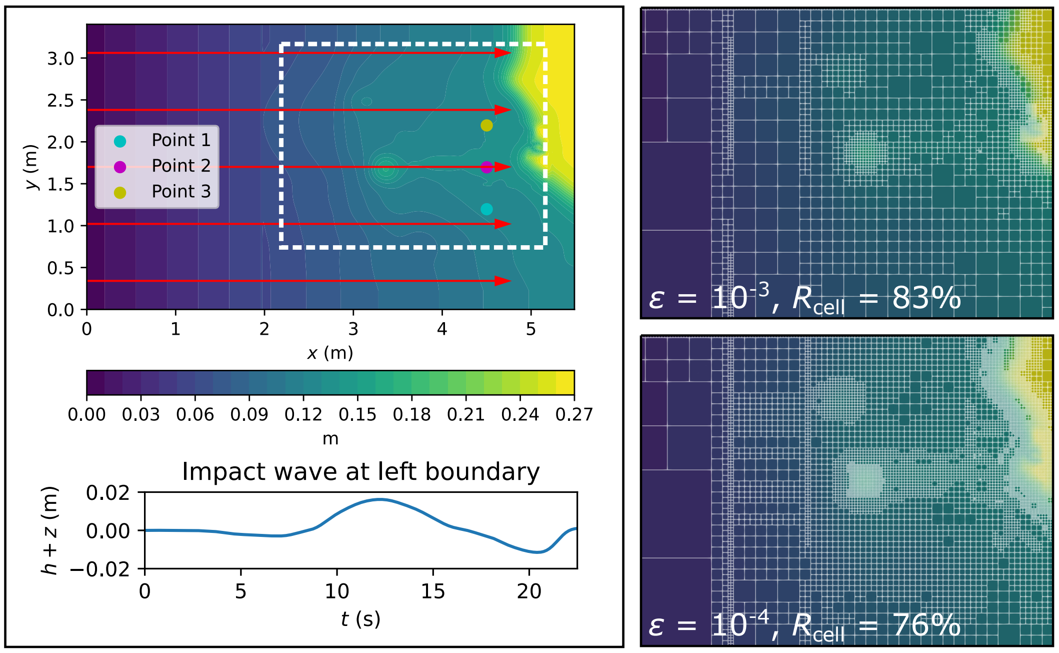


Figure 5.1: *Monai valley test case. Top-down view of bathymetry (top left panel), where the red arrows indicate the direction and distance travelled by the tsunami entering from the left boundary; time history of the free surface elevation of the tsunami at the left boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels).*

In Figure 5.2, an analysis of the runtimes of the GPU-DG2 and GPU-MWDG2 simulations using the time-dependent metrics of Table 4.3 is shown; a time history of Δt is also included. The dashed lines in the top left panel indicate the initial value of N_{cells}, i.e. N_{cells} of the initial GPU-MWDG2 non-uniform grids. Up to 15 s, i.e. before the flooding stage of flow begins, the time history of N_{cells} remains flat, meaning that the number of cells in the GPU-MWDG2 non-uniform grid does not change over time. Once the flooding stage begins, however, N_{cells} increases slightly, particularly at $\varepsilon = 10^{-4}$. With an increased number of cells in the non-uniform grid, the computational effort of performing the DG2 solver updates per timestep should increase, which is confirmed by the time history of R_{DG2}, which is flat before the flooding stage of flow, but thereafter increases, particularly at $\varepsilon = 10^{-4}$. Unlike R_{DG2} though, the time history of R_{MRA} is similar for both values of ε, and stays flat for most of the simulation except for an initial decrease at the start, meaning that the computational effort of performing the MRA process per timestep is similar for both values of ε and remains fixed throughout the simulation. Thus, the drop in the speedup of completing a single timestep of the GPU-MWDG2 simulation compared to the GPU-DG2 simulation is mostly due to the increase in R_{DG2} at $\varepsilon = 10^{-4}$, with S_{inst} dropping from 2.0 to 1.8 (which otherwise stays flat at 2.7 for $\varepsilon = 10^{-3}$).

Besides analysing the computational effort and speedups of the GPU-MWDG2 simulations per timestep, there is also the question of analysing the cumulative computational effort of running the simulations, which depends on the timestep size (Δt) and the number of timesteps taken to reach a given simulation time ($N_{\Delta t}$). In this test case, the time histories of Δt of the GPU-DG2 simulation and the GPU-MWDG2 simulation using $\varepsilon = 10^{-4}$ are very similar, but with $\varepsilon = 10^{-3}$, Δt drops at 7 s, i.e. as soon as the travelling stage of flow begins. This slight drop in Δt is likely dictated by wetting and drying on the cells associated with more frequent and aggressive grid resolution coarsening at $\varepsilon = 10^{-3}$ than 10^{-4}. Due to the smaller Δt at $\varepsilon = 10^{-3}$, the trend in $N_{\Delta t}$ is steeper at $\varepsilon = 10^{-3}$ than 10^{-4}, i.e. more timesteps are taken and thus more computational effort is spent by GPU-MWDG2 to reach a given simulation time at $\varepsilon = 10^{-3}$ than 10^{-4}. Still, despite the steeper trend in $N_{\Delta t}$, the cumulative computational effort of performing the MRA process is similar using both $\varepsilon = 10^{-3}$ and 10^{-4}, which is expected given that R_{MRA} is also very similar for values of ε. In contrast, the cumulative computational effort of performing the DG2 solver updates is considerably higher at $\varepsilon = 10^{-4}$ than 10^{-3}, likely due to the higher R_{DG2} at $\varepsilon = 10^{-4}$, which seems correct since a higher computational effort to perform the DG2 solver updates per timestep should lead to a higher cumulative computational effort (assuming that the time histories of $N_{\Delta t}$ are similar for the different values of ε, which is the case here). Overall, for both values of ε, the total computational effort of running the GPU-MWDG2 simulations is always lower than that of the GPU-DG2 simulation, with C_{tot} always being lower than that of GPU-DG2 at both $\varepsilon = 10^{-3}$ and 10^{-4}. The accumulated speedups of the GPU-MWDG2 simulations, S_{acc}, finish at around 2.5 and 2.0 using $\varepsilon = 10^{-3}$ and 10^{-4}, respectively.

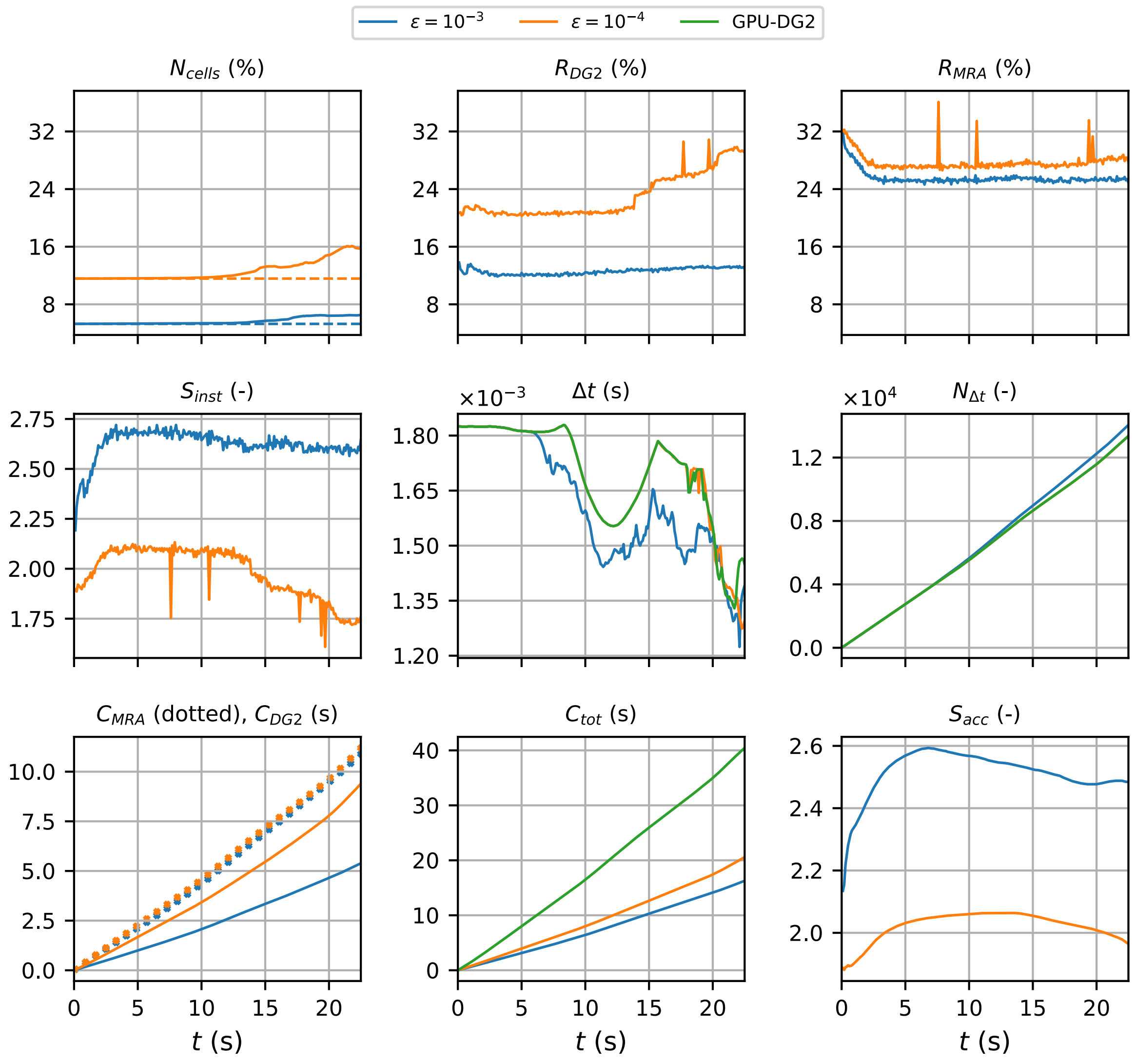


Figure 5.2: *Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Monai valley test case.*

Figure 5.3 shows the water surface elevation time series predicted by GPU-DG2 and GPU-MWDG2 at the coloured points in the top left panel of Figure 5.1. Table 5.2 shows the RMSE and r values computed from the time series, and also from flood maps predicted at the end of the simulation. For both values of ε, the GPU-MWDG2 predictions are very close to the GPU-DG2 predictions, as seen visually in Figure 5.3 and as quantified by the RMSE (order of magnitude of 10^{-3}) and r coefficients (> 0.99).

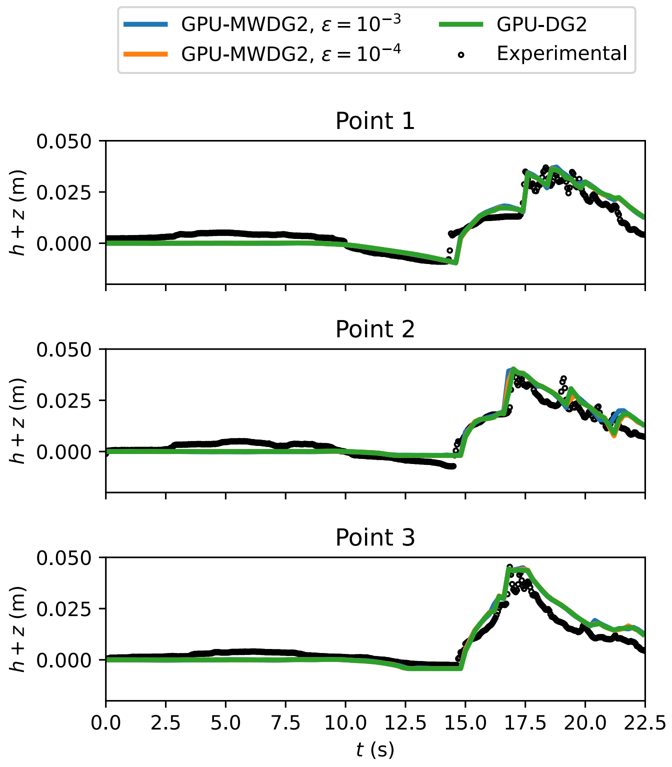


Figure 5.3: *Time series of the water surface elevation $(h + z)$ predicted by GPU-DG2 and GPU-MWDG2 at the three points shown in Figure 5.1 for the Monai valley test case.*

Table 5.2: *RMSE and r values of the water surface elevation predicted by GPU-MWDG2 with respect to the GPU-DG2 predictions for the Monai valley test case.*

	RMSE		$\boldsymbol{r}$	
Prediction dataset	ε = **10^{-3}**	ε = **10^{-4}**	ε = **10^{-3}**	ε = **10^{-4}**
Time series at Point 1	4.19×10^{-4}	1.36×10^{-4}	0.9995	0.9999
Time series at Point 2	1.40×10^{-3}	7.55×10^{-4}	0.9935	0.9979
Time series at Point 3	4.53×10^{-4}	1.91×10^{-4}	0.9994	0.9999
Flood map at t_{end}	1.81×10^{-3}	1.00×10^{-3}	0.9978	0.9993

Overall, GPU-MWDG2 competitively reproduces the GPU-DG2 water surface elevation predictions with both ε values while being more than 2 times faster than GPU-DG2 due to using $L = 10$, i.e. the “borderline” value of L where GPU-MWDG2’s wavelet-based grid adaptation capability reliably yields a speedup. In the next test case, the impact of a larger DEM size, requiring a larger L value, on the speedup of the GPU-MWDG2 solver is evaluated, while further considering the prediction of more complex velocity-related quantities.

5.3 Seaside, Oregon

This is another popular benchmark test case used to validate hydrodynamic solvers for nearshore tsunami inundation simulation (Park et al., 2013; Violeau et al., 2016; Qin et al., 2018; Gao, 2020; Macías et al., 2020b). It involves a 1:50 scaled replica of an urban town in Seaside, Oregon, flooded by a tsunami travelling along a scaled DEM made up of 2181×1091 cells, here requiring a larger $L = 12$ to generate the initial square uniform grid. In Figure 5.4, the bathymetric area is shown (top left panel), which is very plain everywhere except to the right where very complex terrain features of the urban town, such as buildings and streets, are located. The urban town is flooded by a tsunami that enters from the left boundary and travels a distance of 33 m to the right before hitting and flooding the town (as shown by the red arrows). This tsunami is simulated for 40 s, during which it travels through the bathymetric area in four stages of flow over time, much like in the last test case (Section 5.2): the entry stage (0 to 10 s), the travelling stage (10 to 25 s), the flooding stage (25 to 35 s), and the inundation stage (35 to 40 s). During the entry stage, the tsunami is not yet in the bathymetric area. During the travelling stage, the tsunami starts to enter the bathymetric area from the left boundary and travels right towards the town. During the flooding stage, the tsunami hits the town, flooding the streets and overtopping some of the buildings, causing vigorous flow dynamics. Finally, during the inundation stage, the tsunami inundates the town and eventually interacts with the right boundary, causing wave reflections. The water surface elevation hydrograph of the tsunami is plotted in the bottom left panel of Figure 5.4: it only has a single peak, indicating low tsunami complexity. The right panels show the initial GPU-MWDG2 non-uniform grids at $\varepsilon = 10^{-3}$ and 10^{-4}, respectively (again for the portion of the bathymetric area framed by the white box in the top left panel): at $\varepsilon = 10^{-3}$, greater grid coarsening is achieved, with the grid including only 2% of the number of cells as in the GPU-DG2 uniform grid, whereas at $\varepsilon = 10^{-4}$ it is 5% since more cells are used due to retention of finer resolution around and within complex terrain features of the urban town.

In Figure 5.5, an analysis of runtimes of the GPU-MWDG2 and GPU-DG2 simulations are shown. As indicated by the time history of N_{cells}, the number of cells in the GPU-MWDG2 non-

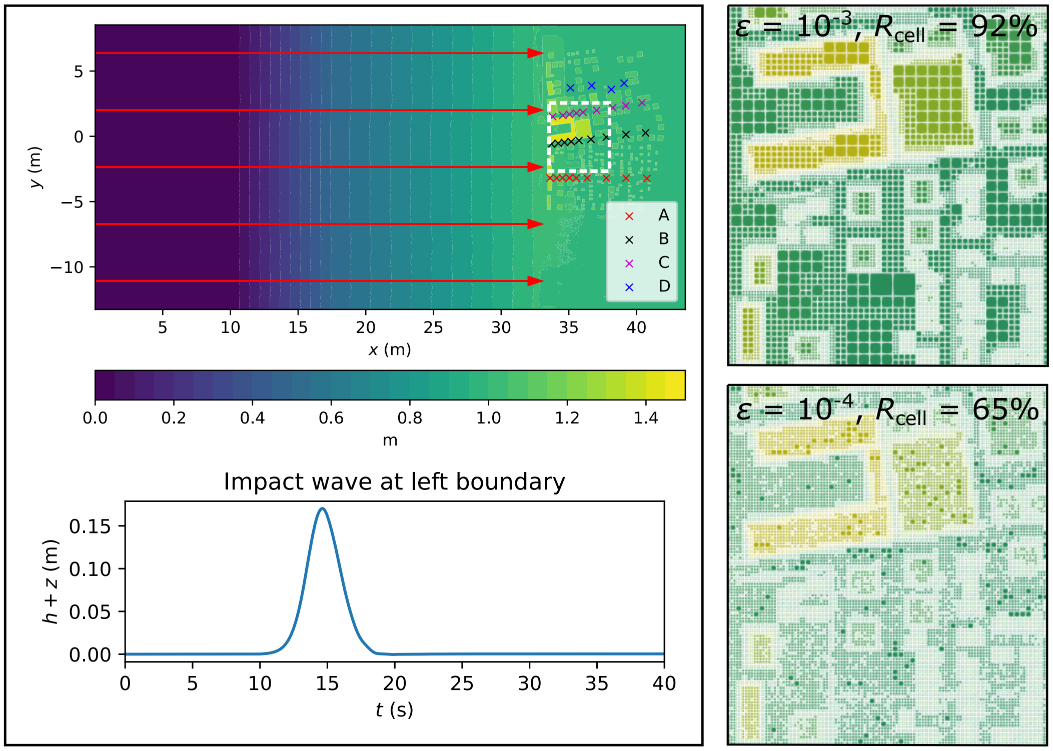


Figure 5.4: *Seaside, Oregon test case. Top-down view of bathymetry (top left panel); tsunami time history at the left boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels).*

uniform grid does not increase significantly for either value of ε until the flooding stage of flow at 25 s, where N_{cells} starts increasing more noticeably. Once the number of cells starts increasing, there is a corresponding increase in R_{DG2}. On the other hand, the time history of R_{MRA} is quite flat during the entire simulation, except for a small decrease during the first 10 s of the simulation, and a small increasing trend in the final 5 s of the simulation, i.e. during the inundation stage of flow when the number of cells increases relatively sharply compared to the rest of the simulation. Driven primarily by the increase in R_{DG2} at the flooding stage at 25 s, the time history of S_{inst} is quite stable until 25 s and thereafter shows a decreasing trend that is particularly steep at $\varepsilon = 10^{-3}$.

Like the previous test case (5.2), the time histories of Δt in the GPU-DG2 simulation and GPU-MWDG2 simulation at $\varepsilon = 10^{-4}$ are very similar, but at $\varepsilon = 10^{-3}$, there is a sharp drop in Δt after 32 s, i.e. when the flooding stage of flow starts transitioning to the inundation stage. This sharp drop in Δt is triggered by wet/dry fronts at coarse cells that are present in the non-uniform grid with $\varepsilon = 10^{-3}$, but not with 10^{-4}. The first drop in Δt, which occurs at 25 s when the flooding stage starts, leads to a locally steeper trend in the time history of $N_{\Delta t}$, as indicated by the kink

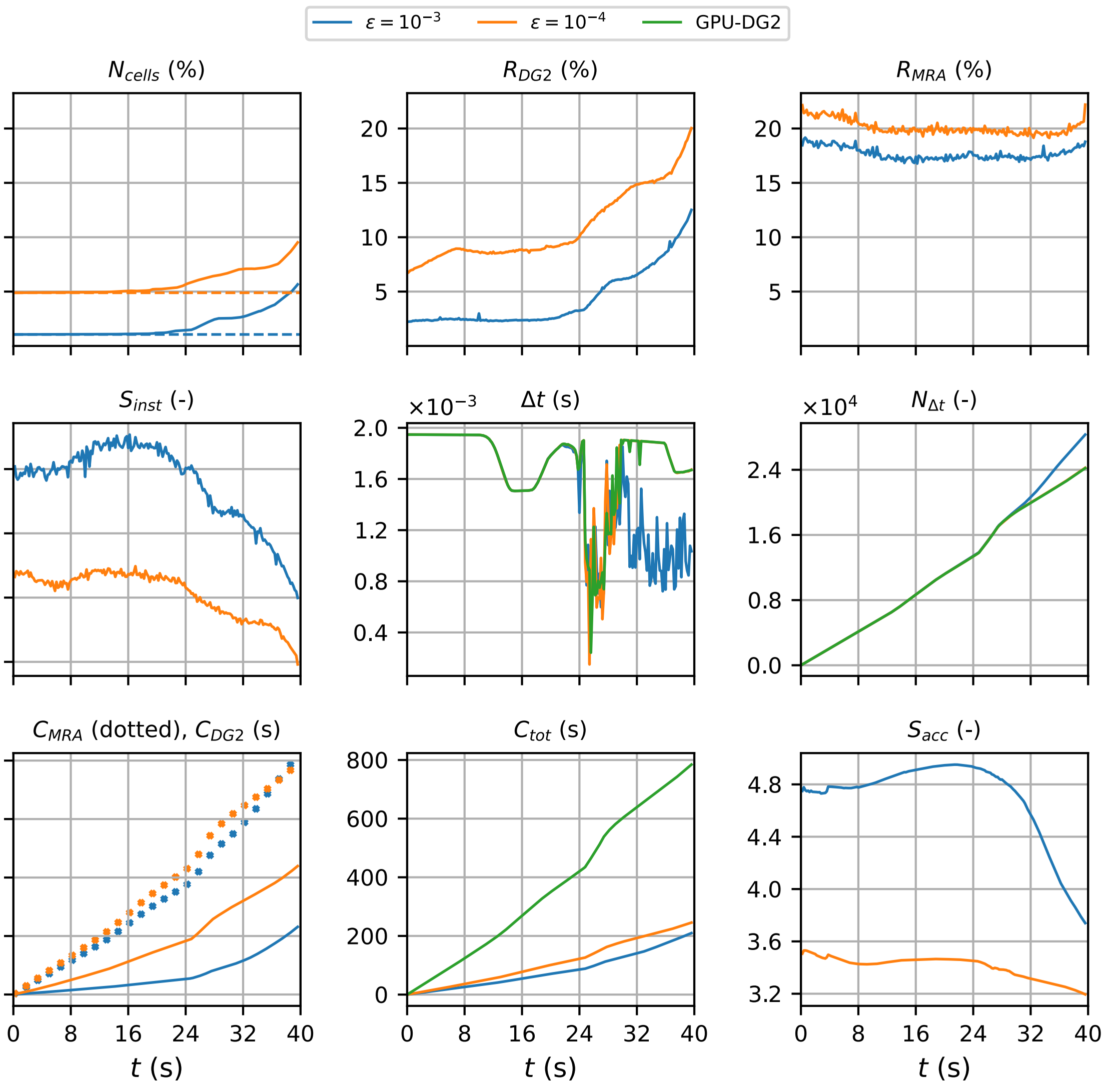


Figure 5.5: *Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Seaside, Oregon test case.*

at 25 s. The second drop in Δt, which is seen only for $\varepsilon = 10^{-3}$ after 32 s, leads to a sustained steepness in the time history of $N_{\Delta t}$ after 32 s. This steepness means that GPU-MWDG2 takes more timesteps and thus accumulates more computational effort to reach a given simulation time at $\varepsilon = 10^{-3}$ than 10^{-4}, which is confirmed by the final value of C_{MRA}, which is higher at the end of the simulation at $\varepsilon = 10^{-3}$ compared to 10^{-4}, even though its time history at $\varepsilon = 10^{-3}$ was consistently lower than at 10^{-4} before this. Since R_{MRA} was always lower at $\varepsilon = 10^{-3}$ than 10^{-4}, this observation about C_{MRA} suggests that even if the computational effort per timestep is lower throughout the simulation, a high timestep count can sufficiently increase the cumulative computational effort such that it becomes higher at $\varepsilon = 10^{-3}$ than 10^{-4}. Nonetheless, the time history of C_{tot} in the GPU-MWDG2 simulations always remains well below that of the GPU-DG2 simulation, with S_{inst} finishing at 3.5 and 3.0 with $\varepsilon = 10^{-3}$ and 10^{-4}, respectively.

Having analysed the computational effort, the GPU-MWDG2 and GPU-DG2 predictions obtained from this test case are compared next, where more complicated velocity-related quantities are considered that were not considered in the last test case (Section 5.2).

In the physical experiment of this test case, time histories of the water surface elevation, velocity, and momentum were recorded at 31 locations, as indicated by the crosses in the top left panel of Figure 5.4. Figure 5.6 shows the water surface elevation, velocity, and momentum time series predicted by GPU-MWDG2 and GPU-DG2 at three of these locations, which are, in increasing order of modelling difficulty: point A1 (one of the left-most crosses in Figure 5.4), point B6 (one of the central crosses), and point D4 (one of the right-most crosses). Table 5.3 shows the RMSE and r values computed from the time series at these points, and also from spatial maps predicted at the end of the simulations.

At point A1, which is located in the impact zone where the tsunami initially hits the town, the GPU-MWDG2 predictions are very close to the GPU-DG2 predictions for both values of ε (RMSE order of magnitude between 10^{-2} and 10^{-3}, r coefficients > 0.99). At point B6, which is located in the middle of the town and is slightly downstream of the tsunami impact zone, the GPU-MWDG2 predictions are still very close to the GPU-DG2 predictions: similar values of RMSE and r as seen for point A1 are obtained. At point D6, however, which is located near the end of the town and is therefore very downstream of the tsunami impact zone and separated by many buildings that could lead to flow discrepancies, there are significant differences between the GPU-MWDG2 and GPU-DG2 velocity and momentum predictions using $\varepsilon = 10^{-3}$, where using $\varepsilon = 10^{-4}$ is necessary for the GPU-MWDG2 predictions to track the GPU-DG2 predictions, as seen in Figure 5.6 and as confirmed by the r values in Table 5.3 (velocity: r is 0.8889 and 0.9176 for $\varepsilon = 10^{-3}$ and 10^{-4}, respectively; momentum: r is 0.7746 and 0.9214). For the spatial maps, a similar pattern in the r coefficients with respect to ε is observed as was seen for point D4. This indicates that running GPU-MWDG2 with $\varepsilon = 10^{-4}$ is a better choice here for obtaining velocity-related predictions.

Overall, GPU-MWDG2 seems to be faster than GPU-DG2 in this test case compared to the last test case due to the larger value of L required – $L = 12$ instead of 10. Running GPU-MWDG2

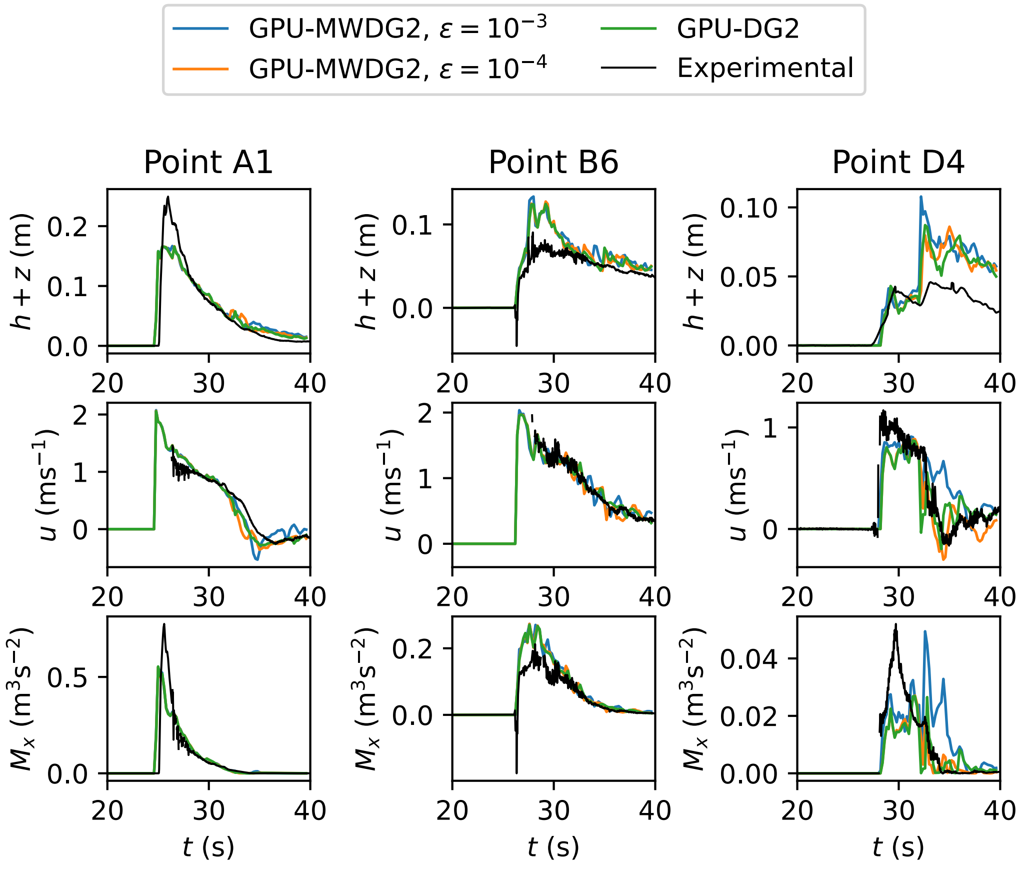


Figure 5.6: *Time series of the water surface elevation ($h + z$), x-directional velocity (u) and momentum (M_x) produced by GPU-DG2 and GPU-MWDG2 at points A1, B6 and D4 in Figure 5.4.*

with $\varepsilon = 10^{-3}$ leads to discrepancies in the velocity and momentum predictions compared to the GPU-DG2 predictions, and to reduce these discrepancies, running GPU-MWDG2 with $\varepsilon = 10^{-4}$ is needed. These results suggest that GPU-MWDG2 should be run here with $\varepsilon = 10^{-3}$ if higher speedups are desired and water surface elevation predictions are sufficient, whereas if velocity and/or momentum predictions are needed, $\varepsilon = 10^{-4}$ is necessary.

In the next test case (Section 5.4), GPU-MWDG2 is compared against GPU-DG2 over simulations of a real-world tsunami event that requires $L = 12$ like in this test case but instead involves a tsunami of considerably higher complexity.

Table 5.3: *RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Seaside, Oregon test case*

		RMSE		$\boldsymbol{r}$	
Prediction dataset	**Quantity**	ε **= 10^{-3}**	ε **= 10^{-4}**	ε **= 10^{-3}**	ε **= 10^{-4}**
Time series at A1	$h + z$	3.04×10^{-3}	2.68×10^{-3}	0.9979	0.9982
	u	6.86×10^{-2}	4.54×10^{-2}	0.9905	0.9958
	M_x	2.90×10^{-3}	1.01×10^{-3}	0.9995	0.9999
Time series at B6	$h + z$	4.48×10^{-3}	2.46×10^{-3}	0.9931	0.9976
	u	5.65×10^{-2}	59×10^{-2}	0.9944	0.9968
	M_x	7.40×10^{-3}	3.87×10^{-3}	0.9943	0.9982
Time series at D4	$h + z$	5.87×10^{-3}	3.75×10^{-3}	0.9841	0.9897
	u	1.30×10^{-1}	9.51×10^{-2}	0.8889	0.9176
	M_x	6.52×10^{-3}	2.20×10^{-3}	0.7746	0.9214
Spatial map at t_{end}	$h + z$	5.58×10^{-3}	1.50×10^{-3}	0.9999	0.9999
	u	6.66×10^{-2}	3.52×10^{-2}	0.8303	0.9502
	M_x	2.03×10^{-3}	5.89×10^{-4}	0.9350	0.9945

5.4 Tauranga harbour

This test case reproduces the 2011 Japan tsunami event in Tauranga Harbour, New Zealand (Borrero et al., 2015; Macías et al., 2015, 2020a). The bathymetric area has a DEM made of 4096×2196 cells, requiring $L = 12$ to generate the initial square uniform grid. As shown by the red arrows in Figure 5.7, the tsunami enters from the top boundary and travels a short distance downwards before quickly hitting the coast at $y = 16$ km. As shown by the time history of the water surface elevation (bottom left panel of Figure 5.7), the tsunami is a wave train made up of three wave peaks and troughs that enter the bathymetric area one after the other at 0, 12, and 24 hr during the 40-hr tsunami event, with the latter two waves also exhibiting noise. Due to the wave train, the short travel distance before flooding the harbour, and the highly irregular bathymetric zones that trigger wave reflections and diffractions, vigorous flow dynamics occur within the bathymetric area from the very beginning of the simulation. The right panels of Figure 5.7 include the GPU-MWDG2 grids generated at $\varepsilon = 10^{-3}$ and 10^{-4} for the region bounded by the white box (top left panel). With $\varepsilon = 10^{-3}$, the number of cells in the grid is 5% of the GPU-DG2 uniform grid, whereas with $\varepsilon = 10^{-4}$, it is 15%, due to less coarsening in and around the irregular bathymetric zones.

In Figure 5.8, an analysis of the runtimes of the GPU-MWDG2 and GPU-DG2 simulations is shown. Unlike the previous test cases (Sections 5.2 and 5.3), the first wave of the tsunami wave train enters the bathymetric area immediately, causing N_{cells} to increase very sharply and immediately

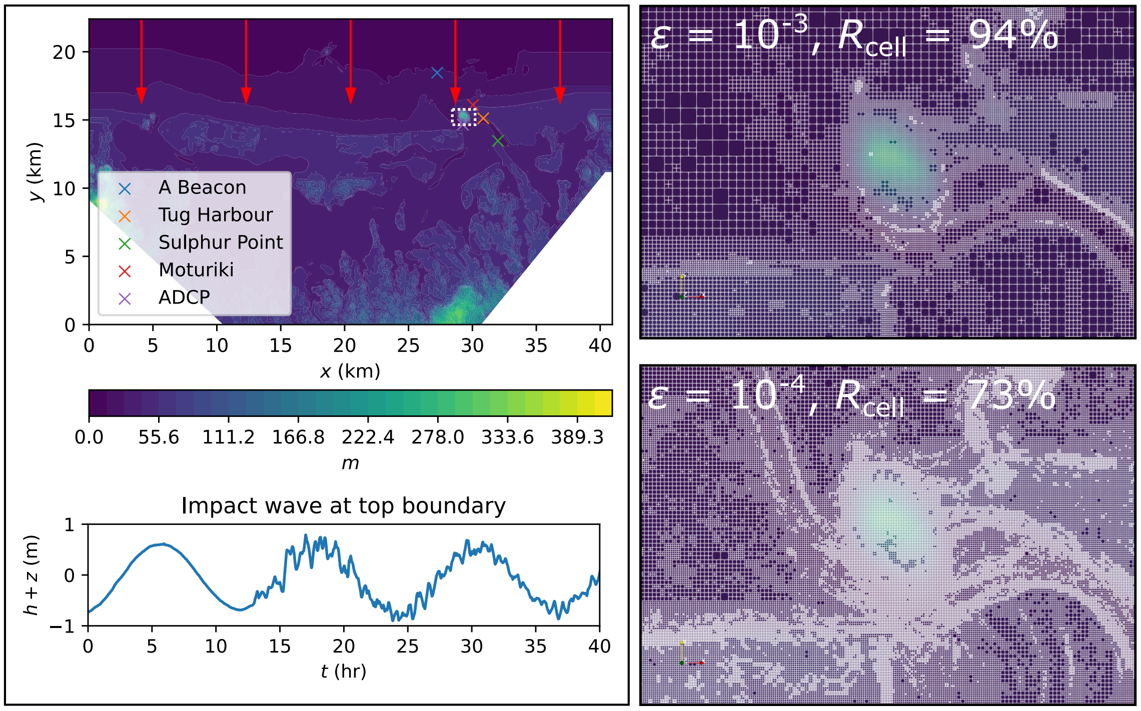


Figure 5.7: *Tauranga harbour test case. Top-down view of bathymetry (top left panel); tsunami time history at the top boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels).*

from its initial value for both values of ε, which thereafter fluctuates due to the periodic tsunami signal. Following the sharp increase and fluctuations in N_{cells}, R_{DG2} also sharply increases and fluctuates. However, R_{MRA} does not and stays stable and flat throughout the simulation. Thus, driven primarily by the sharp decrease in R_{DG2}, S_{inst} decreases sharply from 4.0 to 1.6. The time histories of Δt in the GPU-DG2 simulation and the GPU-MWDG2 simulation using $\varepsilon = 10^{-4}$ follow each other quite closely, but at $\varepsilon = 10^{-3}$, the time history of Δt shows two periodic drops after 24 h, likely due to periodic wetting and drying processes around coarse cells that are present in the non-uniform grid at $\varepsilon = 10^{-3}$ but not at 10^{-4}. Due to the smaller Δt at $\varepsilon = 10^{-3}$, the time history of $N_{\Delta t}$ is locally steeper (see the kinks at 25 and 35 h), but this does not lead to significant differences between the cumulative computational effort at $\varepsilon = 10^{-3}$ versus 10^{-4}. The time history of C_{tot} in the GPU-MWDG2 simulations consistently remains below that of the GPU-DG2 simulation for both values of ε, but they are relatively close to each other compared to the previous test cases (Sections 5.2 and 5.3). Thus, even though S_{acc} starts at around 4, like in the previous test case with $L = 12$ (Section 5.3), it drops sharply to 1.6 and 1.4 at $\varepsilon = 10^{-3}$ and 10^{-4}, respectively.

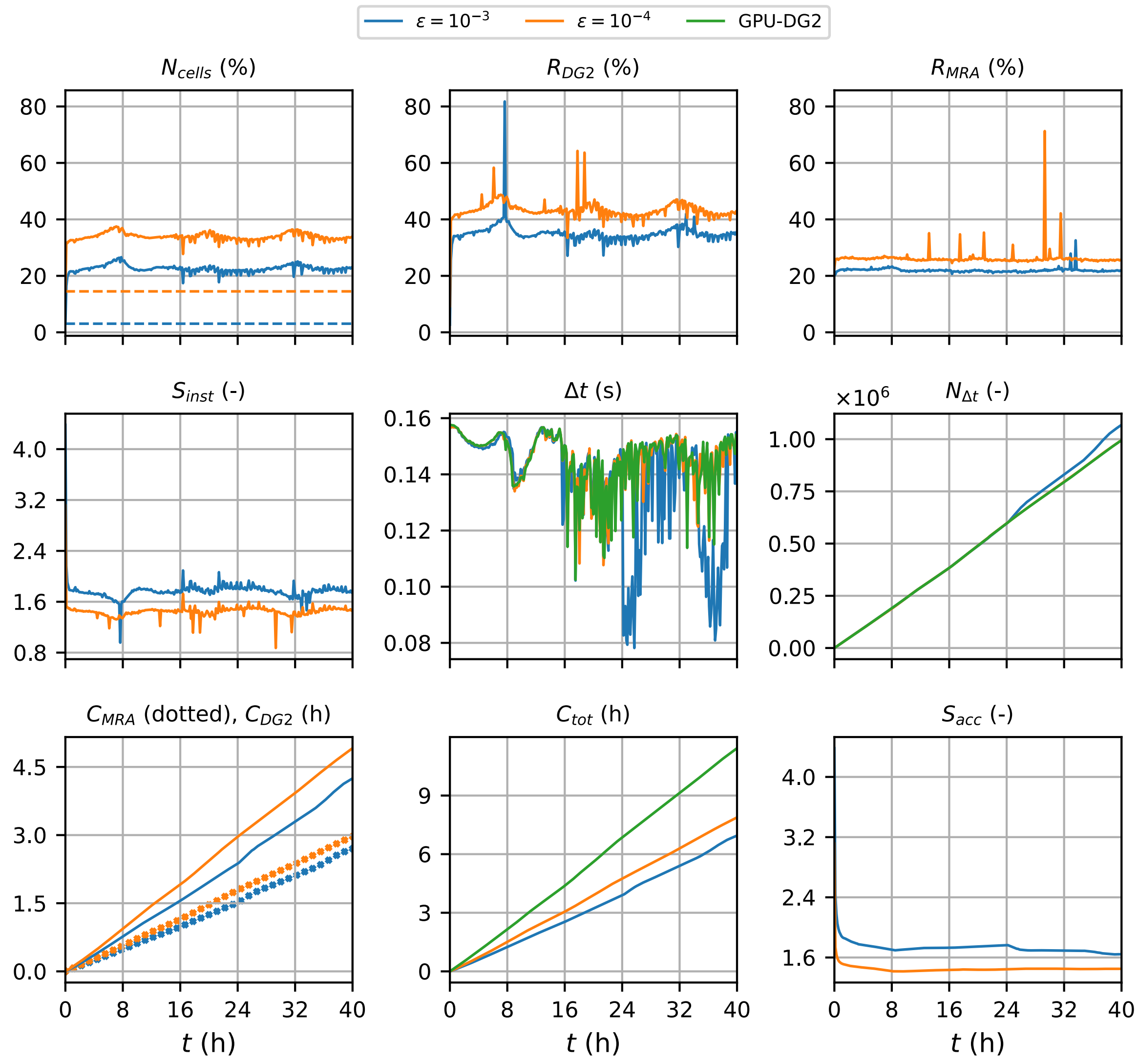


Figure 5.8: *Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Tauranga harbour test case.*

The first four panels of Figure 5.9 show the water surface elevation time series predicted by GPU-MWDG2 and GPU-DG2 at the points labelled "A Beacon", "Tug Harbour", "Sulphur Point" and "Moturiki" in the top left panel of Figure 5.7, while the last panel shows the speed time series predicted at the point labelled "ADCP". Table 5.4 shows the RMSE and r values computed from the time series at these points, as well as from spatial maps predicted at the end of the simulations. For the water surface elevation predictions, both the time series and spatial maps predicted by GPU-MWDG2 are very similar to those predicted by GPU-DG2 regardless of the choice of ε, as seen visually in Figure 5.9 and confirmed by the r values in Table 5.4 (> 0.99). For the speed predictions, however, the GPU-MWDG2 time series are closer to the GPU-DG2 time series using $\varepsilon = 10^{-4}$ than 10^{-3}, where r is 0.9527 and 0.9023, respectively. From the spatial maps, the advantage of using $\varepsilon = 10^{-4}$ than 10^{-3} is even clearer, where r is 0.8950 and 0.7556 – a larger difference of 0.14. Thus, using $\varepsilon = 10^{-4}$ seems to be a more suitable choice here for obtaining speed predictions.

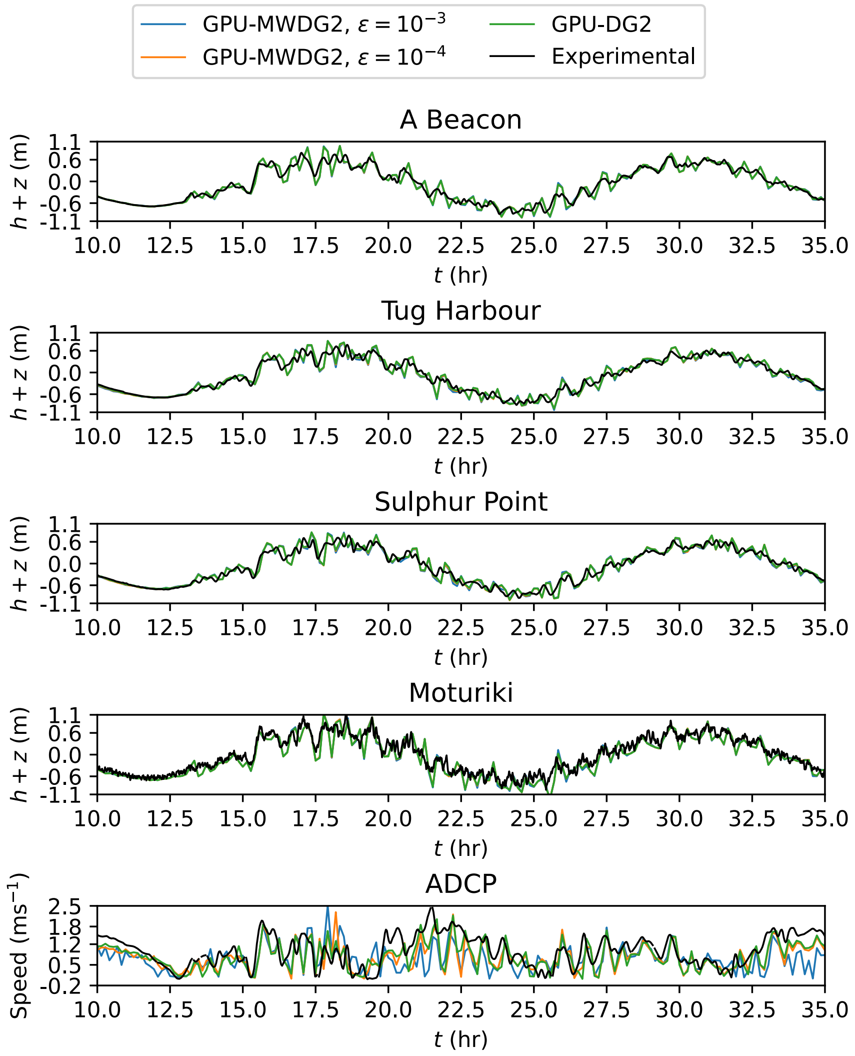


Figure 5.9: *Time series of the water surface elevation ($h + z$) produced by GPU-DG2 and GPU-MWDG2 at the points labelled A Beacon, Tug Harbour, Sulphur Point and Moturiki and of the speed at the point labelled "ADCP" in Figure 5.7.*

Table 5.4: *RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Tauranga harbour test case*

		RMSE		***r***	
Prediction dataset	**Quantity**	$\varepsilon = 10^{-3}$	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-3}$	$\varepsilon = 10^{-4}$
A Beacon	$h + z$	7.46×10^{-2}	1.28×10^{-2}	0.9986	0.9999
Tug Harbour	$h + z$	6.86×10^{-2}	8.99×10^{-2}	0.9984	0.9972
Sulphur Point	$h + z$	1.62×10^{-1}	1.43×10^{-1}	0.9916	0.9933
Moturiki	$h + z$	7.81×10^{-2}	2.35×10^{-2}	0.9935	0.9993
ACDP	Speed	3.45×10^{-1}	2.40×10^{-1}	0.9023	0.9527
Spatial map at t_{end}	$h + z$	7.33×10^{-2}	3.10×10^{-2}	0.9999	0.9999
	Speed	8.01×10^{-2}	4.80×10^{-2}	0.7556	0.8950

Overall, this test case features a more complex tsunami compared to the previous test cases (Sections 5.2 and 5.3), which sharply increases the number of cells in the GPU-MWDG2 non-uniform grid and thus also increases the computational effort of performing the DG2 solver updates. Hence, despite requiring the same $L = 12$ as the previous test case (Section 5.3), the final speedups are lower in this test case due to the more complex tsunami, with S_{acc} finishing at 1.6- and 1.4-fold with $\varepsilon = 10^{-3}$ and 10^{-4}, respectively. Using $\varepsilon = 10^{-4}$ would improve the closeness to the GPU-DG2 predicted velocities, while using $\varepsilon = 10^{-3}$ leads to very close water surface elevation predictions and fairly accurate velocity predictions, although without a major improvement in the speedup. In the next test case, another complex tsunami with higher frequency impact event peaks is considered, but now with a smaller DEM size requiring $L = 10$.

5.5 Hilo harbour

This test case reproduces the 2011 Japan tsunami event at Hilo Harbour in Hawaii, USA (Arcos and LeVeque, 2015; Lynett et al., 2017; Velioglu Sogut and Yalciner, 2019; Macías et al., 2020a). It involves a complex tsunami made up of a high-frequency wave train that propagates for 6 hr into a bathymetric area that is smaller than the previous test case (Section 5.4). The latter bathymetric area has a DEM size made of 702×692 cells, requiring a smaller $L = 10$ to generate the initial square uniform grid. As shown in Figure 5.10, the tsunami enters from the top boundary and travels south to flood and interact with the coast at $y = 4$ km. The wave train occurs over the entire 6 hr simulation time, from a reference timestamp of 7 hr to 13 hr post-earthquake. In Figure 5.10, the time history of the wave train is shown during the 8.5 to 11 hr time period (bottom left panel): from the very beginning and during the entire simulation, violent flow dynamics occur in the bathymetric area. The right panels show the initial GPU-MWDG2 non-uniform grids generated at $\varepsilon = 10^{-3}$ and 10^{-4} for the region bounded by the white box (top left panel): the number of cells in the initial non-uniform grids are at 10.0% and 22.5% of the GPU-DG2 uniform grid at $\varepsilon = 10^{-3}$ and $\varepsilon = 10^{-4}$, respectively.

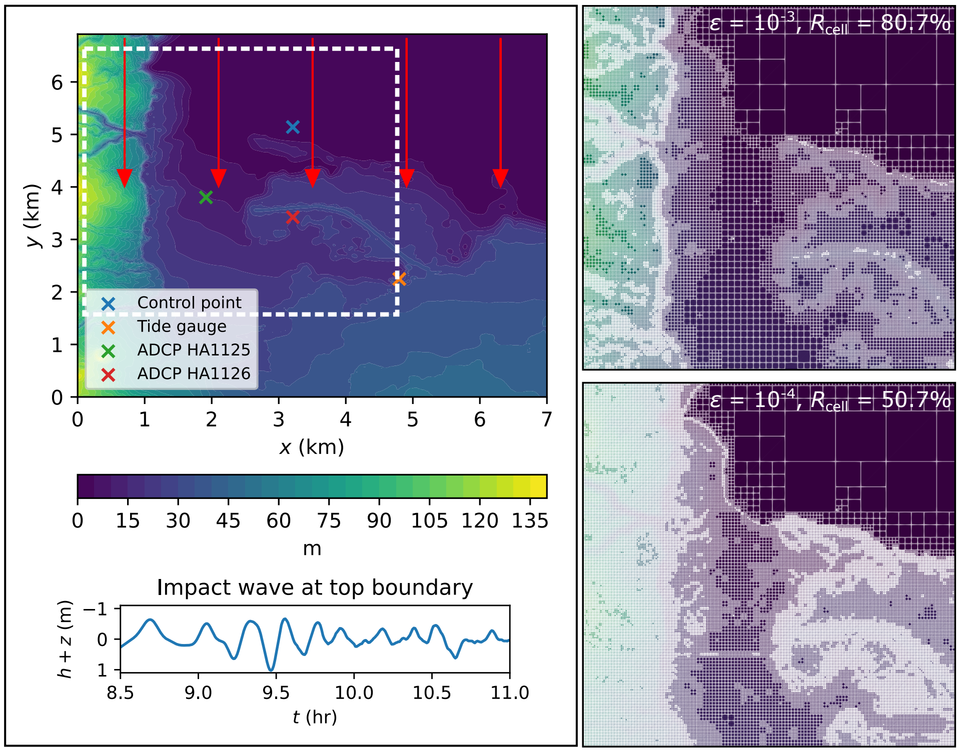

Control point
Tide gauge
ADCP HA1125
ADCP HA1126
y (km)
x (km)
m
Impact wave at top boundary
h + z (m)
t (hr)
ε = 10^-3, R_cell = 80.7%
ε = 10^-4, R_cell = 50.7%

Figure 5.10: *Hilo harbour test case. Top-down view of bathymetry (top left panel); tsunami time history at the top boundary (bottom left panel); initial non-uniform grids generated by GPU-MWDG2 for the region bounded by the dashed white box in the top left panel (right panels).*

In Figure 5.11, an analysis of runtimes of the GPU-MWDG2 and GPU-DG2 simulations is shown. Like the last test case (Section 5.4), since the wave train enters the bathymetric area immediately, N_{cells} increases sharply and immediately to maximum values of 32% and 40% at $\varepsilon = 10^{-3}$ and 10^{-4}, respectively. The time history of N_{cells} stays at this maximum for the simulation except for (somewhat less sharp) localized drops at certain simulation times, e.g., at 8 hr. Following

N_{cells}, R_{DG2} also increases sharply and immediately (to maximum values of 50% and 55% at $\varepsilon = 10^{-3}$ and 10^{-4}, respectively), and shows localized drops at the same timestamps as the drops in N_{cells}. In contrast, the time history of R_{MRA} stays very flat throughout the simulation, except for a small, temporary drop at 8 hr, which is when the largest drop in N_{cells} occurs. Due to the generally flat time histories of R_{DG2} and R_{MRA}, the time history of S_{inst} is also flat except for localized peaks that occur at the same timestamps as the localized drops in R_{DG2} and R_{MRA}. Unlike all of the previous test cases (Sections 5.2 to 5.4), the time history of Δt is very similar between the GPU-DG2 simulation and the GPU-MWDG2 simulations, regardless of the ε value, as they both yield a high number of cells compared to the uniform GPU-DG2 grid. Thus, the time history of $N_{\Delta t}$ is virtually identical across all simulations. Given that the time histories of $N_{\Delta t}$, R_{DG2}, and R_{MRA} are similar for both ε values, the time histories of C_{DG2} and C_{MRA} are also very similar. The time history of C_{tot} is very close between the GPU-DG2 simulation and the GPU-MWDG2 simulations in this test case (even more so than in the last case, Section 5.4), so S_{acc} is the lowest out of all the test cases, finishing at 1.25 and 1.10 using $\varepsilon = 10^{-3}$ and 10^{-4}, respectively.

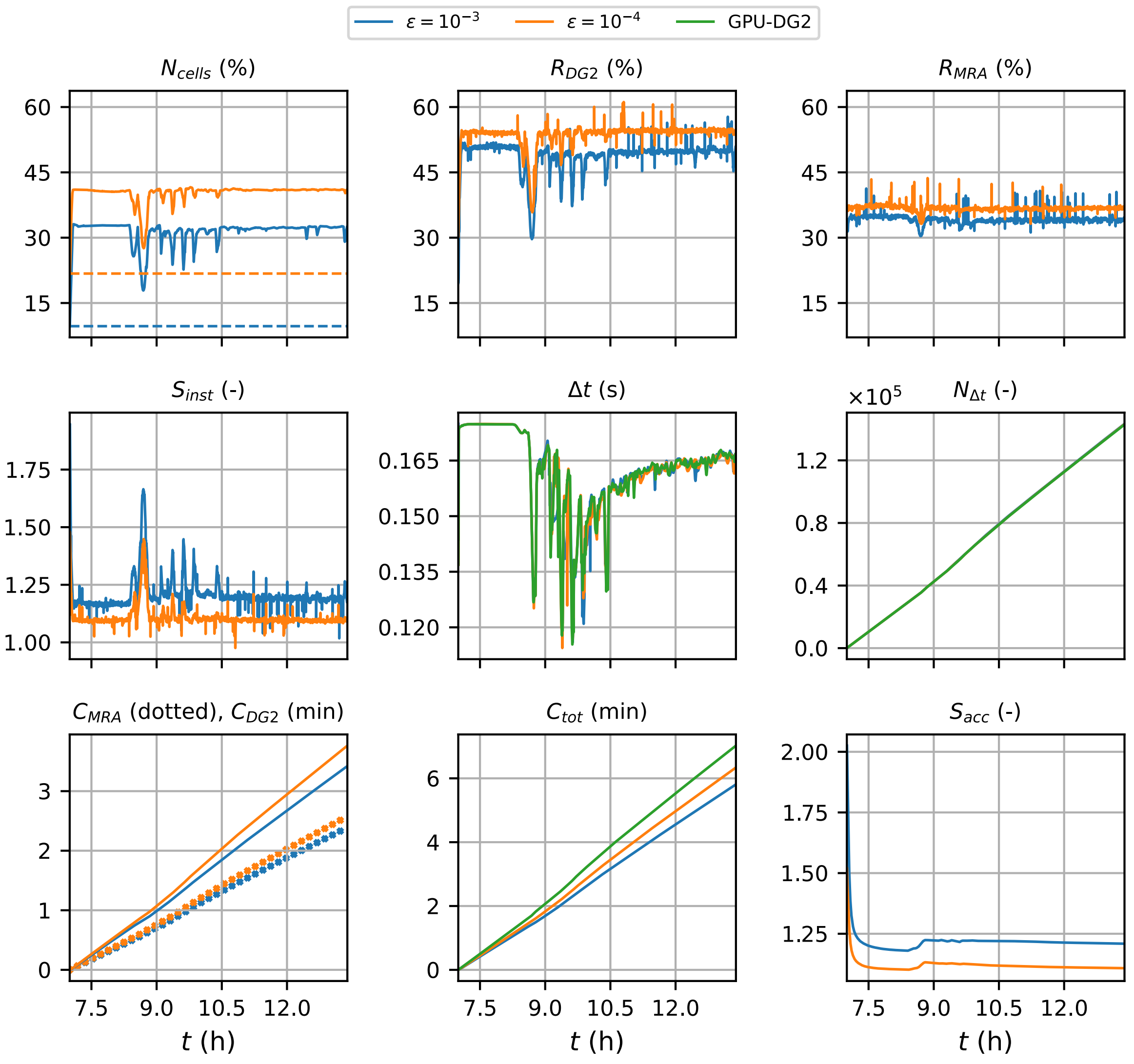


Figure 5.11: *Metrics of Table 4.3 applied to the GPU-MWDG2 and GPU-DG2 simulations. Also shown is a time history of Δt (centre panel) obtained for the Hilo harbour test case.*

The top panels of Figure 5.12 show the time series of the water surface elevation predicted by GPU-MWDG2 and GPU-DG2 at the points labelled "Control point" and "Tide gauge" in the top left panel of Figure 5.10, whereas the bottom panels of Figure 5.12 show the velocity time series at the points labelled "ADCP 1125" and "ADCP 1126". Table 5.5 shows the RMSE and r coefficients computed from the time series, as well as from spatial maps predicted at the end of the simulations. GPU-MWDG2's water surface elevation predictions are very similar to those of GPU-DG2, as seen visually in the top panels of Figure 5.12 and confirmed by the r values in Table 5.5 (> 0.97) for both values of ε. However, for the velocity predictions, using $\varepsilon = 10^{-4}$ gives closer predictions, as seen from the time series at ADCP 1126 (r is 0.8343 versus 0.9193 for $\varepsilon = 10^{-3}$ and 10^{-3}, respectively), and especially seen from the spatial maps, where r is 0.7063 versus 0.8134 for $\varepsilon = 10^{-3}$ and 10^{-3} for the y-directional velocity predictions, respectively, while for the x-directional velocity predictions, r is 0.7465 versus 0.8387. Thus, it appears that running GPU-MWDG2 with $\varepsilon = 10^{-4}$ continues to be the more suitable choice for obtaining velocity predictions close to those of GPU-DG2.

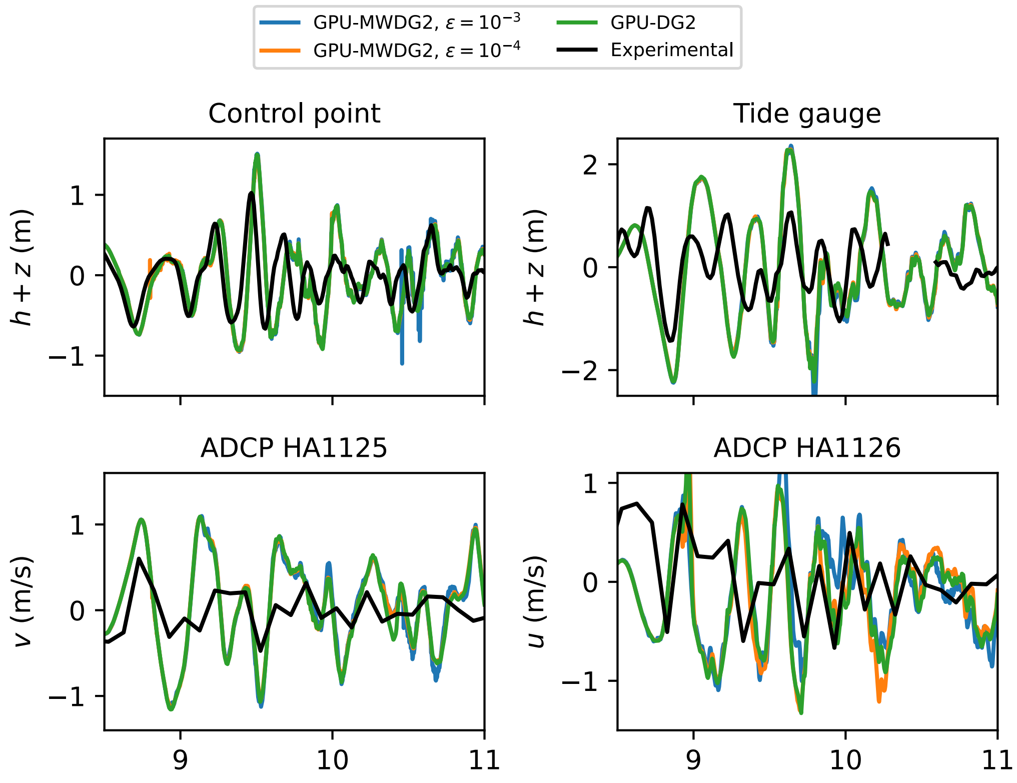


Figure 5.12: *Time series simulated by GPU-DG2 and GPU-MDG2 at the labelled points in Figure 5.10.*

Overall, despite simulating a test case with a complex tsunami impact event and also a DEM size that requires selecting a small $L = 10$, GPU-MWDG2 still manages to attain speedups over GPU-DG2 in this test case: around 1.25 at $\varepsilon = 10^{-3}$ and 1.10 at $\varepsilon = 10^{-4}$. This seems to suggest that the GPU-MWDG2 solver can reliably be used to gain speedups over the GPU-DG2 solver even if simulating complex tsunami impact events, using $\varepsilon = 10^{-3}$ to boost the speedup, or using $\varepsilon = 10^{-4}$ to increase the quality of velocity predictions.

Table 5.5: *RMSE and r values for the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions for the Hilo harbour test case*

		RMSE		***r***	
Prediction dataset	**Quantity**	$\varepsilon = 10^{-3}$	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-3}$	$\varepsilon = 10^{-4}$
Control point	$h + z$	1.11×10^{-1}	5.45×10^{-2}	0.9767	0.9923
Tide gauge	$h + z$	1.81×10^{-1}	1.10×10^{-1}	0.9827	0.9938
ADCP HA1125	v	3.63×10^{-1}	2.91×10^{-1}	0.7884	0.8252
ADCP HA1126	u	3.23×10^{-1}	2.23×10^{-1}	0.8343	0.9193
Spatial map at t_{end}	$h + z$	5.31×10^{-2}	6.20×10^{-3}	0.9999	0.9999
	v	9.17×10^{-2}	6.09×10^{-2}	0.7063	0.8134
	u	9.01×10^{-2}	5.93×10^{-2}	0.7465	0.8387

5.6 Summary

This chapter addressed Objective 3 by comparing GPU-MWDG2 against GPU-DG2 to analyse the impact of GPU-MWDG2's grid adaptation capability on the accuracy and computational effort of DG2 modelling when simulating multiscale rapid flow phenomena, particularly tsunami-induced flooding. In a simulation involving tsunami-induced flooding, the amount of grid coarsening and thus the computational effort and speedup may be particularly affected over time by the flow features produced by the tsunami as it enters and travels through the test case area. Further to the complexity of the tsunami, the speedup achieved by GPU-MWDG2's wavelet-based grid adaptation capability may also depend on the DEM size, as the speedup achieved by wavelet-based grid adaptation when simulating rapid flows was shown in Chapter 3 to scale with the value of L and to require $L \geq 9$ for realistic test cases with complex DEMs: as the DEM size increases, so does the value of L required to accommodate the DEM size, and so does the potential for GPU-MWDG2 to achieve a speedup – hypothetically for $L \geq 9$ in realistic test cases involving rapid flows over complex DEMs.

Therefore, to systematically assess the effect of the DEM size and the tsunami complexity on the speedup achieved by GPU-MWDG2's grid adaptation capability, GPU-MWDG2 was compared against GPU-DG2 by running simulations of four tsunami-induced flooding test cases that each have a unique combination of DEM size and tsunami complexity. GPU-DG2 was always run on a uniform grid at the finest resolution available to GPU-MWDG2. Meanwhile, GPU-MWDG2 was run with two choices for the error threshold ε: 10^{-3} and 10^{-4}, as these values have been shown to preserve a similar level of accuracy as GPU-DG2 for flood and tsunami modelling – in the sense

Table 5.6: *Summary of key set-up parameters and results per test case. The runtime is the time taken to complete a simulation of a test case. The numbers 10^{-3} and 10^{-4} refer to the value of ε chosen for GPU-MWDG2.*

Test case	t_{end}	**Tsunami complexity**	L	**Runtime**			**Average speedup**	
				MWDG2		DG2	MWDG2	
				10^{-3}	10^{-4}	-	10^{-3}	10^{-4}
Monai valley (Section 5.2)	22.5 s	Simple	10	16 s	20 s	40 s	2.5×	2.0×
Seaside, Oregon (Section 5.3)	40 s	Simple	12	3.5 min	5.2 min	13.0 min	4.5×	3.3×
Tauranga harbour (Section 5.4)	40 hr	Complex	12	7.5 hr	8.1 hr	11.3 hr	1.8×	1.4×
Hilo harbour (Section 5.5)	6 hr	Complex	10	5.3 min	5.8 min	6.9 min	1.3×	1.2×

that for this range of ε, GPU-MWDG2 gives flow predictions close to those of GPU-DG2. The closeness of the predictions was shown visually and was also quantified by computing the RMSE and r coefficient of the GPU-MWDG2 predictions with respect to the GPU-DG2 predictions, whereby the lower the value of the RMSE and the closer the value of r to 1, the better the closeness of the predictions.

To identify the combination(s) of DEM size, tsunami complexity and error threshold ε for which GPU-MWDG2 is potentially a better choice than GPU-DG2 for simulating multiscale rapid flow phenomena, Table 5.6 shows a summary of the key results from simulating the tsunami test cases in Sections 5.2 to 5.5. The speedup achieved by GPU-MWDG2 seems to be maximised depending on: (i) the DEM size requiring $L \geq 9$ and (ii) the tsunami complexity being simple. As shown in Table 5.6, for the test cases involving simple tsunamis: when the DEM area required $L = 10$, the average speedups would be around 2.0-fold (i.e. the Monai Valley test case); whereas, with the DEM area requiring a larger $L = 12$, considerable average speedups of 3.3-fold were achieved at $\varepsilon = 10^{-4}$, which increased to 4.5-fold at $\varepsilon = 10^{-3}$ (i.e. the Seaside, Oregon test case). Meanwhile, for the complex tsunamis, the average speedups reduced to 1.8-fold for a DEM area requiring $L = 12$ (i.e. the Tauranga Harbour test case) and to 1.2-fold for a smaller DEM requiring a smaller $L = 10$ (i.e. the Hilo Harbour test case). Thus, GPU-MWDG2's grid adaptation capability best reduces the computational effort when simulating multiscale rapid flows for test cases featuring a simple impact event and a DEM size requiring $L \geq 10$.

Chapter 6

Conclusions

6.1 Thesis summary

This thesis' research aim was to integrate a multiwavelet (MW) grid adaptation capability into the second-order discontinuous Galerkin (DG2) model in LISFLOOD-FP parallelised on the graphics processing unit (GPU) (Shaw et al. 2021), referred to as "GPU-DG2", in order to reduce the computational effort of DG2 modelling when simulating multiscale rapid flow phenomena such as tsunamis – while still achieving a similar level of accuracy as GPU-DG2 run on a uniform mesh. To achieve the research aim, three Objectives were defined:

1. Propose computational ingredients for redesigning the algorithmic structure of a wavelet-based grid adaptation algorithm based on "Haar" wavelets (HW) in order to make it suitable for efficient implementation on the GPU;

2. Extend the use of the computational ingredients proposed in Objective 1 in order to efficiently implement a MW-based grid adaptation algorithm on the GPU and integrate it into the GPU-DG2 model in LISFLOOD-FP;

3. Analyse the impact of the GPU-parallelised MW-based grid adaptation algorithm on the accuracy and computational effort of DG2 modelling when simulating multiscale rapid flow phenomena.

Before addressing the Objectives, a literature review was conducted in Chapter 2. In Section 2.1 of the literature review, it was pointed out that simultaneously adopting both GPU-parallelisation and wavelet-based grid adaptation may substantially reduce the computational effort of DG2 modelling when simulating multiscale rapid flow phenomena without significantly compromising on the level of accuracy, making it a compelling research area, but nevertheless, it was argued that this research area remained uninvestigated due to grid adaptation algorithms being incompatible with GPU computing. To expand on this argument, Section 2.2 gave an overview of GPU computing and clarified that achieving coalesced memory access and avoiding warp divergence are essential for implementing an algorithm on the GPU efficiently. Informed by the overview on GPU computing, Section 2.3 established that achieving coalesced memory access and avoiding warp divergence is difficult when implementing grid adaptation algorithms on the GPU due to irregular and unpredictable operations on tree data structures.

Bearing these difficulties in mind, Objective 1 was addressed in Chapter 3, in which three computational ingredients – a Z-order space-filling curve, a parallel tree traversal algorithm and an array data structure – were proposed to achieve coalesced memory access and avoid warp divergence when implementing a HW-based grid adaptation algorithm on the GPU. The HW-based grid adaptation algorithm was based on the "multiresolution analysis" (MRA) of the HW, which is used to generate a non-uniform grid by "encoding" (coarsening), "decoding" (refining), analysing and traversing modelled data across a deep hierarchy of nested, uniform grids of increasingly coarser resolution, in which the coarsest grid in the hierarchy is made up of a single cell, while the finest grid in the hierarchy is made up of $2^L \times 2^L$ cells, where L is a user-specified maximum refinement level. In the encoding step of the MRA process, a user-specified error threshold ε is needed to flag significant "details" to decide which cells to include in the non-uniform grid, whereby a larger value of ε leads to a higher amount of grid coarsening and thus fewer cells in the non-uniform grid – and thus a lower computational effort to perform "solver computations" – e.g. flux calculations, time integration, etc. The encoding step results in a tree-like structure of significant details that is traversed in the decoding step of the MRA process using a sequential depth-first traversal algorithm to identify "leaf cells", which make up a non-uniform grid. Once the identified leaf cells are assembled into a non-uniform grid, a first-order finite volume (FV1) scheme is applied to perform solver computations over the assembled non-uniform grid.

Attempting to implement the MRA process on the GPU hindered coalesced memory access and incurred warp divergence due to three particular obstacles. The first obstacle was that the modelled data in the hierarchy of grids may be close together in physical space, but they are not guaranteed to be close together in GPU global memory space unless the hierarchy is indexed in a deliberate way, hindering coalesced memory access when the modelled data are accessed to perform the encoding and decoding steps. The second obstacle was that a recursive depth-first traversal algorithm is used in the decoding step to traverse the tree of significant details, which is fundamentally a sequential algorithm that cannot be parallelised on the GPU. The third obstacle was that the non-uniform grid generated by the MRA process is typically indexed and stored using a tree data structure, which hinders coalesced memory access when it is accessed to perform the FV1 solver computations over the non-uniform grid. These three obstacles were overcome using the three computational ingredients proposed above: first, Z-order curves were used to index the hierarchy of grids and thereby ensure coalesced memory access in the encoding and decoding steps. Second, the parallel tree traversal algorithm was used to replace the depth-first traversal algorithm and thereby traverse the tree of significant details on the GPU – with minimal warp divergence. Finally, an array data structure was used instead of a tree data structure to index and store the non-uniform grid and thereby achieve coalesced memory access when performing the FV1 solver computations over the non-uniform grid.

Using the three computational ingredients allowed to implement the HW-based grid adaptation algorithm on the GPU and thereby develop a GPU-parallelised HW adaptive FV1 shallow water model (GPU-HWFV1). To assess the extent to which using the three computational ingredients indeed allowed for an efficient GPU implementation of GPU-HWFV1's grid adaptation capability, GPU-HWFV1's speedup over its sequential CPU version – referred to as CPU-HWFV1 – and over its GPU-parallelised uniform-grid FV1 counterpart – referred to as GPU-FV1 – was analysed. Compared to CPU-HWFV1, GPU-HWFV1's grid adaptation capability was much more efficient (between 20 to 400$\times$ faster) and hinted at weak scaling behaviour. Meanwhile, compared to GPU-FV1, GPU-HWFV1 was generally faster (between 1.1 to 25$\times$ faster) – notably becoming progressively faster as either the maximum refinement level L or the error threshold ε were increased – although requiring $\varepsilon = 10^{-3}$ to remain faster in a realistic test case that used $L = 9$, where using $\varepsilon = 10^{-4}$ led to no overall speedup.

The findings from Chapter 3 suggested that extending the use of the proposed computational ingredients to implement a MW-based grid adaptation algorithm on the GPU was a research direction worth exploring. This direction was explored in Chapter 4, where Objective 2 was addressed: the computational ingredients were used to implement a MW-based grid adaptation algorithm on the GPU in a manner very similar to the HW-based grid adaptation algorithm in Chapter 3, although with a few differences due to being paired with a DG2 scheme instead of the FV1 scheme in GPU-HWFV1 – the DG2 scheme features three scalar coefficients for shaping a piecewise-planar solution per cell and a two-stage Runge-Kutta scheme for time integration (rather than only a single scalar coefficient for shaping a piecewise-constant solution per cell and a forward-Euler scheme for time integration – like in the FV1 scheme).

Implementing the MW-based grid adaptation algorithm on the GPU allowed to develop and integrate a GPU-parallelised MW adaptive DG2 model – referred to as GPU-MWDG2 – into LISFLOOD-FP. When using GPU-MWDG2 within LISFLOOD-FP's framework for running real-world flood simulations, seven extra keywords should be specified in the parameter file (a text file provided by the user to LISFLOOD-FP to specify various model features) to use GPU-MWDG2's grid adaptation capability correctly. Importantly, to run a simulation of a real-world test case involving a digital elevation model (DEM) with a raster grid of $N \times M$ cells, the maximum refinement level L should be specified to be the smallest value of L such that $2^L \geq \max(N, M)$. This should be done so that GPU-MWDG2 constructs an artificial $2^L \times 2^L$ grid – the size of the finest grid in the hierarchy of grids in the MRA process of the MW – that can accommodate the $N \times M$ raster grid of the DEM. Thus, the larger the DEM raster grid, the higher the value of L that must be specified to accommodate the DEM.

The higher the value of L, the higher the amount of memory consumed by GPU-MWDG2; however, GPU-MWDG2 cannot consume more memory than the memory capacity of a given GPU, so the memory capacity of a GPU places an upper limit on the value of L and thus the maximum test case area that can be considered using GPU-MWDG2 with that GPU. As of writing, GPU-MWDG2 cannot be used with L larger than 12 – equivalent to a maximum test case area of 4096 $\times$ 4096 cells – because doing so requires more than 80 GB of GPU memory, which is higher than the memory capacity of most commercially available GPUs.

Having integrated GPU-MWDG2 into LISFLOOD-FP, an analysis of GPU-MWDG2's grid adaptation capability was carried out in Chapter 5, which addressed Objective 3 by comparing GPU-MWDG2's DG2 accuracy and speedup compared to GPU-DG2 over a series of test cases involving tsunami-induced flooding. GPU-MWDG2's speedup over GPU-DG2 was hypothesised to depend on the amount of grid resolution coarsening introduced by GPU-MWDG2's grid adaptation capability: the higher the amount of coarsening in GPU-MWDG2's non-uniform grid, the lower the number of cells in the non-uniform grid and the lower the computational effort of the DG2 solver computations – and thus, the higher the speedup of GPU-MWDG2 over GPU-DG2.

In a simulation involving tsunami-induced flooding, the amount of grid coarsening – and therefore the speedup – may be particularly affected over time by the flow features produced by the tsunami as it enters and travels through the test case area. The speedup achieved by GPU-MWDG2's wavelet-based grid adaptation capability may also scale with the DEM size: the speedup achieved by wavelet-based grid adaptation was shown in Chapter 3 to scale with the value of L and to require $L \geq 9$ for realistic test cases. Thus, as the DEM size increases, so does the value of L required to accommodate the DEM, and hence so does the potential for GPU-MWDG2 to achieve a speedup. To therefore systematically assess the effect of the DEM size and the tsunami complexity on the speedup achieved by GPU-MWDG2's grid adaptation capability, GPU-MWDG2 was compared against GPU-DG2 by running simulations of four tsunami-induced flooding test cases that each had a unique combination of DEM size and tsunami complexity.

The speedup achieved by GPU-MWDG2 seemed to be maximised depending on: (i) the DEM size requiring $L \geq 9$ and (ii) the tsunami complexity being simple. For the test cases involving simple tsunamis: when the DEM area required $L = 10$, the average speedups would be around 2.0-fold (i.e. the Monai Valley test case); whereas, with the DEM area requiring a larger $L = 12$, considerable average speedups of 3.3-fold were achieved at $\varepsilon = 10^{-4}$, which increased to 4.5-fold at $\varepsilon = 10^{-3}$ (i.e. the Seaside, Oregon test case). Meanwhile, for the complex tsunamis, the average speedups reduced to 1.8-fold for a DEM area requiring $L = 12$ (i.e. the Tauranga Harbour test case) and to 1.2-fold for a smaller DEM requiring a smaller $L = 10$ (i.e. the Hilo Harbour test case). Thus, GPU-MWDG2's grid adaptation capability best reduces the computational effort when simulating multiscale rapid flows for test cases featuring a simple impact event and a DEM size requiring $L \geq 10$. Whenever GPU-MWDG2 is chosen, $\varepsilon = 10^{-3}$ should be preferred for higher speedups while still giving very close water surface predictions, but for closer velocity-related predictions, $\varepsilon = 10^{-4}$ is recommended.

Data and scripts for reproducing the Figures in Section 3.2 of Chapter 3 are available at `https://zenodo.org/records/8075133`. The GPU-MWDG2 code as part of LISFLOOD-FP 8.2 – including instructions on how to use GPU-MWDG2 to reproduce the simulations of the test cases in Chapter 5 – will be available at `https://www.seamlesswave.com/LISFLOOD8.0.html`.

Having provided a summary of the thesis, the significant research findings, limitations and potential future work for addressing the limitations can be noted.

6.2 Significant research findings, limitations and future work

The first key finding – identified in Chapter 3 – was that the three computational ingredients proposed to efficiently implement wavelet-based grid adaptation on the GPU – i.e. the Z-order curve, the parallel tree traversal algorithm and the array data structure – were indeed effective in allowing for a suitably efficient GPU implementation of wavelet-based grid adaptation: GPU-HWFV1's HW-based grid adaptation capability was much faster than its CPU version (between 20 to 400$\times$ faster), becoming faster either as the error threshold ε was decreased or as the maximum refinement level L was increased. Both of these actions increase the number of cells in the non-uniform grid and thus the computational throughput, indicating that GPU-HWFV1's grid adaptation capability becomes increasingly faster than its CPU version as the computational throughput is increased – thus hinting at GPU-HWFV1's weak scaling behaviour. This finding suggests that the ingredients could be used to parallelise other CPU-based wavelet models on the GPU.

The second key finding – identified in Chapter 5 – was that GPU-MWDG2's grid adaptation capability indeed reduces the computational effort of DG2 modelling when simulating multiscale rapid flow phenomena while still achieving a similar level of accuracy as GPU-DG2: GPU-MWDG2 is recommended over GPU-DG2 when simulating tsunami-induced flooding test cases featuring a simple impact event and a DEM size requiring $L \geq 10$ – for such test cases, GPU-MWDG2's grid adaptation capability makes it up to four times faster than GPU-DG2.

The main limitation of the research – identified in Chapter 4 – is that integrating GPU-MWDG2 into LISFLOOD-FP results in an enormous GPU memory cost that scales with the value of L required to accommodate the DEM of a test case: at $L = 9$, GPU-MWDG2 consumes only 0.2 GB of memory, but by $L = 13$, GPU-MWDG2 consumes around 100 GB of memory, which is higher than the memory capacity of most commercially available GPUs. Therefore, as of writing, GPU-MWDG2 can only be run with up to $L = 12$: this corresponds to a maximum test case area of 4096×4096 cells.

Handling larger test case areas using GPU-MWDG2 is therefore a clear direction for future research. One way to proceed in this direction would be to reduce GPU-MWDG2's memory consumption, for example by not storing the non-uniform grid and the neighbours in array data structures of fixed length: the non-uniform grid and neighbours consume the largest amount of memory out of all the objects in the GPU-MWDG2 code. This comes with a tradeoff however, as the array data structures are currently essential in facilitating coalesced memory access when performing solver computations over the non-uniform grid. Another way to handle larger test case areas would be by accessing more GPU memory by using multiple GPUs, i.e. by parallelising GPU-MWDG2 across multiple GPUs. Multi-GPU parallelisation of GPU-MWDG2 could be achieved either on a single node in a computing cluster, or across multiple nodes in a computing cluster – e.g. by combining CUDA and MPI. This is a very interesting future work, but it may require completely rewriting the GPU-MWDG2 codebase currently integrated into LISFLOOD-FP, thus entailing a significant research software engineering effort.

Finally, hardware advances must also be considered when implementing wavelet-based grid adaptation in future: having to implement grid adaptation on the GPU in order to avoid the CPU-GPU data transfer bottleneck could become superfluous if GPU hardware vendors in future tightly integrate CPU and GPU memory spaces or sufficiently increase CPU-GPU data transfer bandwiths, as in this case it would be better to simply perform grid adaptation on the CPU.

References

Gilou Agbaglah, Sébastien Delaux, Daniel Fuster, Jérôme Hoepffner, Christophe Josserand, Stéphane Popinet, Pascal Ray, Ruben Scardovelli, and Stéphane Zaleski. Parallel simulation of multiphase flows using octree adaptivity and the volume-of-fluid method. *Comptes Rendus Mécanique*, 339(2):194–207, 2011. ISSN 1631-0721. doi: https://doi.org/10.1016/j.crme.2010.12.006. URL `https://www.sciencedirect.com/science/article/pii/S1631072110002160`. High Performance Computing.

Vadym Aizinger and Clint Dawson. A discontinuous Galerkin method for two-dimensional flow and transport in shallow water. *Advances in Water Resources*, 25(1):67–84, 2002. ISSN 0309-1708. doi: https://doi.org/10.1016/S0309-1708(01)00019-7. URL `https://www.sciencedirect.com/science/article/pii/S0309170801000197`.

Vadym Aizinger, Adam Kosík, Dmitri Kuzmin, and Balthasar Reuter. Anisotropic Slope Limiting for Discontinuous Galerkin Methods. *International Journal for Numerical Methods in Fluids*, 84(9):543–565, 2017. doi: 10.1002/fld.4360. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4360`.

AMD. Advanced Micro Devices Demo, 2024. URL `https://readthedocs.com/projects/advanced-micro-devices-demo/`. Accessed: 5 Feb 2024.

Alex Apotsos, Mark Buckley, Guy Gelfenbaum, Bruce Jaffe, and Deepak Vatvani. Nearshore Tsunami Inundation Model Validation: Toward Sediment Transport Applications. *Pure and Applied Geophysics*, 168(11):2097–2119, 2011. ISSN 1420-9136. doi: 10.1007/s00024-011-0291-5. URL `https://doi.org/10.1007/s00024-011-0291-5`.

M. E. M. Arcos and Randall J. LeVeque. Validating Velocities in the GeoClaw Tsunami Model

Using Observations near Hawaii from the 2011 Tohoku Tsunami. *Pure and Applied Geophysics*, 172(3):849–867, 03 2015. ISSN 1420-9136. doi: 10.1007/s00024-014-0980-y. URL `https://doi.org/10.1007/s00024-014-0980-y`.

Luca Arpaia and Mario Ricchiuto. r-adaptation for Shallow Water flows: conservation, well balancedness, efficiency. *Computers and Fluids*, 160:175–203, 2018. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2017.10.026. URL `https://www.sciencedirect.com/science/article/pii/S0045793017303900`.

Luca Arpaia, Mario Ricchiuto, Andrea Gilberto Filippini, and Rodrigo Pedreros. An efficient covariant frame for the spherical shallow water equations: Well balanced DG approximation and application to tsunami and storm surge. *Ocean Modelling*, 169:101915, 2022. doi: 10.1016/j.ocemod.2021.101915. URL `https://www.sciencedirect.com/science/article/pii/S1463500321001682`.

F. Aureli, A. Maranzoni, P. Mignosa, and C. Ziveri. A weighted surface-depth gradient method for the numerical integration of the 2D shallow water equations with topography. *Advances in Water Resources*, 31(7):962–974, 2008. ISSN 0309-1708. doi: 10.1016/j.advwatres.2008.03.005. URL `https://www.sciencedirect.com/science/article/pii/S0309170808000468`.

Janice Lynn Ayog, Georges Kesserwani, James Shaw, Mohammad Kazem Sharifian, and Domenico Bau. Second-order discontinuous Galerkin flood model: Comparison with industry-standard finite volume models. *Journal of Hydrology*, 594:125924, 2021. ISSN 0022-1694. doi: 10.1016/j.jhydrol.2020.125924. URL `https://www.sciencedirect.com/science/article/pii/S0022169420313858`.

Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Texts in Computational Science and Engineering. Springer Berlin, Heidelberg, 1 edition, 2013. doi: 10.1007/978-3-642-31046-1.

Michael Bader, Christian Böck, Johannes Schwaiger, and Csaba Vigh. Dynamically Adaptive Simulations with Minimal Memory Requirement—Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal on Scientific Computing*, 32(1):212–228, 2010. doi: 10.1137/080728871. URL `https://doi.org/10.1137/080728871`.

Paul D. Bates, Matthew S. Horritt, and Timothy J. Fewtrell. A simple inertial formulation of the shallow water equations for efficient two-dimensional flood inundation modelling. *Journal of Hydrology*, 387(1):33–45, 2010. ISSN 0022-1694. doi: https://doi.org/10.1016/j.jhydrol.2010.03.027. URL `https://www.sciencedirect.com/science/article/pii/S0022169410001538`.

P.D Bates and A.P.J De Roo. A simple raster-based model for flood inundation simulation. *Journal of Hydrology*, 236(1):54–77, 2000. ISSN 0022-1694. doi: 10.1016/S0022-1694(00)00278-X. URL `https://www.sciencedirect.com/science/article/pii/S002216940000278X`.

David Beckingsale, Wayne Gaudin, Andrew Herdman, and Stephen Jarvis. Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units. In *2015 44th International Conference on Parallel Processing*, pages 61–70, 2015. doi: 10.1109/ICPP.2015.15.

Lorenzo Begnudelli and Brett F. Sanders. Conservative Wetting and Drying Methodology for Quadrilateral Grid Finite-Volume Models. *Journal of Hydraulic Engineering*, 133(3):312–322, 2007. doi: 10.1061/(ASCE)0733-9429(2007)133:3(312).

Lorenzo Begnudelli, Brett F. Sanders, and Scott F. Bradford. Adaptive Godunov-Based Model for Flood Simulation. *Journal of Hydraulic Engineering*, 134(6):714–725, 2008. doi: 10.1061/(ASCE)0733-9429(2008)134:6(714). URL `https://ascelibrary.org/doi/abs/10.1061/(ASCE)0733-9429(2008)134:6(714)`.

Nicole Beisiegel, Stefan Vater, Jörn Behrens, and Frédéric Dias. An adaptive discontinuous Galerkin method for the simulation of hurricane storm surge. *Ocean Dynamics*, 70(5):641–666, 2020. ISSN 1616-7228. doi: 10.1007/s10236-020-01352-w. URL `https://doi.org/10.1007/s10236-020-01352-w`.

Marsha J. Berger, David L. George, Randall J. LeVeque, and Kyle T. Mandli. The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources*, 34(9):1195–1206, 2011. ISSN 0309-1708. doi: 10.1016/j.advwatres.2011.02.016. URL `https://www.sciencedirect.com/science/article/pii/S0309170811000480`. New Computational Methods and Software Tools.

M.J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989. doi: 10.1016/0021-9991(89)90035-1. URL `https://www.sciencedirect.com/science/article/pii/0021999189900351`.

Paul-Emile Bernard, Nicolas Chevaugeon, Vincent Legat, Eric Deleersnijder, and Jean-François Remacle. High-order h-adaptive discontinuous Galerkin methods for ocean modelling. *Ocean Dynamics*, 57(2):109–121, 2007. ISSN 1616-7228. doi: 10.1007/s10236-006-0093-y. URL `https://doi.org/10.1007/s10236-006-0093-y`.

Sébastien Blaise and Amik St-Cyr. A Dynamic hp-Adaptive Discontinuous Galerkin Method for Shallow-Water Flows on the Sphere with Application to a Global Tsunami Simulation. *Monthly Weather Review*, 140(3):978–996, 2012. doi: 10.1175/MWR-D-11-00038.1. URL `https://journals.ametsoc.org/view/journals/mwre/140/3/mwr-d-11-00038.1.xml`.

Sébastien Blaise, Amik St-Cyr, Dimitri Mavriplis, and Brian Lockwood. Discontinuous Galerkin unsteady discrete adjoint method for real-time efficient tsunami simulations. *Journal of Computational Physics*, 232(1):416–430, 2013. doi: 10.1016/j.jcp.2012.08.022. URL `https://www.sciencedirect.com/science/article/pii/S0021999112004688`.

Onno Bokhove. Flooding and Drying in Discontinuous Galerkin Finite-Element Discretizations of Shallow-Water Equations. Part 1: One Dimension. *Journal of Scientific Computing*, 22(1):47–82, 2005. ISSN 1573-7691. doi: 10.1007/s10915-004-4136-6. URL `https://doi.org/10.1007/s10915-004-4136-6`.

Boris Bonev, Jan S. Hesthaven, Francis X. Giraldo, and Michal A. Kopera. Discontinuous Galerkin scheme for the spherical shallow water equations with applications to tsunami modeling and prediction. *Journal of Computational Physics*, 362:425–448, 2018. doi: 10.1016/j.jcp.2018.02.008. URL `https://www.sciencedirect.com/science/article/pii/S0021999118300846`.

Jose C. Borrero, Randall J. LeVeque, S. Dougal Greer, S O'neill, and Brisa N. Davis. Observations and modelling of tsunami currents at the port of Tauranga, New Zealand. 2015. URL `https://api.semanticscholar.org/CorpusID:134598338`.

Kolja Brix, Silvia Sorana Melian, Siegfried Müller, and Gero Schieffer. Parallelisation of Multiscale-Based Grid Adaptation using Space-Filling Curves. *ESAIM: Proc.*, 29:108–129, 2009. doi: 10.1051/proc/2009058. URL `https://doi.org/10.1051/proc/2009058`.

Maurizio Brocchini and Nicholas Dodd. Nonlinear Shallow Water Equation Modeling for Coastal Engineering. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 134(2):104–120, 2008. doi: 10.1061/(ASCE)0733-950X(2008)134:2(104). URL `https://ascelibrary.org/doi/abs/10.1061/(ASCE)0733-950X(2008)134:2(104)`.

Donatella Bruno, Francesca De Serio, and Michele Mossa. The FUNWAVE model application and its validation using laboratory data. *Coastal Engineering*, 56(7):773–787, 2009. ISSN 0378-3839. doi: https://doi.org/10.1016/j.coastaleng.2009.02.001. URL `https://www.sciencedirect.com/science/article/pii/S0378383909000234`.

S. R. Brus, D. Wirasaet, J. J. Westerink, and C. Dawson. Performance and Scalability Improvements for Discontinuous Galerkin Solutions to Conservation Laws on Unstructured Grids. *Journal of Scientific Computing*, 70(1):210–242, 2017. ISSN 1573-7691. doi: 10.1007/s10915-016-0249-y. URL `https://doi.org/10.1007/s10915-016-0249-y`.

Shintaro Bunya, Ethan J. Kubatko, Joannes J. Westerink, and Clint Dawson. A wetting and drying treatment for the Runge–Kutta discontinuous Galerkin solution to the shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 198(17):1548–1562, 2009. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2009.01.008. URL `https://www.sciencedirect.com/science/article/pii/S0045782509000383`.

Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-Scale AMR. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2010. doi: 10.1109/SC.2010.25.

Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011. doi: 10.1137/100791634. URL `https://doi.org/10.1137/100791634`.

Andreas Buttinger-Kreuzhuber, Artem Konev, Zsolt Horváth, Daniel Cornel, Ingo Schwerdorf, Günter Blöschl, and Jürgen Waser. An integrated GPU-accelerated modeling framework for high-resolution simulations of rural and urban flash floods. *Environmental Modelling and Software*, 156:105480, 2022. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2022.105480. URL `https://www.sciencedirect.com/science/article/pii/S1364815222001839`.

Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231 (7):2825–2839, 2012. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2011.12.024. URL `https://www.sciencedirect.com/science/article/pii/S0021999111007364`.

Tomas Carlotto, Pedro Luiz Borges Chaffe, Camyla Innocente dos Santos, and Seungsoo Lee. SW2D-GPU: A two-dimensional shallow water model accelerated by GPGPU. *Environmental Modelling and Software*, 145:105205, 2021. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2021.105205. URL `https://www.sciencedirect.com/science/article/pii/S1364815221002474`.

Cristóbal E. Castro, Jörn Behrens, and Christian Pelties. Optimization of the ADER-DG method in GPU applied to linear hyperbolic PDEs. *International Journal for Numerical Methods in Fluids*, 81(4):195–219, 2016. doi: 10.1002/fld.4179. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4179`.

M.J. Castro Díaz, E.D. Fernández-Nieto, A.M. Ferreiro, and C. Parés. Two-dimensional sediment transport models in shallow water equations. A second order finite volume approach on unstructured meshes. *Computer Methods in Applied Mechanics and Engineering*, 198(33):2520–2538, 2009. ISSN 0045-7825. doi: 10.1016/j.cma.2009.03.001. URL `https://www.sciencedirect.com/science/article/pii/S0045782509001169`.

D. Caviedes-Voullième, M. Morales-Hernández, M. R. Norman, and I. Özgen-Xian. SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics. *Geoscientific Model Development*, 16 (3):977–1008, 2023. doi: 10.5194/gmd-16-977-2023. URL `https://gmd.copernicus.org/articles/16/977/2023/`.

Daniel Caviedes-Voullième and Georges Kesserwani. Benchmarking a multiresolution discontinuous Galerkin shallow water model: Implications for computational hydraulics. *Advances in Water Resources*, 86:14–31, 2015. doi: 10.1016/j.advwatres.2015.09.016. URL `https://www.sciencedirect.com/science/article/pii/S0309170815002237`.

Daniel Caviedes-Voullième, Nils Gerhard, Aleksey Sikstel, and Siegfried Müller. Multiwavelet-based mesh adaptivity with Discontinuous Galerkin schemes: Exploring 2D shallow water problems. *Advances in Water Resources*, 138:103559, 2020. doi: 10.1016/j.advwatres.2020.103559. URL `https://www.sciencedirect.com/science/article/pii/S0309170819309121`.

Floyd M. Chitalu, Christophe Dubach, and Taku Komura. Bulk-synchronous parallel simultaneous BVH traversal for collision detection on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357050. doi: 10.1145/3190834.3190848. URL `https://doi.org/10.1145/3190834.3190848`.

Alovya Ahmed Chowdhury, Georges Kesserwani, Charles Rougé, and Paul Richmond. GPU-parallelisation of Haar wavelet-based grid resolution adaptation for fast finite volume modelling: application to shallow water flows. *Journal of Hydroinformatics*, 25(4):1210–1234, 06 2023. ISSN 1464-7141. doi: 10.2166/hydro.2023.154. URL `https://doi.org/10.2166/hydro.2023.154`.

Clint Dawson, Ethan J. Kubatko, Joannes J. Westerink, Corey Trahan, Christopher Mirabito, Craig Michoski, and Nishant Panda. Discontinuous Galerkin methods for modeling Hurricane storm surge. *Advances in Water Resources*, 34(9):1165–1176, 2011. ISSN 0309-1708. doi: 10.1016/j.advwatres.2010.11.004. URL `https://www.sciencedirect.com/science/article/pii/S0309170810002137`. New Computational Methods and Software Tools.

Susanna Dazzi, Renato Vacondio, and Paolo Mignosa. Internal boundary conditions for a GPU-accelerated 2D shallow water model: Implementation and applications. *Advances in Water Resources*, 137:103525, 2020. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2020.103525. URL `https://www.sciencedirect.com/science/article/pii/S0309170819309157`.

M. de la Asunción and M.J. Castro. Simulation of tsunamis generated by landslides using adaptive mesh refinement on GPU. *Journal of Computational Physics*, 345:91–110, 2017. ISSN 0021-9991. doi: 10.1016/j.jcp.2017.05.016. URL `https://www.sciencedirect.com/science/article/pii/S0021999117303856`.

Ralf Deiterding, Margarete Oliveira Domingues, and Kai Schneider. Multiresolution analysis as a criterion for effective dynamic mesh adaptation – A case study for Euler equations in the SAMR framework AMROC. *Computers & Fluids*, 205:104583, 2020. doi: 10.1016/j.compfluid.2020.104583. URL `https://www.sciencedirect.com/science/article/pii/S0045793020301559`.

P. Delandmeter, J. Lambrechts, V. Legat, V. Vallaeys, J. Naithani, W. Thiery, J.-F. Remacle, and E. Deleersnijder. A fully consistent and conservative vertically adaptive coordinate system for SLIM 3D v0.4 with an application to the thermocline oscillations of Lake Tanganyika. *Geoscientific Model Development*, 11(3):1161–1179, 2018. doi: 10.5194/gmd-11-1161-2018. URL `https://gmd.copernicus.org/articles/11/1161/2018/`.

Vincent Delmas and Azzedine Soulaïmani. Multi-GPU implementation of a time-explicit finite volume solver using CUDA and a CUDA-Aware version of OpenMPI with application to shallow water flows. *Computer Physics Communications*, 271:108190, 2022. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2021.108190. URL `https://www.sciencedirect.com/science/article/pii/S0010465521003027`.

Margarete Oliveira Domingues, Ralf Deiterding, Muller Moreira Lopes, Anna Karina Fontes Gomes, Odim Mendes, and Kai Schneider. Wavelet-based parallel dynamic mesh adaptation for magnetohydrodynamics in the AMROC framework. *Computers & Fluids*, 190:374–381, 2019. doi: 10.1016/j.compfluid.2019.06.025. URL `https://www.sciencedirect.com/science/article/pii/S0045793018308405`.

Rosa Donat, M. Carmen Martí, Anna Martínez-Gavara, and Pep Mulet. Well-Balanced Adaptive Mesh Refinement for shallow water flows. *Journal of Computational Physics*, 257:937–953, 2014. doi: 10.1016/j.jcp.2013.09.032. URL `https://www.sciencedirect.com/science/article/pii/S0021999113006463`.

Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Löffler, Brian O'Shea, Erik Schnetter, Brian Van Straalen, and Klaus Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, 2014. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2014.07.001. URL `https://www.sciencedirect.com/science/article/pii/S0743731514001178`. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Anshu Dubey, Martin Berzins, Carsten Burstedde, Michael L. Norman, Didem Unat, and Mohammed Wahib. Structured Adaptive Mesh Refinement Adaptations to Retain Performance Portability With Increasing Heterogeneity. *Computing in Science & Engineering*, 23(5):62–66, 2021. doi: 10.1109/MCSE.2021.3099603.

Daniel J. Dunning, Robert W. Robey, Jeffery A. Kuehn, and Jeanine Cook. A Performance Analysis of Phantom-Cell Adaptive Mesh Refinement on CPUs and GPUs. *Journal of Computing and Information Science in Engineering*, 21(1):011011, 12 2020. ISSN 1530-9827. doi: 10.1115/1.4048717. URL `https://doi.org/10.1115/1.4048717`.

I. Echeverribar, M. Morales-Hernández, P. Brufau, and P. García-Navarro. 2D numerical simulation of unsteady flows for large scale floods prediction in real time. *Advances in Water Resources*, 134:103444, 2019. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2019.103444. URL `https://www.sciencedirect.com/science/article/pii/S0309170819304786`.

Thomas Engels, Kai Schneider, Julius Reiss, and Marie Farge. A Wavelet-Adaptive Method for Multiscale Simulation of Turbulent Flows in Flying Insects. *Communications in Computational Physics*, 30(4):1118–1149, 2021. ISSN 1991-7120. doi: https://doi.org/10.4208/cicp.OA-2020-0246.

A. Ern, S. Piperno, and K. Djadel. A well-balanced Runge–Kutta discontinuous Galerkin method for the shallow-water equations with flooding and drying. *International Journal for Numerical Methods in Fluids*, 58(1):1–25, 2008. doi: https://doi.org/10.1002/fld.1674. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1674`.

C. Eskilsson. An hp-adaptive discontinuous Galerkin method for shallow water flows. *International Journal for Numerical Methods in Fluids*, 67(11):1605–1623, 2011. doi: https://doi.org/10.1002/fld.2434. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.2434`.

Chaulio R. Ferreira and Michael Bader. Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350624. doi: 10.1145/3093172.3093237.

B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, nov 2000. doi: 10.1086/317361. URL `https://dx.doi.org/10.1086/317361`.

Rajesh Gandham, David Medina, and Timothy Warburton. GPU Accelerated Discontinuous Galerkin Methods for Shallow Water Equations. *Communications in Computational Physics*, 18(1):37–64, 2015. doi: 10.4208/cicp.070114.271114a. URL `https://doi.org/10.4208/cicp.070114.271114a`.

Shuang Gao. HIGH RESOLUTION NUMERICAL MODELLING OF TSUNAMI INUNDATION USING QUADTREE METHOD AND GPU ACCELERATION. 2020. URL `https://api.semanticscholar.org/CorpusID:221800432`.

M. L. Garcia and T. L. Popiolek. Adaptive Mesh Refinement in the Dam-Break Problems. *Blucher Mechanical Engineering Proceedings*, 1(1), 2014. URL `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6fa83ddb28a4d4b19a7ee165538bea1da65fd64d`.

R. Gayathri, Prasad K. Bhaskaran, and Felix Jose. Coastal inundation research: an overview of the process. *Current Science*, 112(2):267–278, 2017. ISSN 00113891. URL `http://www.jstor.org/stable/24912354`.

D. L. George. Adaptive finite volume methods with well-balanced Riemann solvers for modeling floods in rugged terrain: Application to the Malpasset dam-break flood (France, 1959). *International Journal for Numerical Methods in Fluids*, 66(8):1000–1018, 2011. doi: https://doi.org/10.1002/fld.2298. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.2298`.

David George. Finite volume methods and adaptive refinement for tsunami propagation and inundation. 2006.

Nils Gerhard, Daniel Caviedes-Voullième, Siegfried Müller, and Georges Kesserwani. Multiwavelet-based grid adaptation with discontinuous Galerkin schemes for shallow water equations. *Journal of Computational Physics*, 301:265–288, 2015a. doi: 10.1016/j.jcp.2015.08.030. URL `https://www.sciencedirect.com/science/article/pii/S0021999115005574`.

Nils Gerhard, Francesca Iacono, Georg May, Siegfried Müller, and Roland Schäfer. A High-Order Discontinuous Galerkin Discretization with Multiwavelet-Based Grid Adaptation for Compressible Flows. *Journal of Scientific Computing*, 62(1):25–52, 2015b. ISSN 1573-7691. doi: 10.1007/s10915-014-9846-9. URL `https://doi.org/10.1007/s10915-014-9846-9`.

Mohammad A. Ghazizadeh, Abdolmajid Mohammadian, and Alexander Kurganov. An adaptive well-balanced positivity preserving central-upwind scheme on quadtree grids for shallow water equations. *Computers & Fluids*, 208:104633, 2020. doi: 10.1016/j.compfluid.2020.104633. URL `https://www.sciencedirect.com/science/article/pii/S004579302030205X`.

Rabih Ghostine, Craig Kapfer, Viswanathan Kannan, and Ibrahim Hoteit. Flood Modeling in Urban Area Using a Well-Balanced Discontinuous Galerkin Scheme on Unstructured Triangular Grids. *International Journal of Mathematical and Computational Sciences*, 13(2):19–29, 2019. ISSN eISSN: 1307-6892. URL `https://publications.waset.org/vol/146`.

Thomas Gillis and Wim M. van Rees. MURPHY—A Scalable Multiresolution Framework for Scientific Computing on 3D Block-Structured Collocated Grids. *SIAM Journal on Scientific Computing*, 44(5):C367–C398, 2022. doi: 10.1137/21M141676X. URL `https://doi.org/10.1137/21M141676X`.

Andrew Giuliani and Lilia Krivodonova. Adaptive mesh refinement on graphics processing units for applications in gas dynamics. *Journal of Computational Physics*, 381:67–90, 2019. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2018.12.019. URL `https://www.sciencedirect.com/science/article/pii/S0021999118308155`.

Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323789. doi: 10.1145/2503210.2503223. URL `https://doi.org/10.1145/2503210.2503223`.

Geovanny Gordillo, Mario Morales-Hernández, I. Echeverribar, Javier Fernández-Pato, and Pilar García-Navarro. A GPU-based 2D shallow water quality model. *Journal of Hydroinformatics*, 22(5):1182–1197, 07 2020. ISSN 1464-7141. doi: 10.2166/hydro.2020.030. URL `https://doi.org/10.2166/hydro.2020.030`.

J. M. Greenberg and A. Y. Leroux. A Well-Balanced Scheme for the Numerical Processing of Source Terms in Hyperbolic Equations. *SIAM Journal on Numerical Analysis*, 33(1):1–16, 1996. doi: 10.1137/0733001. URL `https://doi.org/10.1137/0733001`.

Ernesto Guerrero Fernández, Manuel Jesús Castro-Díaz, and Tomás Morales de Luna. A Second-Order Well-Balanced Finite Volume Scheme for the Multilayer Shallow Water Model with Variable Density. *Mathematics*, 8(5), 2020. ISSN 2227-7390. doi: 10.3390/math8050848. URL `https://www.mdpi.com/2227-7390/8/5/848`.

K. Guo, M. Guan, and D. Yu. Urban surface water flood modelling – a comprehensive review of current models and future challenges. *Hydrology and Earth System Sciences*, 25(5):2843–2860, 2021. doi: 10.5194/hess-25-2843-2021. URL `https://hess.copernicus.org/articles/25/2843/2021/`.

John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, may 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL `https://doi.org/10.1145/42411.42415`.

Mahya Hajihassanpour, Boris Bonev, and Jan S. Hesthaven. A comparative study of earthquake source models in high-order accurate tsunami simulations. *Ocean Modelling*, 141:101429, 2019.

doi: 10.1016/j.ocemod.2019.101429. URL `https://www.sciencedirect.com/science/article/pii/S1463500318302658`.

Dilshad A. Haleem, Georges Kesserwani, and Daniel Caviedes-Voullième. Haar wavelet-based adaptive finite volume shallow water solver. *Journal of Hydroinformatics*, 17(6):857–873, 2015. doi: 10.2166/hydro.2015.039. URL `https://doi.org/10.2166/hydro.2015.039`.

Shawn Hargreaves. Announcing DirectX 12 Ultimate, 2020. URL `https://devblogs.microsoft.com/directx/announcing-directx-12-ultimate/`. Accessed: 5 Feb 2024.

J. Horrillo, A. Wood, G.-B Kim, and A. Parambath. A simplified 3-D Navier-Stokes numerical model for landslide-tsunami: Application to the Gulf of Mexico. *Journal of Geophysical Research: Oceans*, 118(12):6934–6950, 2013. doi: https://doi.org/10.1002/2012JC008689. URL `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2012JC008689`.

Jingming Hou, Qiuhua Liang, Franz Simons, and Reinhard Hinkelmann. A stable 2D unstructured shallow flow model for simulations of wetting and drying over rough terrains. *Computers and Fluids*, 82:132–147, 2013. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2013.04.015. URL `https://www.sciencedirect.com/science/article/pii/S0045793013001552`.

Jingming Hou, Qiuhua Liang, Hongbin Zhang, and Reinhard Hinkelmann. An efficient unstructured MUSCL scheme for solving the 2D shallow water equations. *Environmental Modelling & Software*, 66:131–152, 2015. ISSN 1364-8152. doi: 10.1016/j.envsoft.2014.12.007. URL `https://www.sciencedirect.com/science/article/pii/S1364815214003648`.

Jingming Hou, Run Wang, Qiuhua Liang, Zhanbin Li, Mian Song Huang, and Reihnard Hinkelmann. Efficient surface water flow simulation on static Cartesian grid with local refinement according to key topographic features. *Computers & Fluids*, 176:117–134, 2018. doi: 10.1016/j.compfluid.2018.03.024. URL `https://www.sciencedirect.com/science/article/pii/S0045793018301300`.

Nune Hovhannisyan, Siegfried Müller, and Roland Schäfer. ADAPTIVE MULTIRESOLUTION DISCONTINUOUS GALERKIN SCHEMES FOR CONSERVATION LAWS. *Mathematics of*

Computation, 83(285):113–151, 2014. ISSN 00255718, 10886842. URL `http://www.jstor.org/stable/24488204`.

Yuxin Huang, Ningchuan Zhang, and Yuguo Pei. Well-Balanced Finite Volume Scheme for Shallow Water Flooding and Drying Over Arbitrary Topography. *Engineering Applications of Computational Fluid Mechanics*, 7(1):40–54, 2013. doi: 10.1080/19942060.2013.11015452. URL `https://doi.org/10.1080/19942060.2013.11015452`.

Neil M. Hunter, Matthew S. Horritt, Paul D. Bates, Matthew D. Wilson, and Micha G.F. Werner. An adaptive time step solution for raster-based storage cell modelling of floodplain inundation. *Advances in Water Resources*, 28(9):975–991, 2005. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2005.03.007. URL `https://www.sciencedirect.com/science/article/pii/S0309170805000850`.

Fumihiko Imamura, Ahmet Cevdet Yalciner, and Gulizar Ozyurt. *TSUNAMI MODELLING MANUAL (TUNAMI model)*. 2006. URL `https://www.tsunami.irides.tohoku.ac.jp/media/files/_u/project/manual-ver-3_1.pdf`.

Woochang Jeong, Jae-Seon Yoon, and Yong-Sik Cho. Numerical study on effects of building groups on dam-break flow in urban areas. *Journal of Hydro-environment Research*, 6(2):91–99, 2012. ISSN 1570-6443. doi: https://doi.org/10.1016/j.jher.2012.01.001. URL `https://www.sciencedirect.com/science/article/pii/S1570644312000020`. Special Issue on Ecohydraulics: Recent Research and Applications.

Hua Ji, Fue-Sang Lien, and Fan Zhang. A GPU-accelerated adaptive mesh refinement for immersed boundary methods. *Computers & Fluids*, 118:131–147, 2015. doi: 10.1016/j.compfluid.2015.06.011. URL `https://www.sciencedirect.com/science/article/pii/S0045793015001942`.

Dylan Jude, Jayanarayanan Sitaraman, and Andrew Wissink. An Octree-Based, Cartesian Navier–Stokes Solver for Modern Cluster Architectures. *The Journal of Supercomputing*, 78(9):11409–11440, 2022. ISSN 1573-0484. doi: 10.1007/s11227-022-04324-7. URL `https://doi.org/10.1007/s11227-022-04324-7`.

T. Kärnä, S. C. Kramer, L. Mitchell, D. A. Ham, M. D. Piggott, and A. M. Baptista. Thetis coastal ocean model: discontinuous Galerkin discretization for the three-dimensional hydrostatic equations. *Geoscientific Model Development*, 11(11):4359–4382, 2018. doi: 10.5194/gmd-11-4359-2018. URL `https://gmd.copernicus.org/articles/11/4359/2018/`.

Tero Karras. Thinking Parallel, Part II: Tree Traversal on the GPU, 2012. URL `https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/`.

F. Keinert. *Wavelets and Multiwavelets*. Chapman and Hall/CRC, 1st edition, 2003. doi: 10.1201/9780203011591.

John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language, 2004.

Georges Kesserwani. Topography discretization techniques for Godunov-type shallow water numerical models: a comparative study. *Journal of Hydraulic Research*, 51(4):351–367, 2013. doi: 10.1080/00221686.2013.796574. URL `https://doi.org/10.1080/00221686.2013.796574`.

Georges Kesserwani and Qiuhua Liang. A discontinuous Galerkin algorithm for the two-dimensional shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 199(49):3356–3368, 2010. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2010.07.007. URL `https://www.sciencedirect.com/science/article/pii/S0045782510002197`.

Georges Kesserwani and Qiuhua Liang. Locally Limited and Fully Conserved RKDG2 Shallow Water Solutions with Wetting and Drying. *Journal of Scientific Computing*, 50(1):120–144, 2012a. ISSN 1573-7691. doi: 10.1007/s10915-011-9476-4. URL `https://doi.org/10.1007/s10915-011-9476-4`.

Georges Kesserwani and Qiuhua Liang. Influence of Total-Variation-Diminishing Slope Limiting on Local Discontinuous Galerkin Solutions of the Shallow Water Equations. *Journal of Hydraulic Engineering*, 138(2):216–222, 2012b. doi: 10.1061/(ASCE)HY.1943-7900.0000494.

Georges Kesserwani and Qiuhua Liang. Dynamically adaptive grid based discontinuous Galerkin shallow water model. *Advances in Water Resources*, 37:23–39, 2012c. ISSN 0309-1708.

doi: https://doi.org/10.1016/j.advwatres.2011.11.006. URL `https://www.sciencedirect.com/science/article/pii/S0309170811002181`.

Georges Kesserwani and Qiuhua Liang. RKDG2 shallow-water solver on non-uniform grids with local time steps: Application to 1D and 2D hydrodynamics. *Applied Mathematical Modelling*, 39(3):1317–1340, 2015. ISSN 0307-904X. doi: https://doi.org/10.1016/j.apm.2014.08.009. URL `https://www.sciencedirect.com/science/article/pii/S0307904X14004247`.

Georges Kesserwani and Mohammad Kazem Sharifian. (Multi)wavelets increase both accuracy and efficiency of standard Godunov-type hydrodynamic models: Robust 2D approaches. *Advances in Water Resources*, 144:103693, 2020. doi: 10.1016/j.advwatres.2020.103693. URL `https://www.sciencedirect.com/science/article/pii/S0309170820303079`.

Georges Kesserwani and Mohammad Kazem Sharifian. (Multi)wavelet-based Godunov-type simulators of flood inundation: Static versus dynamic adaptivity. *Advances in Water Resources*, 171:104357, 2023. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2022.104357. URL `https://www.sciencedirect.com/science/article/pii/S0309170822002202`.

Georges Kesserwani and Yueling Wang. Discontinuous Galerkin flood model formulation: Luxury or necessity? *Water Resources Research*, 50(8):6522–6541, 2014. doi: https://doi.org/10.1002/2013WR014906. URL `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2013WR014906`.

Georges Kesserwani, Qiuhua Liang, José Vazquez, and Robert Mosé. Well-balancing issues related to the RKDG2 scheme for the shallow water equations. *International Journal for Numerical Methods in Fluids*, 62(4):428–448, 2010. doi: https://doi.org/10.1002/fld.2027. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.2027`.

Georges Kesserwani, Daniel Caviedes-Voullième, Nils Gerhard, and Siegfried Müller. Multiwavelet discontinuous Galerkin h-adaptive shallow water model. *Computer Methods in Applied Mechanics and Engineering*, 294:56–71, 2015a. doi: 10.1016/j.cma.2015.05.016. URL `https://www.sciencedirect.com/science/article/pii/S0045782515001899`.

Georges Kesserwani, Daniel Caviedes-Voullième, Nils Gerhard, and Siegfried Müller. Multiwavelet discontinuous Galerkin h-adaptive shallow water model. *Computer Methods in Applied Mechanics and Engineering*, 294:56–71, 2015b. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2015.05.016. URL `https://www.sciencedirect.com/science/article/pii/S0045782515001899`.

Georges Kesserwani, Janice Lynn Ayog, and Domenico Bau. Discontinuous Galerkin formulation for 2D hydrodynamic modelling: Trade-offs between theoretical complexity and practical convenience. *Computer Methods in Applied Mechanics and Engineering*, 342:710–741, 2018. ISSN 0045-7825. doi: 10.1016/j.cma.2018.08.003. URL `https://www.sciencedirect.com/science/article/pii/S004578251830389X`.

Georges Kesserwani, James Shaw, Mohammad K Sharifian, Domenico Bau, Christopher J Keylock, Paul D Bates, and Jennifer K Ryan. (Multi)wavelets increase both accuracy and efficiency of standard Godunov-type hydrodynamic models. *Advances in Water Resources*, 129:31–55, 2019. doi: 10.1016/j.advwatres.2019.04.019. URL `https://www.sciencedirect.com/science/article/pii/S0309170819301770`.

Georges Kesserwani, Janice Lynn Ayog, Mohammad Kazem Sharifian, and Domenico Baú. Shallow-Flow Velocity Predictions Using Discontinuous Galerkin Solutions. *Journal of Hydraulic Engineering*, 149(5):04023008, 2023. doi: 10.1061/JHEND8.HYENG-13244. URL `https://ascelibrary.org/doi/abs/10.1061/JHEND8.HYENG-13244`.

A.M Khokhlov. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics*, 143(2):519–543, 1998. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.1998.9998. URL `https://www.sciencedirect.com/science/article/pii/S0021999198999983`.

Khronos. OpenCL Guide, 2024. URL `https://github.com/KhronosGroup/OpenCL-Guide`. Accessed: 5 Feb 2024.

B A Korneev, A V Zakirov, and V D Levchenko. Simulation of shock-body interactions using the Runge–Kutta discontinuous Galerkin method with adaptive mesh refinement implemented on graphic accelerators. *Journal of Physics: Conference Series*, 1787(1):012042, feb 2021. doi: 10.1088/1742-6596/1787/1/012042. URL `https://dx.doi.org/10.1088/1742-6596/1787/1/012042`.

L. Krivodonova, J. Xin, J.-F. Remacle, N. Chevaugeon, and J.E. Flaherty. Shock detection and limiting with discontinuous Galerkin methods for hyperbolic conservation laws. *Applied Numerical Mathematics*, 48(3):323–338, 2004. ISSN 0168-9274. doi: https://doi.org/10.1016/j.apnum.2003.11.002. URL `https://www.sciencedirect.com/science/article/pii/S0168927403001831`. Workshop on Innovative Time Integrators for PDEs.

Lilia Krivodonova. Limiters for high-order discontinuous Galerkin methods. *Journal of Computational Physics*, 226(1):879–896, 2007. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2007.05.011. URL `https://www.sciencedirect.com/science/article/pii/S0021999107002136`.

Ethan J. Kubatko, Joannes J. Westerink, and Clint Dawson. hp Discontinuous Galerkin methods for advection dominated problems in shallow water flow. *Computer Methods in Applied Mechanics and Engineering*, 196(1):437–451, 2006. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2006.05.002. URL `https://www.sciencedirect.com/science/article/pii/S004578250600171X`.

Vijendra Kumar, Kul Vaibhav Sharma, Tommaso Caloiero, Darshan J. Mehta, and Karan Singh. Comprehensive Overview of Flood Modeling Approaches: A Review of Recent Advances. *Hydrology*, 10(7):141, 2023. ISSN 2306-5338. doi: 10.3390/hydrology10070141. URL `https://www.mdpi.com/2306-5338/10/7/141`.

Alexander Kurganov and Guergana Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Communications in Mathematical Sciences*, 5(1):133–160, 2007. ISSN 1539-6746. doi: 10.4310/CMS.2007.v5.n1.a6. URL `https://dx.doi.org/10.4310/CMS.2007.v5.n1.a6`.

Tuomas Kärnä, Benjamin de Brye, Olivier Gourgue, Jonathan Lambrechts, Richard Comblen, Vincent Legat, and Eric Deleersnijder. A fully implicit wetting–drying method for DG-FEM shallow water models, with an application to the Scheldt Estuary. *Computer Methods in*

Applied Mechanics and Engineering, 200(5):509–524, 2011. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2010.07.001. URL https://www.sciencedirect.com/science/article/pii/S0045782510002136.

BOINC Baker Laboratory. BOINC - Baker Laboratory CPU List. URL https://boinc.bakerlab.org/rosetta/cpu_list.php. Accessed: 5 Feb 2024.

Thorne Lay, Yoshiki Yamazaki, Charles J. Ammon, Kwok Fai Cheung, and Hiroo Kanamori. The 2011 Mw 9.0 off the Pacific coast of Tohoku Earthquake: Comparison of deep-water tsunami signals with finite-fault rupture model predictions. *Earth, Planets and Space*, 63(7):797–801, 2011. ISSN 1880-5981. doi: 10.5047/eps.2011.05.030. URL https://doi.org/10.5047/eps.2011.05.030.

Han Soo Lee. Tsunami Run-Up Modeling with Adaptive Mesh Refinement: Monai Valley Benchmark Test. All Days:ISOPE–I–17–191, 06 2017.

Randall J. LeVeque, David L. George, and Marsha J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011. doi: 10.1017/S0962492911000043.

Longxiang Li and Qinghe Zhang. Development of an efficient wetting and drying treatment for shallow-water modeling using the quadrature-free Runge-Kutta discontinuous Galerkin method. *International Journal for Numerical Methods in Fluids*, 93(2):314–338, 2021. doi: 10.1002/fld.4884. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4884.

Qiuhua Liang and Fabien Marche. Numerical resolution of well-balanced shallow water equations with complex source terms. *Advances in Water Resources*, 32(6):873–884, 2009. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2009.02.010. URL https://www.sciencedirect.com/science/article/pii/S0309170809000396.

Qiuhua Liang, Jingming Hou, and Reza Amouzgar. Simulation of Tsunami Propagation Using Adaptive Cartesian Grids. *Coastal Engineering Journal*, 57(4):1550016–1–1550016–30, 2015a. doi: 10.1142/S0578563415500163. URL https://doi.org/10.1142/S0578563415500163.

Qiuhua Liang, Jingming Hou, and Xilin Xia. Contradiction between the C-property and mass conservation in adaptive grid based shallow flow models: cause and solution. *International Journal for Numerical Methods in Fluids*, 78(1):17–36, 2015b. doi: 10.1002/fld.4005. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4005`.

Philip Liu, Seung Buhm Woo, and Yong Sik Cho. Computer Programs for Tsunami Propagation and Inundation. 1998. URL `https://tsunamiportal.nacse.org/documentation/COMCOT_tech.pdf`.

Charles Lohr. GPU-Based Parallel Stackless BVH Traversal for Animated Di stributed Ray Tracing. 2009. URL `https://api.semanticscholar.org/CorpusID:7430697`.

Kevin Lung, Eric Brown-Dymkoski, Victor Guerrero, Eric Doran, Ken Museth, Jo Balme, Bob Urberger, Andre Kessler, Stephen Jones, Billy Moses, and Anthony Crognale. Efficient Combustion Simulation via the Adaptive Wavelet Collocation Method. In *APS March Meeting 2016, abstract id. M1.167*, 2016.

Xisheng Luo, Luying Wang, Wei Ran, and Fenghua Qin. GPU accelerated cell-based adaptive mesh refinement on unstructured quadrilateral grid. *Computer Physics Communications*, 207: 114–122, 2016. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2016.05.018. URL `https://www.sciencedirect.com/science/article/pii/S0010465516301424`.

Patrick J. Lynett, Kara Gately, Rick Wilson, Luis Montoya, Diego Arcas, Betul Aytore, Yefei Bai, Jeremy D. Bricker, Manuel J. Castro, Kwok Fai Cheung, C. Gabriel David, Gozde Guney Dogan, Cipriano Escalante, José Manuel González-Vida, Stephan T. Grilli, Troy W. Heitmann, Juan Horrillo, Utku Kânoğlu, Rozita Kian, James T. Kirby, Wenwen Li, Jorge Macías, Dmitry J. Nicolsky, Sergio Ortega, Alyssa Pampell-Manis, Yong Sung Park, Volker Roeber, Naeimeh Sharghivand, Michael Shelby, Fengyan Shi, Babak Tehranirad, Elena Tolkova, Hong Kie Thio, Deniz Velioğlu, Ahmet Cevdet Yalçıner, Yoshiki Yamazaki, Andrey Zaytsev, and Y.J. Zhang. Inter-model analysis of tsunami-induced coastal currents. *Ocean Modelling*, 114:14–32, 2017. ISSN 1463-5003. doi: https://doi.org/10.1016/j.ocemod.2017.04.003. URL `https://www.sciencedirect.com/science/article/pii/S1463500317300513`.

Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000. ISSN 0010-4655. doi: https://doi.org/10.1016/S0010-4655(99)00501-9. URL `https://www.sciencedirect.com/science/article/pii/S0010465599005019`.

Jorge Macías, Manuel Castro, Sergio Ortega, Cipriano Escalante Sánchez, and José González Vida. Tsunami currents benchmarking results for Tsunami-HySEA, 05 2015.

Jorge Macías, Manuel Castro, Sergio Ortega, and José González-Vida. Performance assessment of Tsunami-HySEA model for NTHMP tsunami currents benchmarking. Field cases. *Ocean Modelling*, 152:101645, 2020a. ISSN 1463-5003. doi: 10.1016/j.ocemod.2020.101645. URL `https://www.sciencedirect.com/science/article/pii/S1463500320301475`.

Jorge Macías, Manuel J. Castro, and Cipriano Escalante. Performance assessment of the Tsunami-HySEA model for NTHMP tsunami currents benchmarking. Laboratory data. *Coastal Engineering*, 158:103667, 2020b. ISSN 0378-3839. doi: 10.1016/j.coastaleng.2020.103667. URL `https://www.sciencedirect.com/science/article/pii/S037838391830351X`.

Kyle T. Mandli and Clint N. Dawson. Adaptive mesh refinement for storm surge. *Ocean Modelling*, 75:36–50, 2014. doi: 10.1016/j.ocemod.2014.01.002. URL `https://www.sciencedirect.com/science/article/pii/S1463500314000031`.

Oliver Meister, Kaveh Rahnema, and Michael Bader. Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. *ACM Trans. Math. Softw.*, 43(3), sep 2016. ISSN 0098-3500. doi: 10.1145/2947668. URL `https://doi.org/10.1145/2947668`.

I Menshov and P Pavlukhin. GPU-native gas dynamic solver on octree-based AMR grids. *Journal of Physics: Conference Series*, 1640(1):012017, oct 2020. doi: 10.1088/1742-6596/1640/1/012017. URL `https://dx.doi.org/10.1088/1742-6596/1640/1/012017`.

Duane Merrill. CUB - CUDA Building Blocks, 2022. URL `https://nvlabs.github.io/cub/`. Accessed: 3 February 2024.

M. Morales-Hernández, M. B. Sharif, S. Gangrade, T. T. Dullo, S.-C. Kao, A. Kalyanapu, S. K. Ghafoor, K. J. Evans, E. Madadi-Kandjani, and B. R. Hodges. High-performance computing in water resources hydrodynamics. *Journal of Hydroinformatics*, 22(5):1217–1235, 03 2020. ISSN 1464-7141. doi: 10.2166/hydro.2020.163. URL `https://doi.org/10.2166/hydro.2020.163`.

M. Morales-Hernández, Md B. Sharif, A. Kalyanapu, S.K. Ghafoor, T.T. Dullo, S. Gangrade, S.-C. Kao, M.R. Norman, and K.J. Evans. TRITON: A Multi-GPU open source 2D hydrodynamic flood model. *Environmental Modelling and Software*, 141:105034, 2021. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2021.105034. URL `https://www.sciencedirect.com/science/article/pii/S1364815221000773`.

MPIForum. *MPI: A Message-Passing Interface Standard.* 2011. URL `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. Parallel Tree Traversal for Nearest Neighbor Query on the GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 113–122, 2016. doi: 10.1109/ICPP.2016.20.

U.C. Nkwunonwo, M. Whitworth, and B. Baily. A review of the current status of flood modelling for urban flood risk management in the developing countries. *Scientific African*, 7:e00269, 2020. ISSN 2468-2276. doi: 10.1016/j.sciaf.2020.e00269. URL `https://www.sciencedirect.com/science/article/pii/S2468227620300077`.

NVIDIA. NVIDIA A100 Data Sheet, 2021. URL `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf`. Accessed: 5 Feb 2024.

NVIDIA. *CUDA C++ Programming Guide.* 2023. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

OpenMP. *OpenMP Application Program Interface.* 1998. URL `https://www.openmp.org/wp-content/uploads/spec30.pdf`.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. doi: https://doi.org/10.1111/j.1467-8659.2007.01012.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01012.x`.

John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. doi: 10.1109/JPROC.2008.917757.

Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C. Stern, and Artem Cherkasov. The transformational role of GPU computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, March 2022. doi: 10.1038/s42256-022-00463-x. URL `https://doi.org/10.1038/s42256-022-00463-x`.

Hyoungsu Park, Daniel T. Cox, Patrick J. Lynett, Dane M. Wiebe, and Sungwon Shin. Tsunami inundation modeling in constructed environments: A physical and numerical comparison of free-surface elevation, velocity, and momentum flux. *Coastal Engineering*, 79:9–21, 2013. ISSN 0378-3839. doi: https://doi.org/10.1016/j.coastaleng.2013.04.002. URL `https://www.sciencedirect.com/science/article/pii/S0378383913000781`.

Junghyun Park, Jin-hee Yuk, WonK yun Joo, and Han Soo Lee. Wave Run-up Modeling with Adaptive Mesh Refinement (AMR) Method in the Busan Marine City during Typhoon Chaba (1618). *Journal of Coastal Research*, 91(SI):56–60, 08 2019. ISSN 0749-0208. doi: 10.2112/SI91-012.1. URL `https://doi.org/10.2112/SI91-012.1`.

Pavel Pavlukhin and Igor Menshov. On Defragmentation Algorithms for GPU-Native Octree-Based AMR Grids. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 235–244, Cham, 2021. Springer International Publishing. ISBN 978-3-030-86359-3.

K. Pons, F. Golay, and R. Marcer. Adaptive Mesh Refinement Method Applied to Shallow Water Model: A Mass Conservative Projection. In David Šimurda and Tomáš Bodnár, editors, *Topical Problems of Fluid Mechanics 2017*, pages 249–258, Prague, 2017. Institute of Thermomechanics of the Czech Academy of Sciences. ISBN 978-80-87012-61-1. doi: 10.14311/TPFM.2017.032. URL `https://doi.org/10.14311/TPFM.2017.032`.

S. Popinet. Adaptive modelling of long-distance wave propagation and fine-scale flooding during the Tohoku tsunami. *Natural Hazards and Earth System Sciences*, 12(4):1213–1227, 2012. doi: 10.5194/nhess-12-1213-2012. URL `https://nhess.copernicus.org/articles/12/1213/2012/`.

Stéphane Popinet. Quadtree-adaptive tsunami modelling. *Ocean Dynamics*, 61(9):1261–1285, 2011. ISSN 1616-7228. doi: 10.1007/s10236-011-0438-z. URL `https://doi.org/10.1007/s10236-011-0438-z`.

Stéphane Popinet and Graham Rickard. A tree-based solver for adaptive ocean modelling. *Ocean Modelling*, 16(3):224–249, 2007. ISSN 1463-5003. doi: 10.1016/j.ocemod.2006.10.002. URL `https://www.sciencedirect.com/science/article/pii/S1463500306000989`.

X. Qin, M. Motley, R. LeVeque, F. Gonzalez, and K. Mueller. A comparison of a two-dimensional depth-averaged flow model and a three-dimensional RANS model for predicting tsunami inundation and fluid forces. *Natural Hazards and Earth System Sciences*, 18(9):2489–2506, 2018. doi: 10.5194/nhess-18-2489-2018. URL `https://nhess.copernicus.org/articles/18/2489/2018/`.

Xinsheng Qin, Randall J. LeVeque, and Michael R. Motley. Accelerating an Adaptive Mesh Refinement Code for Depth-Averaged Flows Using GPUs. *Journal of Advances in Modeling Earth Systems*, 11(8):2606–2628, 2019. doi: 10.1029/2019MS001635. URL `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2019MS001635`.

Hari K Raghavan and Sathish S Vadhiyar. Adaptive executions of hyperbolic block-structured AMR applications on GPU systems. *The International Journal of High Performance Computing Applications*, 29(2):135–153, 2015. doi: 10.1177/1094342014545546. URL `https://doi.org/10.1177/1094342014545546`.

N. Rakowsky, A. Androsov, A. Fuchs, S. Harig, A. Immerz, S. Danilov, W. Hiller, and J. Schröter. Operational tsunami modelling with TsunAWI – recent developments and applications. *Natural*

Hazards and Earth System Sciences, 13(6):1629–1642, 2013. doi: 10.5194/nhess-13-1629-2013. URL `https://nhess.copernicus.org/articles/13/1629/2013/`.

Leonhard Rannabauer, Michael Dumbser, and Michael Bader. ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework. *Computers & Fluids*, 173:299–306, 2018. doi: 10.1016/j.compfluid.2018.01.031. URL `https://www.sciencedirect.com/science/article/pii/S0045793018300392`.

C Reis, J Figueiredo, S Clain, R Omira, M A Baptista, and J M Miranda. Comparison between MUSCL and MOOD techniques in a finite volume well-balanced code to solve SWE. The Tohoku-Oki, 2011 example. *Geophysical Journal International*, 216(2):958–983, 11 2018. ISSN 0956-540X. doi: 10.1093/gji/ggy472. URL `https://doi.org/10.1093/gji/ggy472`.

Jean-François Remacle, Sandra Soares Frazão, Xiangrong Li, and Mark S. Shephard. An adaptive discretization of shallow-water equations based on discontinuous Galerkin methods. *International Journal for Numerical Methods in Fluids*, 52(8):903–923, 2006. doi: https://doi.org/10.1002/fld.1204. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1204`.

Hans Sagan. *Space-Filling Curves*. Universitext. Springer New York, NY, 1 edition, 1994. doi: 10.1007/978-1-4612-0871-6.

Marcos Sanz-Ramos, David López-Gómez, Ernest Bladé, and Danial Dehghan-Souraki. A CUDA Fortran GPU-parallelised hydrodynamic tool for high-resolution and long-term eco-hydraulic modelling. *Environmental Modelling and Software*, 161:105628, 2023. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2023.105628. URL `https://www.sciencedirect.com/science/article/pii/S1364815223000142`.

Hsi-Yu Schive, John A ZuHone, Nathan J Goldbaum, Matthew J Turk, Massimo Gaspari, and Chin-Yu Cheng. gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability. *Monthly Notices of the Royal Astronomical Society*, 481(4):4815–4840, 09 2018a. ISSN 0035-8711. doi: 10.1093/mnras/sty2586. URL `https://doi.org/10.1093/mnras/sty2586`.

Hsi-Yu Schive, John A. ZuHone, Nathan J. Goldbaum, Matthew J. Turk, Massimo Gaspari, and

Chin-Yu Cheng. gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability. *Monthly Notices of the Royal Astronomical Society*, 481(4):4815–4840, 2018b. doi: 10.1093/mnras/sty2586. URL `https://doi.org/10.1093/mnras/sty2586`.

Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

A. N. Semakin and Y. Rastigejev. Optimized wavelet-based adaptive mesh refinement algorithm for numerical modeling of three-dimensional global-scale atmospheric chemical transport. *Quarterly Journal of the Royal Meteorological Society*, 146(729):1564–1574, 2020. doi: 10.1002/qj.3752. URL `https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.3752`.

M. K. Sharifian, G. Kesserwani, A. A. Chowdhury, J. Neal, and P. Bates. LISFLOOD-FP 8.1: new GPU-accelerated solvers for faster fluvial/pluvial flood simulations. *Geoscientific Model Development*, 16(9):2391–2413, 2023. doi: 10.5194/gmd-16-2391-2023. URL `https://gmd.copernicus.org/articles/16/2391/2023/`.

Mohammad Kazem Sharifian, Yousef Hassanzadeh, Georges Kesserwani, and James Shaw. Performance study of the multiwavelet discontinuous Galerkin approach for solving the Green-Naghdi equations. *International Journal for Numerical Methods in Fluids*, 90(10):501–521, 2019. doi: https://doi.org/10.1002/fld.4732. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4732`.

J. Shaw, G. Kesserwani, J. Neal, P. Bates, and M. K. Sharifian. LISFLOOD-FP 8.0: the new discontinuous Galerkin shallow-water solver for multi-core CPUs and GPUs. *Geoscientific Model Development*, 14(6):3577–3602, 2021. doi: 10.5194/gmd-14-3577-2021. URL `https://gmd.copernicus.org/articles/14/3577/2021/`.

Takashi Shimokawabe and Naoyuki Onodera. A High-Productivity Framework for Adaptive Mesh Refinement on Multiple GPUs. In João M. F. Rodrigues, Pedro J. S. Cardoso, Jânio Monteiro, Roberto Lam, Valeria V. Krzhizhanovskaya, Michael H. Lees, Jack J. Dongarra, and Peter M.A. Sloot, editors, *Computational Science – ICCS 2019*, pages 281–294, Cham, 2019. Springer International Publishing. ISBN 978-3-030-22734-0.

Mohammad Shirvani, Georges Kesserwani, and Paul Richmond. Agent-based simulator of dynamic flood-people interactions. *Journal of Flood Risk Management*, 14(2):e12695, 2021. doi: https://doi.org/10.1111/jfr3.12695. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/jfr3.12695.

Sandra Soares-Frazão and Yves Zech. Dam-break flow through an idealised city. *Journal of Hydraulic Research*, 46(5):648–658, 2008. doi: 10.3826/jhr.2008.3164. URL https://doi.org/10.3826/jhr.2008.3164.

Lixiang Song, Jianzhong Zhou, Qingqing Li, Xiaoling Yang, and Yongchuan Zhang. An unstructured finite volume model for dam-break floods with wet/dry fronts over complex topography. *International Journal for Numerical Methods in Fluids*, 67(8):960–980, 2011. doi: https://doi.org/10.1002/fld.2397. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.2397.

V. Soni, A. Hadjadj, O. Roussel, and G. Moebs. Parallel multi-core and multi-processor methods on point-value multiresolution algorithms for hyperbolic conservation laws. *Journal of Parallel and Distributed Computing*, 123:192–203, 2019. doi: 10.1016/j.jpdc.2018.09.016. URL https://www.sciencedirect.com/science/article/pii/S0743731518306981.

Xitong Sun, Georges Kesserwani, Mohammad Kazem Sharifian, and Virginia Stovin. Simulation of laminar to transitional wakes past cylinders with a discontinuous Galerkin inviscid shallow water model. *Journal of Hydraulic Research*, 61(5):631–650, 2023. doi: 10.1080/00221686.2023.2239750. URL https://doi.org/10.1080/00221686.2023.2239750.

C. E. Synolakis, E. N. Bernard, V. V. Titov, U. Kânoğlu, and F. I. González. Validation and Verification of Tsunami Numerical Models. *Pure and Applied Geophysics*, 165(11):2197–2228, 2008. ISSN 1420-9136. doi: 10.1007/s00024-004-0427-y. URL https://doi.org/10.1007/s00024-004-0427-y.

Martin L. Sætra, André R. Brodtkorb, and Knut-Andreas Lie. Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations. *Journal of Scientific Computing*, 63(1):23–48, 2015. ISSN 1573-7691. doi: 10.1007/s10915-014-9883-4. URL https://doi.org/10.1007/s10915-014-9883-4.

J. Teng, A.J. Jakeman, J. Vaze, B.F.W. Croke, D. Dutta, and S. Kim. Flood inundation modelling: A review of methods, recent advances and uncertainty analysis. *Environmental Modelling and Software*, 90:201–216, 2017. ISSN 1364-8152. doi: 10.1016/j.envsoft.2017.01.006. URL `https://www.sciencedirect.com/science/article/pii/S1364815216310040`.

R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement - A new high resolution called RAMSES. *A&A*, 385(1):337–364, 2002. doi: 10.1051/0004-6361:20011817. URL `https://doi.org/10.1051/0004-6361:20011817`.

Vasily Titov, Utku Kânoğlu, and Costas Synolakis. Development of MOST for Real-Time Tsunami Forecasting. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 142(6):03116004, 2016. doi: 10.1061/(ASCE)WW.1943-5460.0000357.

Randa O. Tom, Krhoda O. George, Atela O. Joanes, and Akala Haron. Review of flood modelling and models in developing cities and informal settlements: A case of Nairobi city. *Journal of Hydrology: Regional Studies*, 43:101188, 2022. ISSN 2214-5818. doi: 10.1016/j.ejrh.2022.101188. URL `https://www.sciencedirect.com/science/article/pii/S2214581822002014`.

Eleuterio F. Toro. Shock-Capturing Methods for Free-Surface Shallow Flows. 2001. URL `https://api.semanticscholar.org/CorpusID:118902986`.

Dennis Trujillo, Robert Robey, Neal Davis, and David Nicholaeff. Cell-based Adaptive Mesh Refinement on the GPU with Applications to Exascale Supercomputing. pages 1036–, 10 2011.

R. Vacondio, A. Dal Palù, and P. Mignosa. GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations. *Environmental Modelling and Software*, 57:60–75, 2014. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2014.02.003. URL `https://www.sciencedirect.com/science/article/pii/S136481521400053X`.

Stefan Vater, Nicole Beisiegel, and Jörn Behrens. A limiter-based well-balanced discontinuous Galerkin method for shallow-water flows with wetting and drying: One-dimensional case. *Advances in Water Resources*, 85:1–13, 2015. ISSN 0309-1708. doi: https://doi.org/10.1016/

j.advwatres.2015.08.008. URL `https://www.sciencedirect.com/science/article/pii/S0309170815001943`.

Stefan Vater, Nicole Beisiegel, and Jörn Behrens. Comparison of Wetting and Drying Between a RKDG2 Method and Classical FV Based Second-Order Hydrostatic Reconstruction. In Clément Cancès and Pascal Omnes, editors, *Finite Volumes for Complex Applications VIII - Hyperbolic, Elliptic and Parabolic Problems*, pages 237–245, Cham, 2017. Springer International Publishing. ISBN 978-3-319-57394-6.

Stefan Vater, Nicole Beisiegel, and Jörn Behrens. A limiter-based well-balanced discontinuous Galerkin method for shallow-water flows with wetting and drying: Triangular grids. *International Journal for Numerical Methods in Fluids*, 91(8):395–418, 2019. doi: https://doi.org/10.1002/fld.4762. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4762`.

Deniz Velioglu Sogut and Ahmet Cevdet Yalciner. Performance Comparison of NAMI DANCE and FLOW-3D® Models in Tsunami Propagation, Inundation and Currents using NTHMP Benchmark Problems. *Pure and Applied Geophysics*, 176(7):3115–3153, 07 2019. ISSN 1420-9136. doi: 10.1007/s00024-018-1907-9. URL `https://doi.org/10.1007/s00024-018-1907-9`.

Damien Violeau, Richard Marcer, Pons Kévin, Journeau Camille, Riadh Ata, Michel Benoit, A. Joly, Stéphane Abadie, Lucie Clous, Manuel Martin-Medina, Denis Morichon, J. Chicheportiche, Marine Le Gal, A. Gailler, Hélène Hébert, D. Imbert, Maria Kazolea, Mario Ricchiuto, Sylvestre Le Roy, and Ricardo Silva Jacinto. A database of validation cases for tsunami numerical modelling. 07 2016.

Mohamed Wahib, Naoya Maruyama, and Takayuki Aoki. Daino: A High-Level Framework for Parallel and Efficient AMR on GPUs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 621–632, 2016. doi: 10.1109/SC.2016.52.

Peng Wang, Tom Abel, and Ralf Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, 2010. doi: 10.1016/j.newast.2009.10.002. URL `https://www.sciencedirect.com/science/article/pii/S1384107609001432`.

Tobias Weinzierl and Miriam Mehl. Peano—A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011. doi: 10.1137/100799071. URL `https://doi.org/10.1137/100799071`.

Niklas Wintermeyer, Andrew R. Winters, Gregor J. Gassner, and Timothy Warburton. An entropy stable discontinuous Galerkin method for the shallow water equations on curvilinear meshes with wet/dry fronts accelerated by GPUs. *Journal of Computational Physics*, 375:447–480, 2018. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2018.08.038. URL `https://www.sciencedirect.com/science/article/pii/S0021999118305692`.

Xilin Xia, Qiuhua Liang, and Xiaodong Ming. A full-scale fluvial flood modelling framework based on a high-performance integrated hydrodynamic modelling system (HiPIMS). *Advances in Water Resources*, 132:103392, 2019. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2019.103392. URL `https://www.sciencedirect.com/science/article/pii/S030917081930243X`.

Yulong Xing and Chi-Wang Shu. High order well-balanced finite volume WENO schemes and discontinuous Galerkin methods for a class of hyperbolic systems with source terms. *Journal of Computational Physics*, 214(2):567–598, 2006a. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2005.10.005. URL `https://www.sciencedirect.com/science/article/pii/S0021999105004626`.

Yulong Xing and Chi-Wang Shu. High-Order Well-Balanced Finite Difference WENO Schemes for a Class of Hyperbolic Systems with Source Terms. *Journal of Scientific Computing*, 27(1): 477–494, 2006b. ISSN 1573-7691. doi: 10.1007/s10915-005-9027-y. URL `https://doi.org/10.1007/s10915-005-9027-y`.

Yulong Xing, Xiangxiong Zhang, and Chi-Wang Shu. Positivity-preserving high order well-balanced discontinuous Galerkin methods for the shallow water equations. *Advances in Water Resources*, 33(12):1476–1493, 2010. ISSN 0309-1708. doi: https://doi.org/10.1016/j.advwatres.2010.08.005. URL `https://www.sciencedirect.com/science/article/pii/S0309170810001491`.

Ruize Yang, Yang Yang, and Yulong Xing. High order sign-preserving and well-balanced exponential Runge-Kutta discontinuous Galerkin methods for the shallow water equations with friction. *Journal of Computational Physics*, 444:110543, 2021. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2021.110543. URL `https://www.sciencedirect.com/science/article/pii/S0021999121004381`.

Tae Hoon Yoon and Seok-Koo Kang. Finite Volume Model for Two-Dimensional Shallow Water Flows on Unstructured Grids. *Journal of Hydraulic Engineering*, 130(7):678–688, 2004. doi: 10.1061/(ASCE)0733-9429(2004)130:7(678). URL `https://ascelibrary.org/doi/abs/10.1061/(ASCE)0733-9429(2004)130:7(678)`.

Sergey Zabelok, Robert Arslanbekov, and Vladimir Kolobov. Adaptive kinetic-fluid solvers for heterogeneous computing architectures. *Journal of Computational Physics*, 303:455–469, 2015. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2015.10.003. URL `https://www.sciencedirect.com/science/article/pii/S0021999115006658`.

Andrey Zaytsev, Andrey Kurkin, Efim Pelinovsky, and Ahmet C. Yalciner. NUMERICAL TSUNAMI MODEL NAMI-DANCE. *Science of Tsunami Hazards*, 2019. URL `https://doaj.org/article/a61f205c253c44b6aae0b88047fae928`.

Dia Zeidan, Alex A. Schmidt, Alice J. Kozakevicius, and Stefan Jakobsson. Towards parallel WENO wavelet methods for the simulation of compressible two-fluid models. *AIP Conference Proceedings*, 2425(1):020017, 2022. doi: 10.1063/5.0082038. URL `https://doi.org/10.1063/5.0082038`.

Min Zhang, Weizhang Huang, and Jianxian Qiu. A High-Order Well-Balanced Positivity-Preserving Moving Mesh DG Method for the Shallow Water Equations With Non-Flat Bottom Topography. *Journal of Scientific Computing*, 87(3):88, 2021. doi: 10.1007/s10915-021-01490-3. URL `https://doi.org/10.1007/s10915-021-01490-3`.

Jiaheng Zhao, Ilhan Özgen Xian, Dongfang Liang, Tian Wang, and Reinhard Hinkelmann. An improved multislope MUSCL scheme for solving shallow water equations on unstructured grids. *Computers & Mathematics with Applications*, 77(2):576–596, 2019. ISSN 0898-1221. doi: 10.

1016/j.camwa.2018.09.059. URL `https://www.sciencedirect.com/science/article/pii/S0898122118305790`.

Feng Zhou, Guoxian Chen, Yuefei Huang, Jerry Zhijian Yang, and Hui Feng. An adaptive moving finite volume scheme for modeling flood inundation over dry and complex topography. *Water Resources Research*, 49(4):1914–1928, 2013. doi: 10.1002/wrcr.20179. URL `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/wrcr.20179`.

Nunan Zola, Wagner M. Bona, and Luis C. E. Silva. Fast GPU parallel N-Body tree traversal with Simulated Wide-Warp. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 718–725, 2014. doi: 10.1109/PADSW.2014.7097874.

Appendices

Appendix A

GPU computing

GPU computing is only well-suited for accelerating the speed of programs that possess a very specific algorithmic structure, in particular those with the following characteristics: (1) the program requires an extremely large computational throughput and memory bandwidth (i.e. the program needs to perform a lot of computation on a lot of data), (2) the program involves computations that show substantial data-parallelism (i.e. the program needs to perform many independent yet similar operations on the data), and (3) the program's throughput is more important than its latency (i.e. the program's purpose is to perform a very large number of computations, rather than to perform an single computation very quickly) (Owens et al., 2008). If a program indeed possesses these characteristics, then GPU computing can greatly accelerate the speed of the program by leveraging a significantly higher level of computational power compared to a traditional CPU. Namely, GPUs can achieve a much higher number of floating-point number operations per second (FLOPS) than CPUs: for example, the NVIDIA A100 GPU can achieve up to 9.7×10^{12} FLOPS (NVIDIA, 2021), whereas the AMD 5900X CPU can achieve up to only 7.6×10^{10} FLOPS (Laboratory) – two orders of magnitude lower.

GPUs can achieve a much higher number of FLOPS due to a specialised hardware architecture that originally evolved over time to yield maximum parallel processing speed in computer graphics programs. Computer graphics programs involve geometric computations that require performing a very large number of similar, independent and floating-point-intensive operations – evidently a parallel and compute-intensive problem. The geometric computations are organised in GPUs as a "graphics pipeline" (Owens et al., 2008), and over time, the pipeline became more flexible and programmable and eventually started getting repurposed for accelerating scientific computing programs, which became known as general-purpose GPU (GPGPU) computing (Owens et al., 2007).

GPGPU computing was performed in its early stages using application programming interfaces (API) for graphics programs, such as OpenGL (Kessenich et al., 2004) or DirectX (Hargreaves, 2020). This was a difficult process as scientific computing programs had to be expressed as computer graphics programs. To reduce the difficulty of GPGPU computing, newer GPU programming models emerged that were purposefully developed for GPGPU computing, such as CUDA (NVIDIA, 2023), OpenCL (Khronos, 2024) and ROCm (AMD, 2024). Nonetheless, these GPU programming models are still difficult to use because the programmer must be familiar with GPU hardware architecture and how the hardware's effectiveness depends on the operations in the program: the more efficiently the hardware is used by the program, the better the acceleration of the program from GPU computing.

A.1 GPGPU computing on NVIDIA GPUs using the CUDA programming model

Out of all the GPU programming models that have emerged to date, the proprietary CUDA programming model developed by NVIDIA exclusively for use with NVIDIA GPUs remains one of the most popular (Pandey et al., 2022). The CUDA programming model adopts a "host-device" framework as shown in Figure A.1: the CPU is the "host", which is responsible for everything in a CUDA program except for heavy-duty parallel computation, which is *offloaded* by the host to the "device", the GPU. The GPU handles the parallel computation by executing a program called a "kernel", which is made up of many parallel "threads". The kernel is written as a program for a single thread, but the GPU executes the kernel in parallel with many threads whereby the program is independently evaluated by each thread.

An appropriate number of threads should be launched for handling the computation described by a kernel. For example, a kernel for adding two vectors of length 1024 should launch 1024 threads. In the CUDA programming model, threads are grouped into "blocks", and blocks are grouped into a "grid" (note that this CUDA grid does not refer to the computational grid over which the SWE are solved). The block size and grid size should be chosen to launch a kernel with an appropriate

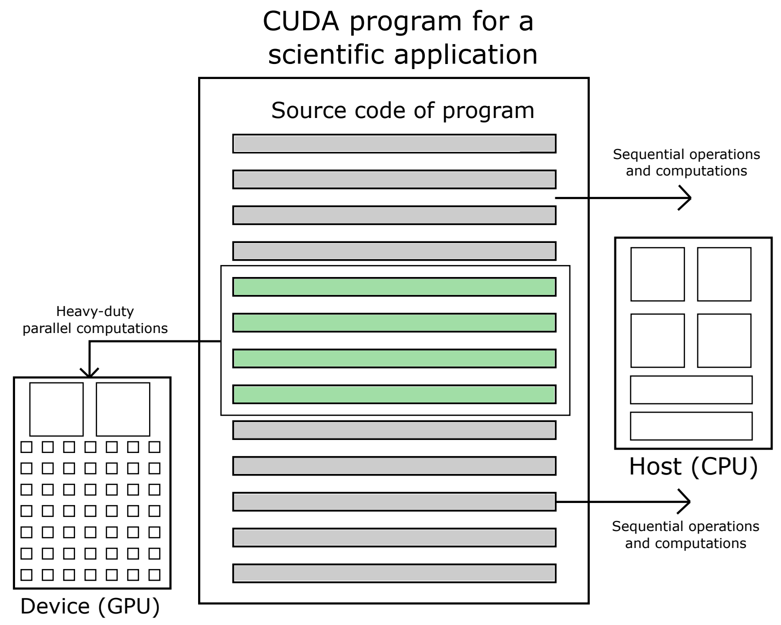

CUDA program for a scientific application
Source code of program
Sequential operations and computations
Heavy-duty parallel computations
Host (CPU)
Sequential operations and computations
Device (GPU)

Figure A.1: *"Host-device" framework of a CUDA program representing a scientific computing program. In a CUDA program, all operations are handled by the "host", the CPU, except for heavy-duty parallel computation, which is offloaded by the host to the "device", the GPU.*

number of threads. For example, to launch a kernel with 1024 threads, a block size of 256 and a grid size of 4 could be chosen, since 256 threads per block $\times$ 4 blocks in the grid = 1024 threads in the grid. The block size and/or grid size are usually chosen empirically to maximise the speed of a kernel.

The speed of the kernel(s) in a CUDA program is one of the most important factors affecting the speed of the overall program: maximally accelerating the speed of a program using CUDA is strongly contingent on maximising the speed of the kernel(s) in the program. To maximise the speed of a kernel, it should be programmed to use the GPU hardware efficiently, but this is not trivial due to the complexity of GPU hardware architecture.

A.1.1 GPU hardware architecture

Figure A.2 shows the hardware architecture of a GPU. As shown in Figure A.2a, a GPU is made up of several "streaming multiprocessors" (SM) which are responsible for computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which the SMs access via an L2 cache. The number of SMs and the amount of DRAM depend on the specific GPU being used. Each SM has several threads for directly performing computations, as well as other hardware for supporting computation: registers, shared/L1/texture cache, constant cache and "warp schedulers", as seen in Figure A.2b. As shown in Figure A.2c, each thread is made up of functional units for executing various instructions, such as 32-bit integer operations (INT32), 32- or 64-bit floating-point operations (FP32 and FP64), load/store operations (LD/ST) and special-function calculations (SFU) for functions like sine, cosine, square root, etc.

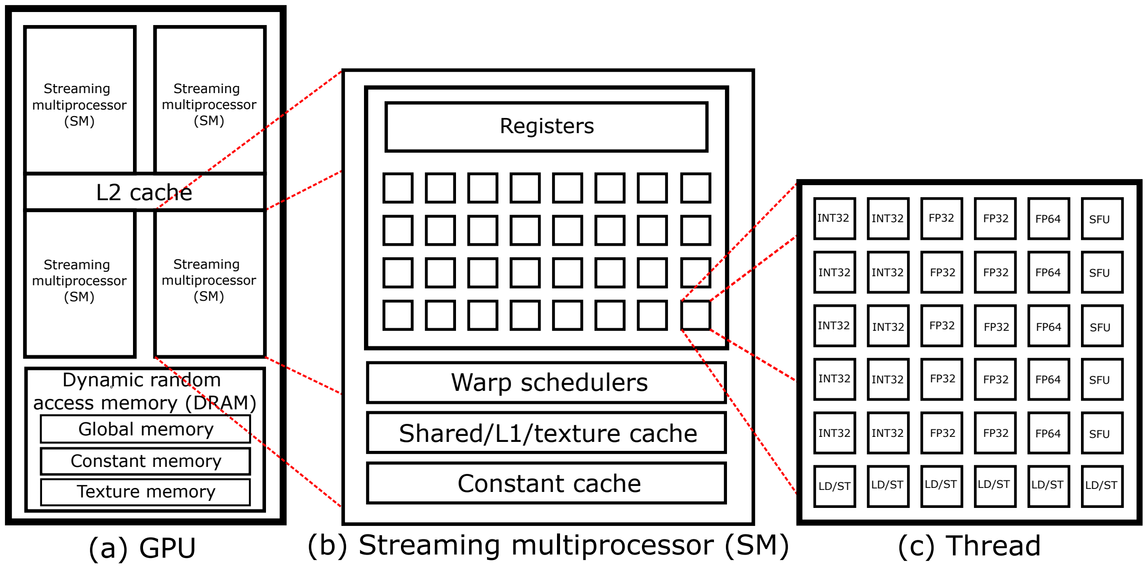

Streaming multiprocessor (SM)
Streaming multiprocessor (SM)
L2 cache
Streaming multiprocessor (SM)
Streaming multiprocessor (SM)
Dynamic random access memory (DRAM)
Global memory
Constant memory
Texture memory
Registers
Warp schedulers
Shared/L1/texture cache
Constant cache
INT32
FP32
FP64
SFU
LD/ST
(a) GPU
(b) Streaming multiprocessor (SM)
(c) Thread

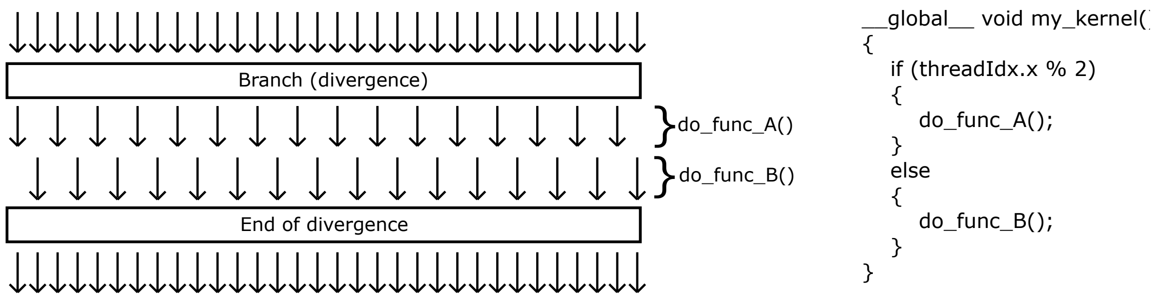
Figure A.2: *GPU hardware architecture; (a) the GPU is made up of several "streaming multiprocessors" (SM) which are responsible for computation, while the data on which the computations are performed reside in dynamic random access memory (DRAM), which SMs access via an L2 cache; (b) each SM is made up of hardware for supporting computation: registers, shared/L1/texture cache, constant cache and "warp schedulers", as well as several "threads" for computation; (c) each thread is made up of functional units for executing various types of instructions.*

The more efficiently the GPU hardware is used by a kernel, the higher the speed of the kernel. Understanding how a kernel uses the hardware requires understanding how kernel execution is coupled with GPU hardware. Recall that during kernel execution, the threads are organised into blocks and the blocks are organised into a grid. The blocks in the grid are distributed among the SMs for execution: for example, if the grid is made up of 128 blocks and there are 16 SMs, there can be up to 128 blocks per 16 SMs = 8 blocks per SM. Blocks are partitioned into "warps", which are groups of 32 consecutive threads: for example, a block made up of 512 threads is made up of: 512 threads per 32 threads per warp = 16 warps. Warps execute instructions in lockstep following a single-instruction multiple-thread (SIMT) computing paradigm (NVIDIA, 2023) in which all threads in the warp execute the same instruction, i.e. in parallel. Warps are scheduled for SIMT execution by one or more "warp schedulers" to "warp slots" in the SM; the number of warp schedulers per SM and the number of warp slots per scheduler depend on the specific GPU being used. For example, if the GPU is made up of SMs that have 2 warp schedulers with 8 warp slots each, then each SM has 2 warp schedulers $\times$ 8 warps slots per scheduler = 16 warps slots, meaning that up to 16 warps can be scheduled for execution in the SM. Whenever one of these scheduled warps finishes execution, it exits the warp slot leaving the slot free for another warp.

Familiarity with the notion of warps and how kernel execution is coupled with GPU hardware allows to discuss the two metrics that quantify how efficiently a kernel uses the hardware: instruction throughput and memory throughput. Instruction throughput is the rate at which the kernel executes instructions, whereas memory throughput is the rate at which the kernel accesses data from memory. Both instruction and memory throughput of a kernel should be maximised to maximise the kernel's speed.

A.1.2 Maximising instruction throughput

Maximising instruction throughput requires understanding the behaviour of a warp under the SIMT computing paradigm. Under SIMT, threads in a warp are allowed to execute differing instructions if needed, but the warp's instruction throughput is best maximised if all threads in the warp execute the same instruction. In particular, if threads in a warp execute differing instructions, then "warp divergence" occurs, which reduces the speed of the warp because the differing instructions cannot be performed by the warp in parallel and must be performed in serial instead. To clarify, an example of warp divergence is shown in Figure A.3, in which the arrows represent 32 threads in a warp; in CUDA code, `threadIdx.x` gives the index of a thread within a block: in this CUDA code, the threads with an odd thread index execute the instructions in the function `func_A()`, whereas the threads with an even thread index execute the instructions in the function `func_B()`. The execution of these functions get serialised: the odd-indexed threads execute `func_A()` while the even-indexed threads wait; then, the even-indexed threads execute `func_B()` while the odd-indexed threads wait. Due to the serialised execution of the functions, the speed of the warp is reduced because it cannot finish executing until both the odd- and even-indexed threads have executed their respective instructions. The warp divergence shown in Figure A.3 is an example of "two-way" divergence because there are two branches in execution: in general, N number of branches in execution lead to "N-way" divergence. The larger the number of branches in execution, the more divergence and therefore serialisation of instructions there is and the slower the speed of the warp becomes. Overall, divergence should be minimised to increase instruction throughput.



```
__global__ void my_kernel(
{
   if (threadIdx.x % 2)
   {
      do_func_A();
   }
   else
   {
      do_func_B();
   }
}
```

Figure A.3: *Warp divergence; a warp is a group of 32 threads with consecutive thread IDs that operate in lockstep.*

Divergence is an instruction throughput issue to consider for a single warp, but the next issue about instruction throughput requires considering the collective behaviour of multiple warps. Recall that multiple warps can be scheduled to warp slots for execution by a warp scheduler: a scheduler can issue an instruction for execution for one warp every clock cycle, and instruction throughput is best maximised if the scheduler can issue instructions at its theoretical *clock cycle frequency*. For example, if a scheduler has a clock cycle frequency of 10^{12} Hz, it can theoretically issue 1 instruction per clock cycle $\times$ 10^{12} clock cycles per second = 10^{12} instructions per second – which is the maximum instruction throughput of the scheduler. However, reaching the maximum instruction throughput depends on if the scheduler is actually able to issue an instruction every clock cycle, which in turn depends on the state of the warps in the warp slots: a warp can either be *active*, meaning it has been scheduled to a warp slot, or it can be *inactive*, meaning it has not been scheduled to any warp slot. Active warps can either be *eligible*, *issued* or *stalled*: an eligible warp is a warp that is ready to execute its next instruction; an issued warp is the warp the scheduler issues an instruction for out of all the eligible warps (if any); a stalled warp is a warp that is not yet ready to execute its next instruction (e.g. due to still executing the previous instruction or due to waiting for data to be accessed from memory). These warp states affect the instruction throughput of the scheduler, explained next using an example.

Figure A.4 shows an example on the instruction throughput of a warp scheduler with eight warp slots over four clock cycles. In the first clock cycle, there are five active warps: three are stalled (shown in red) and two are eligible (shown in light green and green). The scheduler issues an instruction for one of the eligible warps – namely the one in slot five – which is the issued warp in the first clock cycle (shown in green). In the second clock cycle, there are still five active warps: four are stalled and only one warp – in slot four – is eligible. This eligible warp is the issued warp in the second clock cycle. In the third clock cycle, there are four active warps instead of five: the warp in slot four has finished all of its instructions and has therefore exited the warp slot, while all the other warps are stalled so there is no issued warp in the third clock cycle (indicated by the cross). In the fourth clock cycle, there are six active warps: two new warps have been scheduled to the warp slots – three warps are active and three warps are eligible, and the issued warp is the warp in slot three.

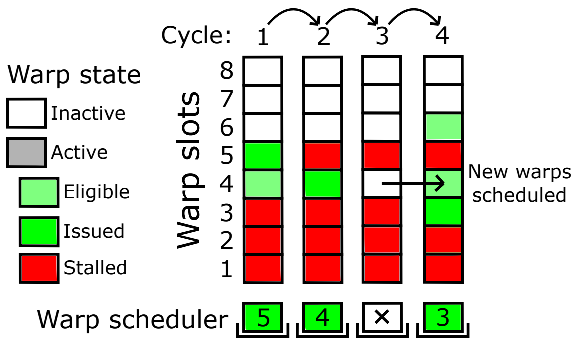


Figure A.4: *Example of the behaviour of a warp scheduler with eight warp slots over four clock cycles.*

To increase the instruction throughput of a scheduler, there should be many active warps per scheduler: having many active warps maximises the chance that there is at least one eligible warp per clock cycle, thereby approaching the theoretical clock cycle frequency of the scheduler. The number of active warps is quantified by a metric termed "occupancy", which is defined as the ratio between the achieved number of active warps per scheduler to the theoretical maximum number of active warps per scheduler. A high occupancy can increase the scheduler's instruction throughput, and to achieve a high occupancy, a high theoretical occupancy is required.

The theoretical occupancy is affected by the following restrictions: (1) the maximum number of warps per scheduler, (2) the maximum number of blocks per SM, (3) the register capacity per SM, and (4) the shared memory per SM, explained as follows. Restriction (1): each SM has a physical maximum number of warps that can be active depending on the GPU (e.g., 16 or 32), which the theoretical occupancy can never exceed. Restriction (2): each SM has a physical maximum number of blocks that can be active depending on the GPU; for example, if a GPU supports 16 active blocks and 64 active warps per SM, than blocks with 32 threads (one warp per block) results in at most 16 active warps, because only 16 blocks can be active and each block has only one warp, leading to theoretical occupancy of 16 active warps per SM per 64 maximum warps per SM = 25%. Restriction (3): each SM has a fixed number of registers for usage; if thread blocks use too many registers, then there will be fewer active blocks per SM. For example, if an SM has a register capacity of 64 kB, and each thread block consumes 8 kB in registers, then there can be at most 64 kB per 8 kB = 8 active blocks. Therefore, register usage should be minimised to increase theoretical occupancy. Restriction (4): each SM has a configurable amount of shared memory for consumption; if individual thread blocks consume too much shared memory, then there will be fewer active blocks per SM. For example, if an SM has a shared memory of 64 kB, and each thread block consumes 16 kB of shared memory, then there can be at most 64 kB per 16 kB = 4 active blocks. Thus, shared memory usage should be minimised to increase theoretical occupancy.

A.1.3 Maximising memory throughput

In addition to maximising instruction throughput, memory throughput should also be maximised. To achieve this, familiarity with the complicated memory hierarchy involved in the CUDA programming model is needed. Figure A.5 shows the memory hierarchy. As shown by the labelled boxes, the memory hierarchy is made up of several memory spaces: CPU DRAM, GPU DRAM, L2 cache, local caches per SM, and registers per thread. Data must pass through some or all of the memory spaces when accessed by a thread, as shown using the small arrows in Figure A.5.

The memory spaces can be ranked from slowest to fastest based on how many clock cycles it takes for a thread to access data from them, as shown using the large arrow in Figure A.5: CPU DRAM, GPU DRAM (at least 100 clock cycles), L2 cache, local caches in an SM (approximately 1 clock cycle) and registers (1 clock cycle). Each of the memory spaces are slower or faster to access depending on whether they are “off-chip” (i.e. far from a thread, like CPU DRAM and GPU DRAM) or “on-chip” (i.e. close to a thread, like the L2 cache, local caches and registers). The memory spaces should be accessed in a way that minimises the use of slow off-chip memory and maximises the use of fast on-chip memory. Accessing the memory spaces in this way results

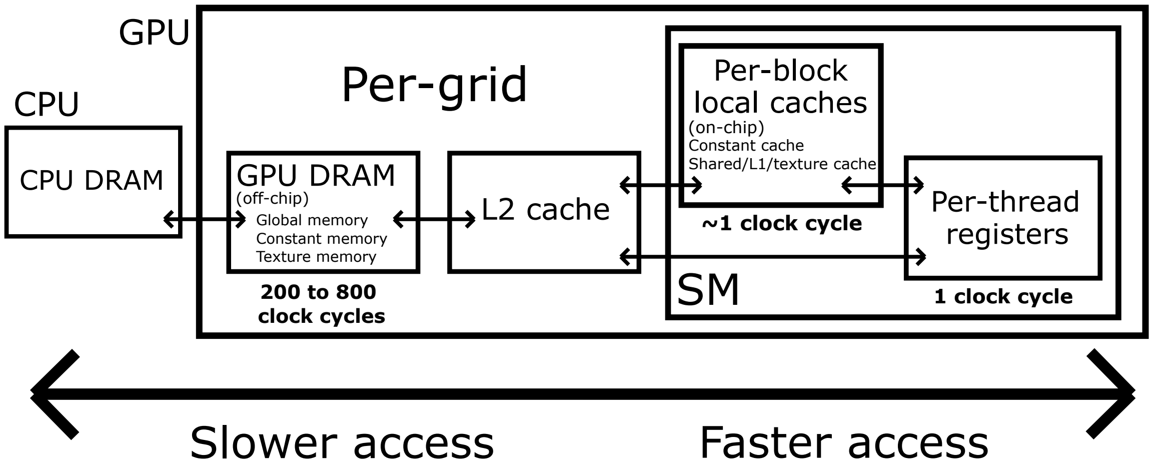


Figure A.5: *Memory hierarchy involved in the CUDA programming model.*

in a strategy with the following steps: (1) copy data from CPU DRAM to GPU DRAM, (2) load data from slow off-chip GPU DRAM to fast on-chip memory, (3) use the data in on-chip memory to perform computations as needed, and (4) store the results from the computations back to slow off-chip GPU DRAM and (5) copy the results from GPU DRAM back to CPU DRAM. When implementing this strategy, each memory space must be accessed by threads in a warp using special access patterns: these special access patterns are necessary to reach the peak access speed of each memory space, particularly when accessing GPU DRAM, the L2 cache and the local caches per SM, explained next.

GPU DRAM is made up of global memory, constant memory and texture memory (see Figure A.2). Texture memory and constant memory allow read-only access to data: texture memory's peak access speed is achieved when threads in a warp read data using access patterns with 2D or 3D spatial locality, whereas constant memory's peak access speed is achieved when threads in warp read data using an access pattern that reads from a constant (i.e. the same) location in memory (NVIDIA, 2023). On the other hand, global memory allows both read and write access to data.

Data in global memory is accessed via the L2 cache as chunks of 32 bytes of data, i.e. as 32-byte *cache lines*. This must be considered when devising an optimal access pattern for global memory. Figures A.6a and A.6b show optimal and non-optimal access patterns for threads in a warp to access global memory, respectively. In Figure A.6a, the upper label around the grey boxes marks off a 32-byte cache line in the L2 cache, and the smaller four green segments represent four L2 cache lines. The arrows indicate 32 threads in a warp. The warp accesses global memory via the L2 cache using an optimal access pattern. In this optimal access pattern, the threads access global memory locations that are aligned with the 32-byte boundaries of each cache line fetched from the L2 cache (indicated by the numbers 0, 32, . . . , 128 in Figure A.6a). This alignment leads to coalesced memory access because the threads make use of all the data in the L2 cache lines: in Figure A.6a, the threads require access to 128 bytes of data, meaning that 128 bytes required / 32 bytes per cache line = 4 cache lines need to be fetched at minimum. Since the warp indeed fetches only four cache lines – as shown by the smaller four green segments in Figure A.6a – this results in 4 cache lines minimum / 4 cache lines fetched = 100% cache line usage.

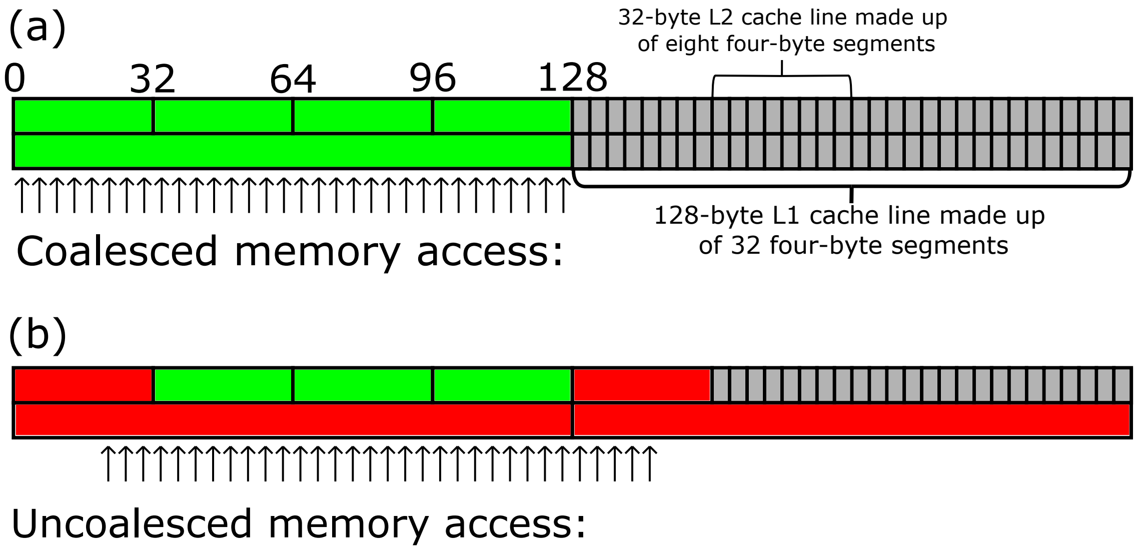

(a)
0
32
64
96
128
32-byte L2 cache line made up of eight four-byte segments
128-byte L1 cache line made up of 32 four-byte segments
Coalesced memory access:
(b)
Uncoalesced memory access:

Figure A.6: *Optimal and non-optimal access patterns by threads in a warp when accessing global memory via the L2 and "L1 caches" (a) an optimal access leading to coalesced memory access; (b) a non-optimal access pattern leading to uncoalesced memory access.*

Less than 100% cache line usage means that the threads in a warp do not make use of all the data moved by the cache lines and implies that the warp is fetching an excess number of cache lines, which reduces the speed of a warp (i.e. uncoalesced memory access occurs). An example of uncoalesced memory access is shown in Figure A.6b, in which the access pattern of the warp is shifted compared to the access pattern of the warp in Figure A.6a. Due to the shifted access pattern, the warp accesses global memory locations that are no longer aligned with the cache line boundaries and five cache lines are fetched, as shown using the small red and green segments in Figure A.6b. However, as the warp still requires only 128 bytes, i.e. 128 bytes required / 32 bytes per cache line = 4 cache lines need to be fetched at minimum, this results in 4 cache lines minimum

/ 5 cache lines fetched = 80% cache line usage.

Data from the L2 cache lines ultimately moves to per-thread registers (right-most box in Figure A.5) where it is used for computation by the threads. Registers are accessible only to individual threads, and take only one clock cycle to access, making them the fastest memory space for a thread to use. Registers receive data either directly from the L2 cache or via local caches per SM, as shown using the small arrows to the right in Figure A.5. A local cache is made up of a constant cache, through which data from constant memory pass, and a shared/L1/texture cache, through which data from global and texture memory pass.

In the shared/L1/texture cache, there is an L1 cache. The L1 cache moves data in cache lines – much like the L2 cache – but using larger 128-byte cache lines, as marked off by the bottom label around the grey boxes in Figure A.6a. The L1 cache has a faster access speed than the L2 cache (NVIDIA, 2023), but using the L1 cache can lead to much bigger losses in cache line usage compared to using the L2 cache if uncoalesced memory access occurs. For example, in Figure A.6b, uncoalesced memory access leads to a cache line usage of 80% if using the L2 cache, but using the L1 cache leads to an even worse cache line usage: two L1 cache lines are accessed, as shown using the two large red segments, meaning that 2 cache lines $\times$ 128 bytes per cache line = 256 bytes are accessed, but only 128 bytes are used, leading to 128 bytes used per 256 bytes read = 50% cache line usage. Thus, uncoalesced memory access should be particularly avoided by the programmer if they have configured a kernel to use the L1 cache.

The shared/L1/texture cache also includes "shared memory", which is very fast to access – nearly as fast as registers. Furthermore, unlike data in registers, which is accessible to individual threads only, data in shared memory is accessible to all threads in a block and thus offers threads a way to share data with each other. However, shared memory must be accessed by threads in a warp using a specific access pattern to reach its peak access speed, explained as follows.

Data in shared memory is stored in 4-byte banks which are arranged in "strides" of 32 banks: the first bank stores 4 bytes of data (4 bytes in total), the next bank stores another 4 bytes of data (8 bytes in total), and so on until the 32nd bank is reached, at which point 32 banks $\times$ 4 bytes per bank = 128 bytes of data are stored. When the next 4 bytes of data need to be stored, there is no 33rd bank: instead, the data "wraps around" after passing the 32nd bank back to the 1st bank and the storage of the data continues as described previously, as shown by the dashed arrow in Figure A.7b.

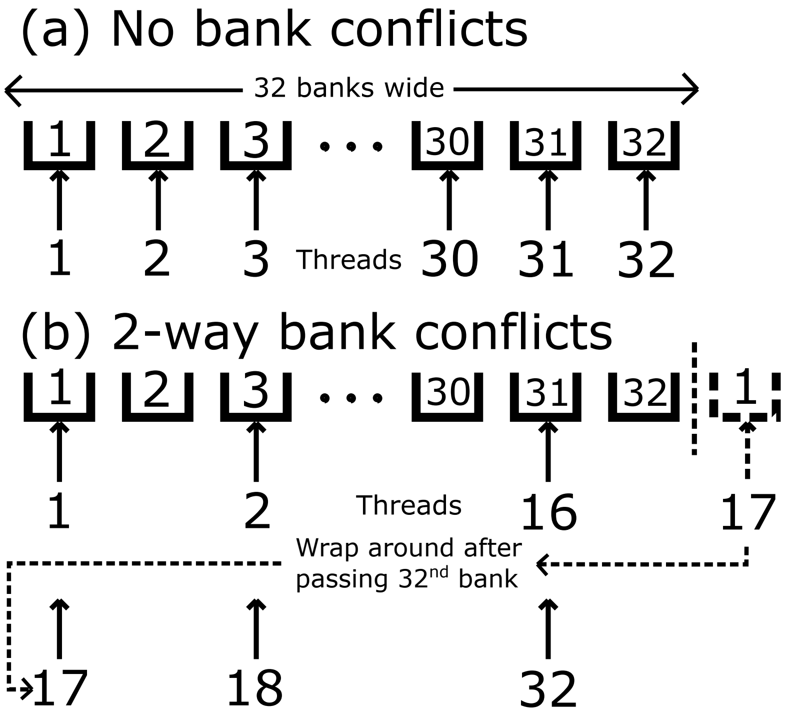


Figure A.7: *Shared memory bank conflicts for threads in a warp; (a) no bank conflicts; (b) two-way bank conflicts.*

If different bytes of data in the same bank are accessed by threads in a warp, a “bank conflict” occurs. Figures A.7a and A.7b show examples of no bank conflicts and “two-way” bank conflicts, respectively; an N-way bank conflict is said to occur if N threads access different bytes of data in the same bank. For example, in Figure A.7a, no bank conflicts occur because none of the threads access different bytes in the same bank: thread 1 accesses bank 1, thread 2 accesses bank 2, and so on until thread 32 accesses bank 32. On the other hand, in Figure A.7b, threads 1 and 17 access different bytes in bank 1, threads 2 and 18 access bank 2, and so on until threads 16 and 32 access bank 16, resulting in 16 two-way bank conflicts. Bank conflicts reduce memory throughput because they serialise data access to threads in a warp: the banks cannot broadcast data to the conflicting threads simultaneously and must broadcast data sequentially. For example, if a two-way bank conflict occurs, the bank must make two separate broadcasts to move the data to the conflicting threads. Bank conflicts should be avoided to maximise memory throughput in shared memory, namely by using an access pattern that ensures that threads in a warp do not access different bytes in the same bank.

Appendix B

Filter bank matrices

These are the filter bank matrices in Equations 4.4a - 4.4d.

$$\mathbf{HH}^0 = \begin{bmatrix} 1/2 & 0 & 0 \\ -\sqrt{3}/4 & 1/4 & 0 \\ -\sqrt{3}/4 & 0 & 1/4 \end{bmatrix}, \quad \mathbf{HH}^1 = \begin{bmatrix} 1/2 & 0 & 0 \\ -\sqrt{3}/4 & 1/4 & 0 \\ \sqrt{3}/4 & 0 & 1/4 \end{bmatrix},$$

$$\mathbf{HH}^2 = \begin{bmatrix} 1/2 & 0 & 0 \\ \sqrt{3}/4 & 1/4 & 0 \\ -\sqrt{3}/4 & 0 & 1/4 \end{bmatrix}, \quad \mathbf{HH}^3 = \begin{bmatrix} 1/2 & 0 & 0 \\ \sqrt{3}/4 & 1/4 & 0 \\ \sqrt{3}/4 & 0 & 1/4 \end{bmatrix},$$

$$\mathbf{GA}^0 = \begin{bmatrix} -\sqrt{14}/14 & -\sqrt{42}/14 & -\sqrt{42}/14 \\ 0 & 0 & -\sqrt{2}/2 \\ 0 & -\sqrt{2}/2 & 0 \end{bmatrix}, \quad \mathbf{GA}^1 = 0,$$

$$\mathbf{GA}^2 = 0, \qquad \mathbf{GA}^3 = \begin{bmatrix} \sqrt{14}/14 & -\sqrt{42}/14 & -\sqrt{42}/14 \\ 0 & 0 & \sqrt{2}/2 \\ 0 & \sqrt{2}/2 & 0 \end{bmatrix},$$

$$\mathbf{GB}^0 = 0, \qquad \mathbf{GB}^1 = \begin{bmatrix} -\sqrt{14}/14 & -\sqrt{42}/14 & \sqrt{42}/14 \\ 0 & 0 & -\sqrt{2}/2 \\ 0 & -\sqrt{2}/2 & 0 \end{bmatrix},$$

$$\mathbf{GB}^2 = \begin{bmatrix} \sqrt{14}/14 & -\sqrt{42}/14 & \sqrt{42}/14 \\ 0 & 0 & \sqrt{2}/2 \\ 0 & \sqrt{2}/2 & 0 \end{bmatrix}, \quad \mathbf{GB}^3 = 0,$$

$$\mathbf{GC}^0 = \begin{bmatrix} 1/2 & 0 & 0 \\ -\sqrt{21}/28 & -3\sqrt{7}/28 & 2\sqrt{7}/14 \\ -\sqrt{21}/28 & 2\sqrt{7}/14 & -3\sqrt{7}/28 \end{bmatrix}, \quad \mathbf{GC}^1 = \begin{bmatrix} -1/2 & 0 & 0 \\ -\sqrt{21}/28 & -3\sqrt{7}/28 & -2\sqrt{7}/14 \\ \sqrt{21}/28 & -2\sqrt{7}/14 & -3\sqrt{7}/28 \end{bmatrix},$$

$$\mathbf{GC}^2 = \begin{bmatrix} -1/2 & 0 & 0 \\ \sqrt{21}/28 & -3\sqrt{7}/28 & -2\sqrt{7}/14 \\ -\sqrt{21}/28 & -2\sqrt{7}/14 & -3\sqrt{7}/28 \end{bmatrix}, \quad \mathbf{GC}^3 = \begin{bmatrix} 1/2 & 0 & 0 \\ \sqrt{21}/28 & -3\sqrt{7}/28 & 2\sqrt{7}/14 \\ \sqrt{21}/28 & 2\sqrt{7}/14 & -3\sqrt{7}/28 \end{bmatrix}.$$