Wavelet-based adaptive mesh refinement on the GPU for shallow water modelling

Alovya Ahmed Chowdhury¹*, Georges Kesserwani¹, Charles Rougé¹, Paul Richmond²

¹Department of Civil and Structural Engineering, University of Sheffield, Mappin St, Sheffield, UK
E-mail: {aachowdhury2, g.kesserwani, c.rouge}@sheffield.ac.uk

²Department of Computer Science, University of Sheffield, Mappin St, Sheffield, UK
E-mail: p.richmond@sheffield.ac.uk

*Corresponding author

Abstract

Wavelet-based adaptive mesh refinement (AMR) uses a rigorous “multiresolution analysis” (MRA) to construct a robust, inherently adaptive numerical solution. Despite its rigour however, there is no parallel implementation of wavelet-based AMR on graphics processing units (GPU) for shallow water modelling since the MRA is recursive and therefore inherently sequential. This paper reworks the recursive nature of MRA into a form suitable for parallelisation by simultaneously adopting two ingredients. First, a Z-order space-filling curve is applied, to ensure coalesced (i.e., adjacent) memory access, and second, a parallel tree traversal algorithm that minimises divergence of GPU threads is adopted. The wavelet-based AMR technique is presented as part of a Haar wavelet (HW) first-order finite volume (FV1) model (GPU-HWFV1). The model’s runtime performance is benchmarked against its central processing unit (CPU) predecessor (CPU-HWFV1) and a GPU-FV1 uniform-grid model for a range of test cases. The test cases demonstrate the robustness of GPU-HWFV1, and show that it runs up to 200× faster than CPU-HWFV1, and up to 30× faster than GPU-FV1, especially in applications that require increased grid resolution and high sensitivity to grid refinement. The magnitude of these runtime performance gains suggest the technique could be applied to speed up other FV1 models.

Keywords: wavelet-based adaptive mesh refinement; computational efficiency assessments; GPU computing; hydraulic modelling; multiresolution analysis
1. Introduction

Computational models solving the two-dimensional (2D) shallow water equations are widely used to support hydraulic engineering applications and often adopt a finite volume scheme involving a uniform computational grid (Toro, 2001; Toro & Garcia-Navarro, 2007). The combination of a large domain with a fine grid resolution generally results in dense grids and leads to prohibitively large runtime costs (Akoh et al., 2017; de Almeida et al., 2016; Echeverribar et al., 2019; Xing et al., 2019). To reduce these runtime costs, parallelisation on graphics processing units (GPU) is an established approach in shallow water modelling (Brodkorb et al., 2012; Castro et al., 2011; Lacasta et al., 2015; Shaw et al., 2021), and uniform-grid finite volume models parallelised on GPUs are the de facto in operational hydraulic modelling packages used for realistic applications (Flood Modeller 2D, 2022; HiPIMS-CUDA, 2021; InfoWorks ICM, 2018; MIKE 21 GPU, 2019; TUFLOW HPC, 2018).

A less common alternative to parallelisation for reducing runtime costs is adaptive mesh refinement (Arpaia & Ricchiuto, 2018; Haleem et al., 2015; Lakhlifi et al., 2018). With AMR, shallow water simulations are run on a non-uniform grid that uses finer resolution only where higher accuracy is needed (Hou et al., 2018). However, shallow water models often need ad-hoc modifications to their underlying finite volume schemes to ensure the numerical solution stays robust over a non-uniform grid (Liang et al., 2015). In shallow water modelling, a numerical solution is robust if the mass balance is conserved over space and time, and if the topography and flux gradients with different wetting and drying conditions are well-balanced (Donat et al., 2014; Liang & Borthwick, 2009; Zhou et al., 2013). In this regard, wavelet-based AMR is especially attractive because it avoids the need for ad-hoc modifications by using a mathematically rigorous “multiresolution analysis” (MRA) to construct a robust, inherently non-uniform numerical solution (Caviedes-Voullième et al., 2020; Caviedes-Voullième & Kesserwani, 2015; Gerhard et al., 2015; Haleem et al., 2015; Kesserwani et al., 2019;
Wavelet-based AMR is rigorous because the MRA applies mathematically derived filter bank formulae (Haleem et al., 2015; Keinert, 2003) to precisely “encode” (coarsen) and “decode” (refine) the numerical solution between resolutions. However, as the MRA is applied to generate a non-uniform grid at every timestep, it imposes a high computational overhead because many operations of encoding and decoding are involved during grid generation. For example, so-called Haar wavelets (HW) (Haleem et al., 2015) have been previously applied to a first-order finite volume (FV1) scheme to allow for the implementation of an adaptive HWFV1 shallow water model (Kesserwani & Sharifian, 2020) that was up to 20× faster than its uniform FV1 counterpart (while having the same accuracy), but it remained too costly to run for realistic test cases.

Parallelising AMR on the GPU to further reduce runtime costs is common in finite volume-type models in different fields such as hydrodynamics and gas dynamics (Beckingsale, 2015; Dunning et al., 2020; Giuliani & Krivodonova, 2019; Sætra et al., 2014; Wahib et al., 2016). However, the AMR techniques adopted by these models construct the non-uniform numerical solution via approximations (e.g., averaging or interpolation). In contrast, despite its use of rigorous filter bank formulae, wavelet-based AMR has yet to be parallelised on GPU and applied to shallow water modelling. Parallelising wavelet-based AMR on the GPU is also necessary to address the overhead of the MRA and make wavelet-based shallow water models such as HWFV1 competitive with GPU-accelerated uniform-grid models. In practice, this is difficult to achieve because the MRA is recursive in nature and inherently sequential. There are two main reasons behind this. First, the encoding and decoding operations needed to coarsen and refine the non-uniform grid, respectively, are performed recursively on a nested hierarchy of grids. Second, the generation of the non-uniform grid involves processing a tree-like structure, which is inherently a recursive task achieved with algorithms such as depth-first traversal (DFT) algorithm (Sedgewick & Wayne, 2011).
This paper aims to overcome these two obstacles in order to implement wavelet-based AMR on the GPU. For this, it combines two existing algorithms and applies them to shallow water modelling for the first time. Our approach (1) introduces so-called space-filling curves (SFC; Sagan, 1994; Bader, 2013) to index the hierarchy of grids, and (2) adopts a parallel tree traversal (PTT) algorithm (Bédorf et al., 2012; Chitalu et al., 2018; Goldfarb et al., 2013; Karras, 2012; Lohr, 2009; Nam et al., 2016; Zola et al., 2014) to replace the DFT algorithm. To be of practical relevance, our implementation of wavelet-based AMR on the GPU must also make the adaptive HWFV1 model competitive with a GPU-parallelised uniform-grid shallow water model.

The rest of this paper is organised as follows. Section 2 describes the parallelisation of an adaptive HWFV1 shallow water model running entirely on the GPU (GPU-HWFV1). Section 3 includes comparisons of GPU-HWFV1 against the uniform-grid FV1 solver of the open-source LISFLOOD-FP 8.0 package (Shaw et al., 2021), referred to hereafter as GPU-FV1. For completeness, GPU-HWFV1 is also compared against its sequential CPU implementation (CPU-HWFV1) which has already been extensively benchmarked for one-dimensional (1D) and 2D shallow water test cases (Kesserwani et al., 2019; Kesserwani & Sharifian, 2020). Section 4 presents conclusions on the potential benefits of GPU-HWFV1 for shallow water modelling applications.

2. Developing GPU-HWFV1

This section presents the parallelisation of the adaptive Haar wavelet (HW) first-order finite volume (FV1) shallow water model (HWFV1) developed in Kesserwani & Sharifian, (2020). The adaptive HWFV1 model is challenging to parallelise because it applies multiresolution analysis (MRA) to generate a non-uniform grid at every timestep – recall that MRA is recursive in nature and therefore inherently sequential. These reasons behind this are examined in Section
2.1 which gives an overview of the original sequential HWFV1 model. Section 2.2 then presents solutions with regards to reformulating the recursive operations featured in the MRA into a form suitable for parallelisation to implement the HWFV1 model entirely on the GPU.

### 2.1. Overview of the sequential HWFV1 model

The HWFV1 model runs shallow water simulations by solving the two-dimensional (2D) shallow water equations (SWE) over a dynamically adaptive grid using an FV1 scheme. HWFV1 adopts wavelet-based adaptive mesh refinement (AMR) to reduce the cost of solving the SWE by generating a non-uniform grid at every timestep. This dynamically adaptive grid uses finer cells only where higher accuracy is needed. Most AMR methods generate a non-uniform grid by selectively refining a baseline uniform grid at the coarsest resolution, but wavelet-based AMR starts with a baseline uniform grid at the finest resolution and coarsens it.

The non-uniform grid is generated by performing a so-called multiresolution analysis (MRA) using Haar wavelets (HW) (Haleem et al., 2015; Kesserwani & Sharifian, 2020). The MRA starts on a uniform grid made up of $2^L \times 2^L$ cells, where $L$ is a user-specified integer denoting the maximum refinement level controlling the resolution of the finest grid. The conservative form of the SWE are initially discretised over this uniform grid, and can be written as:

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) + \partial_y \mathbf{G}(\mathbf{U}) = \mathbf{S}_b(\mathbf{U}) + \mathbf{S}_f(\mathbf{U}). \quad (1)$$

Where $\partial_t$, $\partial_x$ and $\partial_y$ represent partial derivatives with respect to $t$, $x$ and $y$, respectively. The vector $\mathbf{U} = [h, hu, hv]^T$ contains the flow variables, $\mathbf{F} = [hu, (hu)^2/h + 1/2gh^2, hv]^T$, $\mathbf{G} = [hv, huv, (hv)^2/h + 1/2gh^2]^T$ are the components of the flux vector, and $\mathbf{S}_b = [0, -gh\partial_x z, -gh\partial_y z]^T$ and $\mathbf{S}_f = [0, -C_f u \sqrt{u^2 + v^2}, -C_f v \sqrt{u^2 + v^2}]^T$ 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.
In $S_b$, $z(x, y)$ is the topographic elevation (m), and in $S_f$, $C_f = g n_M^2 / h^{1/3}$ where $n_M$ is Manning’s coefficient (s$^{-1}$m$^{1/3}$).

The FV1 scheme locally approximates the variables $h$, $hu$, $hv$ and $z$ to be constant over each cell in the grid, for which scalar coefficients $h_c$, $(hu)_c$, $(hv)_c$ and $z_c$ are assigned per cell. Ordinarily, on a uniform grid, an FV1 update is performed at every timestep using a forward-Euler scheme applied to a spatial operator $L_c$ to advance the vector of flow coefficients $U_c = [h_c, (hu)_c, (hv)_c]^T$ in time as follows:

$$\partial_t U_c(t) = L_c. \quad (2)$$

To compute $L_c$, the local coefficients $S_c = \{h_c, (hu)_c, (hv)_c, z_c\}$ as well as the coefficients of the neighbour cells, denoted by $S_{west}$, $S_{east}$, $S_{north}$, and $S_{south}$, are needed. The expression for $L_c$ is established in previous studies (Liang, 2010) and is therefore outside of the scope of this paper.

In summary, $L_c$ involves flux calculations using a Harten-Lax-van Leer Riemann solver and also includes treatments to ensure the so-called well-balancedness of the FV1 scheme over wet/dry zones and fronts. These treatments are necessary for running accurate simulations of realistic shallow water test cases.

HWFV1 performs an MRA to generate a non-uniform grid that is adapted to the flow and topography at every timestep before performing the FV1 update. The MRA involves a hierarchy of grids of increasingly coarser resolution relative to the finest $2^L \times 2^L$ grid. Figure 1a shows a hierarchy of grids with maximum refinement level $L = 2$. 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 $n = 0$. Let $s^{(n)}$ denote the coefficients $s$ at refinement level $n$, where $s$ denotes any of the quantities in the set $S_c$ for ease of presentation of the MRA.
Figure 1: Multiresolution analysis (MRA). Left panel shows a hierarchy of grids involved in the MRA, with a maximum refinement level \( L = 2 \). Right panel shows how four cells at refinement level \( n + 1 \), called “children”, are related to a single cell at refinement level \( n \), called “parent”. Also shown are the coefficients \( s \) and details \( d \) that are involved in computations to realise the MRA.

When the MRA starts, only \( s^{(L)} \) are available, corresponding to the coefficients obtained from initially discretising the SWE over the finest grid. The MRA continues by producing \( s^{(n)} \) at the lower refinement levels, i.e., for \( n = L - 1, L - 2, \ldots, 1, 0 \). The coefficient \( s^{(n)} \) of a cell at refinement level \( n \) (called “parent”) is produced using the coefficients \( s^{(n+1)}_{[0]}, s^{(n+1)}_{[1]}, s^{(n+1)}_{[2]} \) and \( s^{(n+1)}_{[3]} \) of four cells at refinement level \( n + 1 \) (called “children”), in particular by applying Equation 3a\(^1\). Figure 1b shows the parent-children stencil, i.e., how the children \( s^{(n+1)}_{[0]}, s^{(n+1)}_{[1]}, s^{(n+1)}_{[2]} \) and \( s^{(n+1)}_{[3]} \) at level \( n + 1 \) are positioned relative to the parent \( s^{(n)} \) at level \( n \). Equations 3b - 3d are also applied to produce so-called “details”, denoted by \( d^{(n)}_{\alpha}, d^{(n)}_{\beta}, d^{(n)}_{\gamma} \). These details are used to decide which cells to include in the non-uniform grid based on a normalised detail \( d^{(n)}_{\text{norm}} = \max(d^{(n)}_{\alpha}, d^{(n)}_{\beta}, d^{(n)}_{\gamma})/s_{\text{max}} \), where \( s_{\text{max}} \) is the largest coefficient for all \( s^{(L)} \).

Cells whose \( d^{(n)}_{\text{norm}} \) is greater than \( 2^{n-L} \varepsilon \) are deemed to have significant details, where \( \varepsilon \) is a

---

\(^1\) In Equations 3a - 4d, \( H^0, H^1, G^0 \) and \( G^1 \) are scalar coefficients obtained from the Haar wavelets whose derivation is available in previous works (Haleem et al., 2015; Keinert, 2003), and is outside of the scope of this paper.
user-specified error threshold.

\[
\begin{align*}
  s^{(n)} &= H^0(\mathcal{H}_0 s^{(n+1)}_0 + \mathcal{H}_1 s^{(n+1)}_1) + H^1(\mathcal{H}_0 s^{(n+1)}_2 + \mathcal{H}_1 s^{(n+1)}_3) \quad (3a) \\
  d^{(n)}_\alpha &= H^0(\mathcal{G}_0 s^{(n+1)}_0 + \mathcal{G}_1 s^{(n+1)}_1) + H^1(\mathcal{G}_0 s^{(n+1)}_2 + \mathcal{G}_1 s^{(n+1)}_3) \quad (3b) \\
  d^{(n)}_\beta &= H^0(\mathcal{G}_0 s^{(n+1)}_0 + \mathcal{G}_1 s^{(n+1)}_1) + H^1(\mathcal{G}_0 s^{(n+1)}_2 + \mathcal{G}_1 s^{(n+1)}_3) \quad (3c) \\
  d^{(n)}_\gamma &= H^0(\mathcal{G}_0 s^{(n+1)}_0 + \mathcal{G}_1 s^{(n+1)}_1) + H^1(\mathcal{G}_0 s^{(n+1)}_2 + \mathcal{G}_1 s^{(n+1)}_3) \quad (3d)
\end{align*}
\]

The process of computing \(s^{(n)}\), \(d^{(n)}_\alpha\), \(d^{(n)}_\beta\) and \(d^{(n)}_\gamma\) 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, \(n = L - 1\), to the grid at the lowest refinement level, \(n = 0\). The inner loop iterates through each cell in the grid while applying Eqs. 3a - 3d (lines 5 and 6) and flagging significant details (line 8).

1 \textbf{algorithm} ENCODING(L, \varepsilon) \\
2 \hspace{1em} \textbf{for each} grid in hierarchy from \(L - 1\) to 0 \textbf{do} \\
3 \hspace{2em} \textbf{for each} cell in the grid \textbf{do} \\
4 \hspace{3em} \textbf{load children} \(s^{(n+1)}_0, s^{(n+1)}_1, s^{(n+1)}_2, s^{(n+1)}_3\) \textbf{for} 3a - 3d \\
5 \hspace{3em} \textbf{compute parent coefficient} \(s^{(n)}\) \textbf{using equation} 3a \\
6 \hspace{3em} \textbf{compute details} \(d_\alpha, d_\beta, d_\gamma\) \textbf{using equations} 3b - 3c \\
7 \hspace{3em} \textbf{compute normalised detail} \(d_{\text{norm}}\) \\
8 \hspace{3em} \textbf{flag as significant if} \(d_{\text{norm}} \geq 2^n \cdot \varepsilon\) \\
9 \hspace{2em} \textbf{end for} \\
10 \hspace{1em} \textbf{end for} \\
11 \textbf{end algorithm}

Algorithm 1: Pseudocode for the process of computing \(s^{(n)}\), \(d^{(n)}_\alpha\), \(d^{(n)}_\beta\) and \(d^{(n)}_\gamma\) at the lower refinement levels, called “encoding”.

Flagging significant details during the encoding process results in a tree-like structure of significant details that can be used to identify the cells that make up the non-uniform grid. Figure 2a shows an example of a tree of significant details. The tree is traversed starting from the coarsest cell while applying Equations 4a - 4d, stopping when either a cell on the finest grid
is reached or a cell with detail that is not significant is reached. This way of traversing the tree corresponds to the application of the depth-first traversal (DFT) algorithm to the tree (Sedgewick & Wayne, 2011). The path of the DFT can be traced in Figure 1a by following the cells labelled from 0 to 12 in ascending order. The cells that are visited during the DFT where the tree terminates are identified as “leaf” cells (indicated in blue and green in Figure 1a), which are assembled into a non-uniform grid. Figure 1b shows the non-uniform grid generated after assembling the leaf cells in Figure 1a.

Figure 2: Left panel shows the tree-like structure obtained after flagging significant details during the process of encoding; the cells where the tree terminates are called “leaf” cells (highlighted in green and blue). Right panel shows how the leaf cells are assembled into a non-uniform grid.

\[ s_{[0]}^{(n+1)} = H^0 (H^0 s^{(n)} + G^0 d_\alpha^{(n)}) + G^0 (H^0 d_\beta^{(n)} + G^0 d_\gamma^{(n)}) \]  
\[ s_{[1]}^{(n+1)} = H^1 (H^0 s^{(n)} + G^0 d_\alpha^{(n)}) + G^1 (H^0 d_\beta^{(n)} + G^0 d_\gamma^{(n)}) \]  
\[ s_{[2]}^{(n+1)} = H^0 (H^1 s^{(n)} + G^1 d_\alpha^{(n)}) + G^0 (H^1 d_\beta^{(n)} + G^1 d_\gamma^{(n)}) \]  
\[ s_{[3]}^{(n+1)} = H^1 (H^1 s^{(n)} + G^1 d_\alpha^{(n)}) + G^1 (H^1 d_\beta^{(n)} + G^1 d_\gamma^{(n)}) \]  

The process of performing a DFT while applying Equations 4a - 4d and identifying leaf cells is called “decoding”. Algorithm 2 shows pseudocode describing the decoding process.

The algorithm launches at the coarsest cell in the tree. The children \( s_{[0]}^{(n+1)} \), \( s_{[1]}^{(n+1)} \), \( s_{[2]}^{(n+1)} \) and \( s_{[3]}^{(n+1)} \) of this cell are computed using Equations 4a - 4d (line 6) and then the algorithm is
relaunched using the children coefficients at one refinement level higher (lines 7 to 10). The
algorithm is recursively launched unless 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).

1 recursive algorithm 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 4a - 4d
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 recursive algorithm

Algorithm 2: Pseudocode for performing a depth-first traversal of the tree of significant details
(obtained after encoding) while applying Equations 4a - 4d, called “decoding”.

Decoding allows to identify leaf cells and assemble them into a non-uniform grid. To
compute $L_c$ and perform the FV1 update on this grid, the sets of coefficients of the neighbours
$S_{west}$, $S_{east}$, $S_{north}$ and $S_{south}$ need to be retrieved for each leaf cell. Retrieving $S_{west}$, $S_{east}$, $S_{north}$ and
$S_{south}$ is trivial on a uniform grid because a cell can look left, right, up and down to find its
neighbours, but this is not as straightforward on a non-uniform grid because a cell can have
multiple neighbours in a given direction. Figure 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. HWFV1 avoids
this confusion by taking $S_{east}$ to be the set of coefficients of the eastern neighbour at the same
refinement level (yellow cell in Figure 3), which is readily available since encoding/decoding
produces $s^{(n)}$ at all refinement levels. Being able to find the neighbours and retrieving $S_{west}$,
$S_{east}$, $S_{north}$ and $S_{south}$ makes it trivial to compute $L_c$ and perform the FV1 update per cell.
Figure 3: Finding the neighbours of a cell in a non-uniform grid to retrieve the sets $S_c$, $S_{\text{west}}$, $S_{\text{east}}$, $S_{\text{north}}$, and $S_{\text{south}}$ in order to compute the spatial operator $\mathbf{L}_c$.

This completes the description of the steps involved in the HWFV1 model, and Algorithm 3 shows pseudocode summarising HWFV1. To run HWFV1, the user needs to specify the maximum refinement level ($L$), the error threshold ($\varepsilon$) and the simulation end time ($t_{\text{end}}$) (line 1). HWFV1 runs in a loop until $t_{\text{end}}$ is reached (lines 3 to 10). A non-uniform grid is generated every iteration of the loop, i.e., at every timestep (lines 4 to 6). Note that after the first timestep, the details are zeroed first before re-encoding, and encoding is performed only along the tree of significant details. On this non-uniform grid, an FV1 update is performed (line 7), after which the simulation time is incremented by the current timestep (line 8) while a new timestep is recomputed based on the Courant-Friedrich-Lewy (CFL) condition (line 9); for stability, a CFL number of 0.5 is used (Kesserwani & Sharifian, 2020).

```
1 algorithm HWFV1($L$, $\varepsilon$, $t_{\text{end}}$)
2     get $s^{L_0}$ from initial discretisation of SWE on 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 algorithm
```

Algorithm 3: Pseudocode summarising the steps involved in the HWFV1 model.
2.2. Parallelisation of HWFV1 on the GPU

The HWFV1 model, summarised in Algorithm 3, is parallelised on an NVIDIA GPU using CUDA. In the CUDA programming model, instructions are executed in parallel by threads. When parallelising on a GPU using CUDA, two important considerations are to maximise coalesced memory access and minimise warp divergence. Coalesced memory access is when adjacent threads access contiguous memory locations. Warp divergence is when threads within a warp – a group of 32 threads – execute different instructions. A naive parallelisation of the steps in Algorithm 3 would not properly address the issues of coalesced memory access and/or warp divergence. Consider the parallelisation of the encoding process (line 4 of Algorithm 3; Algorithm 1) assuming the cells in the hierarchy of grids are naively indexed in row-major order. Figure 4 shows the hierarchy of grids in Figure 1a indexed in this 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 (line 3 of Algorithm 1) because there are gaps between the memory locations.

Figure 4: Indexing the hierarchy of grids 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).
A different type of indexing is needed that ensures that coefficients that are nearby in the grid are also nearby in memory. Space-filling curves (SFCs) allow this kind of indexing by definition because they map spatial data to a one-dimensional line such that data close together in space tend to be close together in the line (Bader, 2013; Sagan, 1994). There are a few different types of SFCs such as the Siepinski curve, the Peano curve, the Hilbert curve and the Z-order curve (also known as Lebesgue or Morton curve). All of these SFCs have been previously used in the context of AMR (Brix et al., 2009; Burstedde et al., 2011; Meister et al., 2016; Weinzierl & Mehl, 2011), and also once in the context of wavelet-based AMR (Brix et al., 2009), but these did not perform the AMR on the GPU. To the authors’ knowledge, this is the first work to use a SFC to implement wavelet-based AMR on the GPU. Specifically, the Z-order SFC is chosen because its motif matches exactly with the parent-children square stencil shown in Figure 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 bits of its $i$ and $j$ positional indices in the grid. Figure 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 via its (unique) Morton code.
Figure 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 in binary). The right panel shows how these Morton codes (in decimal form) are followed in ascending order to create a Z-order curve and enforce Z-order indexing of 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 6 shows Z-order indexing of the cells in the hierarchy of grids in Figure 1a, 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. Enforcing this Z-order indexing allows for coalesced memory access during encoding because the children are in contiguous memory locations, as seen in the right panel of Figure 6.

Figure 6: Z-order indexing of the hierarchy of grids so that each cell in the hierarchy has a unique Z-order index (left panel) and the corresponding locations of the children in memory (right panel).
Parallelising the decoding process (line 5 of Algorithm 3; Algorithm 2) is more difficult than parallelising the encoding process because decoding involves DFT which is recursive and therefore inherently sequential. To overcome this difficulty, decoding is broken down into two parts that are parallelised separately: the first part is the application of Equations 4a - 4d and the second part is the identification of leaf cells. Hereafter, decoding refers exclusively to the application of Equations 4a - 4d, not the identification of leaf cells.

Decoding can be parallelised relatively easily because it can be performed using loops (similar to those in Algorithm 1) instead of using DFT. Algorithm 4 shows pseudocode describing how to perform decoding in parallel. The pseudocode involves an outer loop and a parallel 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 $s^{(n)}$ and the details $d^{(n)}_\alpha$, $d^{(n)}_\beta$, and $d^{(n)}_\gamma$ (line 5) and computes the children $s^{(n+1)}_0$, $s^{(n+1)}_1$, $s^{(n+1)}_2$, $s^{(n+1)}_3$ (line 6). This parallel inner loop, in particular the loading of the children coefficients in line 5, has coalesced memory access because of Z-order indexing; this can be seen by interpreting the arrows in Figure 6 in the reverse direction.

```
1 algorithm PARALLEL_DECODING(L)
2    for each grid in hierarchy from 0 to L - 1 do
3        for each cell in the grid do in parallel
4           if detail is significant then
5              load parent $s^{(n)}$ and details $d^{(n)}_\alpha$, $d^{(n)}_\beta$, $d^{(n)}_\gamma$
6              compute children $s^{(n+1)}_0$, $s^{(n+1)}_1$, $s^{(n+1)}_2$, $s^{(n+1)}_3$ 4a - 4d
7           end if
8        end for
9    end for
10 end algorithm
```

Algorithm 4: Pseudocode for the decoding process in parallel.

Unlike decoding, the identification of leaf cells is fundamentally a tree traversal problem. There have been many investigations into parallel tree traversal (PTT) algorithms on
the GPU (Bédorf et al., 2012; Chitalu et al., 2018; Goldfarb et al., 2013; Karras, 2012; Lohr, 2009; Nam et al., 2016; Zola et al., 2014), hinting that the identification of leaf cells can be parallelised by adopting a PTT algorithm. In this work, a modified version of the PTT algorithm developed by Karras (2012) is adopted because it can be easily modified to work with Z-order indexing. The PTT algorithm traverses the tree of significant details as follows, explained by example considering the tree in Figure 2a without loss of generality.

Figure 7a shows the tree after enforcing Z-order indexing of the hierarchy of grids, and different traversal paths are highlighted in cyan, magenta, yellow and grey. The PTT algorithm starts by launching as many threads as there are cells on the finest grid ($2^2 \times 2^2 = 16$). Each thread tries to reach a target cell on the finest grid by traversing progressively finer cells. A thread is denoted by $t_m$, where $m$ is the thread index (here $m = 0, 1, \ldots, 15$). The target cell of $t_m$ is the cell on the finest grid with a Morton code with the same thread index $m$, e.g., $t_3$ has thread index 3 and tries to reach the cell on the finest grid with Morton code 3. A thread stops if it either reaches this target cell or encounters a cell with a detail that is not significant (analogous to identifying a leaf cell). The thread then records the Z-order index of the cell at which it stops. Figure 7b shows the traversal paths of each thread during PTT in terms of the Z-order indices of the cells they traverse. These paths show that divergence is greatly minimised because adjacent threads perform similar traversals. For example, $t_0$ to $t_3$ are adjacent threads and they follow the same cyan path in Figure 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. Figure 7c shows the Z-order indices recorded by each thread after PTT is complete. These Z-order indices correspond to the indices of leaf cells.

2 Note that the Morton code refers to the index of a cell in a single grid, whereas the Z-order index refers to the index of a cell within the hierarchy. For example, the cell on the finest grid with Morton code 3 (see Figure 5b) has a Z-order index in the hierarchy of 8 (see Figure 6).
Figure 7: Parallel tree traversal (PTT). The left panel shows the tree of significant details after enforcing Z-order indexing, with different traversal paths indicated in yellow, cyan, magenta and grey. The middle panel shows the traversal paths of each thread during the PTT 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 PTT is complete.

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

Some of the indices recorded after PTT are duplicates because some threads ($t_0$ to $t_3$ and $t_{12}$ to $t_{15}$) finish at the same leaf cell. These duplicates are used to record the Z-order indices of the neighbour cells in parallel (line 6 of Algorithm 3) by making each thread in the grid in Figure 7c look left, right, up and down. The Z-order indices of the leaf cells and their neighbours are stored in memory. Figure 8a shows how the Z-order indices of the leaf cells and their neighbours are stored in five contiguous arrays. Duplicate groups of indices are removed.
as indicated by the double black lines via so-called parallel stream compaction using the CUB library (Merrill, 2022). Figure 8b shows the Z-order indices of the leaf cells and their neighbours without any duplicates, stored in five arrays. The leaf cells make up a non-uniform grid.

![Figure 8: Parallel FV1 update. Top panel shows the Z-order indices of the leaf cells and their neighbours stored in memory after PTT and neighbour finding. Bottom panel shows the Z-order indices of the leaf cells and their neighbours without any duplicates, used to retrieve \( S_{c}, S_{west}, S_{east}, S_{north} \) and \( S_{south} \) to compute \( L_{c} \) and perform the FV1 update.](image)

Figure 8: Parallel FV1 update. Top panel shows the Z-order indices of the leaf cells and their neighbours stored in memory after PTT and neighbour finding. Bottom panel shows the Z-order indices of the leaf cells and their neighbours without any duplicates, used to retrieve \( S_{c}, S_{west}, S_{east}, S_{north} \) and \( S_{south} \) to compute \( L_{c} \) and perform the FV1 update.

The next step is to parallelise the FV1 update on the leaf cells making up the non-uniform grid (line 7 of Algorithm 3) which is relatively simple. The parallel FV1 update launches one thread per leaf cell. In Figure 8b, there are ten leaf cells, so ten threads \( t_{0} \) to \( t_{9} \) are launched. Each thread uses the arrays of leaf cell and neighbour Z-order indices to retrieve \( S_{c}, S_{west}, S_{east}, S_{north} \) and \( S_{south} \) from within the hierarchy. For example, \( t_{0} \) would use the first column of indices (red box in Figure 8b), \( t_{1} \) would use the next column, and so on. Since each thread \( t_{0} \) to \( t_{9} \) retrieves \( S_{c}, S_{west}, S_{east}, S_{north} \) and \( S_{south} \) for each leaf cell, \( L_{c} \) can be computed to perform the FV1 update in parallel for all leaf cells. After the FV1 update, incrementing the current simulation time by the timestep (line 8 of Algorithm 3) is trivial. The new minimum timestep based on the CFL condition (line 9 of Algorithm 3) is computed by performing a so-called parallel minimum reduction using the CUB library (Merrill, 2022). Hence, lines 4 to 9 of
Algorithm 3 (the steps involved in the HWFV1 model) are all parallelised and GPU-HWFV1 is complete.

3. Numerical results

The proposed GPU-HWFV1 model is benchmarked against two existing shallow water models (Table 1). Benchmarking GPU-HWFV1 against its CPU predecessor, CPU-HWFV1, allows to quantify the benefit of parallelising the wavelet-based AMR of HWFV1. Benchmarking it against the GPU-FV1 flow solver in LISFLOOD-FP 8.0 leads to identifying the simulation scenarios where running simulations on a dynamically adaptive grid outperforms a uniform grid. The benchmarking is performed for five test cases (see Table 2) to explore a range of topographies and flow dynamics. It entails checking the closeness of GPU-HWFV1’s simulation outputs to the predictions made by GPU-FV1 with reference to exact solutions or experimental data (to verify robustness) as well as also assessing its runtime performance (to quantify efficiency).

| Model name   | Developed in              | Grid type                                      | GPU-parallelised |
|--------------|---------------------------|------------------------------------------------|-----------------|
| CPU-HWFV1    | (Kesserwani & Sharifian, 2020) | Same adaptive grid as GPU-HWFV1                | No              |
| GPU-FV1      | (Shaw et al., 2021)       | Uniform grid at the finest resolution accessible to the HWFV1 models | Yes             |

The first test case (Section 3.1) will be used to verify robustness, by confirming the well-balanced property of GPU-HWFV1 (i.e. ability to preserve quiescent flow in the presence of steep terrain including wet/dry zones and fronts), and by checking its ability to replicate the accuracy of reference GPU-FV1 predictions when simulating dynamic flows over the same steep terrain.
Table 2: List of the test cases used to benchmark GPU-HWFV1.

| Test name | Test type | Reason for use | Previously used in |
|-----------|-----------|----------------|--------------------|
| Quiescent flow over irregular topographies with different steepness. Dam-break flow over realistic terrain with friction effects (Section 3.1) | Synthetic | Verifying robustness | (Huang et al., 2013; Kesserwani et al., 2018; Kesserwani & Sharifian, 2020; Shirvani et al., 2021; Song et al., 2011) |
| Circular 2D dam-break flow (Section 3.2) | Synthetic | Assessing runtime performance | (Kesserwani & Sharifian, 2020; Wang et al., 2011) |
| Pseudo-2D dam-break flow (Section 3.2) | Synthetic | Assessing runtime performance | (Kesserwani et al., 2019; Kesserwani & Sharifian, 2020) |
| Dam-break wave interaction with an urban district (Section 3.3) | Experimental | Assessing runtime performance | (Caviedes-Voullième et al., 2020; Jeong et al., 2012; Kesserwani & Sharifian, 2020) |
| Tsunami wave propagation over a complex beach (Section 3.3) | Experimental | Assessing runtime performance | (Arpaia & Ricchiuto, 2018; Caviedes-Voullième et al., 2020; Hou et al., 2018; Kesserwani & Sharifian, 2020) |

After confirming the validity of using the proposed GPU-HWFV1 model, the second and third test cases (Section 3.2) will further focus on assessing the runtime performance of GPU-HWFV1 against CPU-HWFV1 and GPU-FV1 for synthetic dam break flows over flat terrain. As there is no terrain data to consider in these scenarios, they will be used to systematically analyse the runtime performance in relation to the sensitivity to triggering grid refinement ($\varepsilon$), the depth in grid resolution ($L$), and the flow type (vigorous or smooth). Therefore, simulations are run by pairing different values for $\varepsilon$ and $L$, with $\varepsilon = \{10^{-4}, 10^{-3}, 10^{-2}\}$ to cover the recommended ranges for maintaining a fair balance between the predictive accuracy and runtime performance (Kesserwani et al., 2019; Kesserwani & Sharifian, 2020).
and, $L = \{8, 9, 10, 11\}$ as no gains in runtime performance was identified for $L \leq 7$ and using $L \geq 12$ was not affordable within the memory capacity of the GPU card used (RTX 2070). The effects of these parameters and the flow type on the wavelet-based AMR of the HWFV1 models is summarised in Table 3.

Table 3: Aspects against which the runtime performance of GPU-HWFV1 over CPU-HWFV1 and over GPU-FV1 are assessed.

| Aspects | Description | Finest grid resolution |
|---------|-------------|------------------------|
| $L$     | Controls the finest accessible grid resolution | Deeper with higher $L$ |
| $\varepsilon$ | Controls how far the finest grid resolution is accessed | More accessible with smaller $\varepsilon$ |
| Flow    | Vigorous (with discontinuities) to smooth (up to flat) | Triggered often for vigorous flows |

The expectations on runtime performance established from the synthetic test cases will be finally explored in the fourth and fifth test cases (Section 3.3), which involve realistic topographies represented by Digital Elevation Models (DEM) and using experimental data for model verification. Running the HWFV1 model with the presence of a DEM means that the maximum refinement $L$ must be set to accommodate the DEM resolution without allowing any coarsening in its grid beyond what the MRA of the DEM suggests.

3.1. Verification of robustness

The first synthetic test case in Table 2 is considered to verify the robustness of GPU-HWFV1, with a dual objective: (i) to verify its well-balanced property in the presence of wet-dry fronts with different levels of steepness in topography and different wetting conditions and, (ii) to reproduce a realistic dam-break flow with friction effects and moving wet-dry fronts. The domain area is $70 \text{ m} \times 30 \text{ m}$ with closed wall boundaries and includes humps to represent an irregular topography profile. To verify the well-balanced property for realistic topographies,
three hump shapes are considered with increasingly steeper bed slopes as shown in the top panels of Figure 9 (smooth on the left, steeper in the middle and rectangular on the right). For each hump shape, appropriate initial conditions are applied (Table 4) with zero velocities to generate an unmoving free-surface elevation that leads to different wetting conditions around and/or at the humps.

Figure 9: Verification of robustness. Well-balanced property verification in the presence of wet-dry fronts with different levels of steepness in topography and different wetting conditions: smooth humps (left panels), steeper humps (middle panels) and rectangular humps (right panels). The top panels show the geometrical profiles of the humps in the domain area, and the lower panels include the time history of the maximum discharge errors where $q_x = hu$ and $q_y = hv$. 

$\textbf{Figure 9:}$ Verification of robustness. Well-balanced property verification in the presence of wet-dry fronts with different levels of steepness in topography and different wetting conditions: smooth humps (left panels), steeper humps (middle panels) and rectangular humps (right panels). The top panels show the geometrical profiles of the humps in the domain area, and the lower panels include the time history of the maximum discharge errors where $q_x = hu$ and $q_y = hv$. 

$10^{-12}$
Table 4: Zero-velocity initial conditions applied to generate an unmoving free-surface flow for the three hump profiles shown in Figure 9.

| 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 | (Huang et al., 2013; Shirvani et al., 2021; Song et al., 2011) |
| 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) |

GPU-HWFV1 simulations are run up to 100 s with a maximum refinement level $L = 8$ and an error threshold $\varepsilon = 10^{-3}$ (requiring around 3,000 timesteps to complete). The time histories of the maximum discharge errors are shown in the bottom panels of Figure 9 for the three hump profiles. These errors are seen to become increasingly higher with increased irregularity in the hump profile but remain bounded as also observed for the CPU model counterparts (Kesserwani & Sharifian, 2020). This demonstrates that GPU-HWFV1 is numerically well-balanced irrespective of the steepness of the bed slope and the presence of wet-dry zones and fronts in the domain area.

Next, GPU-HWFV1 is applied to reproduce a frictional dam-break flow ($n_M = 0.018$ m$^{1/3}$/s) for the smooth hump profile (top left panel, Figure 9). The initial dam-break flow conditions assume a water body of $h = 1.875$ m held by an imaginary dam located at $x = 16$ m with zero discharges. Using the same choice of $\varepsilon$ and $L$, a GPU-HWFV1 simulation is run up to 12 s. A GPU-FV1 simulation on the fine uniform grid is also performed to allow for like-for-like comparisons of flood depth profiles at outputs times reported in previous studies (Shirvani et al., 2021; Song et al., 2011). Figure 10 includes the 2D contour maps of the flood
depths predicted by GPU-HWFV1 (left panel) compared to those predicted by GPU-FV1 (right panel) at 0, 6 and 12 s. At 0 s (top panel), both models are seen to start from the same flood depth profile. 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, and the $L^1$ error difference between depths predicted by GPU-HWFV1 and GPU-FV1 is $4.6 \times 10^{-4}$. There are similar wave patterns surrounding the large hump by 12 s (bottom panel) and the $L^1$ error is $9.2 \times 10^{-4}$. In all the predictions, GPU-HWFV1 shows symmetrical flood extent profiles that are similar to those reproduced GPU-FV1 and other hydrodynamic profiles reported in previous works (Shirvani et al., 2021; Song et al., 2011). This indicates that the GPU-HWFV1 implementation is as robust as well-established models used for real-world applications.

Figure 10: Verification of robustness. Realistic dam-break flow with friction effects and moving wet-dry fronts. Flood depth profiles predicted by GPU-HWFV1 and GPU-FV1 on the left and right panels, respectively. At 0 s, the dam break wave emerges (top panels). At 6 s, the wave has submerged the small humps (middle panels). At 12 s, the wave starts to surround the large hump (bottom panels).
3.2. Assessing runtime performance for synthetic test cases

**Circular 2D dam-break flow.** This test case has often been used to verify new model implementations by capturing the symmetric propagation of shocks and rarefaction waves in the closed [-20 m, 20 m]² domain area (Toro, 2001). Initially, the water depth inside the cylindrical dam is 2.5 m, separating it from a water depth of 0.5 m elsewhere. The dam-break flow occurs over a frictionless and flat terrain, resulting in a shock moving radially outwards and a rarefaction wave moving radially inwards, which eventually collapses to form a secondary shock. It is first used to further verify GPU-HWFV1 using the same choice of $\varepsilon$ and $L$ as in Section 3.1 and by comparing its simulation outputs to those of CPU-HWFV1 and FV1-GPU. As in (Kesserwani & Sharifian, 2020), simulations are run up to $t = 3.5$ s for GPU-HWFV1, CPU-HWFV1 and FV1-GPU. Figure 11 shows the water depth centrelines predicted by the three models. GPU-HWFV1 predicts water depths that are identical to those predicted by CPU-HWFV1, GPU-FV1 and the reference solution. The reference solution was produced using the FV1 numerical solution to 1D radial form of the 2D shallow water equations using $256 \times 256$ cells, following (Toro, 2001).

![Figure 11: Circular 2D dam-break flow. Verification of GPU-HWFV1 using the same choice of $\varepsilon$ and $L$ as in Section 3.1 ($L = 8$ and $\varepsilon = 10^{-3}$): water depth centrelines at 3.5 s predicted by GPU-HWFV1, CPU-HWFV1, and GPU-FV1 compared to the reference solution.](image)

To perform speed-up analysis, the models are rerun for the combinations of $\{\varepsilon, L\}$, and their runtimes were recorded for producing the speed-up ratios of GPU-HWFV1 relative to
CPU-HWFV1 and GPU-FV1, respectively. Figure 12 contains the plots of the speed-up ratios with increasing maximum refinement level $L$, relative to CPU-HWFV1 in the left panel and to GPU-FV1 in the right panel. The black lines indicate the average speed-up ratios obtained for the three error thresholds and the dash-dotted lines indicate the breakeven point above which GPU-HWFV1 demonstrates speed-up (used also in the subsequent figures).

![Figure 12: Circular 2D dam-break flow. Speed-up ratios to accomplish a 3.5 s simulation: GPU-HWFV1 over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel).](image)

GPU-HWFV1 is identified to be 5 to 46× faster than CPU-HWFV1. This speed-up is proportional to the increase in $L$ and decrease in $\varepsilon$. This suggests that wavelet-based AMR is much more efficient when parallelised on the GPU, in particular as the maximum resolution refinement level is deepened and the sensitivity to refine resolution is increased. Compared to the runtime performance of FV1-GPU, GPU-HWFV1 is not faster in this test (right panel of Figure 12) until $L \geq 9$ for all the $\varepsilon$ values, reaching a maximum of 3× for the largest $\varepsilon = 10^{-2}$, and around 2× for the smaller $\varepsilon = 10^{-3}$ and $10^{-4}$. This means that GPU-HWFV1, despite the overhead costs from the MRA process, can still compete with the speed of a fine uniform grid GPU-FV1 simulation even for a vigorous flow that would cause overrefinement on the adaptive grid. Namely, GPU-HWFV1 is likely to be faster than GPU-FV1 the deeper the grid resolution (which would lead to an excessively fine uniform grid for GPU-FV1) and the lower the sensitivity for triggering grid refinement. Next, a transient analysis of the speed-ups is
performed in a longer simulation that sees a gradual change in the flow from vigorous to very smooth.

**Pseudo-2D dam-break flow.** This 1D dam-break flow test case has conventionally been used to verify shallow water models for a short simulation run (2.5 s) involving transient shock and rarefaction waves propagation in two opposite directions. It was used recently for a much longer simulation time (40 s) to assess speed-up for CPU-based adaptive grid models to their uniform grid counterparts by considering a flow with gradual transition from vigorous to smooth (Kesserwani et al., 2019; Kesserwani & Sharifian, 2020).

![Figure 13: Pseudo-2D dam-break flow. Verification of GPU-HWFV1 using the same choice of $\varepsilon$ and $L$ as in Section 3.1 ($L = 8$ and $\varepsilon = 10^{-3}$): Water depths centerlines predicted by GPU-HWFV1, CPU-HWFV1 and GPU-FV1 at 2.5 s compared with the exact solution.](image)

The domain area is $50 \times 25$ m and assumed to be flat and frictionless with open boundary conditions. The dam, located at $x = 10$ m, initially separates an upstream water depth of 6 m from a downstream water depth of 2 m. After the dam removal, at $t = 0$ s, the shock and rarefaction waves remain present in the domain area up to 2.5 s. After 3 s, the shock wave has left the domain area from the downstream and the flow dynamics are only driven by the presence of the rarefaction wave until 10 s, after which it exits from the upstream. Therefore, after 10s, the flow dissipates gradually with increased smoothness until 40 s. The models are first verified by running simulations up to 2.5 s using the same choice of $\varepsilon$ and $L$ as in Section 3.1 ($L = 8$ and $\varepsilon = 10^{-3}$) for the HWFV1-based models and the finest uniform grid for the GPU-
FV1 model. Figure 13 shows the plots of the water depth centerlines predicted by the models all showing a good agreement with the exact solution (Delestre et al., 2011).

To assess speed-up, GPU-HWFV1 and CPU-HWFV1 simulations are rerun for up to 40 s for the combinations of \( \{\varepsilon, L\} \) alongside GPU-FV1 simulations on the finest uniform grid. Time histories of the runtimes are recorded throughout the 40 s simulations during which the flow transitions from vigorous to very smooth. Time series of the speed-up ratios of GPU-HWFV1 over CPU-HWFV1 and GPU-FV1 for the different values of \( L \) and \( \varepsilon \) are plotted in Figure 14.

Looking at the speed-up over CPU-HWFV1 (Figure 14, top panels), GPU-HWFV1 exceeds the breakeven for all but the largest \( \varepsilon = 10^{-2} \) and the lowest maximum refinement level \( L = 8 \) up to 2.5 s (Figure 14, top left panel). This means that CPU-HWFV1 only remained as fast as GPU-HWFV1 when the flow included the shock and the rarefaction waves and for the setting with the least depth in resolution refinement and the least sensitivity to trigger grid
refinement. However, even at $\varepsilon = 10^{-2}$ up to $8\times$ speed-up is noted after 3 s when the shock wave is not present anymore. With any other combinations of $\{\varepsilon, L\}$, there is a significant demonstration of speed-up: with reduced $\varepsilon$ and increased $L$, GPU-HWFV1 becomes increasingly faster than CPU-HWFV1 up to reaching, for the highest $L$ and smallest $\varepsilon$, an average speed-up of $68\times$ throughout the simulation and a maximum speed-up of $88\times$ at 2.5 s when flow discontinuities were still present. This confirms the benefit of parallelising the wavelet-based AMR on the GPU as an alternative to the CPU version for general purpose modelling involving all types of flow.

In terms of speed-ups over GPU-FV1 (Figure 14, bottom panels), at $\varepsilon = 10^{-2}$, GPU-HWFV1 demonstrates a maximum speed-up of $25\times$ when $L = 11$, though it could only outrun GPU-FV1 for $L \geq 9$, beyond which GPU-HWFV1 increasingly shows speed-up within increased smoothening in the flow. At $\varepsilon = 10^{-3}$, GPU-HWFV1’s maximum speed-up reduces to $12\times$ and outruns GPU-FV1 for $L \geq 10$, whereas for $L \leq 9$, it begins to demonstrate speed-up only after 10 s when the flow starts smoothening. This suggests to expect less speed-up over GPU-FV1 with increased sensitivity for triggering grid refinement with GPU-HWFV1 and reduced depth of the finest resolution. The same can be noted with $\varepsilon = 10^{-4}$, but here GPU-HWFV1 starts to be faster than FV1-GPU for $L \geq 9$ and the overall maximum speed-up reduces to $8\times$ (reached again after 10 s when the flow is smoothening). These analyses indicate that an adaptive-grid GPU-HWFV1 simulation is likely to be more efficient than a uniform-grid GPU-FV1 simulation for very fine resolution modelling of gradual to smooth flows, with $L \geq 9$, and when the sensitivity to grid refinement is not maximal, with $\varepsilon > 10^{-4}$.

3.3. Further investigations into runtime performance: realistic flow simulations

Dam-break wave interaction with an urban district. This test case has widely been used for model verification (e.g. Caviedes-Voullième et al. (2020)) as it has a set of spatial
experimental data for the water depth and the velocities (Soares-Frazão & Zech, 2008). It involves a dam-break wave propagation in a 36 m × 3.6 m smooth channel ($n_M = 0.01$) that includes a wall barrier with a gate initially separating an upstream water body of 0.4 m from a water depth of 0.011 m (Figure 15). Downstream of the gate, there are twenty-five 0.3 m × 0.3 m square blocks, with 0.1 m gaps. The ground height for the wall barrier and the square blocks is 2 m. Based on this height and the dimension reported in (Soares-Frazão & Zech, 2008), a DEM file was built at a resolution of 0.02 m × 0.02 m, made of 324,000 cells. The DEM includes the two rectangular blocks forming the wall barrier linked to the gate and the twenty-five square blocks. These discontinuous blocks are included in the grid and are accounted for as part of the well-balanced topography integration.

As the gate opens abruptly, 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 (e.g. Fig. 24 in Soares-Frazão and Zech (2008)).

![Figure 15: Dam-break wave interaction with an urban district. Top down view of the smooth channel with the gate indicated in green and topographic blocks coloured in yellow. Experimental depth and velocity data available along $y = 0.2$ m, indicated in red.](image)

A 10 s simulation is run using GPU-HWFV1 with $L = 11$ for two values of $\varepsilon = \{10^{-4}, 10^{-3}\}$, and using GPU-FV1 on a uniform grid using the finest resolution accessible to GPU-HWFV1. Figure 16 shows the water depth (left panel) and velocity (right panel) profiles along $y = 0.2$ m at 6 s predicted by the GPU-HWFV1 and GPU-FV1 as well as the experimental profiles. All the models predicted profiles are within the expected range of agreement with the experimental profiles (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.

![Graph showing depth and velocity profiles predicted by GPU-HWFV1 and GPU-FV1 compared to experiments.](image)

Figure 16: Dam-break wave interaction with an urban district. Depth (left panel) and velocity (right panel) profiles predicted along $y = 0.2$ m at 6 s by GPU-HWFV1 and GPU-FV1 compared to the experiments.

To analyse speed-ups, a CPU-HWFV1 simulation is also run. The recorded runtimes for the three models were used to calculate the time series of speed-up ratios of GPU-HWFV1 over CPU-HWFV1 and over GPU-FV1, which are plotted in the left and right panels of Figure 17. On average, GPU-HWFV1 is found 19× and 25× faster to run than CPU-HWFV1, with $\varepsilon = 10^{-3}$ and $10^{-4}$ respectively, throughout the 10 s simulation. Higher levels of speed-up are demonstrated with larger $\varepsilon$, which is in line with the findings in Section 3.2. GPU-HWFV1 is also faster than GPU-FV1 in this test, on average ~2.4× faster with both $\varepsilon = 10^{-3}$ and $10^{-4}$. This can be expected for a run with $L = 11$ accommodating the very fine resolution of the DEM. Up to 2 s, the run with GPU-HWFV1 at $\varepsilon = 10^{-3}$ demonstrates higher levels of speed-up than at $\varepsilon = 10^{-4}$, which is in line with the observations made in Section 3.2. In contrast, after 2 s, GPU-HWFV1 at $\varepsilon = 10^{-3}$ reduces the level of speed-up to become lower than with GPU-HWFV1 at $\varepsilon = 10^{-4}$. This could be due to GPU-HWFV1’s higher sensitivity to grid refinement around the rectangular and square topographic blocks. Overall, GPU-HWFV1, besides being more...
performant than CPU-HWFV1, also remains faster than GPU-FV1 for this test. Supported by the analysis in Section 3.2, this can be expected given the maximised depth in the resolution level ($L = 11$) needed to accommodate the domain size to the fine resolution of the DEM.

Figure 17: Dam-break wave interaction with an urban district. Speed-up ratios of GPU-HWFV1 over CPU-HWFV1 (left panel) and GPU-FV1 (right panel).

**Tsunami wave propagation over a complex beach.** The test case considers a 1:400 scaled replica of the 1993 Okushiri tsunami (Matsuyama & Tanaka, 2001). It has been used in other works for model verification and for runtime performance assessments of wavelet-based adaptive models versus their uniform counterparts for simulations on the CPU (Caviedes-Voullième et al., 2020; Kesserwani & Sharifian, 2020). It is here used to assess the runtime performance of the adaptive GPU-HWFV1 model versus CPU-HWFV1 and GPU-FV1 models.
The physical replica consists of a $5.488 \times 3.402$ m smooth area ($n_M = 0.01 \text{ m}^{1/3}/\text{s}$) that has a uniform resolution of $0.014 \times 0.014$ m on a DEM made of 163,840 cells (i.e. around twice fewer cells than the previous test case). The domain area has closed boundaries except for the western boundary through which a tsunami-generated inflow (Kesserwani & Sharifian, 2020) enters and eventually reaches the coastal area to the east, before which there is a gauge point ($x = 4.521$ m, $y = 1.696$ m) hit by the tsunami-generated flood wave. Experimental time histories of the free-surface water elevation are available at this point and will be used to verify the GPU-HWFV1 and GPU-FV1 models’ ability to achieve a 22.5 s simulation. Figure 18a displays a view of the domain area including the gauge point location, marked by a red dot, and the coastal area at the eastern end (yellow colour). Given the smaller size of the domain area, the depth in the resolution level for the DEM to the domain size requires using $L = 9$ in this test to run the GPU-HWFV1 simulations with $\varepsilon = \{10^{-3}, 10^{-4}\}$. The GPU-FV1 simulation was run on a uniform grid at the DEM resolution. Figure 18b contains time histories of the free-surface water elevations predicted by the models, which are in a good
agreement with the experimental time histories. It can be seen that all the models predict the expected gradual retraction in the free-surface elevation between 12 s and 15 s, followed by a sharp increase that peaks at around 17 s. GPU-HWFV1 at $\epsilon = 10^{-4}$ leads to predictions that are visually indistinguishable from those predicted by GPU-FV1. With $\epsilon = 10^{-3}$, the predictions remain comparable subject to small, localised discrepancies at times where there is a sharp flow transition such as at around 18 and 21 s.

Figure 19: Tsunami wave propagation over a complex beach. Speed-up ratios of GPU-HWFV1 over CPU-HWFV1 (left panel) and over GPU-FV1 (right panel).

Figure 19 contains the plots of the time histories of the speed-up ratios for GPU-HWFV1 over CPU-HWFV1 run with similar setting (left panel) and over GPU-FV1 (right panel) during the 22.5 s simulation. GPU-HWFV1 is seen to be significantly faster than CPU-HWFV1, leading to average speed-ups of 200× and 400× with $\epsilon = 10^{-3}$ and $10^{-4}$, respectively. Compared to the previous test case, the terrain is complex all over the domain and the grid cannot be coarsened much, leading to an overrefined grid that is further refined by flow disturbances during HWFV1 simulations. In such a case, GPU-HWFV1 demonstrates remarkable speed-up over CPU-HWFV1, which increases as $\epsilon$ is decreased from $10^{-3}$ to $\epsilon = 10^{-4}$ in line with the observations in Section 3.2, but the speed-up doubles in this test.
Compared to GPU-FV1, GPU-HWFV1 only demonstrates speed-up with $\varepsilon = 10^{-3}$ (around 1.25×) and is slightly slower to run with $\varepsilon = 10^{-4}$ where its speed-up falls below the breakeven line. This implies that it is worthwhile to perform wavelet-based AMR in this test for $\varepsilon = 10^{-3}$ but not for $\varepsilon = 10^{-4}$, at which the grid is overrefined (due to the terrain and flow disturbances). Overall, GPU-HWFV1 is shown to be generally faster than CPU-HWFV1 but could not outrun GPU-FV1 for $\varepsilon = 10^{-4}$ (which leads to an overrefined grid) and for $L = 9$ (to accommodate a relatively small domain). Nonetheless, for $\varepsilon = 10^{-3}$, GPU-HWFV1 remains a viable choice over GPU-FV1.

4. Conclusion

This paper presented an adaptive Haar wavelet (HW) first-order finite volume (FV1) shallow water model running entirely on the graphics processing unit (GPU), termed GPU-HWFV1. The GPU-HWFV1 model adopts a parallel wavelet-based adaptive mesh refinement (AMR) technique driven by the multiresolution analysis (MRA) of the HWs to generate a non-uniform grid at every timestep. The MRA involves a nested hierarchy of two-dimensional (2D) grids, where the coarsest grid in the hierarchy is made up of a single cell while the finest grid is made up of $2^L \times 2^L$ cells, and $L$ is a user-specified maximum refinement level. In the “encoding” (coarsening) process, a user-specified error threshold $\varepsilon$ is needed to flag significant “details” to decide which cells to include in the non-uniform grid. The process of encoding results in a tree-like structure of significant details which is traversed in the “decoding” (refinement) process by applying a depth-first traversal (DFT) algorithm to identify the cells making up the non-uniform grid. Encoding and decoding were parallelised on the GPU by adopting the indexing of a Z-order space-filling curve to ensure coalesced memory access. Meanwhile, the DFT algorithm was replaced with a parallel tree traversal algorithm to traverse the tree of significant details on the GPU with minimal warp divergence.
GPU-HWFV1 was first verified and then its runtime performance assessed against a sequential predecessor running on the central processing unit (CPU-HWFV1) as well as an operational GPU-FV1 uniform-grid shallow water model ran on the finest grid accessible to the HWFV1 models. The verification was performed using $\varepsilon = 10^{-3}$ (recommended for flood modelling) and $L = 8$ for four synthetic test cases involving motionless, vigorous, gradual and smooth flows. A systematic runtime performance assessment was performed for two synthetic test cases involving dam-break flow, where a lower and higher order-of-magnitude for $\varepsilon = \{10^{-2}, 10^{-3}, 10^{-4}\}$ was also considered in combination with an increase in the maximum refinement level $L = \{8, 9, 10, 11\}$. Verification and runtime performance assessments were finally performed for realistic test cases with digital elevation models (DEM) for which the value of $L$ was selected to match the resolution of the DEM, and by running GPU-HWFV1 for $\varepsilon = \{10^{-3}, 10^{-4}\}$.

The overall performance of GPU-HWFV1 for all the test cases provided strong evidence that it is as robust as GPU-FV1 in replicating the realistic flows including the presence of uneven topographies, wet-dry fronts and friction effects. In terms of runtime performance over CPU-HWFV1, GPU-HWFV1 yields significant speed-ups for all the test cases, ranging between $20\times$ to $400\times$. Hence, this work offers compelling evidence to apply the parallel wavelet-based AMR technique to other fields in computational engineering. From the systematic runtime performance assessment for the synthetic test cases, GPU-HWFV1 tends to demonstrate speed-up of around $1.1\times$ to $30\times$ over GPU-FV1 for $L \geq 9$ and/or by avoiding the smallest $\varepsilon = 10^{-4}$. For the test cases involving realistic flows over real DEMs, GPU-HWFV1 could comfortably show speed-up over GPU-FV1 for the test with $L = 11$ and for $\varepsilon = 10^{-3}$ for the test with $L = 9$. Hence, GPU-HWFV1 can be favoured to gain runtime performance over GPU-FV1 for shallow water modelling over real DEMs, namely with an increased fineness in the DEM resolution and an increased size of the domain area.
References

Akoh, R., Ishikawa, T., Kojima, T., Tomaru, M., & Maeno, S. (2017). High-resolution modeling of tsunami run-up flooding: A case study of flooding in Kamaishi city, Japan, induced by the 2011 Tohoku tsunami. *Natural Hazards and Earth System Sciences, 17*, 1871–1883. https://doi.org/10.5194/nhess-17-1871-2017

Arpaia, L., & Ricchiuto, M. (2018). r–adaptation for Shallow Water flows: conservation, well balancedness, efficiency. *Computers & Fluids, 160*, 175–203. https://doi.org/10.1016/J.COMPFLUID.2017.10.026

Bader, M. (2013). *Space-Filling Curves* (1st ed.). Springer Berlin, Heidelberg.

Beckingsale, D. (2015). *Towards scalable adaptive mesh refinement on future parallel architectures* [PhD]. University of Warwick.

Bédorf, J., Gaburov, E., & Portegies Zwart, S. (2012). A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics, 231*(7), 2825–2839. https://doi.org/10.1016/j.jcp.2011.12.024

Brix, K., Melian, S. S., Müller, S., & Schieffer, G. (2009). Parallelisation of Multiscale-Based Grid Adaptation using Space-Filling Curves. *Esaim: Proceedings, 29*, 108–129.

Brodtkorb, A. R., Setra, M. L., & Altinakar, M. (2012). Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids, 55*, 1–12. https://doi.org/10.1016/J.COMPFLUID.2011.10.012

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

Castro, M. J., Ortega, S., de la Asunción, M., Mantas, J. M., & Gallardo, J. M. (2011). GPU computing for shallow water flow simulation based on finite volume schemes. *Comptes Rendus Mécanique, 339*(2–3), 165–184. https://doi.org/10.1016/J.CRME.2010.12.004

Caviedes-Voullième, D., Gerhard, N., Sikstel, A., & Müller, S. (2020). Multiwavelet-based mesh adaptivity with Discontinuous Galerkin schemes: Exploring 2D shallow water problems. *Advances in Water Resources, 138*, 103559. https://doi.org/10.1016/J.ADVWATRES.2020.103559

Caviedes-Voullième, D., & Kesserwani, G. (2015). Benchmarking a multiresolution discontinuous Galerkin shallow water model: Implications for computational hydraulics. *Advances in Water Resources, 86*, 14–31. https://doi.org/10.1016/J.ADVWATRES.2015.09.016

Chitalu, F. M., Dubach, C., & Komura, T. (2018). Bulk-Synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. https://doi.org/10.1145/3190834.3190848

de Almeida, G., Bates, P., & Ozdemir, H. (2016). Modelling urban floods at sub-metre resolution: challenges or opportunities for flood risk management? *Journal of Flood Risk Management, 11*. https://doi.org/10.1111/jfr3.12276

Delestre, O., Lucas, C., Ksniat, P.-A., Darboux, F., Laguerre, C., Vo, T., James, F., & Cordier, S. (2011). *SWASHES: a library of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies*. 
Donat, R., Martí, M. C., Martínez-Gavara, A., & Mulet, P. (2014). Well-Balanced Adaptive Mesh Refinement for shallow water flows. *Journal of Computational Physics*, 257, 937–953. https://doi.org/10.1016/J.JCP.2013.09.032

Dunning, D., Marts, W., Robey, R. W., & Bridges, P. (2020). Adaptive mesh refinement in the fast lane. *Journal of Computational Physics*, 406, 109193. https://doi.org/https://doi.org/10.1016/j.jcp.2019.109193

Echeverribar, I., Morales-Hernández, M., Brufau, P., & García-Navarro, P. (2019). 2D numerical simulation of unsteady flows for large scale floods prediction in real time. *Advances in Water Resources*, 134, 103444. https://doi.org/10.1016/J.ADVWATRES.2019.103444

Flood Modeller 2D. (2022). *Flood Modeller by Jacobs*.

Gerhard, N., Caviedes-Voullième, D., Müller, S., & Kesserwani, G. (2015). Multiwavelet-based grid adaptation with discontinuous Galerkin schemes for shallow water equations. *Journal of Computational Physics*, 301, 265–288. https://doi.org/10.1016/J.JCP.2015.08.030

Giuliani, A., & Krivodonova, L. (2019). Adaptive mesh refinement on graphics processing units for applications in gas dynamics. *Journal of Computational Physics*, 381, 67–90. https://doi.org/https://doi.org/10.1016/j.jcp.2018.12.019

Goldfarb, M., Jo, Y., & Kulkarni, M. (2013). General Transformations for GPU Execution of Tree Traversals. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/2503210.2503223

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

HiPIMS-CUDA. (2021). *HiPIMS-CUDA*.

Hou, J., Wang, R., Liang, Q., Li, Z., Huang, M. S., & Hinkelmann, R. (2018). Efficient surface water flow simulation on static Cartesian grid with local refinement according to key topographic features. *Computers & Fluids*, 176, 117–134. https://doi.org/10.1016/J.COMPFLUID.2018.03.024

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

InfoWorks ICM. (2018). *GPU Runtime Results for 2D InfoWorks ICM Models*.

Jeong, W., Yoon, J. S., & Cho, Y. S. (2012). Numerical study on effects of building groups on dam-break flow in urban areas. *Journal of Hydro-Environment Research*, 6(2), 91–99. https://doi.org/10.1016/J.JHER.2012.01.001

Karrass, T. (2012, November 26). *Thinking Parallel, Part II: Tree Traversal on the GPU*. Keinert, F. (2003). *Wavelets and Multiscale Methods* (Vol. 42). Chapman & Hall/CRC Press.

Kesserwani, G., Ayog, J. L., & Bau, D. (2018). 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. https://doi.org/10.1016/J.CMA.2018.08.003
Kesserwani, G., & Sharifian, M. K. (2020). (Multi)wavelets increase both accuracy and efficiency of standard Godunov-type hydrodynamic models: Robust 2D approaches. *Advances in Water Resources, 144*, 103693. https://doi.org/10.1016/J.ADVWATRES.2020.103693

Kesserwani, G., Shaw, J., Sharifian, M. K., Bau, D., Keylock, C. J., Bates, P. D., & Ryan, J. K. (2019). (Multi)wavelets increase both accuracy and efficiency of standard Godunov-type hydrodynamic models. *Advances in Water Resources, 129*, 31–55. https://doi.org/10.1016/J.ADVWATRES.2019.04.019

Lacasta, A., Morales-Hernández, M., Murillo, J., & Garcia-Navarro, P. (2015). GPU implementation of the 2D shallow water equations for the simulation of rainfall/runoff events. *Environmental Earth Sciences*. https://doi.org/10.1007/s12665-015-4215-z

Lakhlifi, Y., Daoudi, S., & Boushaba, F. (2018). Dam-Break Computations by a Dynamical Adaptive Finite Volume Method. *Journal of Applied Fluid Mechanics, 11*(6), 1543–1556. https://doi.org/10.29252/jafm.11.06.28564

Liang, Q. (2010). Flood Simulation Using a Well-Balanced Shallow Flow Model. *Journal of Hydraulic Engineering, 136*(9), 669–675. https://doi.org/10.1061/(ASCE)HY.1943-7900.0000219

Liang, Q., & Borthwick, A. G. L. (2009). Adaptive quadtree simulation of shallow flows with wet–dry fronts over complex topography. *Computers & Fluids, 38*(2), 221–234. https://doi.org/https://doi.org/10.1016/j.compfluid.2008.02.008

Liang, Q., Hou, J., & Xia, X. (2015). 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. https://doi.org/https://doi.org/10.1002/fld.4005

Lohr, C. (2009). *GPU-Based Parallel Stackless BVH Traversal for Animated Distributed Ray Tracing*.

Matsuyama, M., & Tanaka, H. (2001). An experimental study of the highest run-up height in the 1993 Hokkaido Nansei-Oki earthquake tsunami. 7.

Meister, O., Rahnema, K., & Bader, M. (2016). Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. *ACM Trans. Math. Softw.*, 43(3). https://doi.org/10.1145/2947668

Merrill, D. (2022). *CUB software package*. https://nvlabs.github.io/cub/

MIKE 21 GPU. (2019). *MIKE Powered by DHI: GPU - Guidelines*.

Nam, M., Kim, J., & Nam, B. (2016). Parallel Tree Traversal for Nearest Neighbor Query on the GPU. *2016 45th International Conference on Parallel Processing (ICPP)*, 113–122. https://doi.org/10.1109/ICPP.2016.20

Sætra, M., Brodkorb, A., & Lie, K.-A. (2014). Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations. *Journal of Scientific Computing, 63*. https://doi.org/10.1007/s10915-014-9883-4

Sagan, H. (1994). *Space-Filling Curves* (1st ed.). Springer New York, NY.

Sedgewick, R., & Wayne, K. (2011). *Algorithms, 4th Edition*. Addison-Wesley.

Shaw, J., Kesserwani, G., Neal, J., Bates, P., & Sharifian, M. K. (2021). LISFLOOD-FP 8.0: the new discontinuous Galerkin shallow-water solver for multi-core CPUs and GPUs. *Geosci. Model Dev., 14*(6), 3577–3602. https://doi.org/10.5194/gmd-14-3577-2021
Shirvani, M., Kesserwani, G., & Richmond, P. (2021). Agent-based simulator of dynamic flood-people interactions. *Journal of Flood Risk Management, 14*(2), e12695. https://doi.org/10.1111/jfr3.12695

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

Song, L., Zhou, J., Li, Q., Yang, X., & Zhang, Y. (2011). 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. https://doi.org/10.1002/fld.2397

Toro, E. F. (2001). *Shock-capturing methods for free-surface shallow flows*. Wiley-Blackwell.

Toro, E. F., & Garcia-Navarro, P. (2007). Godunov-type methods for free-surface shallow flows: A review. *Journal of Hydraulic Research, 45*(6), 736–751. https://doi.org/10.1080/00221686.2007.9521812

TUFLOW HPC. (2018). *TUFLOW Classic/HPC User Manual*.

Wahib, M., Maruyama, N., & Aoki, T. (2016). Daino: A High-Level Framework for Parallel and Efficient AMR on GPUs. *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 621–632. https://doi.org/10.1109/SC.2016.52

Wang, Y., Liang, Q., Kesserwani, G., & Hall, J. W. (2011). A 2D shallow flow model for practical dam-break simulations. *Journal of Hydraulic Research, 49*(3), 307–316. https://doi.org/10.1080/00221686.2011.566248

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

Xing, Y., Liang, Q., Wang, G., Ming, X., & Xia, X. (2019). City-scale hydrodynamic modelling of urban flash floods: the issues of scale and resolution. *Natural Hazards, 96*. https://doi.org/10.1007/s11069-018-3553-z

Zhou, F., Chen, G., Huang, Y., Yang, J. Z., & Feng, H. (2013). An adaptive moving finite volume scheme for modeling flood inundation over dry and complex topography. *Water Resources Research, 49*(4), 1914–1928. https://doi.org/10.1002/wrcr.20179

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