Contents

Figures

Tables

Abbreviations

SWE	Shallow water equations
HLLC	Harten-Lax-van Leer-Contact
CFL	Courant-Friedrichs-Lewy
GPU	Graphics Processing Unit
CPU	Central Processing Unit
OpenMP	Open Multi-Processing
API	Application Programming Interface
MPI	Message Passing Interface
SM	Streaming Multiprocessor
SIMT	Single Instruction, Multiple Threads
GPGPU	General-Purpose computing on GPUs
CUDA	Compute Unified Device Architecture
AMD	Advanced Micro Devices
HIP	Heterogeneous-computing Interface for Portability
OpenCL	Open Computing Language
OpenACC	Open Accelerators
HPC	High Performance Computing
LUMI	Large Unified Modern Infrastructure
SYCL	SYstem-wide Compute Language
FPGA	Field-Programmable Gate Array
NPU	Neural Processing Unit
DPC++	Data Parallel C++
OpenGL	Open Graphics Library

1 Introduction

hook problématique exemple parlant de pq on doit accelere

2

Governing equations

2.1 Shallow water equations

The Watlab hydraulic simulator solves the two-dimensional shallow water equations (SWE), which describe depth-averaged conservation of mass and momentum in a horizontal plane. These equations neglect vertical velocities, making them suitable for flood modeling where horizontal flow dominates. They are expressed in conservative vector form as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}(\boldsymbol{U})}{\partial x} + \frac{\partial \boldsymbol{G}(\boldsymbol{U})}{\partial y} = \boldsymbol{S}(\boldsymbol{U}) \tag{1}$$

where

$$\boldsymbol{U} = \begin{bmatrix} h \\ q_x \\ q_y \end{bmatrix} \quad \boldsymbol{F}(\boldsymbol{U}) = \begin{bmatrix} q_x \\ \frac{q_x^2}{h} + \frac{1}{2}gh^2 \\ \frac{q_x q_y}{h} \end{bmatrix} \quad \boldsymbol{G}(\boldsymbol{U}) = \begin{bmatrix} q_y \\ \frac{q_x q_y}{h} \\ \frac{q_y^2}{h} + \frac{1}{2}gh^2 \end{bmatrix} \quad \boldsymbol{S}(\boldsymbol{U}) = \begin{bmatrix} 0 \\ gh(S_{0x} - S_{fx}) \\ gh(S_{0y} - S_{fy}) \end{bmatrix}$$

Here, $h\,[\mathrm{L}]$ is the water depth, and $q_x = uh\,[\mathrm{L}^2\,\mathrm{s}^{-1}]$, $q_y = vh\,[\mathrm{L}^2\,\mathrm{s}^{-1}]$ are the unit discharges in the x and y directions, respectively, with velocity components u and v. Furthermore, the bed slope effects are modeled as

$$S_{0x} = -\frac{\partial z_b}{\partial x} \qquad S_{0y} = -\frac{\partial z_b}{\partial y}$$

where z_b [L] is the bed elevation. Friction losses are given by

$$S_{fx} = \frac{n^2 u\sqrt{u^2+v^2}}{h^{\frac{4}{3}}} \qquad S_{fy} = \frac{n^2 v\sqrt{u^2+v^2}}{h^{\frac{4}{3}}}$$

where n is the Manning's roughness coefficient.

2.2 Numerical scheme

To solve Equation 1, we divide the computational domain into smaller two-dimensional *cells*, denoted by $\mathcal{C}_i$ (Figure 2.1), and assume the hydraulic variables in $\boldsymbol{U}$ remain constant within each cell. The segments forming the boundaries are called *interfaces*, and the cells are typically triangular. Finally, the cell vertices are called *nodes*, and the collection of cells, interfaces, and nodes forms a *mesh* used to solve the shallow water equations.

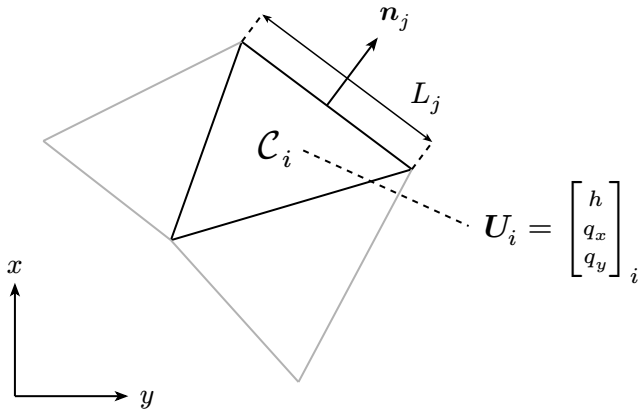


Figure 2.1 – Cell geometry

The finite volume formulation follows from conservation laws imposed on the control volumes $\Omega_i \coloneqq \mathcal{C}_i \times \Delta t$ (Figure 2.2). Mathematically, this is expressed as:

$$\int_{\Omega_i} \left(\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}(\boldsymbol{U})}{\partial x} + \frac{\partial \boldsymbol{G}(\boldsymbol{U})}{\partial y} \right) d\Omega = \int_{\Omega_i} \boldsymbol{S}(\boldsymbol{U}) d\Omega \tag{2}$$

The left-hand side of Equation 2 represents the divergence of the vector

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{F} \\ \boldsymbol{G} \\ \boldsymbol{U} \end{bmatrix}$$

in the (x, y, t) space. Applying the Green-Gauss theorem, we transform the volume integral into a surface integral:

$$\int_{\Omega_i} (\nabla \cdot \boldsymbol{H})\, d\Omega = \oint_{\partial\Omega_i} (\boldsymbol{H} \cdot \boldsymbol{n})\, dS$$

where $\boldsymbol{n}$ is the outward normal vector. To compute the integral, we sum the contributions from each face of the control volume.

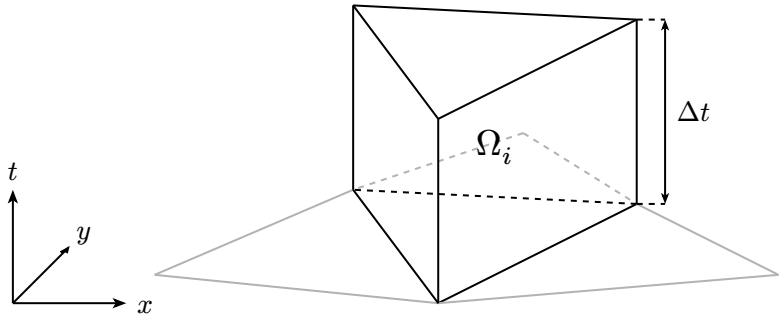


Figure 2.2 – Control volume

Since Watlab handles **unstructured** meshes, determining flux contributions at interfaces is nontrivial. We use a $*$ superscript for time-averaged quantities over Δt and denote the area of $\mathcal{C}_i$ as $|\mathcal{C}_i|$. Instead of directly computing $\boldsymbol{F}^*$ and $\boldsymbol{G}^*$ at interfaces, we solve a locally equivalent problem using the basis transformation:

$$\overline{\boldsymbol{U}} \coloneqq \boldsymbol{T}\boldsymbol{U} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & n_x & n_y \\ 0 & -n_y & n_x \end{bmatrix} \begin{bmatrix} h \\ uh \\ vh \end{bmatrix} = \begin{bmatrix} h \\ u_n h \\ v_t h \end{bmatrix}$$

where $\boldsymbol{n} = \left(n_x, n_y\right)$ is the normal vector of a cell interface. Solving Equation 1 in the (x_n, y_t, t) basis yields:

$$\frac{\partial \overline{\boldsymbol{U}}}{\partial t} + \frac{\partial \boldsymbol{F}\left(\overline{\boldsymbol{U}}\right)}{\partial x_n} = \boldsymbol{T}\boldsymbol{S} \qquad \text{with} \qquad \boldsymbol{F}\left(\overline{\boldsymbol{U}}\right) = \begin{bmatrix} u_n h \\ u_n^2 h + \frac{1}{2} g h^2 \\ u_n v_t h \end{bmatrix}$$

Multiplying by $\boldsymbol{T}^{-1}$ recovers the average flux across the interface in global coordinates, i.e. an appropriate weighted summation of vertical and horizontal flux contributions, yielding:

$$\oint_{\partial\Omega_i} (\boldsymbol{H}\cdot\boldsymbol{n})\,dS = [\boldsymbol{F}\ \ \boldsymbol{G}\ \ \boldsymbol{U}]_i^{n+1}\begin{bmatrix}0\\0\\1\end{bmatrix}|\mathcal{C}_i| + [\boldsymbol{F}\ \ \boldsymbol{G}\ \ \boldsymbol{U}]_i^{n}\begin{bmatrix}0\\0\\-1\end{bmatrix}|\mathcal{C}_i|$$
$$+\Delta t\sum_j \boldsymbol{T}_j^{-1}\boldsymbol{F}_j^*\left(\overline{\boldsymbol{U}}_i^n\right)L_j$$

The right-hand side of Equation 2 integrates as:

$$\int_{\Omega_i}\boldsymbol{S}(\boldsymbol{U})\,d\Omega = \boldsymbol{S}_i^*\Delta t|\mathcal{C}_i|$$

Combining and rearranging the terms, the discrete form of Equation 1 corresponding to the finite volume explicit scheme is given by

$$\boldsymbol{U}_i^{n+1} = \boldsymbol{U}_i^n - \frac{\Delta t}{|\mathcal{C}_i|}\sum_j \boldsymbol{T}_j^{-1}\boldsymbol{F}_j^* L_j + \boldsymbol{S}_i^*\Delta t \tag{3}$$

The key challenge in this scheme is computing the numerical fluxes $\boldsymbol{F}_j^*$ at cell interfaces while ensuring mass and momentum conservation [3]. Since conserved variables are piecewise constant, discontinuities at interfaces create Riemann problems. In practice, Watlab reconstructs numerical fluxes using the Harten-Lax-van Leer-Contact (HLLC) solver. A full discussion of this solver is beyond this introduction, but one may retain that it is computationally heavier than finite volume updates, as confirmed by Watlab profiling in the next sections. Flux computation may also vary at boundary interfaces, depending on boundary conditions.

Additionally, Watlab implements a morphodynamic model based on the Exner equation. Since this module does not alter the program's core architecture, only increasing computational cost, it is not discussed further. More details are available in the official documentation [4].

Finally, as this scheme is explicit, the time step must be carefully chosen to ensure numerical stability. The Courant-Friedrichs-Lewy (CFL) condition dictates:

$$\Delta t^n = \min_i\left(\frac{\Delta x}{|\boldsymbol{u}^n| + c^n}\right)_i \tag{4}$$

where

$$\Delta x_i = 2\frac{|\mathcal{C}_i|}{|\partial\mathcal{C}_i|} \qquad |\boldsymbol{u}_i^n| = \sqrt{(u_i^n)^2 + (v_i^n)^2} \qquad c_i^n = \sqrt{gh_i^n}$$

2.3 Data parallelism

The finite volume scheme derived in the previous section (Equation 3) exhibits a high degree of *data parallelism* or *fine-grained parallelism*. The **same operation** of updating hydraulic variables is performed on **multiple data**. Additionally, fluxes must be computed at each interface, and source terms evaluated for each cell, all of which are repeated at every time step. Given Equation 4, these time steps can be very small and beyond our control.
Therefore, the main limitations of an implementation of this scheme are the mesh size, defined by the number of cells and interfaces, and the desired simulation time. This inherent data parallelism guides our efforts to accelerate Watlab. One approach is to distribute computations across multiple processors, known as *parallelization*. This also motivates the use of coprocessors such as Graphics Processing Units (GPUs), which execute instructions simultaneously across many processors. Finally, since computations access different data while some cells share edges, optimizing data access patterns is essential.

2.4 Program structure

Based on the discretization scheme of the governing equations, Watlab's program structure is straightforward to understand.

To prevent confusion, we first provide a high-level overview of the program architecture. Watlab is primarily divided into two distinct parts, written in different languages for different purposes. The first part handles preprocessing and postprocessing tasks such as mesh parsing, boundary condition setup, and result visualization, all implemented in Python. Most hydraulic studies conducted by researchers or external users are performed through this Python API as a black box. However, the core computational code is written in C++, a compiled language known for its efficiency. The finite volume scheme is implemented in C++ using an object-oriented approach, producing a single executable after compilation. This binary reads input files containing geometry and simulation parameters from the Python interface and outputs hydraulic variables at user-specified time steps.

Watlab is publicly available [4] as a Python package and can be easily installed via `pip` [5]. When downloaded, only the Python files and compiled binaries are installed on the user's machine.

Our work primarily focuses on the computational code written in C++, though we may also modify the Python module if, for example, mesh reorganization is needed. A summary diagram of the high-level architecture is shown in Figure 2.3. Also note that the C++ code may sometimes be referred as *Hydroflow*.

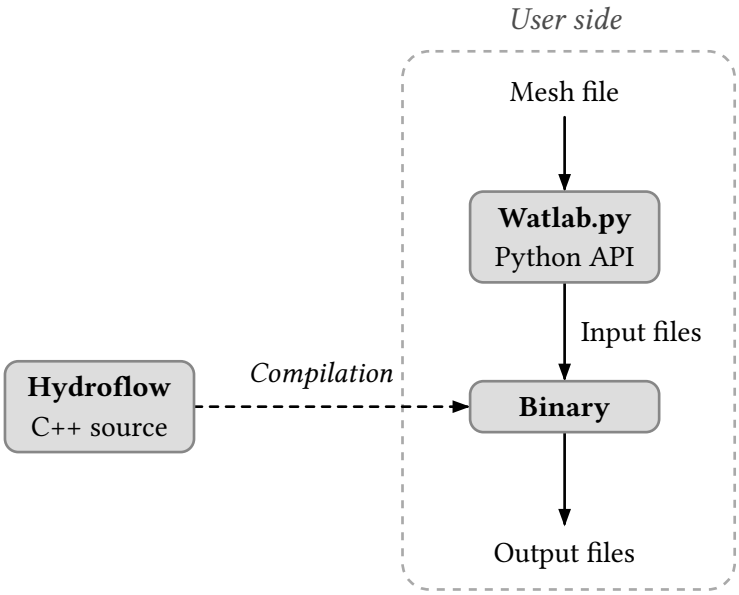


Figure 2.3 – High-level architecture

We now focus on the structure of the computational code, including the implementation of the finite volume scheme. The program begins by reading the provided geometry and instantiating objects representing the domain, cells, interfaces, and nodes. It then enters the

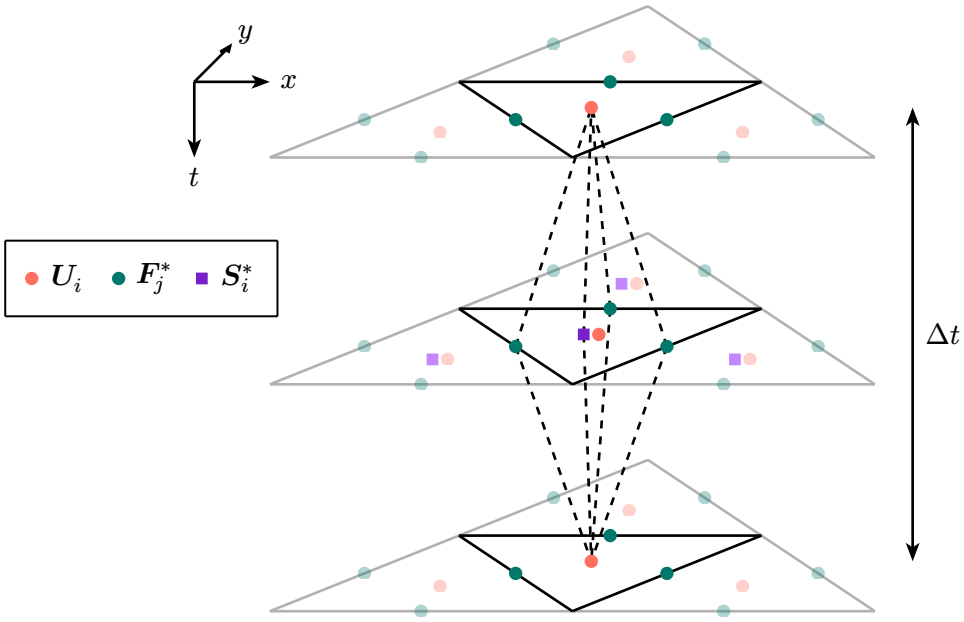


Figure 2.4 – Dependencies

main loop.
At each iteration, the program first checks whether hydraulic variables need to be output. Watlab provides several data outputs, including hydraulic variables at user-specified gauges, the maximum water height in the domain (*enveloppes*), snapshots of hydraulic variables in each cell (*pictures*), and total discharge across interface sections. If output is required, it is processed first; otherwise, the domain is updated.

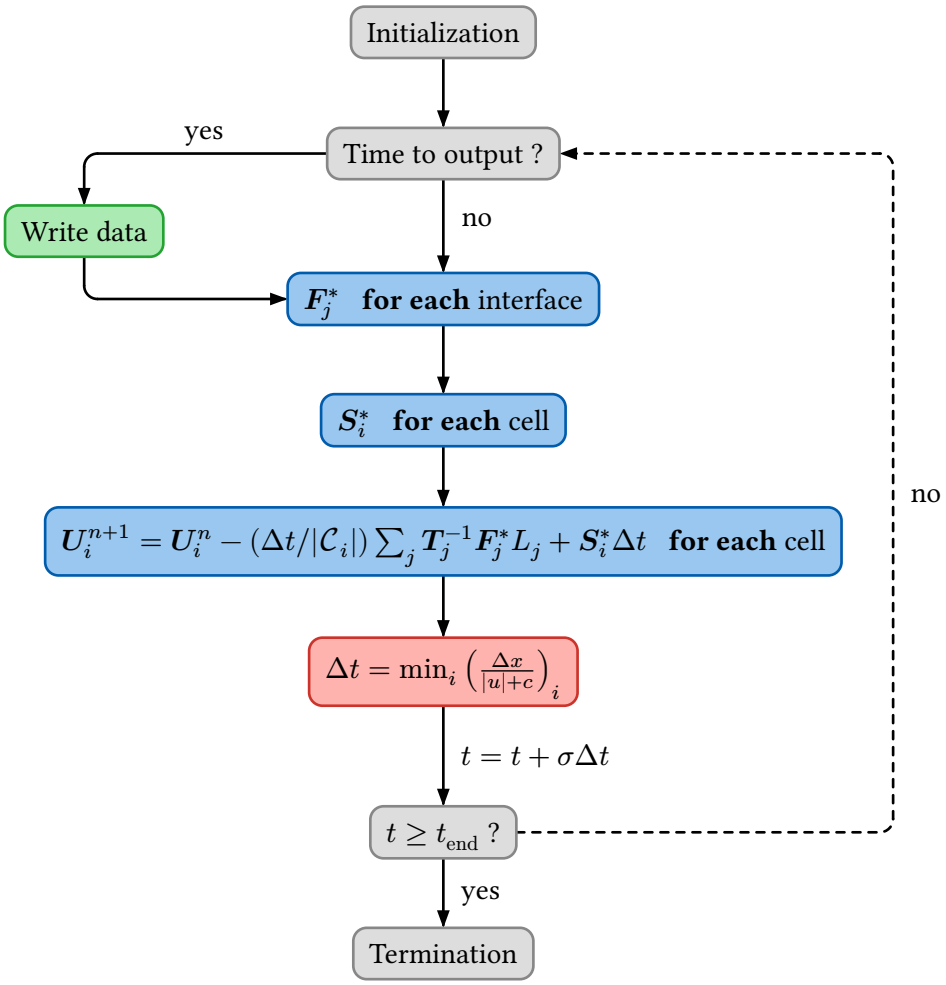


Figure 2.5 – Program structure

The update process involves computing numerical fluxes at each interface, calculating source terms for each cell, and updating hydraulic quantities based on the finite-volume scheme (Equation 3). The computational dependencies between variables over time are illustrated in Figure 2.4. The next time step is then determined using the CFL condition (Equation 4). This corresponds to the theoretically maximum stable time step. In practice, it is rescaled using the user-provided CFL number $0 < \sigma \leq 1$ to enhance stability. If the updated simulation time

exceeds the end time, the program terminates and cleans up the data. Otherwise, the time step is incremented, and the loop continues. An overview of the program structure is provided in Figure 2.5.

3
State of the art

3.1 Parallelization methods

Modern Central Processing Units (CPUs) contain multiple processing cores, which are independent units that execute streams of instructions. Supercomputers, on the other hand, are machines with many CPUs that can work together on the same problem. Both types of parallel architectures rely on parallelism, that is, the ability to break a computational task into smaller parts that can be executed independently and simultaneously. Because it occurs at multiple levels, parallelism is difficult to define precisely.

To review the current parallelization methods, we will consider the more general abstraction of a parallel computer, which can be thought of as an architecture with multiple processors that allow multiple instruction sequences to be executed simultaneously. As the processors work together to solve a common problem, they need to access a common pool of data.
Parallel machines differ in how they tackle the problem of reconciling multiple memory accesses from multiple processors. According to the *distributed memory* paradigm, each processor has its own address space, and no conflicting shared accesses can arise. In the *shared memory* paradigm, all processors access a common address space. The existing programming

models for taking advantage of parallel processors follow one of these paradigms, each of which involves a different set of constraints and capabilities.

Note that this section and the following are primarily based on the work of [6].

3.1.1 Threads

The building block of a parallel programming model implementing the shared memory paradigm is the thread. A thread is an execution context, i.e. a set of register values that enables the CPU to execute a sequence of instructions. Each thread has its own stack for storing local variables and function calls, but it shares the heap of the parent process (i.e., an instance of a program) with other threads, and therefore shares global variables as well.
It is the operating system, through its scheduler, that decides which thread is executed, when, and on which processor. These scheduling decisions are often made at runtime and may vary from one execution to another [7]. Therefore, if shared data is accessed concurrently by different threads, the final result may depend on which thread executes first. This is known as a race condition. To solve this problem, inter-thread synchronization is needed, typically through mechanisms such as *mutexes*. These synchronization primitives allow only one thread to access a shared resource at a time while others wait.
Finally, threads are dynamic in the sense that they can be created and terminated during program execution. This is commonly referred as the *fork-join* model[1].

C++ provides native support for threads and mutexes through the `<thread>` and `<mutex>` libraries. The pseudocode below demonstrates an example of thread spawning and joining. The instructions to perform take the form of function.

```
void writeOutput(arg){ ... }              // job to do

std::thread myThread(writeOutput, args);  // launch
myThread.join();                          // wait for thread to finish
```

3.1.2 OpenMP

OpenMP [8] is a directive-based API and the most widely used shared memory programming model in scientific codes. It is built on threads, inheriting related paradigms, and hides thread spawning and joining from the developer through compiler directives that automatically generate parallel code. For example, a `for` loop can be parallelized as follows:

[1]As threads are launched, the original execution sequence of the program splits into multiple parallel execution threads that run simultaneously, which can be thought of as a reversed fork. Later, the threads finish their tasks and join back with the main execution thread.

```
#pragma omp parallel for                    // compiler directive
for (int i = 0; i < nCells; i++) {
  ...
}
```

OpenMP is designed to be easy to deploy and supports incremental parallelization. Since it primarily aims to distribute loop iterations across parallel processors, it is suited for data parallelism, where the same independent operations must be performed on different pieces of data.

3.1.3 MPI

MPI (Message Passing Interface) [9] is the standard solution for implementing parallel programming adopting the distributed memory paradigm. This library interface enables both data and *task parallelism*, i.e., executing whole subprograms in parallel. In this paradigm, MPI processes cannot access each other's data directly, as memory is distributed either virtually or physically. Therefore, processes must perform communication operations, namely one-sided, point-to-point, or collective, to exchange data.

The distributed memory model also requires an initial partitioning of data, such as the mesh in SWE solvers. In such cases, authors often rely on external libraries like METIS [10] to partition the domain in a way that minimizes inter-process communication. Although MPI can be used on a single multiprocessor system, it truly demonstrates its power when deployed across a cluster of CPU nodes.

3.2 Graphics Processing Units

Decoding and processing instructions is an expensive operation in terms of time, energy, and the number of required transistors. Graphics Processing Units were developed from the idea of reducing the functionality of parallel cores to save on each of these costs, thereby enabling a much higher number of cores. These massively parallel architectures were originally designed to offload from the CPU the task of rendering two-dimensional images from a three-dimensional virtual world [2], by processing each pixel in parallel.

The functional equivalent of CPU cores in GPUs are called *Streaming Multiprocessors* (SMs). Each SM contains execution cores, both single- and double-precision floating-point units, as well as special function units, schedulers, caches, and registers. A sequence of instructions is abstracted as *threads*, which are grouped into *warps* that execute concurrently and truly simultaneously in lockstep on the SMs. Consequently, execution on GPUs follows the Single Instruction, Multiple Threads (SIMT) paradigm.

The growing number of numerical simulations exhibiting massive data parallelism in both industry and academia has made the use of GPUs for non-graphics applications increasingly popular [11]. This trend is known as General-Purpose computing on GPUs (GPGPU). However, since GPUs are fundamentally different hardware with their own architectural paradigms, they also require different programming languages than those commonly used for CPU programming.

3.3 The zoo of GPGPU languages

At the time of the previous study [12], the options for porting existing C++ code to a GPU-capable implementation were quite limited and often hardware-dependent. The most popular among them is the Compute Unified Device Architecture (CUDA) [13], a proprietary framework developed by NVIDIA that supports only its GPUs. As a strategic response, AMD developed HIP [14], a C++ runtime API that allows developers to write code compilable for both AMD and NVIDIA GPUs. Its syntax is intentionally close to CUDA[2] to ease code conversion[3] from CUDA to HIP. Originally, OpenCL [15] was the primary method for GPGPU on AMD and Intel hardware. It is a cross-platform parallel programming standard compatible with various accelerators, including NVIDIA, AMD, and Intel GPUs. However, it is less popular in the scientific community due to limited performance portability [16] and lower productivity, requiring more code to achieve the same results [17].

Additionally, the OpenACC [18] programming standard uses directives for NVIDIA/AMD GPU offloading, making it the GPGPU equivalent of OpenMP. This approach is well-suited for inexperienced developers [19]. More recently, OpenMP introduced GPU offloading via compiler macros, but its performance remains slower than OpenACC [20], [21].

During the past decades, NVIDIA was the uncontested leader in the consumer GPU and HPC markets. Nowadays, AMD and Intel have become serious competitors [22]. For example, many new HPC infrastructures use non-NVIDIA hardware, such as the European LUMI supercomputer with AMD Instinct GPUs [23]. Meanwhile, Intel has launched its Arc GPUs, increasingly featured in mainstream laptops.

Facing increasing heterogeneity in computing platforms, the HPC community has developed high-level abstraction libraries such as Kokkos [24], RAJA [25], SYCL [26], and Alpaka [27]. These are built on traditional GPGPU languages like CUDA, HIP, and OpenCL, allowing users to avoid hardware-specific details by handling backend code within the libraries. This enables developers to write self-contained C++ programs, unlike CUDA and OpenCL, which separate host and device code. Indeed, since they extend the regular C++ language with new

[2] Mainly, replacing `cuda` prefixes with `hip`.

[3] AMD also provides an automation translation tool called `hipify`.

keywords, the kernel code must be embedded in `.cu` and `.cl` files, respectively. The key advantage brought by abstraction libraries is a single source file supporting multiple backend targets, including serial and OpenMP-based CPU versions, while ensuring minimal overhead and performance portability. Additionally, these implementations are future-proof, as maintainers can adapt the libraries to emerging hardware like Field-Programmable Gate Arrays (FPGAs) or Neural Processing Units (NPUs) without requiring major user code rewrites. Notably, Kokkos documentation indicates ongoing backend development.

The latest SYCL standard, SYCL 2020 [28], was developed by Khronos, the group behind OpenCL. The main difference from other libraries is that SYCL is merely a specification defining an API with expected behaviors, without providing an implementation. Various implementations support SYCL 2020 specification to different extents, using different compilers and backends. Notable examples include DPC++ [29] (Intel), ComputeCpp [30] (Codeplay), triSYCL [31], AdaptiveCpp (formerly hipSYCL) [32], and neoSYCL [33], each targeting different hardware platforms. Note that SYCL can also serve as a backend for both Kokkos [34] and RAJA [35].

A visual summary of the approaches discussed so far is presented in Figure 3.6.

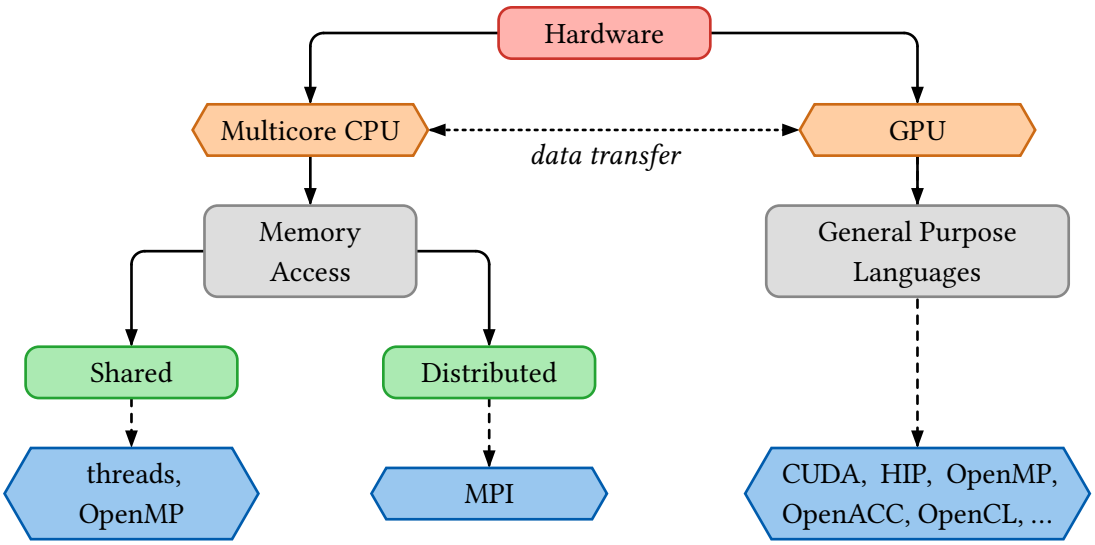


Figure 3.6 – Visual diagram of discussed approaches

3.4 Trends in SWE solvers

Various implementations of SWE solvers exist. To accelerate their codes, hydraulic researchers initially focused on CPU parallelization. Some implementations decompose the computational domain into smaller regions and use MPI to solve subproblems simultaneously [36], [37], [38],

[39], [40], [41], [42], [43]. The main argument for using the distributed memory paradigm is the ability to deal with gigantic domains consisting of many cells and interfaces that could not be stored on a single machine due to the limited amount of memory available. Others still adopted a shared memory model with OpenMP [44], [45] which allows to get rid of any inter-process communication. However, since the achievable speed-up with these methods is limited by the number of processor cores, interest grew in leveraging GPUs for solving SWE equations even before the advent of GPGPU languages like CUDA. For example, Lamb et al. [46] used Microsoft DirectX 9 [47], while [48] employed ClearSpeed [49] accelerator cards, a now-defunct manufacturer. Other implementations [50], [51], [52] rely on Cg [53] and OpenGL [54], the graphical predecessor of OpenCL. Although these GPU implementations achieved significant speedups, the graphical nature of their implementation language made them difficult to understand and maintain. Therefore, the release of CUDA in 2007 motivated many authors [55], [56], [57], [58], [59], [60], [61], [62], [63], [64] to deploy their SWE solvers on GPUs because of the reduced development effort. Meanwhile, others [64], [65] chose OpenCL to improve the portability of their GPU implementation. Finally, OpenACC implementations [21], [64], [66], [67] were also explored, motivated by the minimal recoding effort.

With the growing use of GPU-accelerated hydrocodes, the next step was to leverage multiple GPUs [68] by decomposing the domain for further speedup. This multi-GPU approach, implemented with CUDA+MPI [69] or OpenACC+MPI [70], achieved low run times but showed limited scalability with a large number of GPUs due to bottlenecks on communication and I/O tasks. Besides, given the growing hardware heterogeneity in HPC infrastructures, Caviedes-Voullième et al. [71] developed a high-performance, portable shallow-water solver with multi-CPU and multi-GPU support using Kokkos. Similarly, Büttner et al. [72] compared GPU and FPGA performance for a shallow-water solver using the finite element method with SYCL.

All the presented methods benefit from different acceleration factors compared to the serial version of the code, depending on the numerical scheme, case study, implementation details, profiling protocol, and hardware used. As a result, their performance may not directly reflect what we can expect for Watlab. To provide better insights, we present in Table 3.1 an overview of the speedups achieved with different parallel technologies, along with relevant hardware and test case contexts. Given the broad range of cited articles, this comparative table includes a non-exhaustive selection of studies chosen for relevance. Priority is given to recent implementations that either:

- Use a numerical scheme similar to Watlab,
- Serve as representative examples of a given parallel technology, or
- Feature novel optimizations or programming approaches.

Therefore, all hydrocodes use an explicit finite volume method, though the flux computation and reconstruction schemes may differ (as noted in the fourth column). We focused on test cases involving wet and dry cells, which present a worst-case scenario for GPU acceleration due to warp divergence (see Section 6.2.1). These case studies include analytical circular dam breaks, realistic dam breaks, and flood simulations. We believe these are the primary applications of the Watlab program and are likely to be run at large scales, requiring significant execution times, making them the most relevant for focus.

Reference	API	Hardware	Scheme	Mesh	Cells	Max. speedup
ParBreZo [42]	MPI	Intel Xeon E5472	Roe	Unstruct.	374 414 Dry/Wet	**27×/48 cores**
Petaccia et al. [62]	OpenMP	Intel Core i7 3.4 GHz	Roe	Unstruct.	200 000 Dry/Wet	**2.35×/4 cores**
Castro et al. [55]	CUDA	NVIDIA GTX 480	Roe	Unstruct.	1 001 898 Wet	**152×/480 cores** (single precision) **41×/480 cores** (double precision)
G-Flood [60]	CUDA	NVIDIA GTX 580	HLLC	Struct.	3 141 592 Dry/Wet	**74×/512 cores** (single precision)
Lacasta et al. [61]	CUDA	NVIDIA Tesla C2070	Roe	Unstruct.	400 000 Dry/Wet	**59×/448 cores** (double precision)
Liu et al. [66]	OpenACC	NVIDIA Kepler GK110	MUSCL-Hancock + HLLC	Unstruct.	2 868 736 Dry/Wet	**31×/2496 cores**
TRITON [69]	CUDA + MPI	386 NVIDIA Volta V100	Augmented Roe	Struct.	68 000 000 Dry/Wet	**≤43176× /1966080 cores**[4]
Saleem and Norman [70]	OpenACC + MPI	8 NVIDIA Volta V100	MUSCL + HLLC	Unstruct.	2 062 372 Dry/Wet	**≤1989× /40960 cores**
SERGHEI [71]	Kokkos	NVIDIA RTX 3070	Roe	Struct.	515 262 Dry/Wet	**51×/5888 cores**

Table 3.1 – Overview of reported speedups compared to serial implementations

[4]The parallel speedup over serial time is not measured in this paper; instead, it is computed relative to a CPU multithreaded implementation. The presented speedup factor is an estimate, assuming the multithreaded implementation achieves perfect speedup with respect to the available cores.

4
Case studies

To evaluate the pros and cons of our ideas for accelerating Watlab, we need case studies where the hydraulic solver can be run to measure execution time. We will use two examples: one small-scale and one large-scale.

4.1 Toce

In [73], the authors described flash flood experiments conducted on a 1:100 scale physical model of the Toce River valley, built in concrete. An urban district made of several concrete block buildings was placed in the riverbed to simulate flooding in a populated area and study the complex flow patterns caused by water interactions. Multiple water depth gauges were installed to collect bathymetric data, and an electric pump controlled the inflow discharge, determining the flood intensity. Together, the topography, inflow, and gauge measurements form a dataset well-suited for validating mathematical flood models.

In our case, we use a mesh file based on this experimental setup, shown in Figure 4.7. The inflow enters the domain through the west boundary indicated in the figure and exits through a transmissive boundary at the far end. Note the 20 aligned squares representing scaled models of buildings. Sediment transport is not considered in this case study.

The key properties of the mesh geometry are listed in Table 2a. Notice that we generated another mesh, much larger, on the same domain by significantly reducing the mesh size. This results in a mesh with about 800 000 small triangles, which we call Toce *XL*. This geometry will be used in Section 6.3, where we only need a large number of cells to leverage GPU compu-

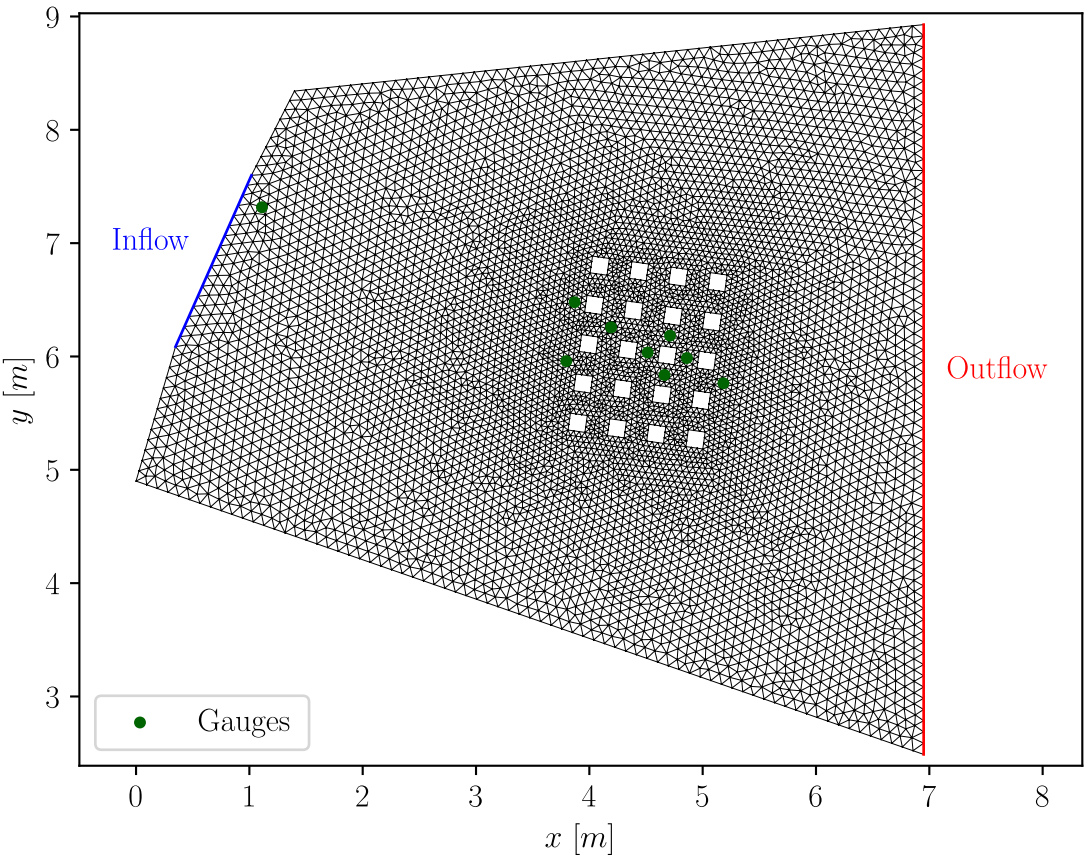


Figure 4.7 – Mesh corresponding to the small-scale model of the Toce River

tational power for toy examples. However, we will not run Watlab on it, as very small cells may cause inconsistencies. First, due to Equation 4, which significantly reduces the acceptable step size. Second, it may lead to arithmetic precision loss because of the difference in order of magnitude between the numerical fluxes, source terms, hydraulic variables, and the geometric quantities related to the cells in Equation 3.

Regarding simulation parameters, the CFL number is set to $\sigma = 0.9$, and the simulation runs from 7 [s] to 60 [s]. This choice is based on the boundary hydrograph provided with the dataset (Figure 4.8). We also periodically generate pictures of the hydraulic variables and gauge measurements, with gauge positions matching those in [73] as described in Table 2b. The gauge coordinates are shown in Figure 4.7. We also recorded envelopes every second, which correspond to the maximum values of height and velocities for each computational cell. Finally, we recorded the discharge measurements over the inflow and outflow sections to exploit all output capabilities of the program.

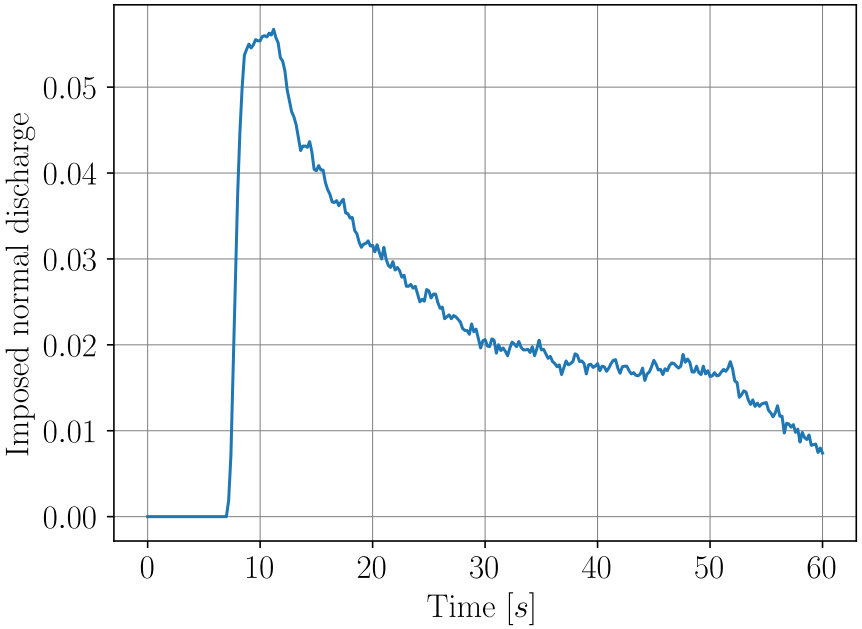


Figure 4.8 – Inflow hydrograph

Case study	**# Nodes**	**# Interfaces**	**# Cells**
Toce	6 716	19 731	12 996
Toce *XL*	406 115	1 214 591	808 457
Square basin	290 132	868 393	578 262

(a) Geometric description of meshes

Parameter	**Toce**	**Square basin**
t_{start} [s]	7	0
t_{end} [s]	60	60
n [—]	0.0162	0.06
σ [—]	0.9	0.95
Δt_{pics} [s]	1	1
Δt_{gauges} [s]	1	-
$\Delta t_{\text{sections}}$ [s]	1	-
$\Delta t_{\text{enveloppes}}$ [s]	1	1

(b) Simulation parameters

Table 4.2 – Case study descriptions

4.2 A square basin

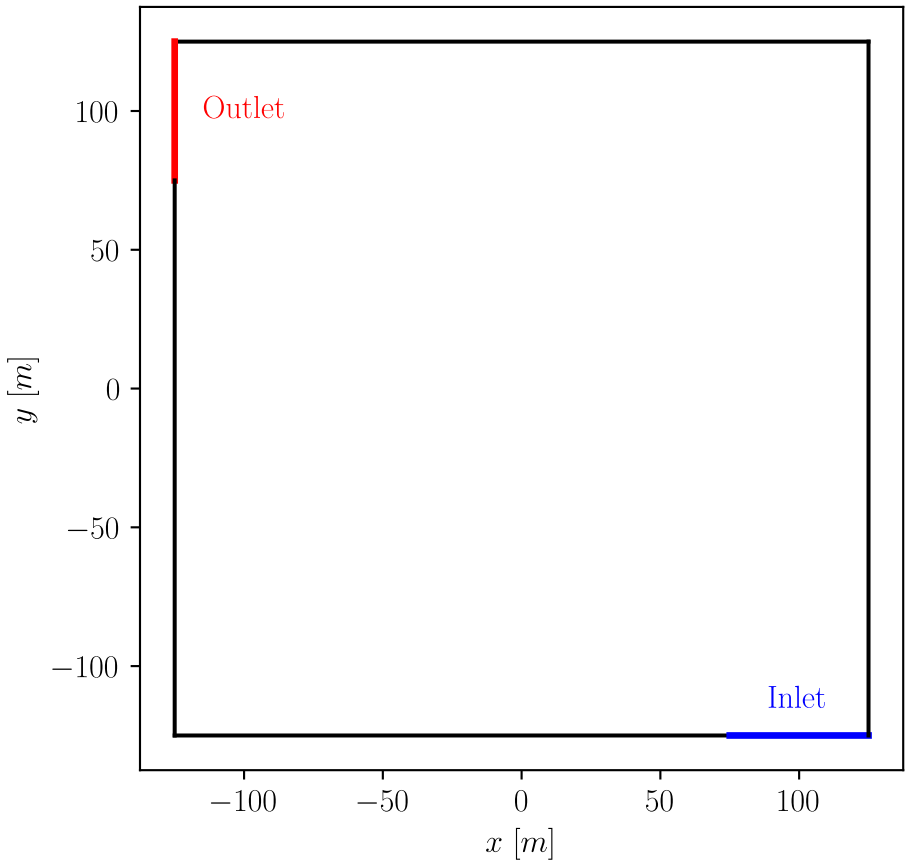


Figure 4.9 – Square basin geometry

The mesh of the previous case study contains about 13 000 cells. In terms of computational load, it is quite reasonable. As we investigate solutions that logically divide the computational domain into smaller parts to be solved concurrently, we need to ensure each computational core receives sufficient workload to assess the real benefits of our approach. Therefore, we rely on a second, larger-scale case study.

This case involves an artificial square basin with one inlet, where an inflow discharge of 400 $[\mathrm{m}^3\ \mathrm{s}^{-1}]$ is imposed, and one outlet through which water can escape. An overview of the geometry is given in Figure 4.9. The initial water level is uniformly set to 1 [m] across the domain, resulting in a fully wet simulation. The large scale results from the basin's considerable dimensions, measuring 125 [m] by 125 [m]. Using cells with a characteristic length of 0.5 [m], the resulting mesh contains a significant number of geometrical elements, as shown in Table 2a.

The simulation runs for 60 seconds, with a picture and envelope generated each second. Some of them are shown in Figure 4.10.

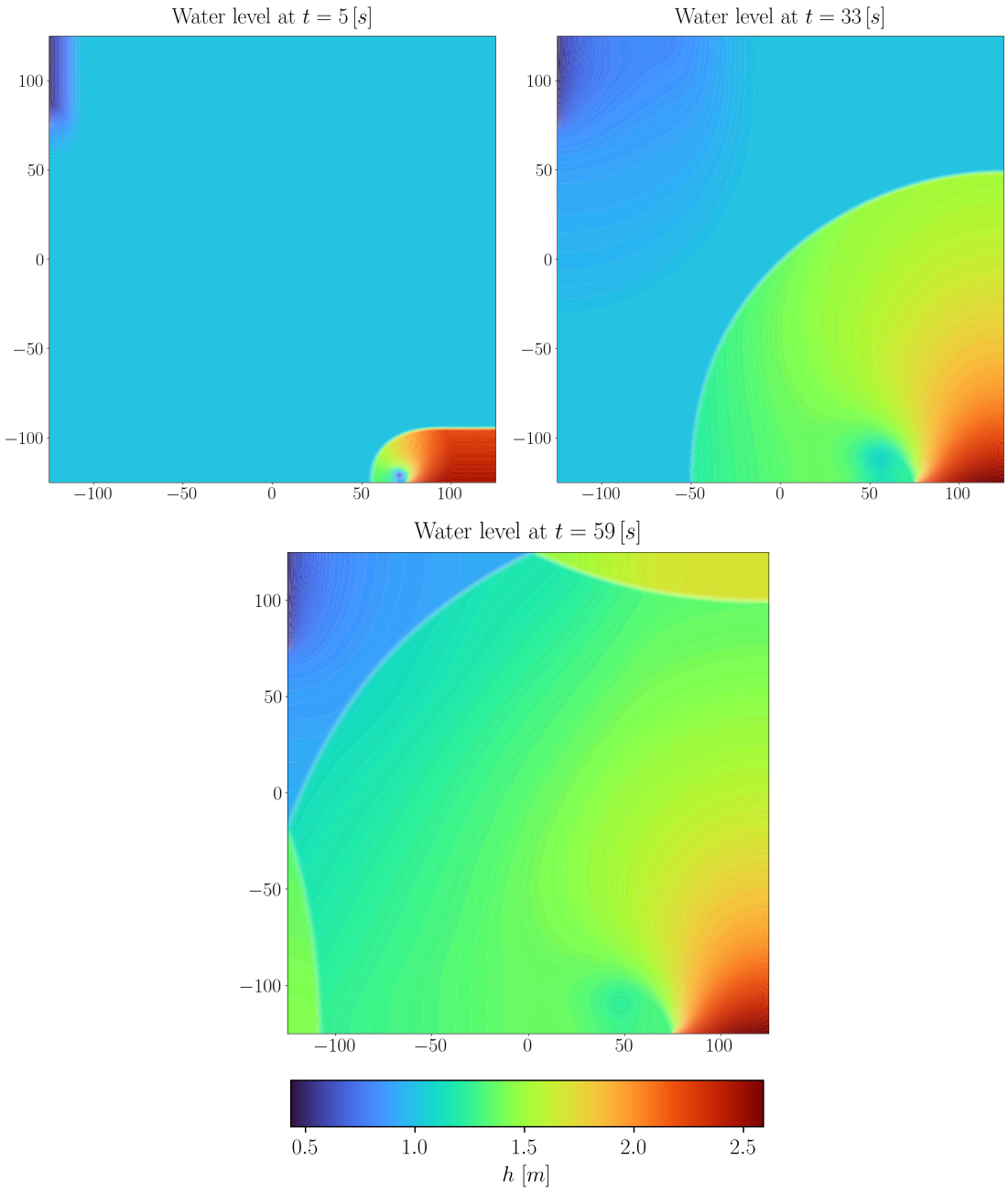


Figure 4.10 – Snapshots from the square basin simulation

5
Implementations on CPU

The following section reviews the main steps explored to make Watlab faster. In a nutshell, we begin by profiling the code to empirically identify bottlenecks and analyze which parts can be accelerated and which cannot. We then present the main leads explored from a theoretical perspective. At the end of the section, execution times from the case studies are presented to empirically assess the benefits of each optimization.

5.1 Toce river: Profiling

We define the serial version of the code as a version of Watlab without any form of parallelism. As a side note, we clarify that this version does not correspond to the original one received at the beginning of the master thesis. On one hand, we corrected some minor bugs in the first weeks, including memory leaks, output irregularities, and logic inconsistencies. On the other hand, the original version still contained remnants of parallelism and synchronization for output tasks, except for pictures, inherited from the previous study [12]. We believe that a

fully sequential, cleaner version that is also closer to the parallel implementations presented later is more relevant for comparison and assessment of our work's efficiency. Therefore, we refer to this version when speaking about the serial version. It fully follows the execution diagram of Figure 2.5.

Thanks to our work in Section 2, where we related Watlab's architecture to the underlying governing equations it simulates, we already know that execution time increases with mesh size and the number of simulated time steps. More precisely, from Figure 2.5 we derive that the program's complexity is $\Theta(T(N+M))$, with T the number of time steps, N the number of cells, and M the number of interfaces. To illustrate and verify this theoretical analysis, we executed and profiled the serial version on the Toce river case study. Snapshots of the flow are shown in Figure 5.11. The first picture exhibits the dry start related to this case study, as the domain is initially empty. Water enters through the inflow boundary and propagates over time. Note that no friction term needs to be computed for dry cells, and no fluxes travel between two dry cells. As a result, this phase is faster to process, and parallelization or GPU use may not reach maximum efficiency because the computational load is low at the beginning of the simulation.

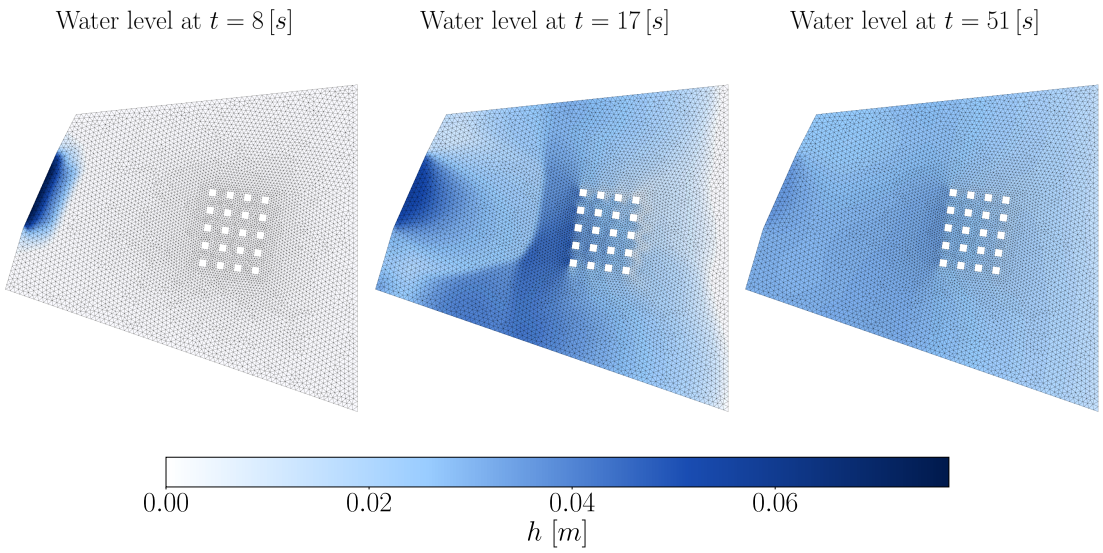


Figure 5.11 – Snapshots from the Toce river simulation

We profiled the execution of the code using `gprof` [74] and `gprof2dot` to visualize the results as a call tree. Figure 5.12 shows this partial call graph, omitting functions that have little or no impact on the total computation time. The percentages shown reflect the share of each call, with the total measured CPU time being 18.66 [s]. It is clear that the execution time of the Watlab computational code is dominated by flux computations, which account for about 73 % of the total. The most time consuming step is the processing of inner interfaces, taking 58

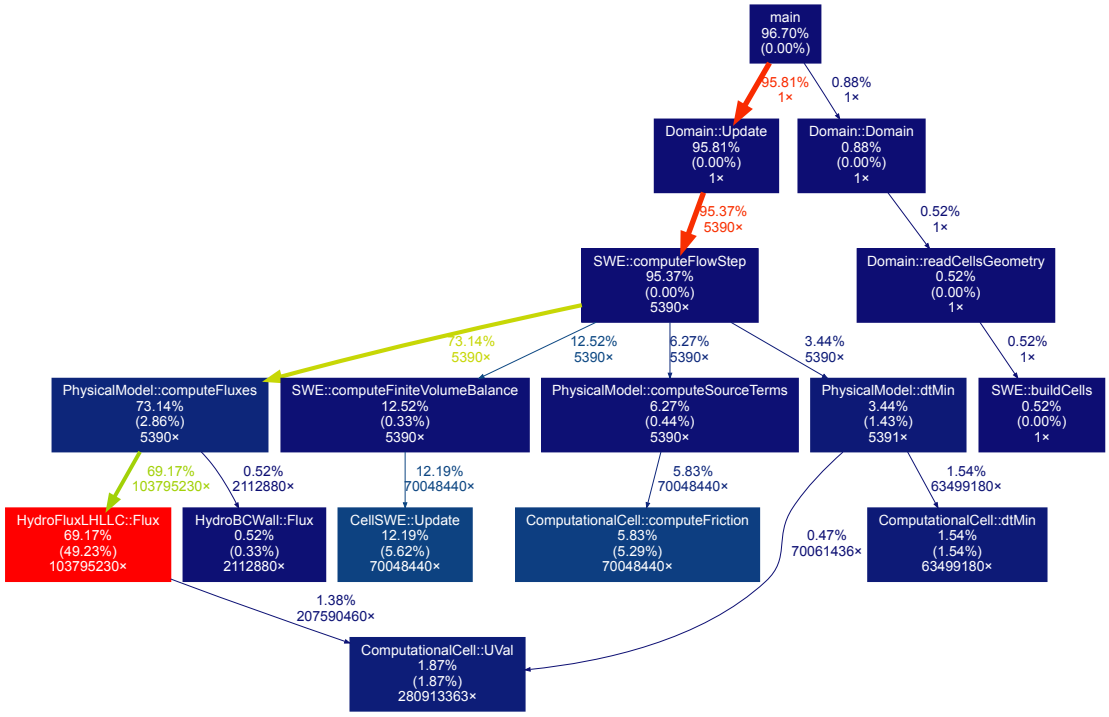


Figure 5.12 – Profiling results of hydroflow execution

%. The second most time consuming step is the finite volume update of the cells, followed by the computation of source terms and finally the time step. Output functions are not shown, as they are negligible in comparison.

From the profiling data, we see that writing the envelope and pictures takes only 0.02 %, while gauge and section measurements each take less than one hundredth of a percent and are therefore not captured in further detail by the profiling tools. It is no surprise that the computeFluxes method accounts for a large share of the time, as it is called 5390 times during the 53 [s] of simulation, compared to only 53 pictures taken. However, if we compute the percentage per call, we get 0.0135 % and 0.0004 % respectively. This confirms that I/O operations remain a task that can be handled efficiently. Since writing to a file is expected to grow linearly with the number of cells, these proportions should hold for larger mesh sizes.

The profiling tree also shows that the long execution times of the functions in the finite volume scheme come from the fact that each cell and interface needs to be processed independently. To speed up computations, it is quite intuitive to leverage parallelism and the multicore architecture of modern processors to process them concurrently and accelerate the whole program.

But what kind of speedup can we actually expect? Amdahl's law [75] is a simple formula that helps predict the maximum expected speedup based on the sequential part of the program f and the number of parallel processors P. It is given by:

$$S = \frac{1}{f + \frac{1-f}{P}} \tag{5}$$

In our case, all steps of the finite volume scheme can be parallelized, and they account for $95.37\,\%$ of the total execution time. Depending on the number of processors, we obtain:

$$S(2) = 1.91 \qquad S(4) = 3.51 \qquad S(8) = 6.04 \qquad S(16) = 9.44$$

We observe that the efficiency[5] fastly decreases as the number of processor increases. Of course, this estimate should not be seen as a very precise estimate, as it neglects factors such as interprocess communication, synchronization, and the overhead from thread initialization and scheduling. On the other hand, as we increase the simulated time, the cost of geometry parsing, which depends only on the mesh size, becomes smaller compared to the finite volume computations, and so does the sequential part. As a result, the achievable speedups should increase.

5.2 Going parallel

There are several steps that can be parallelized in the original program structure Figure 2.5. First, one may imagine that writing processes such as pictures or gauge snapshots can be done in parallel with the main finite-volume computations. This corresponds to the green box in the diagram. Furthermore, the flux and source term computations, as well as the finite-volume update, are mostly independent from one interface or cell to another, so we could distribute the loop iterations over multiple processor cores. These correspond to the blue boxes. Finally, the minimum involved in the time step computation can also be parallelized by dividing the domain into subdomains, computing the local minimum in each subdomain, and then merging the results. This corresponds to the red box.

As recalled in the state of the art, several technologies exist to implement these parallelizations. First, we have to choose the paradigm to follow between the shared-memory and distributed-memory approaches. We chose to focus on a thread-based parallelization for several reasons:

[5] In parallel computing, parallel efficiency measures the ratio of parallel speedup to the number of parallel processors. The scalability of the program is characterized by the evolution of the efficiency as the number of processors varies. Ideally, we would like to have a constant efficiency of 100% or, equivalently, a linear speedup.

- We aim for a parallel version that benefits the largest number of users, while MPI implementations are more suited for use on clusters;
- The flux computation at each interface requires knowledge of the hydraulic variables $\boldsymbol{U}$ from the left and right cells. Partitioning the cell array would then involve communication overhead to share ghost cell states surrounding boundary interfaces;
- Although distributed memory implementations allow the processing of larger meshes that would not fit into a single computer's RAM, we argue that the current Watlab implementation is still mainly limited by execution time, not memory.

The previous study showed that using threads and mutex-based synchronization to parallelize output writing and OpenMP to parallelize loops was the most efficient and easy-to-deploy way to implement the shared-memory model [12]. The original Watlab implementation we received at the beginning of the year still contained a broken version of this approach, so our work was more a rehabilitation than a development from scratch.

Regarding output, there is a small subtlety: writing and computation processes are not fully independent since they rely on the same data, namely the hydraulic variables. If the writing thread outputs concurrently while the finite-volume update is in progress, it leads to race conditions, as written values may vary between runs. To avoid this, careful synchronization is required. For example, computational threads can compute fluxes and source terms while cell data is being output since these intermediate values are not saved, but they must wait for the writing process to finish before altering the hydraulic variables during the finite-volume update.
To minimize idle times and avoid significant delays for computational threads and minimize idle times, writer threads make a local copy of the data to output using a preallocated buffer, which is faster than I/O operations. On the other hand, synchronization relies on mutexes: when launched, writers lock a mutex to indicate they intend to buffer cell data. Once done, they release the mutex, signaling that the backup copy is ready and proceed to write to files. Meanwhile, after computing fluxes and source terms, the finite-volume threads try to lock the same mutexes and will wait if writers have not yet finished buffering.

Finally, we slightly modified the implementation of the minimum computation. The original approach from [12] parallelized the loop over cells using OpenMP, with each thread storing intermediate results in a thread-local variable before merging them to find the global minimum. Since loop iterations are distributed across OpenMP threads, this acts like a domain subdivision, although the cells within each local region may not be contiguous. However, the merging strategy was suboptimal: each thread compared its result with a global variable inside a critical section, making the merging inherently sequential with linear complexity in the number of threads.

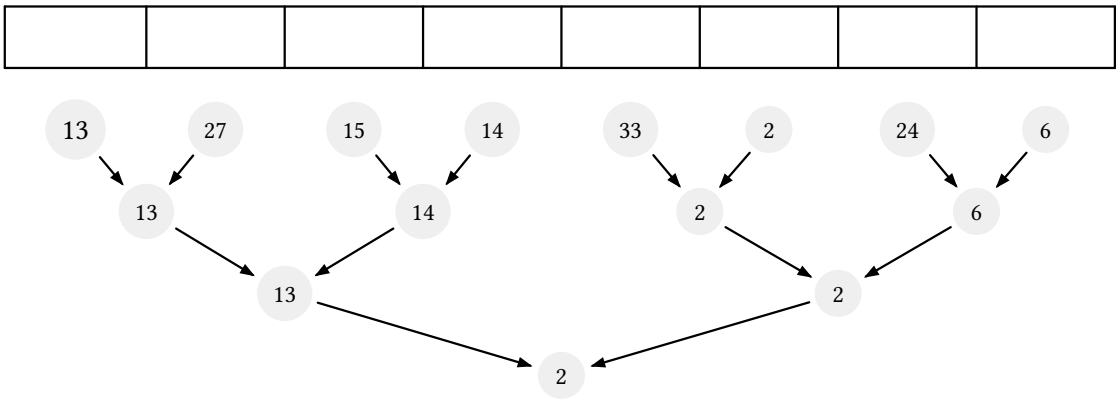


Figure 5.13 – Minimum reduction

This can be improved using a parallel reduction, which resembles a tournament. First, threads form pairs and compare values in parallel, then winners form new pairs and repeat the process until one thread remains. This reduces the global minimum in $\log_2(T)$ steps instead of T, where T is the number of threads. An illustrative diagram of the reduction strategy is shown in Figure 5.13. While this optimization may have little impact with the small number of cores typical in modern CPUs, it becomes significant for GPU implementations with many more cores. For example, 2560 parallel processes would require only 12 reduction rounds. Furthermore, it allows us to leverage cache memory because threads only need to know their local minimum and the local minimums of the other threads with which they compete.

Fortunately, OpenMP provides built-in support for parallel reduction via the reduction keyword, abstracting implementation details and simplifying the code, as shown in the following pseudocode:

```
#pragma omp parallel for reduction(min:tmin)
for (int i = 0; i < nCells; i++) {
  if (Cell[i].tmin < tmin) tmin = Cell[i].tmin;
}
```

The updated structure of the final parallel implementation is shown in Figure 5.14.

A subtlety remains in the program structure shown in Figure 2.5 and Figure 5.14. The sum in Equation 3 is not performed while iterating over cells. Instead, each cell is linked to a buffer, reinitialized at each time step, where fluxes computed at each interface accumulate[6] during the first step of flux computation. This requires each interface to store references to its left and right cells, unless it lies on the boundary. In the update step, the flux sum is retrieved

[6]Actually, multiplied by constants reflecting coordinate transforms and scaled by interface lengths

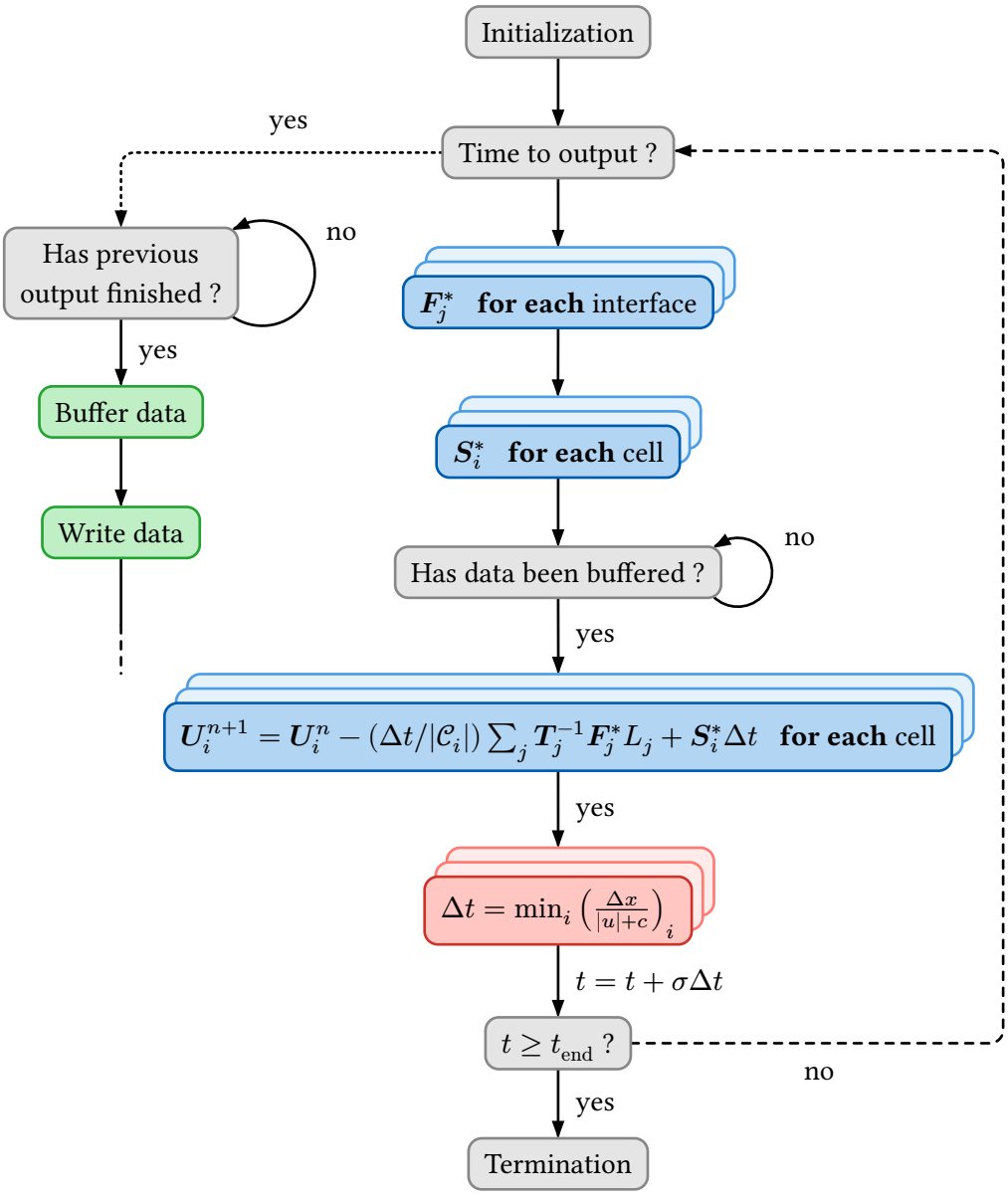
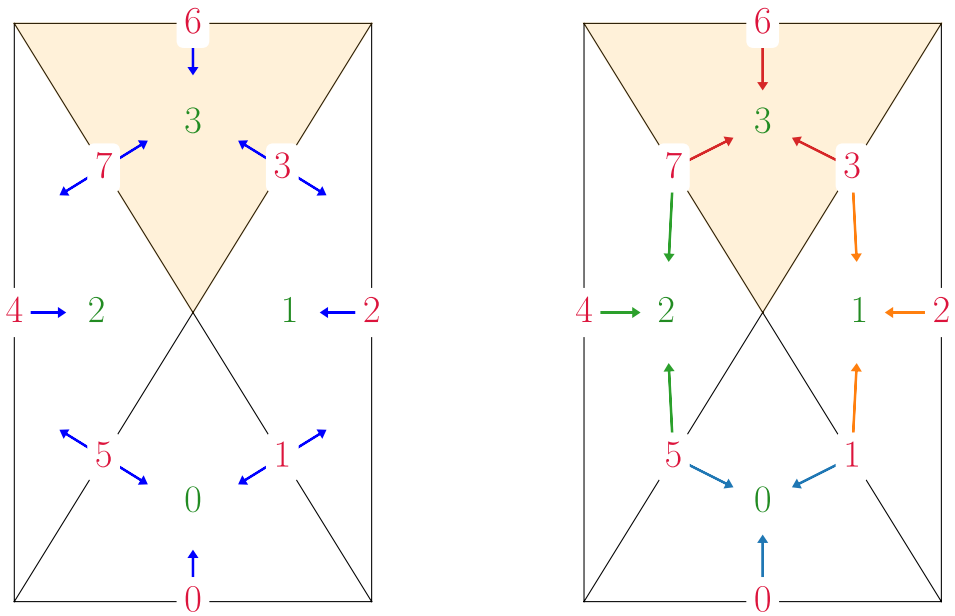


Figure 5.14 – Updated program structure of the parallel implementation

from the buffer as a constant and combined with source term contributions. A diagram of this interface-based approach appears in Figure 15a.

In contrast, a cell-based approach as in Figure 15b may feel more intuitive given the mathematical formulation. To support it, each cell must store a list of references to its adjacent interfaces. Though both approaches require the same number of floating point operations, they differ in memory access patterns. As shown in Figure 5.15, both lead to the same number



(a) Interface-based approach (b) Cell-based approach

Figure 5.15 – Memory accesses of flux balances

of memory accesses (i.e., same number of arrows in the figures). However, in the interface-based approach, flux-balancing memory accesses happen earlier since computing fluxes with the HLLC scheme requires hydraulic variables from both sides. These are likely already in cache and thus cheap to access. A reminder on cache mechanisms is given in Section 5.5. On the other hand, the cell-based approach involves three uncached and costly memory accesses per cell during the update step. This can only be partially mitigated by reordering so that neighboring interfaces have nearby indices.

The main caveat of the interface-based approach is the greater difficulty in parallelizing it. While interfaces are processed in parallel by multiple threads, two threads may attempt to access the same cell at the same time, potentially causing a race condition. To avoid this, cell buffers must be protected with mutexes, ensuring only one thread modifies them at a time. This adds some serialization overhead, but it remains low, as the typically large number of interfaces spread across threads makes simultaneous access to the same cell unlikely.

5.3 The precision issue

Real numbers are used in most scientific software. However, computers have finite memory to represent this infinite continuum. As a result, only a discrete subset can be stored, introducing round-off errors. The IEEE 754 [76] standard defines how floating point numbers are stored

in hardware and specifies rounding rules for arithmetic operations. In particular, floating point operations must follow correct rounding, that is, the result of an operation should be the exact value, whether representable or not, rounded to the nearest machine number. One consequence is that addition is not associative, especially when combining numbers of different magnitudes. The following Python example shows this.

```
(1e20 + (-1e20)) + 3.14 # outputs 3.14
1e20 + ((-1e20) + 3.14) # outputs 0.0
```

A consequence of this rounding behavior is that multiple executions of a parallel program can produce different results if the order of arithmetic operations depends on scheduling decisions. In our case, one of our unit tests simulates Watlab on an artificial circular dam break with sediment transport. It considers a domain $\Omega = [0, 5] \times [0, 5]$, with a circular subregion $\Omega_D = \left\{(x, y) \in \Omega \mid \left(x - \frac{5}{2}\right)^2 + \left(y - \frac{5}{2}\right)^2 \leq \left(\frac{3}{4}\right)^2\right\}$ representing a dam that suddenly breaks. The mathematical initial state of the hydraulic variables for this test case is as follows:

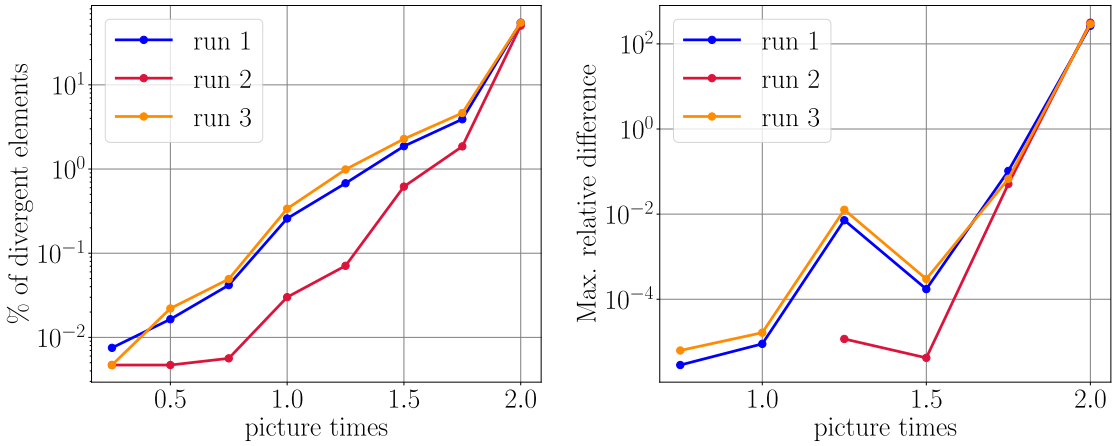
$$h^0 = \begin{cases} 1 & \text{if } (x, y) \in \Omega_D \\ 0 & \text{if } (x, y) \in \Omega \setminus \Omega_D \end{cases} \qquad h_s^0 = \begin{cases} 0 & \text{if } (x, y) \in \Omega_D \\ \frac{1}{5} & \text{if } (x, y) \in \Omega \setminus \Omega_D \end{cases}$$

with h_s^n denoting the sediment level. The difference in water height along the radial direction simulates a dam break over a few seconds. The parallel version described earlier fails this test. Moreover, the differences between the expected and actual results vary across executions. To better understand the issue, we recorded snapshots every $0.25\,[\mathrm{s}]$ and compared them with the serial version. Results are shown in Figure 5.16. We observe that the differences are nondeterministic. In addition, both the proportion of diverging elements and the maximum relative error tend to increase over time.

The only source of nondeterminism in the program comes from scheduling decisions. After carefully verifying that no race conditions occur, we found that the execution order of threads affects the order of flux balancing in the interface-based approach, which can alter the result. For example, consider Figure 5.15, and in particular the cell with index 3. In the serial version, interfaces are processed in order, so the accumulation in the buffer results in

$$\sum_j \boldsymbol{T}_j^{-1} \boldsymbol{F}_j L_j = \boldsymbol{T}_3^{-1} \boldsymbol{F}_3 L_3 + \boldsymbol{T}_6^{-1} \boldsymbol{F}_6 L_6 + \boldsymbol{T}_7^{-1} \boldsymbol{F}_7 L_7$$

Whereas in parallel, interfaces are processed without any guaranteed order, so we have no control over the sequence in which the boundaries of cell 3 are handled, which might be



(a) Evolution of diverging proportion (b) Evolution of maximum relative error

Figure 5.16 – Nondeterministic behavior of parallel implementation

$$
\begin{aligned}
& \boldsymbol{T}_3^{-1}\boldsymbol{F}_3 L_3 + \boldsymbol{T}_6^{-1}\boldsymbol{F}_6 L_6 + \boldsymbol{T}_7^{-1}\boldsymbol{F}_7 L_7 \\
\text{or} \quad & \boldsymbol{T}_3^{-1}\boldsymbol{F}_3 L_3 + \boldsymbol{T}_7^{-1}\boldsymbol{F}_7 L_7 + \boldsymbol{T}_6^{-1}\boldsymbol{F}_6 L_6 \\
\text{or} \quad & \boldsymbol{T}_7^{-1}\boldsymbol{F}_7 L_7 + \boldsymbol{T}_3^{-1}\boldsymbol{F}_3 L_3 + \boldsymbol{T}_6^{-1}\boldsymbol{F}_6 L_6 \\
\text{or} \quad & \dots
\end{aligned}
$$

Due to round-off errors, all these operations, though mathematically equivalent, can produce different results in the computer implementation. Further investigation revealed an example of such discrepancies. The flux balance of a cell yielded in the first run was:

```
(3.74744 - 4.61448) + 0.867004 = 9.70675e-18
```

While in the other:

```
(0.867004 + 3.74744) - 4.61448 = 0
```

This is an example of *catastrophic cancellation*, a well-known round-off issue that causes significant precision loss when two close floating-point numbers cancel each other. Although the differences are relatively small, they propagate throughout the simulation due to the explicit nature of the computational scheme, which constantly reuses computed values, and to neighboring cells through the numerical fluxes. It is therefore not surprising to observe the patterns seen in Figure 5.16. Of course, these differences are merely numerical artifacts resulting from the computer-based discretization of the shallow water equations, which already introduce approximation errors. Thus, no execution can be considered physically more correct than another. However, the desire for reproducibility and the unbounded nature of error growth

motivate us to adapt the parallel implementation to eliminate this nondeterministic behavior. The following sections present two attempts to achieve this goal.

Note that we did not observe such nondeterministic behavior in the Toce simulation case study, regardless of how long the simulation ran or how large the mesh was. The example highlighted in the circular dam break occurs because the cell appears to be in steady state: inflow and outflow cancel out. This does not seem to happen in the Toce simulation.

5.3.1 First attempt

To always get hydraulic values identical to the serial version, we just need to sum the fluxes in the same order, i.e. by increasing interface indices. This is not entirely trivial in the interface-based approach since a thread processing an interface cannot know if the previous ones have already been handled without expensive inter-thread communication. The simplest solution is to shift the summation to the finite-volume update step, adopting a cell-based approach. At this point, it suffices to retrieve the fluxes from the surrounding interfaces and accumulate their contributions in increasing order. As previously mentioned, the main caveat is the overhead from additional memory accesses. Finally, note that this approach removes the need for mutexes.

5.3.2 Second attempt

A better approach that maintains performance while ensuring determinism is to transform each cell's buffer into a vector of buffers. Each interface has precomputed local left and right indices that point to the appropriate position in the buffer vector where the fluxes should be stored. In the update step, we simply accumulate the vector entries, avoiding any additional memory accesses outside the cell's buffer. Therefore, the number of uncached memory accesses is nearly the same as the interface-based implementation.
However, one detail remains. Interfaces do not always add fluxes to the left and right cell buffers. When both water heights are below the user-defined threshold representing zero depth, the hydraulic variables are preserved but not added. The program handles this with if-else logic that checks the water levels in both cells. When shifting the flux balance to the cell update step, these conditions needed to be rechecked before accumulating the boundary interface fluxes in the first version. These checks require extra memory access and increase overhead. In this second version, we filled the buffer with +0 in these cases, since adding +0 leaves values unchanged in IEEE 754-compliant systems[7].

[7]More precisely, we have $x + (+0) = x$ for every floating point number x except the negatively signed zero -0, for which $-0 + (+0) = +0$.

5.4 Scheduling

Concerning OpenMP loops, OpenMP provides several scheduling policies that can be passed to the compiler using the `schedule(<policy>, <block_size>)` keyword in the directive. Users can choose between `static`, `dynamic`, or `guided` scheduling policies. In static scheduling, loop iterations are split into contiguous blocks of roughly equal size at compile time and assigned to threads. The `block_size` argument can reduce block size and distribute them cyclically among threads. Dynamic scheduling assigns blocks at runtime, adding overhead. Threads receive a new block of size `block_size` as soon as they finish the previous one. If not specified, `block_size` defaults to 1. Guided scheduling is a dynamic strategy with decreasing block sizes. The scheduler chooses a block size proportional to $\max($`block_size`$,$ `unassigned_iterations / num_threads`$)$.

A diagram summarizing the different strategies for 4 threads is shown in Figure 5.17.

Naturally, the overhead of dynamic scheduling is offset by its ability to better handle load imbalance. When iterations vary in computational cost, some threads may remain idle while others are still processing, resulting in uneven workload distribution and reduced parallel efficiency. A static partitioning cannot account for this variation and is more prone to load imbalance.
In Watlab, imbalance stems from the presence of dry and wet cells. No flux or source terms are computed between two dry cells, making iterations over wet cells longer and requiring careful distribution among threads. Thus, the parallel flux and source term steps in each finite-

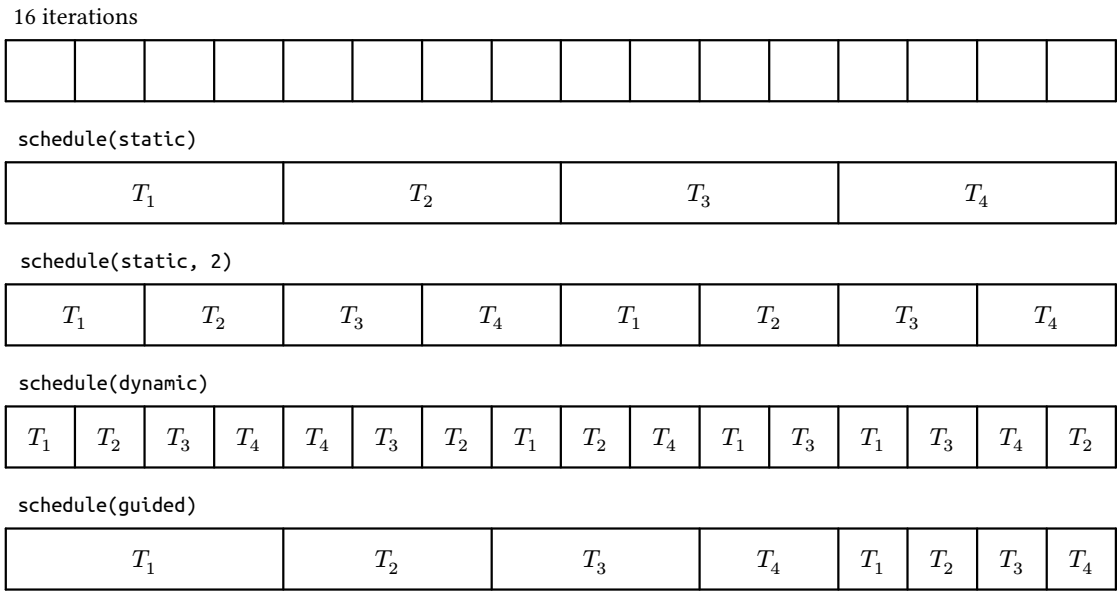


Figure 5.17 – Scheduling policies

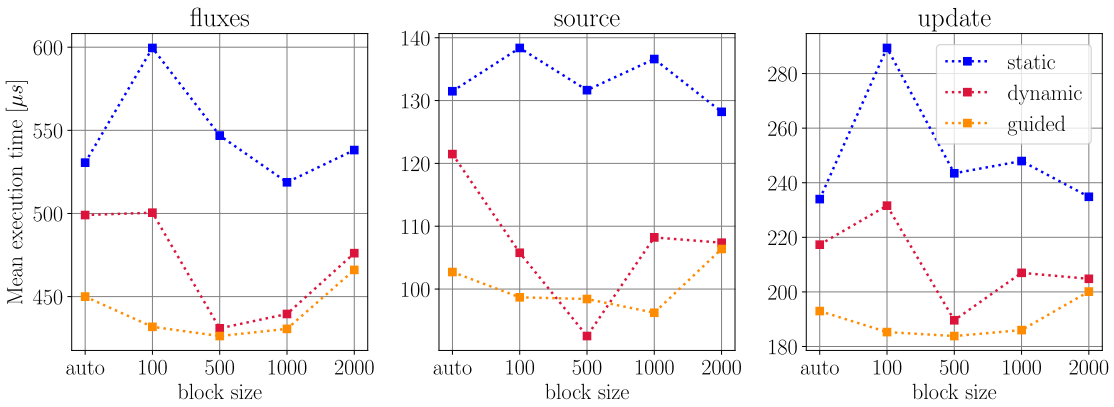


Figure 5.18 – Mean kernel execution time with each scheduling policy

volume time step likely benefit from dynamic scheduling. In contrast, updating the hydraulic variables is more uniform, mainly summing cell flux buffers that are zero for dry cells, making static scheduling more appropriate. To confirm this, we analyzed the mean execution time of each computational kernel during the Toce simulation (Figure 5.18).

We observe that guided scheduling is the most efficient for all steps with the default block size. When varying the block size, timings remain fairly stable, though chunks of 500 iterations appear to be a good choice. However, this value likely depends on the mesh studied, so we retain the default. Increasing the block size brings performance closer to that of guided scheduling but remains more variable. In contrast, the static policy suffers significantly from load imbalance and performs much worse than the other two.

5.5 Memory

5.5.1 Roofline analysis

To gain further insights into the bottlenecks in the program and identify what exactly should be optimized, we should first determine whether the program is *compute bound*, meaning the CPU reaches its floating point functional limits, or *memory bound*, meaning the CPU spends most of its cycles waiting on data loads and stores. This is mainly determined by the *data intensity* of the implemented algorithm, which refers to the ratio of floating point operations to the number of bytes moved. The *Roofline model* [77] relates data intensity to the number of floating point operations per second by showing the maximum achievable performance on the system, depending on the peak achievable bandwidth and the peak performance of the processor.

If we can empirically measure the data intensity and the floating point throughput, we can place an algorithm on the Roofline. The roof that intersects with the vertical line drawn from that point allows us to determine whether it is compute bound or memory bound. Furthermore, the vertical gap between the algorithm and the roof helps assess whether there is room for improvement or if the program already pushes the hardware to its functional limits.

To do this, we used Intel Advisor, which is able to analyze each function separately. The resulting analysis is shown in Figure 19a. Each circle represents a function call, with its size and color proportional to its self execution time. Therefore, large red circles are the most important to optimize according to Amdahl's law. The green circle in the lower left represents the `Domain::Update` function, which roughly corresponds to the full program execution. Its low data intensity indicates that Watlab is essentially a memory bound program. We can zoom into the different child functions to gain further insights, as shown in Figure 19b.

We see that all functions except dtMin lie below the peak bandwidth of the third cache. This suggests that greater use of the first and second cache levels may improve performance.

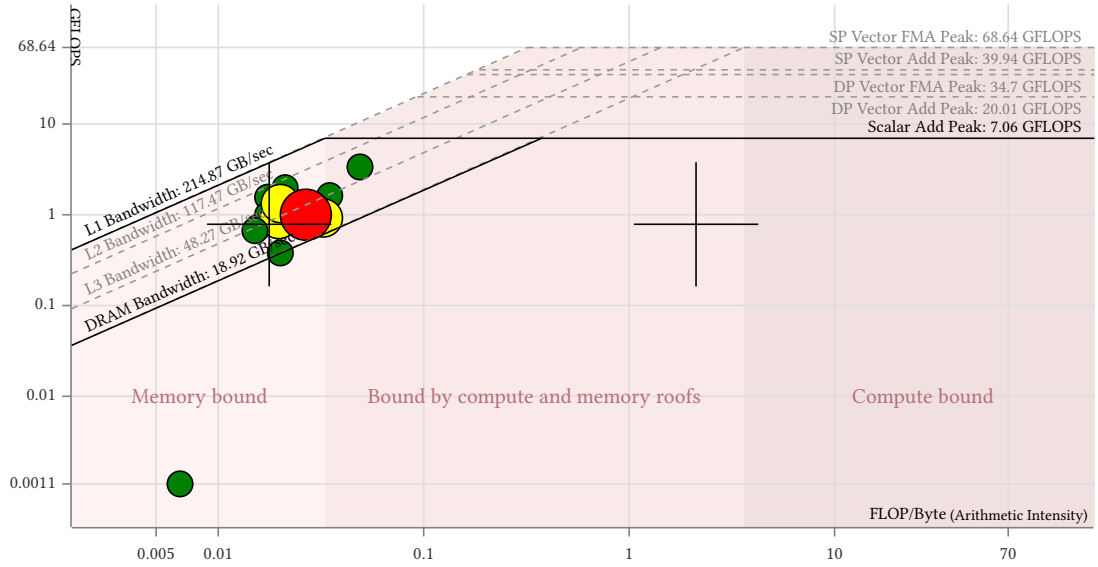
5.5.2 Leveraging caches

From their inception, CPUs have continually become faster. Memory speeds have also increased, but not at the same pace as CPUs [78]. While a processor can perform several operations per clock cycle, accessing data from main memory can take tens of cycles [6]. This phenomenon is known as the *memory wall* [79].

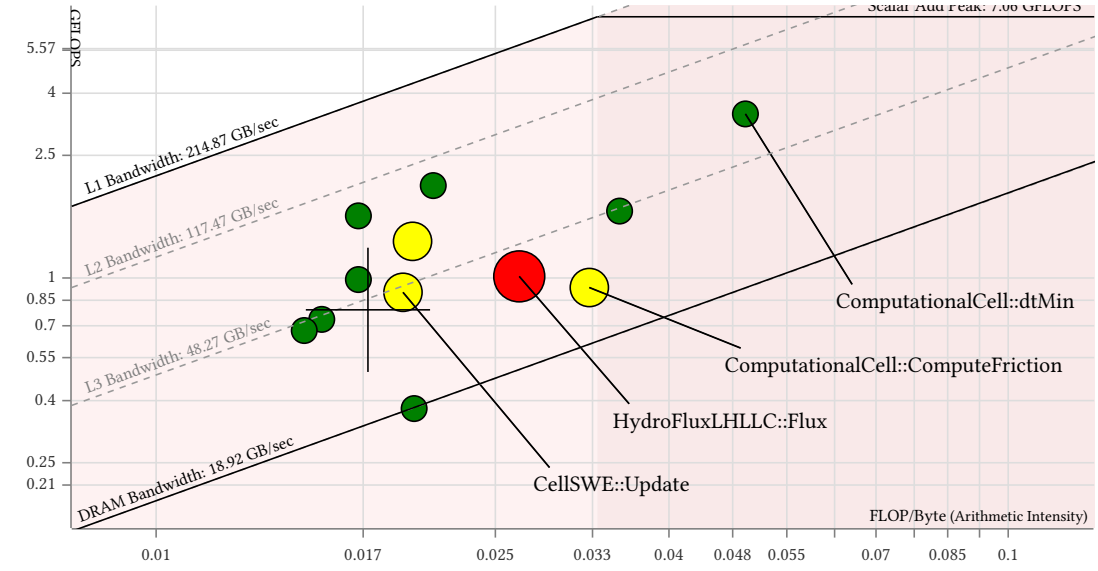
To alleviate this bottleneck, faster on-chip memory called *cache* was developed, greatly reducing memory access times. However, cache hardware consumes more power and requires more transistors than main memory, which limits its capacity. Caches are divided into *cache lines*, typically 64 bytes long on modern processors. Each cache line stores a copy of a memory block from main memory.

When the CPU needs to read an address from main memory, it first checks whether the data is already present in the cache. If it is, this is called a *cache hit*. If not, it results in a *cache miss*, and the data must be loaded from main memory into a cache line, possibly evicting another line to make room, before being moved from the cache to the registers.

Although cache behavior is managed by the hardware and not under direct programmer control, writing code with cache usage in mind can significantly improve performance. Specifically, maximizing *temporal* and *spatial locality* increases the likelihood of cache hits. Temporal locality refers to the tendency of programs to reuse the same data within short time intervals. Spatial locality means that programs tend to access memory locations that are close to each other [6].



(a) Distribution of functions in the Roofline model



(b) Zoom

Figure 5.19 – Roofline analysis of Hydroflow using Intel Advisor

To evaluate the number of cache hits and misses during the execution of Watlab, we used Cachegrind, a tool from the Valgrind framework [80], which simulates cache behavior and provides line-by-line annotations of source code indicating the number of cache read and write misses.

In the original Watlab implementation, intermediate results used in the flux computations at boundary and inner interfaces were stored in member variables, i.e., variables that are part of a class object and accessible by all functions within the class. Cachegrind analysis revealed that these member variables caused numerous cache misses, as they were accessed only once per iteration and stored in main memory. Furthermore, most of these variables were only used inside the flux computation methods.

As a first memory optimization, we converted these into local variables, restricting their scope to the flux computation function whenever possible. This ensured they were stored on the stack or in registers, which are theorically faster to access.

Additionally, other cache misses were related to memory accesses during iterations over cells, for computing source terms, updating hydraulic variables, computing the next time step, or over interfaces for flux computation. These accesses involve simple traversals of arrays of structures (cells and interfaces), and cannot be further optimized. The memory layout of cells, nodes, and interfaces follows the order in the input files generated by the preprocessing Python API.

However, during flux computations at each interface, we must access the left and right cells to retrieve associated water heights for inner interfaces, or only the left cell for boundary interfaces. If the cells are renumbered such that the index difference between left and right cells is minimized, we can benefit from improved spatial locality, as the accessed cells are likely to be closer in memory.
Fortunately, this is precisely what the reverse Cuthill-McKee algorithm (RCM) can help us to do. It is initially designed to permute a symmetric sparse matrix in order to reduce its bandwidth. In our case, this matrix corresponds to the adjacency matrix derived from the dual graph associated with the mesh. If you consider each cell as a node in a graph connected to neighboring cells, i.e., each inner interface represents an edge, you can easily derive an adjacency matrix.
To illustrate, we consider a simple square mesh composed of 26 triangular elements (Figure 20a). The lower triangular part of the original symmetric adjacency matrix is shown in Figure 21a, while the upper triangular part represents the updated matrix after applying the reverse Cuthill-McKee algorithm. The bandwidth of the resulting matrix is much smaller, meaning that left and right cell indices of each interface are now closer in memory.
However, However, this renumbering strategy alone will not reduce the number of cache misses. This is because, while spatial locality is improved, temporal locality is simultaneously degraded. The colors of each entry in the adjacency matrices represent the corresponding interface indices, ranging from cyan (low indices) to pink (high indices). In the initial lower matrix, we observe a smooth gradient from left to right. This is linked to how GMSH [81], the mesh generator used to produce the mesh files feeding Watlab, numbers the interfaces. A closer look at Figure 20a reveals that it first numbers boundary interfaces in a counter-

clockwise manner. Then, inner interfaces are ordered by the left cell index and, for interfaces sharing the same left index, by the right cell index. As interfaces are processed in order, this improves temporal locality by increasing the likelihood of accessing common neighboring cells consecutively.
After reordering the cells, we lose this benefit. The new indices disrupt the original sorting, as shown by the shuffled colors in the updated adjacency matrix of Figure 21a. The solution, however, is quite straightforward. We can simply renumber the interfaces by sorting them based on the new left and right indices, using an algorithm like Quicksort [82] (Figure 21b).

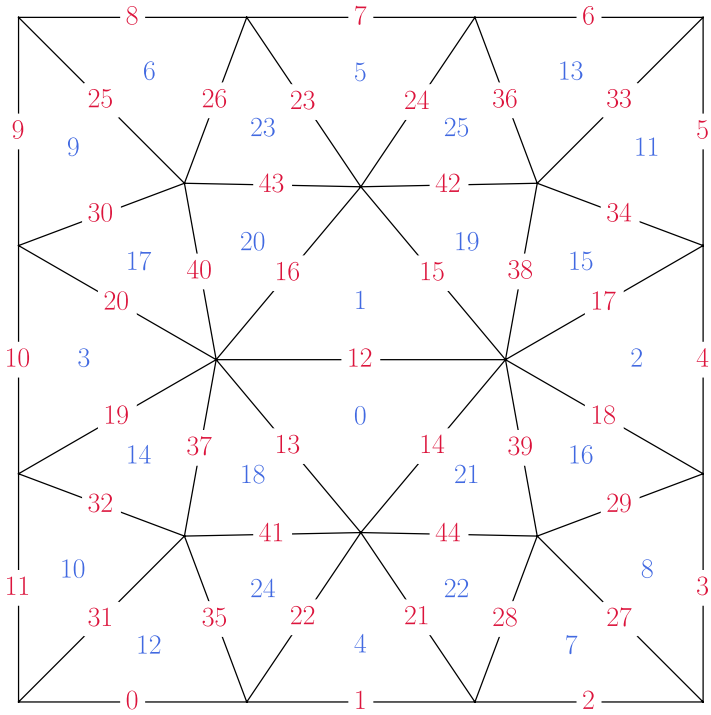
5.6 Benchmarks

To assess the performance of each version presented in the previous subsections, we recorded the mean execution time of the Toce case study over 100 runs, discarding 10 warm-up executions (Figure 5.22). Note that all parallel versions use 14 OpenMP threads, matching the physical number of cores of our processor. First, we observe that reordering the mesh slightly reduces execution time regardless of parallelism. Then, we see that the first attempt at a deterministic OpenMP enabled version introduced significant overhead compared to the initial parallel implementation based on locks. While this was mitigated by converting member variables to local ones when possible, the second attempt is clearly faster and further reduces execution time. The lowest bar corresponds to our final parallel version, which includes all the presented features and will be referred to as the parallel version of Watlab from now on.

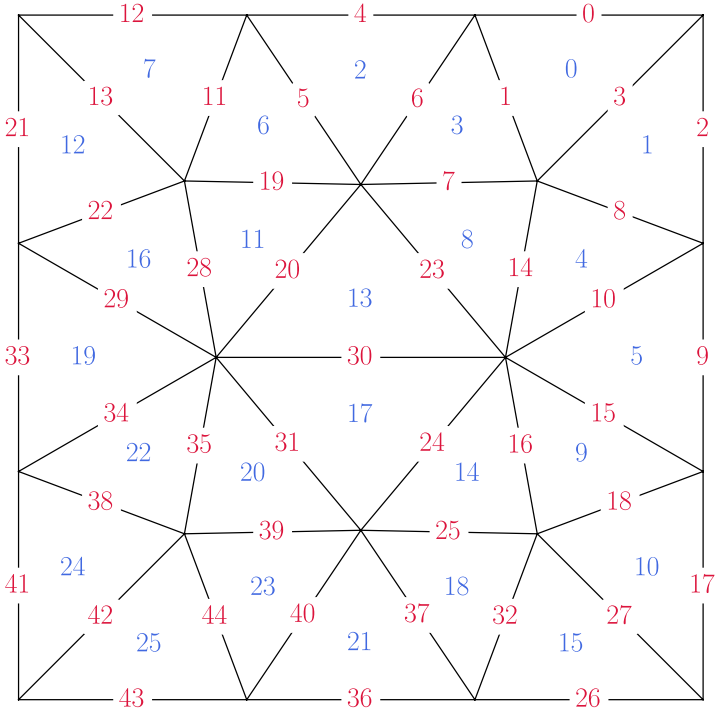
To further assess the potential gains brought by our memory optimization in the Flux function, we show the profiling update in the Roofline model in Figure 5.23. We observe that the arithmetic intensity has increased. Subsequently, the arithmetic throughput increases by $41.47\,\%$ percent, reducing execution time by $43.78\,\%$. The peak bandwidths differ from the previous figure as we now consider all 14 cores of our processor.

Finally, we inspect the impact of varying the number of OpenMP threads on the measured elapsed time. The measurements appear in Figure 5.24. As the thread count increases, execution time falls until all 14 physical cores are used. The Intel Core i7 13700H we use has 6 performance cores that support 12 logical threads via Intel Hyperthreading. By default OpenMP can use up to 20 threads. However, performance with 20 threads is worse than with 4. In that case the total concurrent threads can reach 24 when outputs are recorded, exceeding available resources and forcing interleaved execution. With 16 threads the results vary widely: one run may be very fast, another much slower.

With these measurements we can also compute the speedup factors relative to the sequential version (Table 5.3) to analyze scalability. The program scales fairly well up to 8 threads. Beyond that, using more threads increases performance only slightly. This may be due to

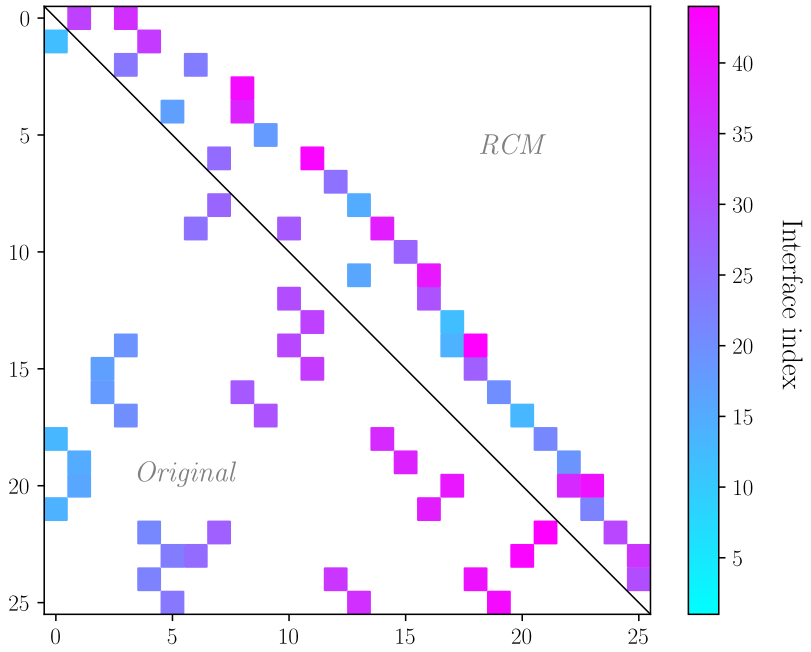


(a) Before reordering

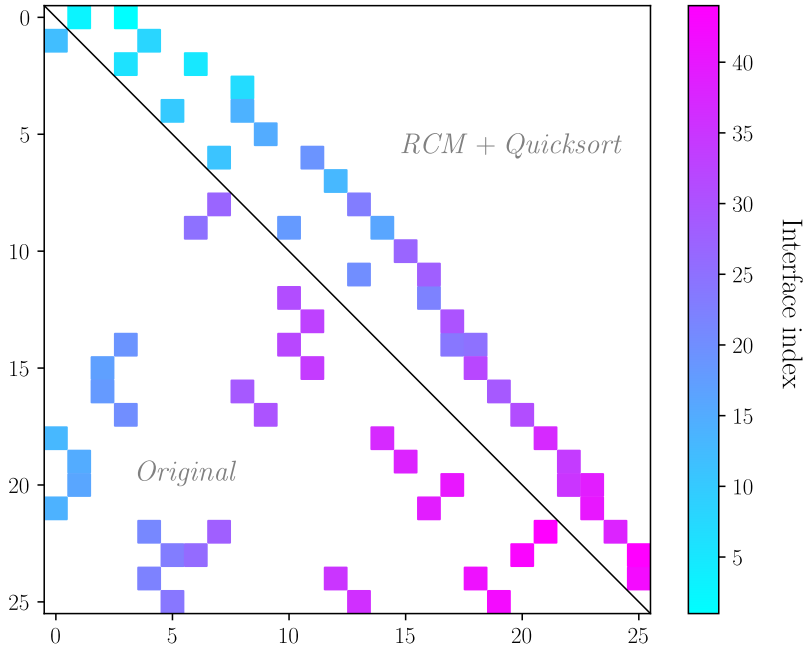


(b) After reordering

Figure 5.20 – A simple square mesh with 26 cells and 45 interfaces



(a) Renumbering cells using the reverse Cuthill-McKee algorithm



(b) Renumbering cells using the reverse Cuthill-McKee algorithm and interfaces using Quicksort

Figure 5.21 – Adjacency matrices related to the square mesh

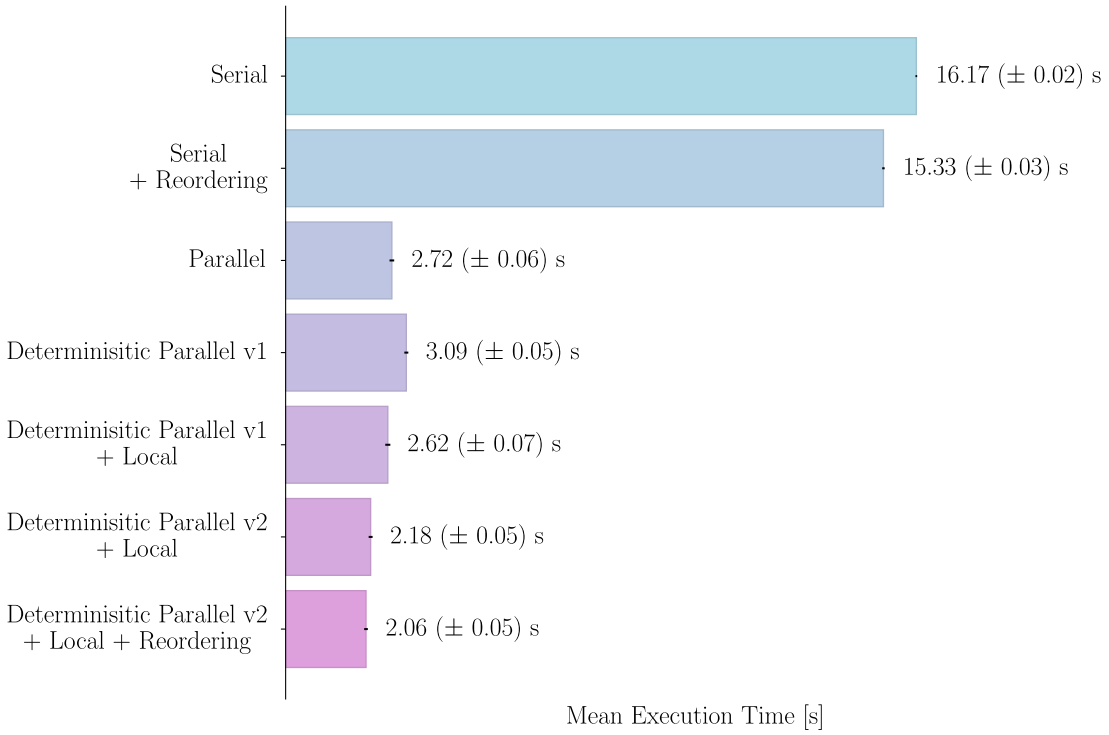


Figure 5.22 – Comparison of serial and parallel versions using 14 threads (Toce)

the small scale of the case study, which does not generate enough workload to offset thread

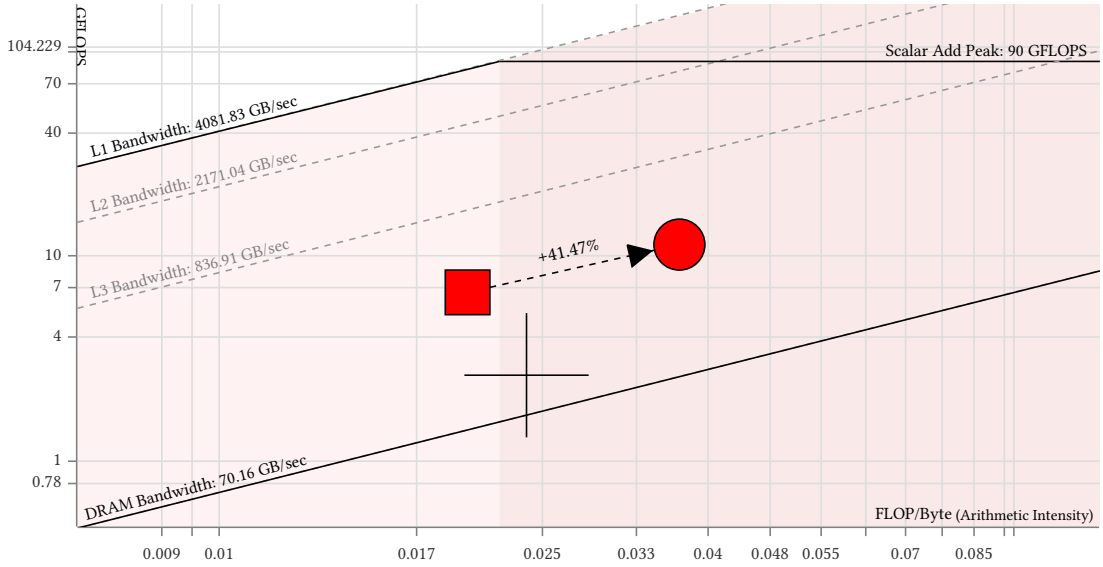


Figure 5.23 – Comparison of `HydroFluxLHLLC` roofline analysis

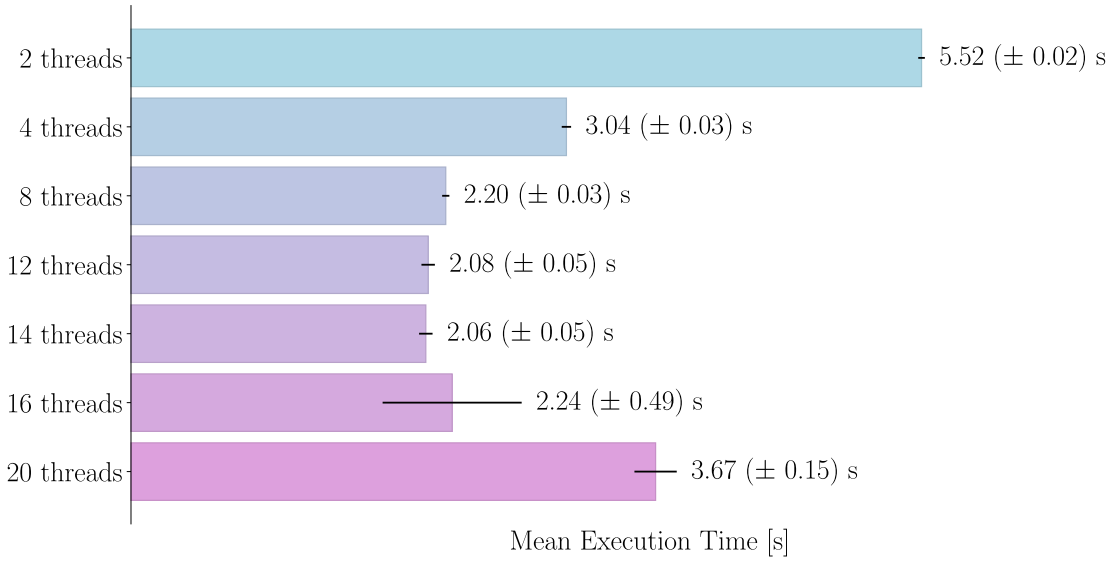


Figure 5.24 – Impact of the number of OpenMP threads on the execution time (Toce)

Threads	**2**	**4**	**8**	**12**	**14**	**16**	**20**
$\text{Speedup}_{\text{serial}}$	2.93	5.31	7.35	7.79	7.85	7.20	4.41

Table 5.3 – Speedup factors with respect to the serial implementation

spawning overhead. More likely, threads access data on the same cache lines, causing *false sharing*. Therefore, we examined the results from our larger scale case study to confirm those assumptions.

Due to technical constraints, the number of cores that comprise contemporary multicore processors is limited, thereby constraining the achievable speed-up of CPU-parallelized programs in accordance with Ahmdal's law (Equation 5). At the time of this writing, the latest generation of Intel Core i9 processors for consumers has a maximum of 24 cores. Consequently, to further reduce execution times, it is necessary to explore alternative hardware solutions, such as GPUs or FPGAs, which have emerged in the HPC field over the past decades [83].

We decided to develop a GPU implementation of Watlab to pursue the work initiated in the previous thesis [12] and leverage the widespread presence of GPUs in consumer computers, originally designed for gaming. A functional and efficient GPU-enabled program would benefit many Watlab users.

6
Implementations on GPU

As introduced in the state of the art, Graphics Processing Units were originally used for video rendering. Historically, their main users were in the gaming community. In most video games, a dynamic virtual world must be rendered many times per second to ensure a smooth visual experience. This process involves computing the color of each pixel based on lighting, textures, and perspective. The massively parallel architecture of GPUs is well suited for this task.

We observe a clear parallel with our hydraulic solver: pixels are replaced by interfaces and cells, and the small, repeated computations become the calculation of fluxes, source terms, and finite-volume updates (Figure 6.25). This intuition has driven research into using GPUs to

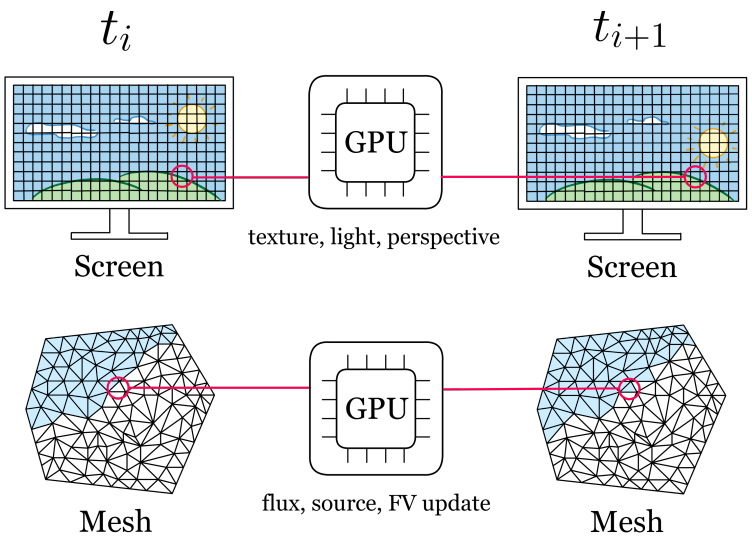


Figure 6.25 – Insight behind Watlab's GPU implementation

accelerate SWE solvers and, more broadly, scientific simulations for over a decade. As reported in Table 3.1, many GPU implementations have achieved impressive speedups that are not attainable on CPUs due to their limited core count. It is therefore natural to want Watlab to benefit from this technology as well.

Reviewing the literature revealed that a key to efficient implementation on Graphics Processing Units is a solid understanding of their underlying architecture. We will therefore first present the hardware, followed by the programming abstraction built on top of it.

6.1 Hardware

Graphics Processing Units are built around an array of Streaming Multiprocessors[8] [84]. Each SM contains a fixed number of cores, which are processing units capable of executing instructions, shown in green on Figure 6.26.
The capabilities of these SMs vary between generations and are defined by the GPU's *Compute Capability* (CC). For example, the throughput of single and double precision arithmetic instructions per clock cycle for various Compute Capabilities is reported in Table 6.4. The cores within each SM are simple and execute the same instruction at the same time, following the Single Instruction, Multiple Threads (SIMT) paradigm.
Figure 6.26 also shows the memory hierarchy. L1 caches are private to each SM, while the

[8]For AMD chips, these are called *Compute Units*. The rest of the section focuses on NVIDIA's terminology and architecture, which dominates the HPC community. Other vendors' hardware is broadly similar apart from naming.

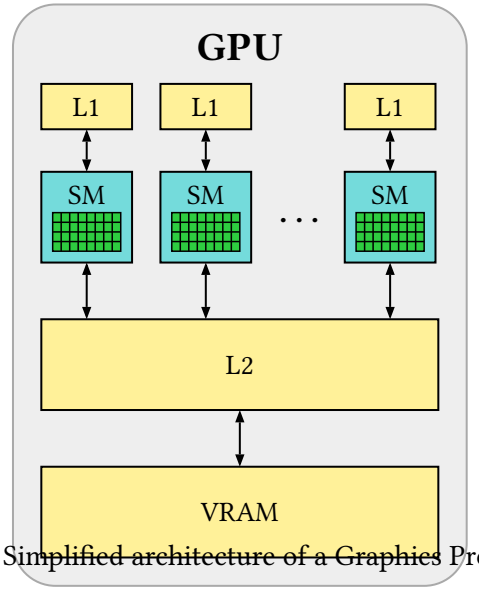


Figure 6.26 – Simplified architecture of a Graphics Processing Unit

larger L2 cache is shared across all SMs. Access to the off-chip device[9] memory, known as *Video Random Access Memory* (VRAM), goes through the L2 cache. Since the number of SMs varies between GPUs, NVIDIA introduced the CUDA programming model to abstract execution and memory, allowing programs to scale transparently with the number of cores available on the hardware.

6.2 CUDA programming model

6.2.1 Execution

CUDA C++ handles the heterogeneity of GPU architectures by introducing the thread abstraction, which represents the smallest unit of execution and corresponds to a sequence of

CC	**5.0, 5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.x**	**8.0**	**8.6**	**8.9**	**9.0**
32-bit (single precision)	128		64	128		64			128	
64-bit (double precision)	4		32	4		32	32	2		64

Table 6.4 – Floating-point add, multiply, multiply-add per clock cyle per SM

[9]Note that we will use the terms device and GPU interchangeably. In NVIDIA terminology, device commonly refers to the GPU in the CUDA programming model. Higher level abstraction libraries such as SYCL or Kokkos have reused this term to refer to the accelerator being used. In this context, it can refer not only to GPUs but also to FPGAs, for example. In the rest of this writing, when we use this term, we will refer specifically to Graphics Processing Units. In contrast, the term host will refer to the main CPU in the system.

instructions run on the cores of an SM. In practice, developers define a function to be executed on the GPU much like a regular C++ function, as follows:

```
// Regular function
void updateCPU(Cell *cells, int n_cells)
{
  for (int i = 0; i < n_cells; i++)
  {
    cells[i].U += ...;
  }
}

// GPU's version
__global__ void updateGPU(Cell *cells, int n_cells)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n_cells)
  {
    cells[i].U += ...;
  }
}

updateGPU<<<blocksPerGrid, threadsPerBlock>>>(...);
```

The function compiled to run on the GPU is called a kernel and is marked by the `__global__` keyword from the CUDA C++ extension. In this case, it corresponds to writing a single iteration of the for loop we aim to parallelize in the CPU version. The kernel body must be executed for many values of `i`, representing each cell index. The resulting sequences of instructions for each index are the threads.

To determine which core of which multiprocessor runs each thread, CUDA uses a hierarchical organization. Threads are grouped into blocks, and blocks are distributed across available SMs. The number of threads per block is defined by the user and passed through the `threadsPerBlock` variable using the *execution configuration* syntax `<<<...>>>`, another CUDA extension.

Since the number of threads in a block may exceed the number of available cores in an SM, each block is split into warps of 32 threads. Warps are executed concurrently and scheduled by the SM's schedulers. Traditionally, all threads in a warp shared the same program counter, which holds the address of the next instruction. As a result, they executed the same instruction in lockstep.

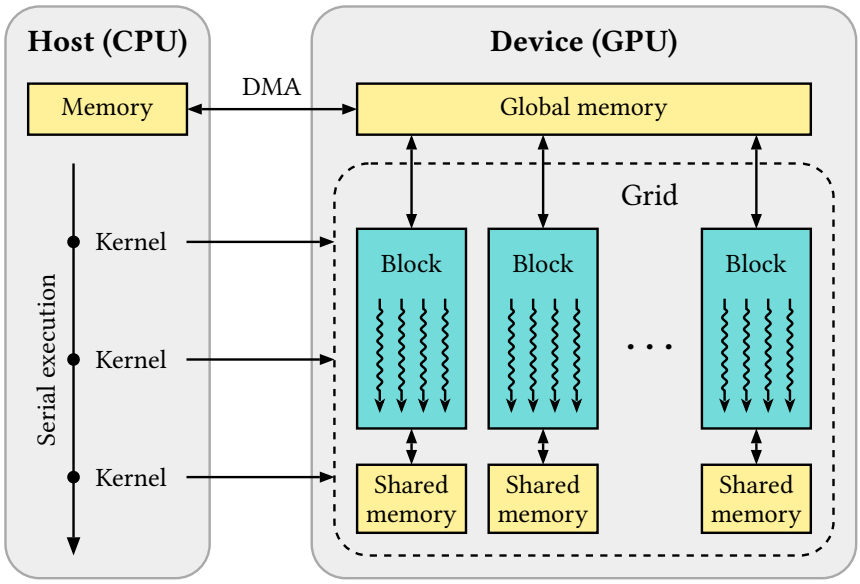


Figure 6.27 – CUDA thread hierarchy (adapted from [1])

A major drawback of this model appears with conditional branches. When some threads follow an if-branch while others do not, only the active ones execute while the rest are masked and wait. Once that path completes, the others execute. These paths are serialized and run one after the other[10]. This behavior is called *warp divergence* and should be avoided to maximize parallel efficiency. Figure 6.28 illustrates this phenomenon with a dummy pseudocode.

Another important consideration is the number of threads per block. Since warp size is fixed at 32, the block size should be a multiple of 32. For example, if 48 threads are assigned to a block, two warps of 32 are created, but only 48 threads are active. The extra 16 threads in

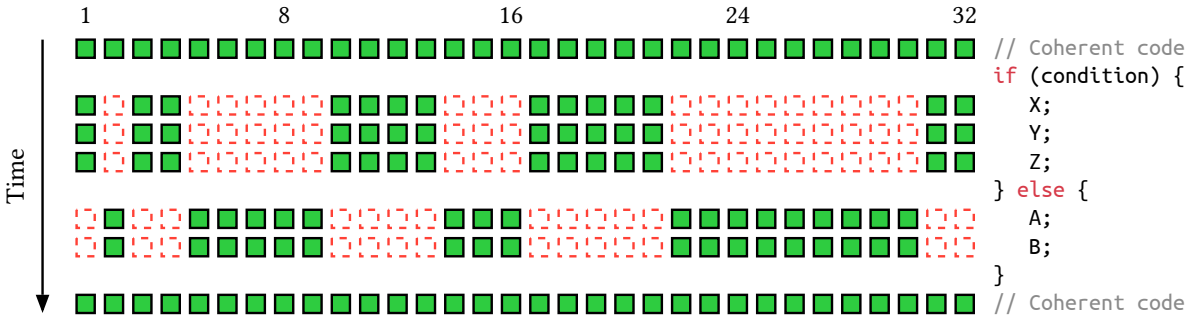


Figure 6.28 – Warp divergence (adapted from [2])

[10]Starting with the *Volta* architecture (Compute Capability 7.0), threads have independent program counters. When divergence occurs, execution paths are interleaved, enabling intra-warp synchronization

the second warp remain idle, and their results are discarded when execution ends.Blocks can also be one, two, or three dimensional for convenience. For example, applications that work with matrices may find two-dimensional blocks more suitable for partitioning data.
Finally, thread blocks are organized into a grid to complete the thread hierarchy. The grid can also be one, two, or three dimensional. In the code snippet above, both the block and grid are assumed to be one dimensional. The global thread index, which corresponds to the cell to be processed, is computed using:

`blockIdx.x` the block index within the grid along the x direction

`blockDim.x` the number of threads per block along the x direction

`threadIdx.x` the thread's local index within the block along the x direction

The discussed organization is schematized in Figure 6.27, where threads are depicted by wavy arrows.

To conclude, GPU execution features multiple levels of concurrency. At the highest level, multiprocessors execute blocks in parallel. Within each multiprocessor, warps are executed concurrently using the SM's resources, either interleaved or simultaneously depending on the number of schedulers and available cores. Finally, threads within a warp run in parallel on the SM's CUDA cores.

6.2.2 Memory

The CUDA memory hierarchy includes three logical address spaces. Each thread has its own private local memory. All threads within a block can access a shared memory space, which exists for the lifetime of the block. In modern NVIDIA GPUs, this shared memory is located in the L1 cache, providing faster access. The largest memory space is global memory, which mainly consists of VRAM, along with cache partitions that help speed up memory transactions. Data stored in global memory is accessible to all threads and persists across kernel launches by the same program.

To process data on the device, it must first be transferred into global memory from the host. This transfer is handled over the PCIe (Peripheral Component Interconnect Express) bus using Direct Memory Access (DMA), a mechanism that enables memory transfers without CPU involvement. The PCIe bus has limited bandwidth and can become a bottleneck for performance [11]. For example, it peaks at 15.75 GB/s on our system, while the global memory bandwidth of our NVIDIA RTX 4050 reaches 216.0 GB/s.

Despite this high bandwidth, global memory has significant latency. Fetching a single element can take hundreds of clock cycles [2]. To reduce this overhead, individual memory transactions should be minimized. Once a transaction is initiated, it should transfer as much

data as possible to make effective use of the available bandwidth.
Furthermore, the CUDA programming manual explains that when a warp executes an instruction accessing global memory, the memory accesses of its threads are coalesced into one or more memory transactions. This depends on the size of the word each thread accesses and how memory addresses are distributed across the threads. While word size is mostly outside the programmer's control, the memory layout can be adjusted so that nearby threads access nearby data. This optimization technique, known as memory coalescing, is essential for achieving efficient GPU performance.

Another key technique GPUs use to maximize performance is *latency hiding*. When a warp stalls on a memory fetch, the GPU quickly switches to another ready-to-run warp, avoiding idle time. This fast context switching requires each streaming multiprocessor to have enough warps to choose from. This is captured by the *occupancy*, defined as the ratio of active warps on an SM to the theoretical maximum it can support. The number of active warps depends not only on block size but also on each thread's resource usage, such as registers and shared memory. However, higher occupancy does not always mean better performance. Once memory latency is fully hidden, increasing occupancy further can reduce per-thread resources and hurt overall performance.

These considerations show that GPGPU is not a miracle solution for speeding up every algorithm. It is effective only for highly parallel tasks that can fully utilize hardware resources by reaching sufficient occupancy. Given the high memory latency, the number of floating-point operations should outweigh memory accesses. [11] suggests a minimum ratio of two operations per memory access to make efficient use of a GPU.

6.3 Proof of Concept

As introduced in the state of the art, various solutions exist to enable an existing application to use GPUs. To choose the most appropriate framework for our project, it was necessary to first clearly define the objectives of the future GPU port. The most important are listed below.

Portability The Watlab public software is widely used by researchers at UCLouvain and final-year students with various GPUs. It also runs on the university's supercomputing infrastructure, which is heterogeneous and may evolve over time.

Simplicity Since most Watlab maintainers are hydraulics experts rather than developers, it is crucial to avoid low-level hardware-specific instructions and maintain a high level of abstraction. This ensures a focus on physical modeling and guarantees modularity for future hydraulic features. Writing the GPU version purely in C++ would further simplify the architecture.

Unified source To minimize development effort and errors, we prioritize a single source code that compiles identically for all GPUs. Additionally, we aim to reduce duplication between the CPU and GPU-enabled versions, given that Watlab already includes serial and OpenMP-based parallel implementations.

Performance Since our primary goal is accelerating Watlab, we seek a solution with minimal performance overhead.

Given the frameworks presented in the first section, these criteria help narrow down candidates but are insufficient for a final choice. Based on documentation alone, several frameworks meet the requirements, and benchmarks show similar performance between them [16], [17], [23], [85].

To gain deeper insight into the development experience and make the appropriate choice, we designed a small *Proof of Concept* (PoC) to test the frameworks. The PoC is a simplified, mathematically meaningless version of Watlab that approximates its key components. As noted in section REF, the finite volume implementation consists of flux computation at each interface, updating hydraulic variables in each cell, and a minimum reduction to determine the smallest time step (Fig REF).

In the PoC, only the water height variable and interface fluxes remain. Starting from a random water level distribution, flux computation averages water heights between adjacent cells. The cell update adds a constant of $2 \cdot 10^{-6}$ to the water height. Finally, a minimum reduction identifies the smallest water height at each time step. Each implementation must allocate and transfer data between host and GPU and retrieve results post-computation. A schema illustrating the PoC is shown in Figure 6.29.

For the final step, the reduction must be performed in parallel with optimal local memory usage. To achieve this, we use built-in reduction algorithms when available, ensuring they meet these requirements.

The candidate frameworks implemented for the PoC, alongside the serial version, include CUDA, OpenMP (CPU and GPU offloading), OpenACC, Kokkos, RAJA, SYCL, and Alpaka. The OpenMP CPU version is for comparison only. Similarly, while CUDA does not meet our previous criteria, it helps assess the overhead of abstraction libraries compared to native CUDA. OpenCL was excluded due to its incompatibility with simplicity and unified source requirements. OpenACC, primarily supported by NVIDIA, is experimentally available on AMD GPUs via GCC [86] hence it does not constitute a viable solution. However, given its simple implementation, we included it for a broader comparison.

Attentive readers may wonder which SYCL2020 implementation we used for our PoC. We chose AdaptiveCpp, as it supports CPUs via OpenMP and targets NVIDIA, AMD, and Intel GPUs while delivering competitive performance [72], [87], [88]. AdaptiveCpp is also the only

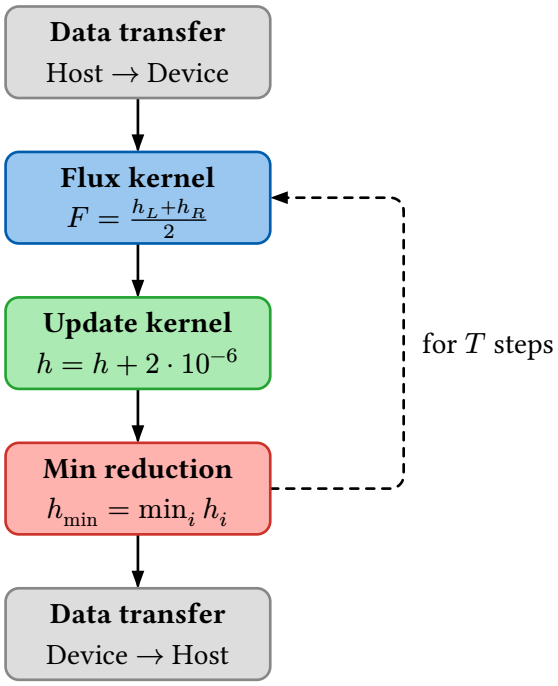


Figure 6.29 – *Proof of Concept* operating diagram

implementation that does not rely exclusively on OpenCL but also on native backends (e.g. CUDA and HIP). It might be beneficial since OpenCL has shown to be less performant than HIP and CUDA on equivalent benchmarks [16]. Finally, the key property that led us to choose AdaptiveCpp is its novel single-source, single-compiler-pass (SSCP) design [89].

Indeed, although several solutions can generate source code compilable for a broad range of architectures, this does not equate to producing a single universal binary that runs on all architectures. For instance, with the CUDA compiler, nvcc must explicitly specify Compute Capabilities to produce a fat binary compatible with GPUs following these architectures. However, the resulting executable cannot be used with AMD or Intel GPUs since no unified code representation exists between drivers. The situation is even worse for AMD, as it does not provide a device-independent code representation. Thus, compiling for all AMD GPUs requires separate compilation for each. This limitation also affects higher-level libraries like Kokkos, preventing simultaneous compilation with both CUDA and HIP backends.[11]

This is where AdaptiveCpp's generic compilation becomes attractive. The SSCP compiler stores an intermediate representation (LLVM IR) at compile time, which is backend- and

[11]Note, however, that Serial, CPU-parallel (OpenMP), and CUDA/HIP/OpenMP GPU offloading versions can be combined.

device-independent. At runtime, the architecture is detected Just-in-Time, and the code is translated accordingly to meet driver expectations.

6.3.1 Results

When compiling OpenMP with GPU offloading, we received warnings about copying non-trivially copyable data to GPU memory, meaning the mapping was not guaranteed to be correct. Our C++ program uses custom classes for mesh cells and interfaces, which have custom constructors, making them non-trivially copyable. This is a fundamental aspect of the current Watlab architecture that we cannot bypass for now. When executing the binary, it crashed inexplicably, likely due to incorrect memory mapping. As a result, the OpenMP solution has been dropped and will no longer appear in the following results.

Furthermore, implementing the PoC with RAJA revealed its backend-dependent API. Many constants and functions are prefixed with `cuda`/`hip`/`omp`/... requiring code duplication and macros to manage hardware-specific compilation. This makes the code impractical and insufficiently abstracted compared to other libraries. Consequently, RAJA will not be discussed further.

Abstraction libraries such as Kokkos, Alpaka, and SYCL allow writing code that is easily understandable for developers with no prior GPU programming knowledge. Their abstraction paradigms differ significantly from one framework to another. Alpaka mainly relies on compile time templates with a syntax relatively close to CUDA by keeping kernel definitions. In contrast, Kokkos offers a higher level of abstraction by encapsulating data into `View` objects and by providing `parallel_for` constructs that easily offload loop processing to the GPU. The approach followed by the SYCL standard is quite singular and is based on events and queues. Jobs are submitted to a queue and executed on the GPU, either in order or out of order depending on the configuration. An illustrative example of Kokkos and SYCL pseudocode compared to CUDA implementation is given below.

```
                        /* Transfer data host -> device */
// CUDA
cudaMalloc((void **) &d_cells, N * sizeof(Cell));
cudaMemcpy(d_cells, h_cells, N * sizeof(Cell), cudaMemcpyHostToDevice);
// Kokkos
Kokkos::View<Cell*, Kokkos::HostSpace> h_cells(data, N);
Kokkos::View<Cell*> d_cells("cells", N);
Kokkos::deep_copy(d_cells, h_cells);
// SYCL
sycl::queue q(sycl::default_selector{}, sycl::property::queue::in_order{});
Cell* d_cells = sycl::malloc_device<Cell>(N, q);
q.memcpy(d_cells, h_cells, N * sizeof(Cell));
```

```
                                    /* Kernel launch */
// CUDA
__global__ void updateKernel(Cell* d_cells, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
    {
        d_cells[i].h += 2e-6;
    }
}
int threadsPerBlock = 128
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
updateKernel<<<blocksPerGrid, threadsPerBlock>>>(d_cells, N);
cudaDeviceSynchronize();
// Kokkos
Kokkos::parallel_for("UpdateKernel", d_cells.extent(0), KOKKOS_LAMBDA(const int
i){
    d_theCells(i).h += 2e-6;
});
// SYCL
q.submit([&](sycl::handler &h) {
    h.parallel_for<class UpdateKernel>(sycl::range<1>(N), [=](sycl::id<1> i) {
        d_cells[i].h += 2e-6;
    });
});
```

Another interesting feature of abstraction libraries shown by the above pseudocode is the automatic assignment of the number of threads per block, which is done by querying the hardware and using occupancy heuristics. This makes the choice robust across different devices and enhances portability.

As in the Watlab implementation, the PoC includes a minimum reduction to compute the minimum water height in the domain. The need for an efficient parallel reduction algorithm is even more critical than in the CPU implementation given the high number of threads that must communicate. Furthermore, due to the high latency of global memory, intrablock communication should be preferred through shared memory when possible. To achieve a backend independent parallel implementation that meets this requirement, we rely on built in optimized algorithms provided by the libraries, which handle technical details and ease the programmer's work. This choice aligns with the PoC objectives stated at the beginning of the section.

Framework	**Duration [s]**	$\mathbf{Speedup_{serial}}$
Serial	$56.18\ (\pm 0.29)$	1
OpenMP (20 threads)	$3.31\ (\pm 0.19)$	16.95
CUDA	$2.13\ (\pm 0.01)$	26.41
Kokkos	$2.10\ (\pm 0.02)$	26.81
AdaptiveCpp	$2.32\ (\pm 0.04)$	24.22

Table 6.5 – Timings of the PoC on Toce *XL* over 1000 steps

For this reason, Alpaka remains less attractive as it does not provide such reductions, unlike CUDA, Kokkos, or AdaptiveCpp. To effectively compare the remaining candidates, we considered the following test case. We used the Toce *XL* geometry as input to create sufficient computational load so that the benefits of GPU offloading could be clearly observed. Water heights were initialized with pseudorandom values between 0 and 1 according to a user defined seed. The rest of the execution follows the chart shown in Figure 6.29 and we simulated $T = 1000$ steps. Only the execution of this part was recorded. The measurements for the different implementations are listed in Table 6.5. Execution times are measured over 100 runs, discarding the first ten to avoid warm up effects.

We observe that the speedups achieved with the GPU implementations are significant and greater than those of the OpenMP parallel version. The results are quite similar among the GPU implementations. We would also expect the CUDA implementation to yield the lowest mean execution times as it does not introduce abstraction overhead. The slightly larger execution time of the AdaptiveCpp implementation mainly results from the initialization of the queue. The point is that, as CPU and GPU are different hardware that execute independently, CPU based timings like those presented may not be accurate given the asynchronous nature of the program. A more reliable way to compare them is to use GPU specific profiling tools such as NVIDIA Nsight Compute, which is part of the CUDA toolkit and allows to perform kernel profiling. The results are shown in Table 6.6.

This analysis shows that abstraction libraries are competitive with naive CUDA code while offering simplicity and portability. Furthermore, the low compute throughput shows that this PoC is an example of a memory bound algorithm. Between the two libraries we will choose AdaptiveCpp because its generic compiler feature allows the use of a single binary while maintaining competitive performance.

Naturally, the Just-In-Time (JIT) mechanism introduces a small overhead. However, the authors of [89] report that it is of the same order of magnitude as the cost of the compilation step from IR to machine code already performed by backend drivers in current SYCL implementations. Only the first run of the program involves this JIT mechanism when a kernel

Framework	Duration [ms]	Compute Throughput [%]	Memory Throughput [%]
CUDA	1.38 ($\pm$0.004)	2.8 ($\pm$0.009)	47.758 ($\pm$0.123)
Kokkos	1.384 ($\pm$0.005)	2.796 ($\pm$0.008)	47.596 ($\pm$0.140)
AdaptiveCpp	1.38 ($\pm$0.005)	2.799 ($\pm$0.009)	47.757 ($\pm$0.140)

(a) Flux kernel

Framework	Duration [ms]	Compute Throughput [%]	Memory Throughput [%]
CUDA	0.29 ($\pm$0.004)	4.522 ($\pm$0.058)	86.744 ($\pm$0.408)
Kokkos	0.293 ($\pm$0.005)	4.46 ($\pm$0.029)	85.448 ($\pm$0.466)
AdaptiveCpp	0.291 ($\pm$0.004)	4.515 ($\pm$0.058)	86.578 ($\pm$0.516)

(b) Update kernel

Framework	Duration [ms]	Compute Throughput [%]	Memory Throughput [%]
CUDA	0.15 ($\pm$0.000)	14.18 ($\pm$0.067)	97.57 ($\pm$0.364)
Kokkos	0.151 ($\pm$0.003)	12.434 ($\pm$0.044)	91.054 ($\pm$0.290)
AdaptiveCpp	0.15 ($\pm$0.000)	9.42 ($\pm$0.022)	95.30 ($\pm$0.171)

(c) Min reduction

Table 6.6 – Per-kernel profiling

execution is requested. Kernels are then cached as executable objects so that following runs do not need to retranslate them for the target architecture. To measure it precisely, we rebuilt the SYCL executable 100 times to discard cached kernels and executed it after the CUDA version to isolate the JIT overhead from GPU initialization, which also occurs during the first run of a GPU enabled program (for example, setting up drivers). The obtained mean execution time is 2.49 ($\pm$0.05) seconds, giving an average overhead of 0.42 seconds. Note that the overhead is independent of the test case size and depends only on the number of instructions and the complexity of the kernels.

6.3.2 Optimizations

In the first subsections, we identified bottlenecks inherent to GPU architecture that can impact performance, such as warp divergence and uncoalesced memory access. Some optimizations can address these issues and improve execution speed. We implemented a few on the *Proof of Concept* to assess their effectiveness.

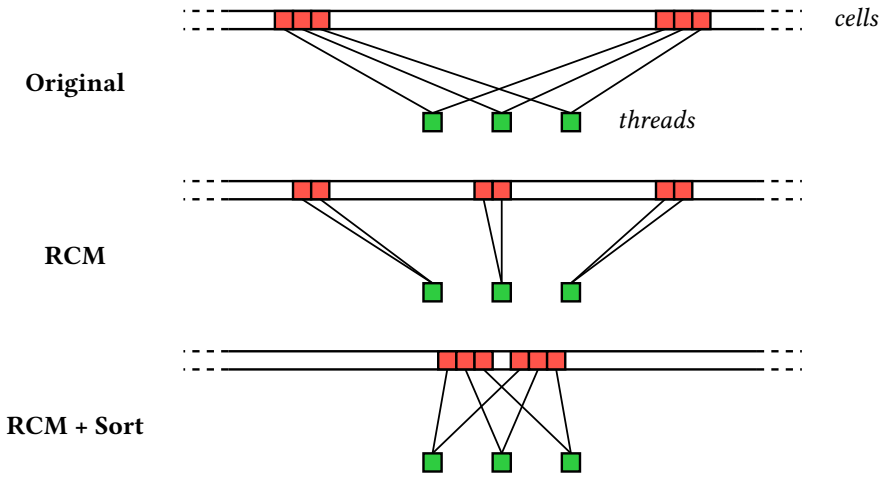


Figure 6.30 – Idealized memory accesses in the flux kernel to retrieve h_L and h_R

6.3.2.1 Mesh reordering

To maximize coalesced memory access, neighboring threads should access neighboring entries in the cell and interface arrays. In our parallel implementation, this applies primarily to the flux kernel, where each interface reads water heights from adjacent cells. As discussed in Section 5.5, applying the Reverse Cuthill-McKee algorithm to renumber cell indices brings neighboring cells closer in memory, improving per-thread coalescence. Sorting interfaces based on these new indices further enhances coalescence at the warp level, allowing threads to reuse loaded data or access nearby values. A visualization of these accesses is shown in Figure 6.30.

Coalesced memory is not the only benefit of mesh reordering. While clearly visible in the Proof of Concept due to its simplicity, it also reduces warp divergence. In Watlab, flux computations and cell updates vary when cell water heights fall below a user-defined threshold. These cells are considered dry, and no flux is computed between two dry cells.
Physically, water tends to form a contiguous wet zone as it propagates, with only the moving front expanding. This makes it likely that dry cells are surrounded by other dry cells, and similarly for wet cells, except near the wet-dry boundary. The RCM algorithm renumbers cells so that neighboring cells have nearby indices. Sorting interfaces by their left and right cells ensures that neighboring interfaces also have close indices.
As seen in Figure 20b, cell indices increase from the top-right to the bottom-left corner, and so do interface indices. This means contiguous threads in a warp are more likely to process cells in the same state—either wet or dry—leading to more uniform execution paths. Threads handling the moving front are the main exceptions.

The reasoning can be extended to boundary interfaces, which differ from inner ones as they have no right-side cells. Their flux computations are either simplified or follow hydrograph or limnigraph inputs. Since boundary conditions are applied uniformly across domain edges, which the mesh generator segments into similar interfaces, a good strategy is to start renumbering boundary edges in a counterclockwise order. This reduces warp divergence near domain edges.
GMSH already follows this approach, as seen in Figure 20a.

We tested these strategies on the same test case as above to evaluate their effectiveness using the AdaptiveCpp implementation. The results are reported in Table 7a and Table 7b.

We observe that the new reordering effectively reduces the execution time and the speedup factor increases from ~ 24 to ~ 30.5. Kernel profiling shows an increase in memory throughput, indicating an improvement in memory coalescence, that results in a decrease in

Reordering	**Duration [s]**	$\mathbf{Speedup_{serial}}$
RCM + Sort	1.84 ($\pm$0.04)	30.52
RCM + Sort with boundary variant	1.84 ($\pm$0.04)	30.54

(a) Total execution times

Reordering	**Duration [ms]**	**Compute Throughput [%]**	**Memory Throughput [%]**
RCM + Sort	0.90 ($\pm$0.003)	4.33 ($\pm$0.015)	63.93 ($\pm$0.214)
Variant	0.90 ($\pm$0.003)	4.32 ($\pm$0.014)	63.90 ($\pm$0.210)

(b) Flux kernel profiling

Reordering	**Duration [ms]**	**Compute Throughput [%]**	**Memory Throughput [%]**
RCM + Sort	0.29 ($\pm$0.004)	4.51 ($\pm$0.053)	86.68 ($\pm$0.487)
Variant	0.29 ($\pm$0.004)	4.52 ($\pm$0.056)	86.62 ($\pm$0.500)

(c) Update kernel profiling

Reordering	**Duration [ms]**	**Compute Throughput [%]**	**Memory Throughput [%]**
RCM + Sort	0.15 ($\pm$0.000)	9.42 ($\pm$0.025)	95.32 ($\pm$0.214)
Variant	0.15 ($\pm$0.000)	9.42 ($\pm$0.025)	95.29 ($\pm$0.183)

(d) Min reduction profiling

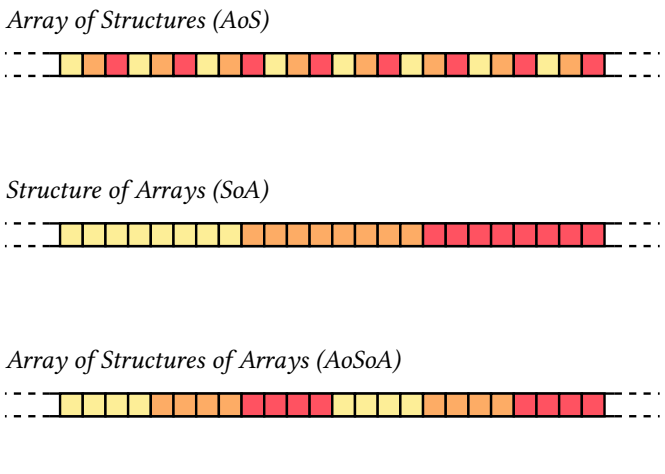
Table 6.7 – Timings of AdaptiveCpp implementation with reordered mesh (Toce *XL*)

the execution time. However, the boundary variant does not improve performance. We justify this by noting that warp divergence is not very pronounced in the PoC: boundary edges simply skip the mean water height computation, but as shown by kernel profiling, the limiting factor is memory access rather than computational load. Moreover, the reordering variant, while reducing warp divergence, also reduces spatial locality because the indices of the cells sharing a boundary interface are not direct neighbors except at corners. Finally, the number of boundary interfaces remains negligible compared to the number of inner interfaces in a large mesh as used in the test case.

6.3.2.2 Data layout

In the previous section, we saw how reordering the mesh can change the distribution of memory addresses accessed by threads and ultimately improve memory coalescence. Although threads access contiguous entries in the array of interfaces or cells, multiple memory transactions may still be required depending on the size of each C++ structure or class instance. Consider a simple example. Assume that cells are represented by a lightweight structure that stores only their hydraulic variables h, p, q as shown in Figure 6.31. When a warp executes an instruction that accesses global memory, it combines the memory accesses of all threads in the warp into as few memory transactions as possible. Global memory is accessed through 32, 64, or 128 byte transactions, and a double precision floating point number is 8 bytes. For this example, we assume a simple kernel that iterates over the cells to access hydraulic variables, as would happen during the update step of the finite volume scheme.

First, all threads in a warp try to retrieve the h values. If the cells are stored using an Array of Structures (AoS) layout, only the first 6 threads can coalesce their memory accesses into a single 128 byte transaction, since each (h, p, q) triplet takes 24 bytes. As a result, at least 6




```
struct Cell {
  double h, p, q;
};
Cell cells[N];
```

```
struct Cells {
  double h[N], p[N], q[N];
};
Cells cells;
```

```
struct BlockCell {
  double h[4], p[4], q[4];
};
BlockCell cells[N/4];
```

Figure 6.31 – Data layouts

transactions are needed to load all h values. Although this layout is intuitive for programmers, it leads to poor memory coalescence. On the other hand, since p and q are accessed in the next instruction, AoS provides good spatial locality because they were already loaded in the previous transaction. An alternative is the Structure of Arrays (SoA) layout, which has been shown to be more efficient in GPU implementations [90]. In this layout, all h values are stored in one large array, followed by all p values and all q values. This enables excellent memory coalescence. In our example, only two transactions are needed to load all h values in a warp: one for the first 16 threads and one for the last 16. With FP32, a single transaction is enough. However, SoA has weaker spatial locality, since accesses to h, p, and q are distant in memory across instructions. Still, it is generally more attractive because GPU caches are much smaller than those on CPUs.
A final approach combines the best of both layouts and is known as Array of Structures of Arrays (AoSoA). As the name suggests, it uses a tiled block organization that preserves coalescence while also providing good spatial locality. The block size can be tuned experimentally, and index-based access must be carefully managed to align with the tiled structure.

To assess the effectiveness of each memory layout, we preprocessed the original data containers (formerly classes with additional methods to set up geometry) into lightweight structures to better isolate the impact of memory layout. The structure for cells follows the pseudocode of Figure 6.31, while similar structures are used for the interfaces with fields for the flux and the indices of the left and right cells. We also added two accesses to the new variables p and q and added $2 \cdot 10^{-6}$ to both of them in the update kernel. The results of this modified version on the Toce *XL* case study for each data layout are reported in Table 6.8.

We observe that the lowest execution times are achieved with the Structure of Arrays layout. Furthermore, the larger the block size used in the AoSoA layout[12], the lower the execution times. This suggests that memory coalescence is more important than cache awareness in GPU algorithms. It is also worth noting that, regardless of the data layout, execution times are overall lower than in the previous sections. For this assessment, we removed all member variables and functions related to mesh geometry, keeping only the hydraulic variables. Reducing the stride between edge and cell instances helps speed up computation. This change should also be preferred in CPU implementations, as running the serial version with this layout yields a mean duration of 15.06 ± 0.187 seconds, resulting in a speedup of 3.74. In the previous version, the suboptimal memory layout was mitigated by reordering the mesh and leveraging cache behavior, but here, reordering provides no benefit regardless of the layout. Worse, it actually increases execution times. As before, we performed a per kernel analysis for further insight, presented in Table 6.9.

[12]Indicated in {...} in the table.

Layout	Duration [ms]
AoS	674 (±21)
AoS (reordered mesh)	749 (±22)
SoA	**516 (±12)**
SoA (reordered mesh)	536 (±14)
AoSoA{4}	639 (±20)
AoSoA{8}	626 (±13)
AoSoA{16}	599 (±14)
AoSoA{32}	600 (±14)
AoSoA{64}	593 (±14)
AoSoA{4} (reordered mesh)	649 (±14)
AoSoA{8} (reordered mesh)	654 (±17)
AoSoA{16} (reordered mesh)	622 (±16)
AoSoA{32} (reordered mesh)	623 (±15)
AoSoA{64} (reordered mesh)	629 (±16)

Table 6.8 – Total execution times of the PoC with the modified update kernel

The kernel timings align more closely with expectations. In the update kernel, the AoSoA and SoA layouts slightly improve memory throughput, but since throughput was already high in AoS, execution time differences are minor. In contrast, the flux kernel shows more variation. Here, mesh reordering significantly increases bandwidth usage across layouts, reducing mean execution times. Even without reordering, the AoSoA layout with block size 16 benefits from high memory throughput due to its tiled structure. However, compute throughput is also higher because of additional operations needed to access elements in the blocked layout, leading to suboptimal performance compared to SoA. In a more compute-intensive kernel, the indexing overhead may become negligible, making the layout more favorable. SoA generally performs best, especially with mesh reordering, achieving very low timings and showing a speedup of 14x compared to Table 6.6 profiling. Finally, the AoS layout underperforms relative to the other two in the reduction kernel.

6.3.2.3 Arithmetic precision

A final point is that GPUs were originally built for graphics, where floating point precision matters less than in scientific computing. As a result, more transistors were historically dedicated to FP32 units than FP64. FP32 also uses half the memory of FP64, effectively doubling bandwidth.

Layout	Duration [µs]	Compute Throughput [%]	Memory Throughput [%]
AoS	378.09 (±24.682)	10.21 (±0.653)	71.89 (±0.130)
AoS (reordered mesh)	271.06 (±18.221)	14.30 (±0.948)	95.81 (±0.184)
SoA	140.38 (±1.013)	27.72 (±0.204)	65.89 (±0.503)
SoA (reordered mesh)	96.73 (±0.238)	42.06 (±0.192)	94.36 (±0.141)
AoSoA{16}	195.02 (±2.531)	40.66 (±0.526)	80.12 (±0.125)
AoSoA{16} (reordered mesh)	162.82 (±0.288)	49.13 (±0.139)	95.23 (±0.083)

(a) Flux kernel

Framework	Duration [µs]	Compute Throughput [%]	Memory Throughput [%]
AoS	133.10 (±0.663)	28.98 (±0.146)	93.46 (±0.158)
AoS (reordered mesh)	133.41 (±1.945)	28.90 (±0.378)	93.45 (±0.163)
SoA	130.29 (±2.124)	30.76 (±0.386)	95.39 (±0.172)
SoA (reordered mesh)	130.24 (±2.048)	30.78 (±0.388)	95.39 (±0.211)
AoSoA{16}	128.33 (±2.529)	31.12 (±0.400)	95.34 (±0.177)
AoSoA{16} (reordered mesh)	128.09 (±1.877)	31.11 (±0.418)	95.34 (±0.183)

(b) Modified update kernel

Layout	Duration [µs]	Compute Throughput [%]	Memory Throughput [%]
AoS	112.10 (±2.624)	12.32 (±0.259)	94.44 (±0.230)
AoS (reordered mesh)	111.63 (±0.292)	12.37 (±0.034)	94.43 (±0.248)
SoA	44.21 (±0.544)	32.76 (±0.460)	83.67 (±0.711)
SoA (reordered mesh)	44.12 (±0.363)	32.75 (±0.243)	83.80 (±0.750)
AoSoA{16}	45.21 (±0.352)	31.93 (±0.263)	81.59 (±0.687)
AoSoA{16} (reordered mesh)	45.29 (±0.329)	31.95 (±0.293)	81.41 (±0.635)

(c) Min reduction

Table 6.9 – Per-kernel profiling

On high end datacenter hardware like the NVIDIA A100 (CC 8.0) or H100 (CC 9.0), double

Layout (precision)	Duration [s]
SoA (reordered mesh + floats)	260 ($\pm$16)

(a) Total execution time

Kernel	Duration [µs]	Compute Throughput [%]	Memory Throughput [%]
Flux	72.28 ($\pm$2.559)	20.21 ($\pm$0.645)	94.63 ($\pm$0.130)
Update	55.64 ($\pm$2.042)	20.68 ($\pm$0.688)	93.25 ($\pm$0.196)
Reduction	33.98 ($\pm$0.504)	8.55 ($\pm$0.127)	55.85 ($\pm$1.018)

(b) Per-kernel profilings

Table 6.10 – Profile of AdaptiveCpp implementation with FP32 precision

precision throughput is about half that of single precision, as shown in Table 6.4. On consumer GPUs designed for gaming or video editing, the gap is much larger. For example, on our NVIDIA RTX 4050 (CC 8.9), FP64 throughput is 64 times slower than FP32. Therefore, if a scientific application can tolerate reduced precision, it is always more efficient to use single precision.

To illustrate the impact of arithmetic precision on GPU performance, we implemented a version of the SYCL code using the Structure of Arrays layout with reordered mesh, replacing doubles with floats. As before, profiling metrics are reported in Table 6.10.(

Although this proof of concept is memory bound rather than compute bound, total execution time is reduced by about a factor of two. Kernel profiles show lower timings despite decreased compute throughput, indicating that FP32 resources remain underutilized. The update kernel is now over twice as fast, and the other two also show notable gains.

A single-precision version of the Watlab solver could be worth exploring, as the numerical schemes already introduce approximation errors and the hydraulic solver often relies on simplifying assumptions, especially in flood studies. A faster, less precise version could provide quick preliminary estimates before running more detailed and time-consuming simulations.

6.4 GPU port of Watlab

6.4.1 Program structure

The significant speedups achieved with the GPU port of the proof of concept motivate us to port the full Watlab version as well. Its architecture is similar to the parallel implementation described in Section 5.2. The main difference is that it uses two independent chips that run

separately. The GPU handles the computations of the three loops in Figure 2.5 as well as the minimum reduction, while the CPU writes output data to files. To do this, cell arrays are transferred back from the GPU when it is time to output. The caveat of using two independent hardware units is the need to transfer shared data between them. In a sense, this can be seen as an instance of task parallelism under the distributed memory paradigm. At each iteration, we also need to transfer the computed time step since the global logic is managed by the processor. The resulting program structure is shown in Figure 6.32.

6.4.2 Polymorphism challenges

When implementing Figure 6.32, the main challenge was handling polymorphism. In programming, polymorphism refers to the ability of objects to share a common interface while having different behavior depending on their type. Watlab's architecture relies heavily on this concept. For example, the solver includes several types of boundary conditions and therefore various types of interfaces. In the code, we define a base class `GenericInterface` that includes functions expected in every interface, typically a `Flux` method to compute hydraulic transfers and some helper functions used to calculate physical quantities common to all numerical schemes. Each derived class inherits from `GenericInterface`, overrides the `Flux` method, and still uses the other functions from the base class. When computing fluxes, we can call the `Flux` method on each interface directly, taking advantage of runtime polymorphism. A diagram showing interface polymorphism is provided in Figure 6.33. The `Flux` function is declared as virtual, meaning it is not implemented in the base class and must be defined in the derived classes, while all derived classes can access the helper function `Sigma`.

To ensure that the correct version of the `Flux` virtual method is called based on the type, C++ uses virtual method tables (vtables) and virtual table pointers (vpointers). At the class level, a vtable is created containing function pointers for each virtual function the class implements. This table is shared by all instances of the class. Each instance includes a hidden member, the vpointer, which is set at construction to point to the correct vtable. When a virtual function is called, the vpointer is dereferenced and the vtable it points to is indexed to execute the appropriate overriding function.

In typical GPU-enabled code, data is first initialized in host memory and then copied to device memory. This copy includes all class members and may duplicate member functions to generate code for execution on the device. However, the vpointer is also copied, and it may point to a vtable that resides in host memory. At runtime, this causes an illegal memory access when the device attempts to use a function pointer located in host memory. For this reason, virtual functions are not supported in the SYCL 2020 standard.

The Kokkos documentation [91], on the other hand, discourages the use of virtual functions but provides a workaround. This workaround is not portable across all backends and is

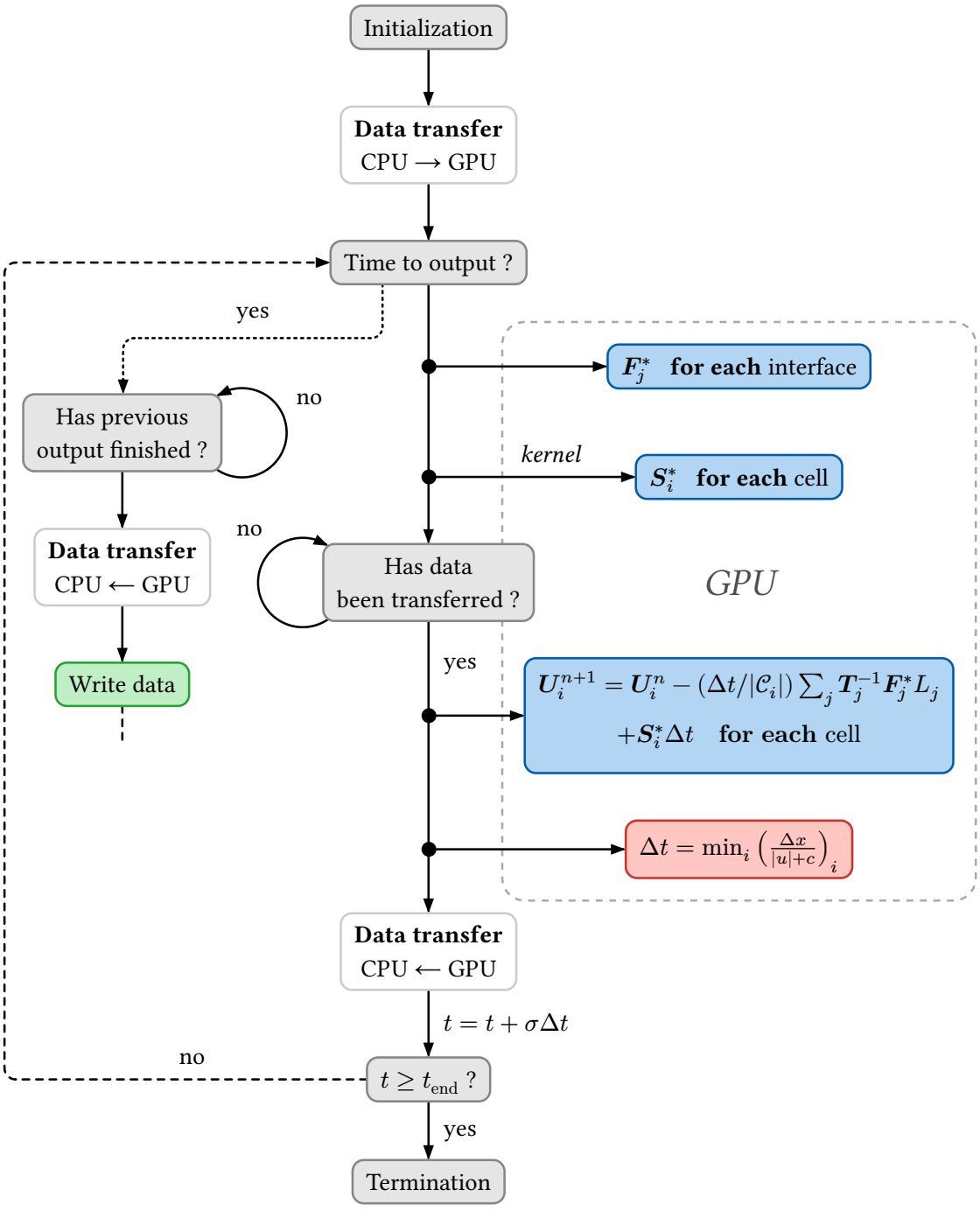


Figure 6.32 – Updated program structure of the parallel implementation

suboptimal in terms of performance. In a nutshell, it relies on the placement new technique, which allows using the new operator to construct an object in a preallocated memory buffer.

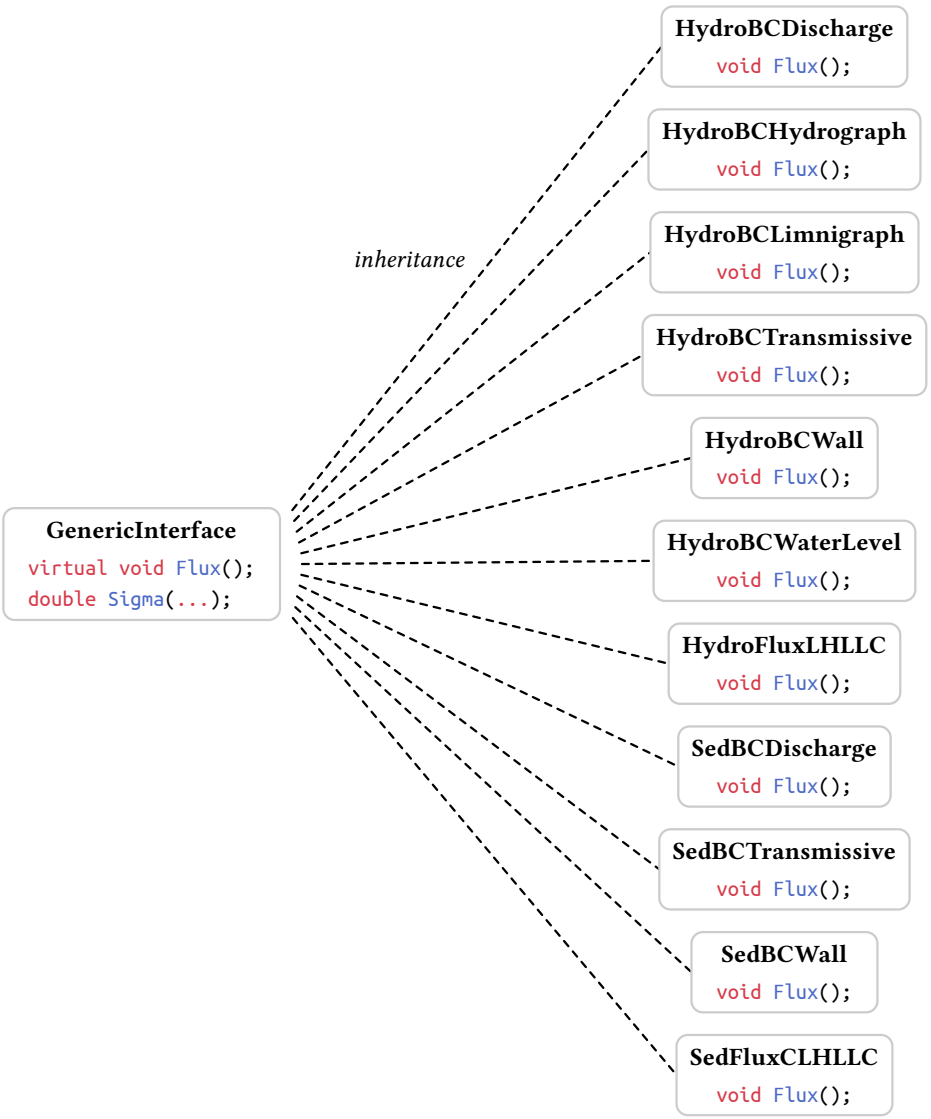


Figure 6.33 – An example of polymorphism used in Watlab

In this case, the buffer is allocated in the device memory, and the object instantiation is performed within kernels. Since objects are instantiated on the device, their vtables point to device code rather than host code. More information about the exact pseudocode can be found in the Kokkos documentation. This technique results in unproductive and hard to read code. We tried it using AdaptiveCpp, but it led to memory corruption errors related to alignment issues. For these reasons, this solution is not viable in the long term for Watlab development.

A simpler way to handle virtual functions is to remove them entirely. To do so, we reworked the Watlab source code to eliminate runtime polymorphism. This led to highly redundant

code. For instance, instead of having a single array of `GenericInterface` pointers on which we can directly call the Flux method, we now have separate arrays for `HydroFluxLHLLC` interfaces, `HydroBCWall` interfaces, `HydroBCTransmissive` interfaces, and so on. These arrays must be processed one after the other during flux computations. Separate memory allocations in the GPU memory are also required. Therefore, if someone wants to implement a new type of interface boundary, the code must be carefully adapted.

Polymorphism is also used in Watlab when sediment transport is considered. As shown in Figure 6.33, each interface type is adapted for this case and has an equivalent class prefixed by Sed that also derives from the GenericInterface class. The cell programming implementations also differ. We have a base class, `ComputationalCell`, with two derived subclasses, `CellSWE` and `CellSWE_Exner`, to distinguish between the two cases, as the numerical schemes and the number of hydraulic variables differ. In the original implementation, they were referenced through base class pointers, allowing for uniform handling in most cases. With the removal of polymorphism, additional if-else logic is needed to handle the different cases, which further reduces modularity.

However, this GPU implementation provides a clear way to analyze in depth the performance gains from a GPU port. It helps assess whether a full refactoring of the code architecture to eliminate runtime polymorphism and improve GPU compatibility is truly worthwhile.

6.4.3 Managing data transfers

We now detail the data migration mechanisms that distinguish the GPU capable implementations from the previous ones. First, SYCL provides several APIs for managing memory. The *Unified Shared Memory* (USM) model includes three allocation functions:

`sycl::malloc_host` allocates memory in host memory. Unlike a standard C++ `malloc` call, the memory is pinned, meaning it cannot be paged out by the operating system and always resides in RAM. It is also accessible by the device. However, such access occurs remotely through PCIe queries, so no data migration is involved and the data stays in host memory.

`sycl::malloc_device` allocates memory in device memory. It is not accessible by the host and requires explicit data copies to move data between host and device.

`sycl::malloc_shared` allocates memory accessible by both host and device. Migrations are handled at runtime by the operating system and drivers using a page fault mechanism. As a result, performance depends on their quality and is backend dependent, as noted in [92]. According to [87], this model is unstable on AMD backends and can cause random errors.

SYCL also includes the `sycl::buffer` and `sycl::accessor` model, which provides an abstract view of memory accessible by both host and device. These are implemented using USM under

the hood. Data migrations are resolved at runtime based on encountered data dependencies, which increases kernel launch latency, as shown by [87].

Based on these considerations, we excluded buffers and shared USM memory for handling data transfers. [87] shows that relying only on `malloc_host` is not viable because the data transfer time increases linearly with the number of device memory accesses to host memory. Instead, it is more efficient to load the data once at program initialization on the device and copy data back to the host only when needed, for example when output is required or to retrieve the minimum time step.

The remaining question is how to allocate memory on the host that must be sent to the device, as well as buffers that will receive cell arrays and reduced minimum values during execution. We can either use standard C++ heap allocations such as `new`, `malloc`, or `std::vector`, or rely on `malloc_host`. Again, [87] shows that non-pinned allocations are faster for host to device transfers if the number of copies is fewer than three. In our case, interface and cell arrays are transferred only once from CPU to GPU, so there is no need to use `malloc_host` for this data. This also allows us to use `std::vector`, which can be resized dynamically since the number of each interface type is unknown when parsing input files.

On the other hand, [87] demonstrates that using `malloc_host` for receiving buffers is preferable, as the CUDA driver cannot write directly to pageable memory and performs a temporary copy to pinned memory, adding overhead.

6.4.4 CPU-GPU synchronization

The event based approach of the SYCL programming model is convenient for handling synchronization between the host and device when data needs to be output. Each task submitted to the SYCL queue is associated with a `sycl::event` on which the host can wait for completion. We store a pointer to this event, which is instantiated when the copy from device to host is requested. We wait on the event if the previous transfer has not finished or if one of the working output threads has not completed its previous task. Similarly, we also wait on this event before launching the update kernel to avoid overwriting cell data that is still being transferred back to the host.

6.4.5 Benchmarks

We begin benchmarking our GPU implementation by measuring the mean execution time on the Toce simulation, as in the previous chapter. Results are shown in Table 6.11. We also computed the obtained speedups with respect to the serial and parallel implementations, using 14 OpenMP threads and a reordered mesh for the latter.

Version	**Mean Execution Time [s]**	$\mathbf{Speedup_{serial}}$	$\mathbf{Speedup_{parallel}}$
GPU	$1.30(\pm 0.03)$	12.43	1.58
GPU (reordering)	$1.28(\pm 0.03)$	12.63	1.61

Table 6.11 – Benchmarks of the GPU implementation on Toce case study

7
Perspectives

8
Conclusion

9
Acknowledgements

Bibliography

[1] A. Li, R. Maunder, B. Al-Hashimi, and L. Hanzo, "Implementation of a Fully-Parallel Turbo Decoder on a General-Purpose Graphics Processing Unit," *IEEE Access*, vol. 4, p. 1, 2016, doi: 10.1109/ACCESS.2016.2586309°.

[2] A. Brodtkorb, T. Hagen, and M. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 4–13, 2013, doi: 10.1016/j.jpdc.2012.04.003°.

[3] S. Soares Frazão, "Dam-break Induced Flows in Complex Topographies: Theoretical, Numerical and Experimental Approaches," 2007.

[4] Hydraulics Group, Université catholique de Louvain, "Watlab." [Online]. Available: https://sites.uclouvain.be/hydraulics-group/watlab/°

[5] T. P. P. A. (PyPA), "pip - The Python Package Installer." 2024. [Online]. Available: https://pip.pypa.io/°

[6] V. Eijkhout, R. van de Geijn, and E. Chow, *Introduction to High Performance Scientific Computing*. 2016. doi: 10.5281/zenodo.49897°.

[7] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems—A Cyber-Physical Systems Approach*, 2nd ed. MIT Press, 2016.

[8] OpenMP Architecture Review Board, "OpenMP Application Programming Interface." Accessed: Jun. 20, 2024. [Online]. Available: https://www.openmp.org/specifications/°

[9] M. P. I. Forum, *MPI: A Message-Passing Interface Standard, Version 4.0*. University of Tennessee, 2021.

[10] G. Karypis and V. Kumar, "METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices," 1997. Accessed: Jun. 20, 2024. [Online]. Available: https://hdl.handle.net/11299/215346°

[11] G. Colin de Verdière, "Introduction to GPGPU, a hardware and software background," *Comptes Rendus Mécanique*, vol. 339, no. 2, pp. 78–89, 2011, doi: https://doi.org/10.1016/j.crme.2010.11.003°.

[12] E. Gamba and J.-B. Macq, “Parallélisation d'un outil de simulation hydraulique numérique,” 2018.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming with CUDA,” *Queue*, vol. 6, pp. 40–53, 2008, doi: 10.1145/1401132.1401152°.

[14] A. M. D. (AMD), “Heterogeneous-Compute Interface for Portability (HIP).” [Online]. Available: https://rocm.docs.amd.com/°

[15] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science and Engg.*, vol. 12, no. 3, pp. 66–73, May 2010.

[16] T. Henriksen, “A Comparison of OpenCL, CUDA, and HIP as Compilation Targets for a Functional Array Language,” in *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Programming for Productivity and Performance*, in FProPer 2024. Milan, Italy: Association for Computing Machinery, 2024, pp. 1–9. doi: 10.1145/3677997.3678226°.

[17] M. Martineau, S. McIntosh-Smith, and W. Gaudin, “Assessing the performance portability of modern parallel programming models using TeaLeaf,” *Concurrency and Computation: Practice and Experience*, vol. 29, p. e4117, 2017, doi: 10.1002/cpe.4117°.

[18] S. Wienke, P. Springer, C. Terboven, and D. an Mey, “OpenACC — First Experiences with Real-World Applications,” in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870.

[19] P. Czarnul, J. Proficz, and K. Drypczewski, “Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems,” *Scientific Programming*, vol. 2020, no. 1, p. 4176794, 2020, doi: https://doi.org/10.1155/2020/4176794°.

[20] R. Usha, P. Pandey, and N. Mangala, “A Comprehensive Comparison and Analysis of OpenACC and OpenMP 4.5 for NVIDIA GPUs,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6. doi: 10.1109/HPEC43674.2020.9286203°.

[21] S. Zhang, R. Yuan, Y. Wu, and Y. Yi, “Parallel Computation of a Dam-Break Flow Model Using OpenACC Applications,” *Journal of Hydraulic Engineering*, vol. 143, no. 1, p. 4016070, 2017, doi: 10.1061/(ASCE)HY.1943-7900.0001225°.

[22] J. Wang, “Research on the Competitive Development and Prospects of Nvidia,” *Advances in Economics, Management and Political Sciences*, 2025, [Online]. Available: https://api.semanticscholar.org/CorpusID:275503189°

[23] G. S. Markomanolis *et al.*, "Evaluating GPU Programming Models for the LUMI Supercomputer," in *Supercomputing Frontiers*, D. K. Panda and M. Sullivan, Eds., Cham: Springer International Publishing, 2022, pp. 79–101.

[24] C. R. Trott *et al.*, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022, doi: 10.1109/TPDS.2021.3097283°.

[25] D. A. Beckingsale *et al.*, "RAJA: Portable Performance for Large-Scale Scientific Applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81. doi: 10.1109/P3HPC49587.2019.00012°.

[26] R. Reyes and V. Lomüller, "SYCL: Single-source C++ accelerator programming," in *International Conference on Parallel Computing*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6979118°

[27] E. Zenker *et al.*, "Alpaka - An Abstraction Library for Parallel Kernel Acceleration," IEEE Computer Society, May 2016. [Online]. Available: http://arxiv.org/abs/1602.08477°

[28] Khronos Group, "SYCL 2020 Specification." [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf°

[29] I. Corporation, "Data Parallel C++ (DPC++)." 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html°

[30] C. S. Ltd., "ComputeCPP." 2024. [Online]. Available: https://developer.codeplay.com/products/computecpp°

[31] R. Keryell and L.-Y. Yu, "Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation," in *Proceedings of the International Workshop on OpenCL*, in IWOCL '18. Oxford, United Kingdom: Association for Computing Machinery, 2018. doi: 10.1145/3204919.3204937°.

[32] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL," 2020, p. 1. doi: 10.1145/3388333.3388658°.

[33] Y. Ke, M. Agung, and H. Takizawa, "neoSYCL: a SYCL implementation for SX-Aurora TSUBASA," in *The International Conference on High Performance Computing in Asia-Pacific Region*, in HPCAsia '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 50–57. doi: 10.1145/3432261.3432268°.

[34] D. Arndt, D. Lebrun-Grandie, and C. Trott, "Experiences with implementing Kokkos' SYCL backend," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, in IWOCL '24. Chicago, IL, USA: Association for Computing Machinery, 2024. doi: 10.1145/3648115.3648118°.

[35] B. Homerding, A. Vargas, T. Scogland, R. Chen, M. Davis, and R. Hornung, "Enabling RAJA on Intel GPUs with SYCL," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, in IWOCL '24. Chicago, IL, USA: Association for Computing Machinery, 2024. doi: 10.1145/3648115.3648131°.

[36] A. Delis and E. Mathioudakis, "A finite volume method parallelization for the simulation of free surface shallow water flows," *Mathematics and Computers in Simulation*, vol. 79, no. 11, pp. 3339–3359, 2009, doi: https://doi.org/10.1016/j.matcom.2009.05.010°.

[37] P. Rao, "A parallel hydrodynamic model for shallow water equations," *Applied Mathematics and Computation*, vol. 150, no. 1, pp. 291–302, 2004, doi: https://doi.org/10.1016/S0096-3003(03)00228-5°.

[38] M. Castro, J. García-Rodríguez, J. González-Vida, and C. Parés, "A parallel 2d finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 19, pp. 2788–2815, 2006, doi: https://doi.org/10.1016/j.cma.2005.07.007°.

[39] J.-M. Hervouet, "A high resolution 2-D dam-break model using parallelization," *Hydrological Processes*, vol. 14, no. 13, pp. 2211–2230, 2000, doi: https://doi.org/10.1002/1099-1085(200009)14:13<2211::AID-HYP24>3.0.CO;2-8°.

[40] J. Pau and B. Sanders, "Performance of Parallel Implementations of an Explicit Finite-Volume Shallow-Water Model," *Journal of Computing in Civil Engineering - J COMPUT CIVIL ENG*, vol. 20, p. , 2006, doi: 10.1061/(ASCE)0887-3801(2006)20:2(99)°.

[41] I. Villanueva and N. Wright, "An efficient multi-processor solver for the 2D shallow water equations," 2006, p. .

[42] B. Sanders, J. Schubert, and R. Detwiler, "ParBreZo: A parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale," *Advances in Water Resources - ADV WATER RESOUR*, vol. 33, p. , 2010, doi: 10.1016/j.advwatres.2010.07.007°.

[43] A. Lacasta, P. García-Navarro, J. Burguete, and J. Murillo, "Preprocess static subdomain decomposition in practical cases of 2D unsteady hydraulic simulation," *Computers & Fluids*, vol. 80, pp. 225–232, 2013, doi: https://doi.org/10.1016/j.compfluid.2012.03.010°.

[44] J. Neal, T. Fewtrell, and M. Trigg, "Parallelisation of storage cell flood models using OpenMP," *Environmental Modelling & Software*, vol. 24, no. 7, pp. 872–877, 2009, doi: https://doi.org/10.1016/j.envsoft.2008.12.004°.

[45] S. Zhang, Z. Xia, R. Yuan, and X. Jiang, "Parallel computation of a dam-break flow model using OpenMP on a multi-core computer," *Journal of Hydrology*, vol. 512, pp. 126–133, 2014, doi: https://doi.org/10.1016/j.jhydrol.2014.02.035°.

[46] R. Lamb, A. Crossley, and S. T. Waller, "A fast 2D floodplain inundation model," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:61732991°

[47] Microsoft, "Microsoft DirectX 9.0 Software Development Kit." 2002.

[48] J. C. Neal, T. J. Fewtrell, P. D. Bates, and N. G. Wright, "A comparison of three parallelisation methods for 2D flood inundation models," *Environmental Modelling & Software*, vol. 25, no. 4, pp. 398–411, 2010, doi: https://doi.org/10.1016/j.envsoft.2009.11.007°.

[49] ClearSpeed, "CSX Processor Architecture." Bristol, UK, 2007. [Online]. Available: http://developer.clearspeed.com/resources/library/°

[50] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen, "Visual simulation of shallow-water waves," *Simulation Modelling Practice and Theory*, vol. 13, pp. 716–726, 2005, doi: 10.1016/j.simpat.2005.08.006°.

[51] M. Lastra, J. M. Mantas, C. Ureña, M. J. Castro, and J. A. García-Rodríguez, "Simulation of shallow-water systems using graphics processing units," *Mathematics and Computers in Simulation*, vol. 80, no. 3, pp. 598–618, 2009, doi: https://doi.org/10.1016/j.matcom.2009.09.012°.

[52] W.-Y. Liang *et al.*, "A GPU-Based Simulation of Tsunami Propagation and Inundation," 2009, pp. 593–603. doi: 10.1007/978-3-642-03095-6_56°.

[53] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[54] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3 and Later*, 8th ed. Addison-Wesley, 2013.

[55] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, and J. M. Gallardo, "GPU computing for shallow water flow simulation based on finite volume schemes," *Comptes Rendus Mécanique*, vol. 339, no. 2, pp. 165–184, 2011, doi: https://doi.org/10.1016/j.crme.2010.12.004°.

[56] A. Brodtkorb, T. Hagen, K.-A. Lie, and J. Natvig, "Simulation and visualization of the Saint-Venant system using GPUs," *Computing and Visualization in Science*, vol. 13, pp. 341–353, 2010, doi: 10.1007/s00791-010-0149-x°.

[57] M. de la Asunción, J. Mantas, and M. Castro, "Simulation of one-layer shallow water systems on multicore and CUDA architectures," *The Journal of Supercomputing*, vol. 58, pp. 206–214, 2011, doi: 10.1007/s11227-010-0406-2°.

[58] A. J. Kalyanapu, S. Shankar, E. R. Pardyjak, D. R. Judi, and S. J. Burian, "Assessment of GPU computational enhancement to a 2D flood model," *Environmental Modelling &*

Software, vol. 26, no. 8, pp. 1009–1016, 2011, doi: https://doi.org/10.1016/j.envsoft.2011.02.014°.

[59] M. de la Asunción, M. J. Castro, E. Fernández-Nieto, J. M. Mantas, S. O. Acosta, and J. M. González-Vida, "Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes," *Computers & Fluids*, vol. 80, pp. 441–452, 2013, doi: https://doi.org/10.1016/j.compfluid.2012.01.012°.

[60] R. Vacondio, A. Dal Palù, and P. Mignosa, "GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations," *Environmental Modelling & Software*, vol. 57, pp. 60–75, 2014, doi: https://doi.org/10.1016/j.envsoft.2014.02.003°.

[61] A. Lacasta, M. Morales-Hernández, J. Murillo, and P. García-Navarro, "An optimized GPU implementation of a 2D free surface simulation model on unstructured meshes," *Advances in Engineering Software*, vol. 78, pp. 1–15, 2014, doi: https://doi.org/10.1016/j.advengsoft.2014.08.007°.

[62] G. Petaccia, F. Leporati, and E. Torti, "OpenMP and CUDA simulations of Sella Zerbino Dam break on unstructured grids," *Computational Geosciences*, vol. 20, no. 5, pp. 1123–1132, Oct. 2016, doi: 10.1007/s10596-016-9580-5°.

[63] T. Carlotto, P. L. Borges Chaffe, C. Innocente dos Santos, and S. Lee, "SW2D-GPU: A two-dimensional shallow water model accelerated by GPGPU," *Environmental Modelling & Software*, vol. 145, p. 105205, 2021, doi: https://doi.org/10.1016/j.envsoft.2021.105205°.

[64] J. A. Herdman *et al.*, "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 465–471. doi: 10.1109/SC.Companion.2012.66°.

[65] L. S. Smith and Q. Liang, "Towards a generalised GPU/CPU shallow-flow modelling tool," *Computers & Fluids*, vol. 88, pp. 334–343, 2013, doi: https://doi.org/10.1016/j.compfluid.2013.09.018°.

[66] Q. Liu, Y. Qin, and G. Li, "Fast Simulation of Large-Scale Floods Based on GPU Parallel Computing," *Water*, vol. 10, no. 5, 2018, doi: 10.3390/w10050589°.

[67] X. Hu and L. Song, "Hydrodynamic modeling of flash flood in mountain watersheds based on high-performance GPU computing," *Natural Hazards*, vol. 91, p. , 2018, doi: 10.1007/s11069-017-3141-7°.

[68] M. Sætra and A. Brodtkorb, "Shallow Water Simulations on Multiple GPUs," 2010, pp. 56–66. doi: 10.1007/978-3-642-28145-7_6°.

[69] M. Morales-Hernández *et al.*, "TRITON: A Multi-GPU open source 2D hydrodynamic flood model," *Environmental Modelling & Software*, vol. 141, p. 105034, 2021, doi: https://doi.org/10.1016/j.envsoft.2021.105034°.

[70] A. H. Saleem and M. R. Norman, "Accelerated numerical modeling of shallow water flows with MPI, OpenACC, and GPUs," *Environmental Modelling & Software*, vol. 180, p. 106141, 2024, doi: https://doi.org/10.1016/j.envsoft.2024.106141°.

[71] 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*, vol. 16, pp. 977–1008, 2023, doi: 10.5194/gmd-16-977-2023°.

[72] M. Büttner, C. Alt, T. Kenter, H. Köstler, C. Plessl, and V. Aizinger, "Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, in PASC '24. Zurich, Switzerland: Association for Computing Machinery, 2024. doi: 10.1145/3659914.3659925°.

[73] G. Testa, D. Zuccalà, F. Alcrudo, J. Mulet, and S. Soares-Frazão, "Flash Flood Flow Experiment in a Simplified Urban District," *Journal of Hydraulic Research*, vol. 45, no. sup1, pp. 37–44, 2007, doi: 10.1080/00221686.2007.9521831°.

[74] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004, doi: 10.1145/989393.989401°.

[75] Y. Robert *et al.*, "Encyclopedia of Parallel Computing," 2011, pp. 2025–2029. doi: 10.1007/978-0-387-09766-4_59°.

[76] "IEEE Standard for Floating-Point Arithmetic." [Online]. Available: https://doi.org/10.1109/IEEESTD.2019.8766229°

[77] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, doi: 10.1145/1498765.1498785°.

[78] S. Oliveira and D. Stewart, "Writing Scientific Software: A Guide to Good Style," *Writing Scientific Software: A Guide for Good Style*, p. , 2006, doi: 10.1017/CBO9780511617973°.

[79] W. Wulf and S. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, p. , 1996.

[80] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation (PLDI), ACM, 2007, pp. 89–100. doi: 10.1145/1250734.1250746°.

[81] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, pp. 1309–1331, 2009, doi: 10.1002/nme.2579°.

[82] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[83] M. Véstias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6. doi: 10.1109/FPL.2014.6927483°.

[84] NVIDIA Corporation, "CUDA C++ Programming Guide." 2024.

[85] M. Breyer, A. Van Craen, and D. Pflüger, "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware," in *Proceedings of the 10th International Workshop on OpenCL*, in IWOCL '22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. doi: 10.1145/3529538.3529980°.

[86] I. Free Software Foundation, "GNU Compiler Collection (GCC)." 2024. [Online]. Available: https://gcc.gnu.org/°

[87] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, in IWOCL '24. Chicago, IL, USA: Association for Computing Machinery, 2024. doi: 10.1145/3648115.3648120°.

[88] M. Breyer, A. Van Craen, and D. Pflüger, "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware," in *Proceedings of the 10th International Workshop on OpenCL*, in IWOCL '22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. doi: 10.1145/3529538.3529980°.

[89] A. Alpay and V. Heuveline, "One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends," in *Proceedings of the 2023 International Workshop on OpenCL*, in IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. doi: 10.1145/3585341.3585351°.

[90] J. Xu *et al.*, "Generalized GPU Acceleration for Applications Employing Finite-Volume Methods," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 126–135. doi: 10.1109/CCGrid.2016.30°.

[91] C. Edwards, C. Trott, D. Sunderland, and others, “Kokkos Programming Guide, Chapter 14: Kokkos and Virtual Functions.” 2022.

[92] AdaptiveCpp Contributors, “Performance Documentation – AdaptiveCpp.” 2025.