Contents

Figures

1 Introduction

hook problématique exemple parlant de pq on doit accelerer

2 Governing equations

2.1 Shallow water equations

The Watlab hydraulic simulator solves the two-dimensional shallow water equations (SWE), which describe depth-averaged mass and momentum conservation 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

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

Here, h [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 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.

The finite volume formulation follows from conservation laws imposed on the control volumes $\Omega_i := \mathcal{C}_i \times \Delta t$ (Figure 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:

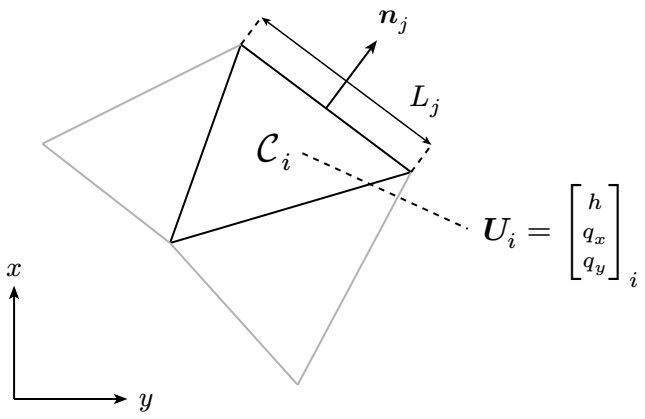


Figure 1 - Cell geometry

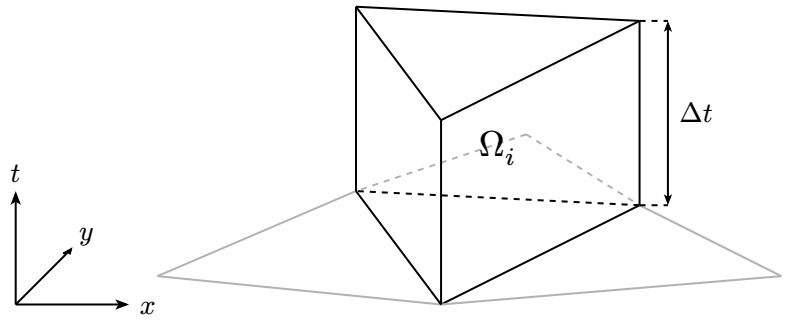


Figure 2 - Control volume

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

where $\boldsymbol{n}$ is the outward normal vector. To compute the integral, we sum the contributions from each face of the control volume.
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) 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 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 [1]. 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 [2].

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

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

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 [2] as a Python package and can be easily installed via `pip` [3]. 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 3. Also note that the C++ code may sometimes be referred as *Hydroflow*.

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 main loop.

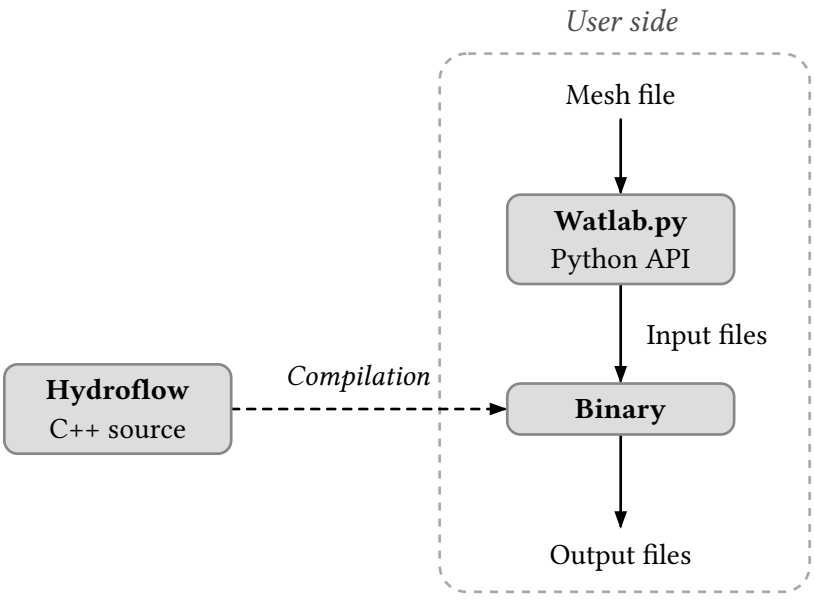


Figure 3 - High-level architecture

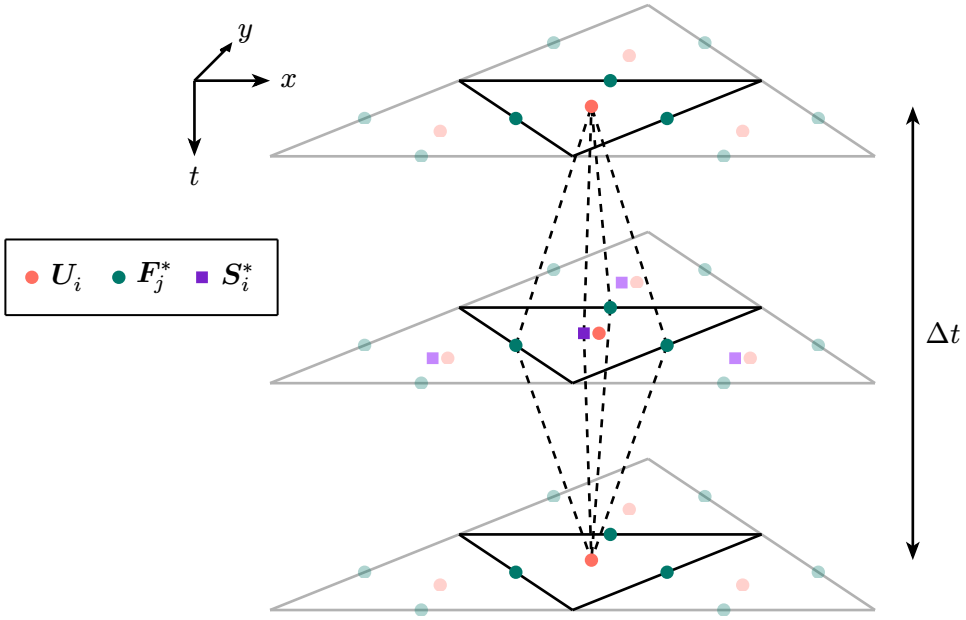


Figure 4 - Dependencies

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, 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. The update process involves computing numerical fluxes at each interface, calculating source terms for each cell, and updating hydraulic quantities based on the FV scheme (Equation 3). The computational dependencies between variables over time are illustrated in Figure 4. The next time step is then determined using the CFL condition (Equation 4). 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 5.

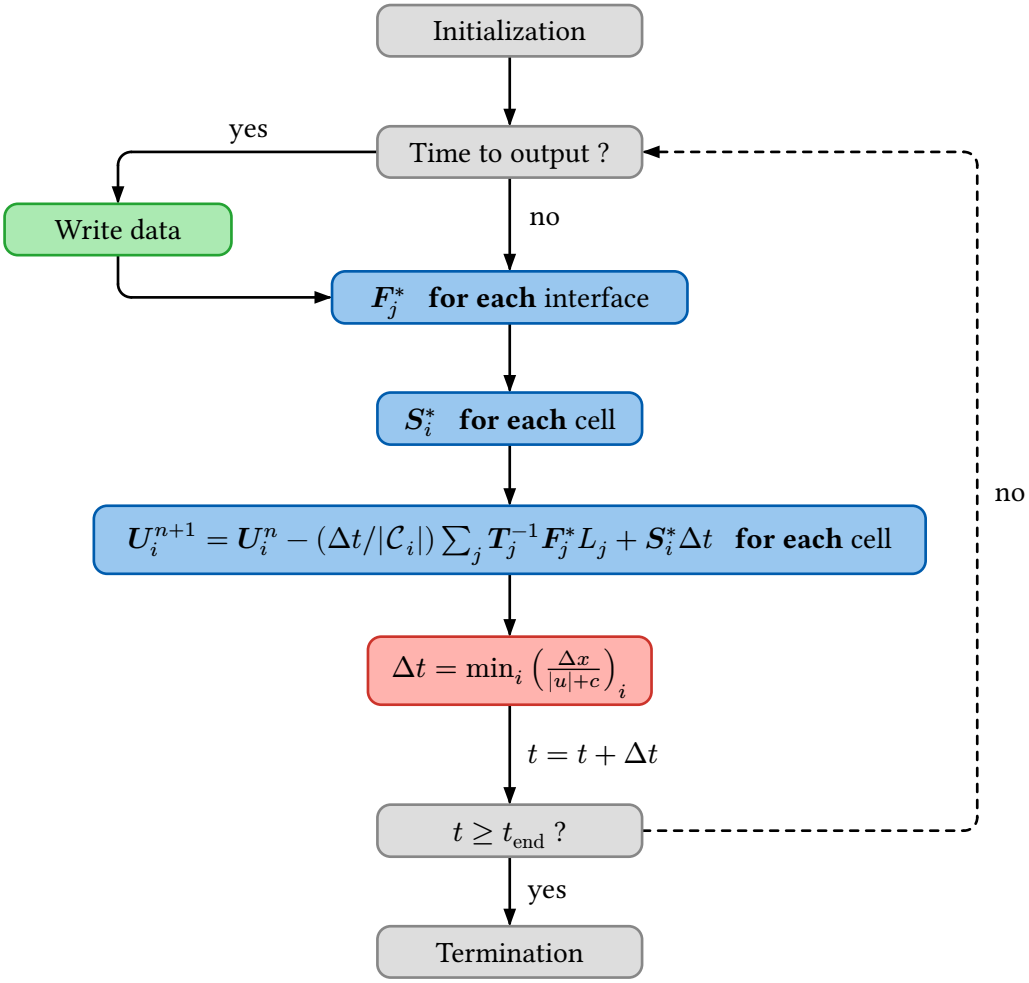


Figure 5 - Program structure

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—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 [4].

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—that is, 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 [5]. 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, which 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 during program execution.

C++ provides native support for threads and mutexes through the and libraries. The pseudocode below demonstrates an example of thread spawning. 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 [6] 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:

```
#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) [7] is the standard solution for implementing distributed memory parallel programming. This library interface enables both data and task parallelism, i.e., executing 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—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 shallow water equation (SWE) solvers. In such cases, authors often rely on external libraries like METIS [8] 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 (GPUs) 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 [9], 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 [10]. 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 [11], 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) [12], a proprietary framework developed by NVIDIA that supports only its GPUs. As a strategic response, AMD developed HIP [13], a C++ runtime API that allows developers to write code compilable for both AMD and NVIDIA GPUs. Its syntax is intentionally close to CUDA[1] to ease code conversion[2] from CUDA to HIP. Originally, OpenCL [14] was the primary

[1] Mainly, replacing `cuda` prefixes with `hip`.
[2] AMD also provides an automation tool called `hipify`.

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 [15] and lower productivity, requiring more code to achieve the same results [16].

Additionally, the OpenACC [17] programming standard uses directives for NVIDIA/AMD GPU offloading, making it the GPGPU equivalent of OpenMP. This approach is well-suited for inexperienced developers [18]. More recently, OpenMP introduced GPU offloading via compiler macros, but its performance remains slower than OpenACC [19], [20].

During the past decades, NVIDIA was the uncontested leader in the consumer GPU and HPC markets. Nowadays, AMD and Intel have become serious competitors [21]. For example, many new HPC infrastructures use non-NVIDIA hardware, such as the European LUMI supercomputer with AMD Instinct GPUs [22]. 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 [23], RAJA [24], SYCL [25], and Alpaka [26]. 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. The key advantage 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 FPGAs or DSPs without requiring major user code rewrites. Notably, Kokkos documentation indicates ongoing backend development.

The latest SYCL standard, SYCL 2020 [27], was developed by Khronos, the group behind OpenCL. Various implementations support SYCL 2020 specification to different extents, using different compilers and backends. Notable examples include DPC++ [28] (Intel), ComputeCpp [29] (Codeplay), triSYCL [30], AdaptiveCpp (formerly hipSYCL) [31], and neoSYCL [32], each targeting different hardware platforms. Note that SYCL can also serve as a backend for both Kokkos [33] and RAJA [34].

A visual summary of the approaches discussed above is presented in Figure 6.

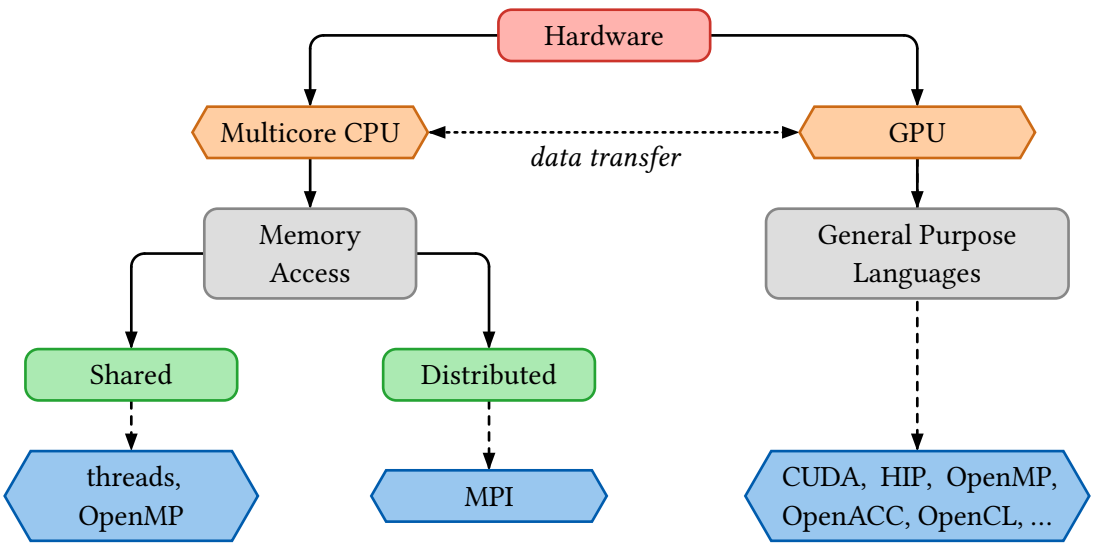


Figure 6 - Visual diagram of discussed approaches

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 [35], [36], [37], [38], [39], [40], [41], [42]. The main argument for using the paradigm is the ability to deal with gigantic domains consisting of many cells and interfaces that could not be stored on a single node due to the limited amount of memory available. Others still adopted a shared memory model with OpenMP [43], [44] 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. [45] used Microsoft DirectX 9 [46], while [47] employed ClearSpeed [48] accelerator cards, a now-defunct manufacturer. Other implementations [49], [50], [51] rely on Cg [52] and OpenGL [53], 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 motivated many authors [54], [55], [56], [57], [58], [59], [60], [61], [62], [63] to deploy their SWE solvers on GPUs because of the reduced development effort. Meanwhile, others [63], [64] chose OpenCL to improve the portability of their GPU implementation. Finally, OpenACC implementations [20], [63], [65], [66] 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 [67] by decomposing the domain for further speedup. This multi-GPU approach, implemented with CUDA+MPI [68] or OpenACC+MPI [69], 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. [70] developed a high-performance, portable shallow-water solver with multi-CPU and multi-GPU support using Kokkos. Similarly, Büttner et al. [71] 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 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.

Reference	API	Hardware	Scheme	Mesh	Cells	Max. speedup
ParBreZo [41]	MPI	Intel Xeon E5472	Roe	Unstruct.	374 414 Dry/Wet	**27×/48 cores**
Petaccia et al. [61]	OpenMP	Intel i7 3.4 GHz	Roe	Unstruct.	200 000 Dry/Wet	**2.35×/4 cores**
Castro et al. [54]	CUDA	NVIDIA GTX 480	Roe	Unstruct.	1 001 898 Wet	**152×/480 cores** (single precision) **41×/480 cores** (double precision)
G-Flood [59]	CUDA	NVIDIA GTX 580	HLLC	Struct.	3 141 592 Dry/Wet	**74×/512 cores** (single precision)
Lacasta et al. [60]	CUDA	NVIDIA Tesla C2070	Roe	Unstruct.	400 000 Dry/Wet	**59×/448 cores** (double precision)
Liu et al. [65]	OpenACC	NVIDIA Kepler GK110	MUSCL-Hancock + HLLC	Unstruct.	2 868 736 Dry/Wet	**31×/2496 cores**
TRITON [68]	CUDA + MPI	386 NVIDIA Volta V100	Augmented Roe	Struct.	68 000 000 Dry/Wet	**≤43176 ×/1966080 cores**[3]
Saleem and Norman [69]	OpenACC + MPI	8 NVIDIA Volta V100	MUSCL + HLLC	Unstruct.	2 062 372 Dry/Wet	**≤1989×/40960 cores**
SERGHEI [70]	Kokkos	NVIDIA RTX 3070	Roe	Struct.	515 262 Dry/Wet	**51×/5888 cores**

Table 1 - An overview of the most significant reported speedups compared to serial implementations is provided. 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 thread divergence (see Section REF). 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.

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

4 Case studies

4.1 Toce

4.2 Theux

4.3 Profiling

5 Implementations

5.1 CPU

5.1.1 What I did first ?

5.1.2 Going parallel

There are several steps that can be parallelized in the original program structure Figure 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:

- 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 wouldn't 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 to parallelize output writing and OpenMP to parallelize loops was the most efficient and easy-to-deploy way to implement the shared-memory model [11]. 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's 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.
However, I/O operations are time-consuming in C++, which would cause significant delays for computational threads. Instead, writer threads make a local copy of the data to output using a preallocated buffer, which is faster. 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.

Concerning OpenMP loops, OpenMP provides several scheduling policies that can be passed to the compiler using the schedule(...) keyword in the directive. to do:

- explain scheduling policies

- compare scheduling policies in depth
- scheduling allows for better load balance
- dynamic scheduling may be good for flux computation to differentiate cell and dry cells
- but for FV update no divergence thus maybe better static scheduling ?
- verify that guided is the best

Finally, we slightly modified the implementation of the minimum computation. The original approach 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.

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 7. 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.

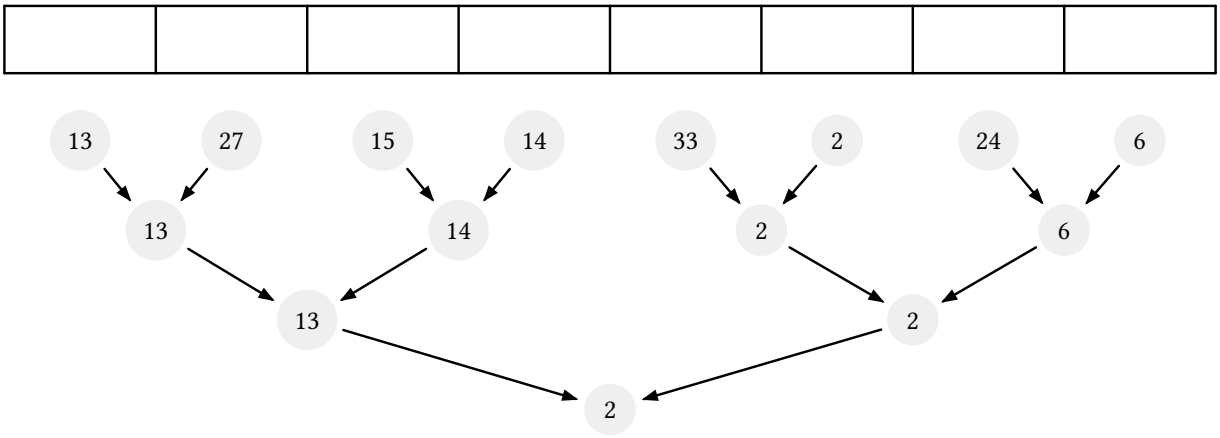


Figure 7 - Minimum reduction

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 8.

A finir

5.1.3 The precision issue

la parallelisation entraine du non detemrinisme du aux changement d'ordre des ops –> Eijkhout

a faire:

si je mets des 0 dans la cell based approach est ce toujours aussi lent ?

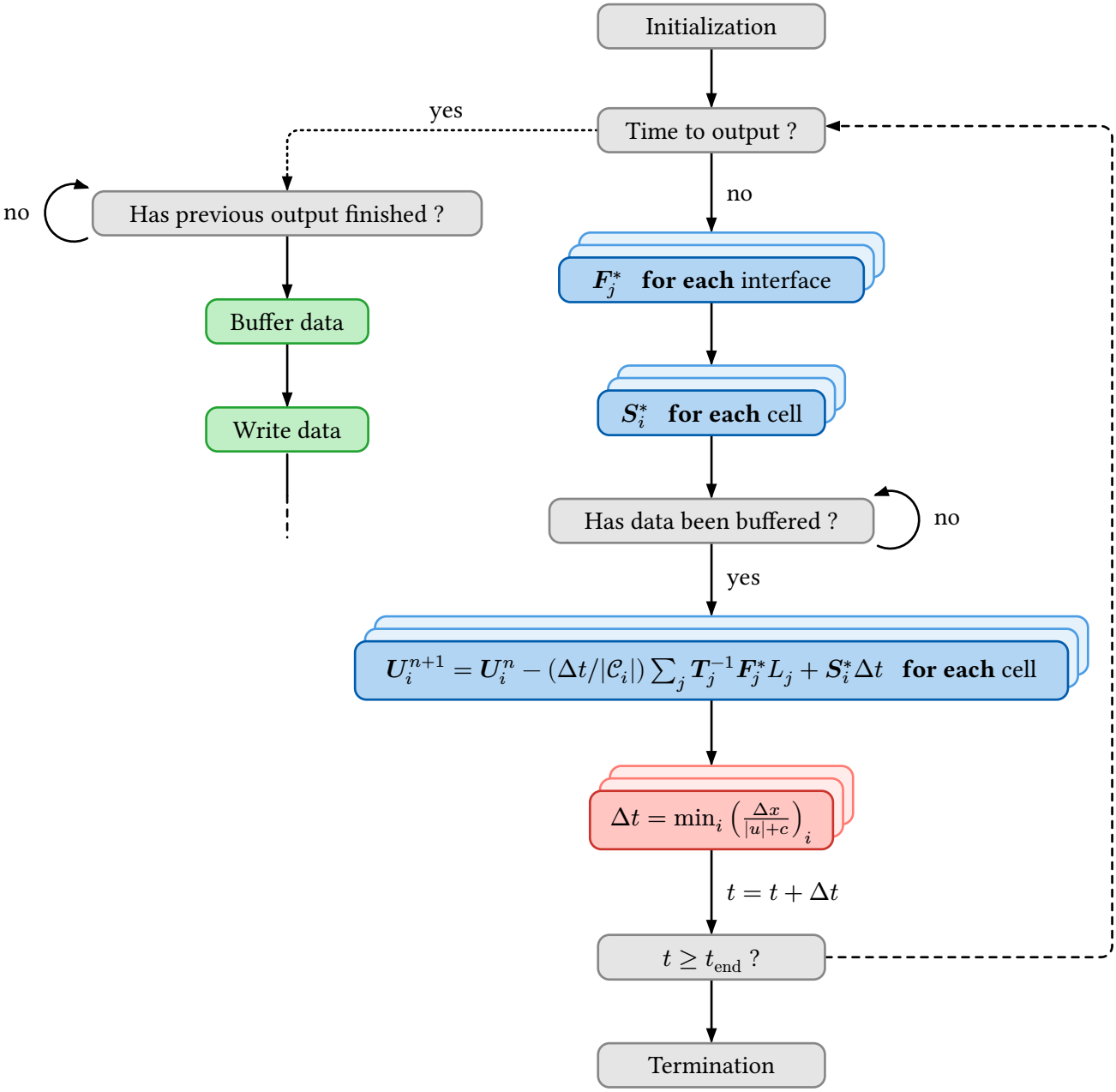


Figure 8 - Updated program structure of the parallel implementation

je vois un truc comme ca:

présenté problématique et les 2 graphes

cause

csq (tests + explosion erreur + non reproductibilité)

solutions ?

calculer indexs locaux pour toujours faire les ops dans le meme ordre.

- essai 1: cell based

–> pourquoi est ce plus lent ?

- essai 2: buffer de flux

5.1.4 Memory

It will be cool to prove that Watlab is memory bound and analyze data intensity.

From their inception, CPUs have continually become faster. Memory speeds have also increased, but not at the same pace as CPUs [72]. While a processor can perform several operations per clock cycle, accessing data from main memory can take tens of cycles [4]. This phenomenon is known as the memory wall [73].

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 [4].

To evaluate the number of cache hits and misses during the execution of Watlab, we used Cachegrind, a tool from the Valgrind framework [74], 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 function.

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, which slightly reduced execution times as shown in [REF].

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 interface,you can easily derive an adjacency matrix.

To illustrate, we consider a simple square mesh composed of 26 triangular elements (Figure 9a). The lower triangular part of the original symmetric adjacency matrix is shown in Figure 10a, 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, REF shows that these reordered meshes do not lead to reduced execution times. 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 blue (low indices) to rose (high indices). In the initial lower matrix, we observe a smooth gradient from left to right. This is linked to how GMSH [75], the mesh generator used to produce the mesh files feeding Watlab, numbers the interfaces. A closer look at Figure 9a reveals that it first numbers boundary interfaces in a counterclockwise 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 10a. 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 [76] (Figure 10b).
The obtained timings are reported in REF.

dire pourquoi j'ai mis le reordering dans python

tester si cest ap mieux de faire comme GMSH pour numeroter les edges i.e. d'abord numeroter les frontieres puis les inners

5.2 Case studies results

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 (REF). 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 Graphic Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs), which have emerged in the HPC field over the past decades [77].

We decided to develop a GPU implementation of Watlab to pursue the work initiated in the previous thesis [11] on the previous 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.

5.3 GPU

5.3.1 Hardware

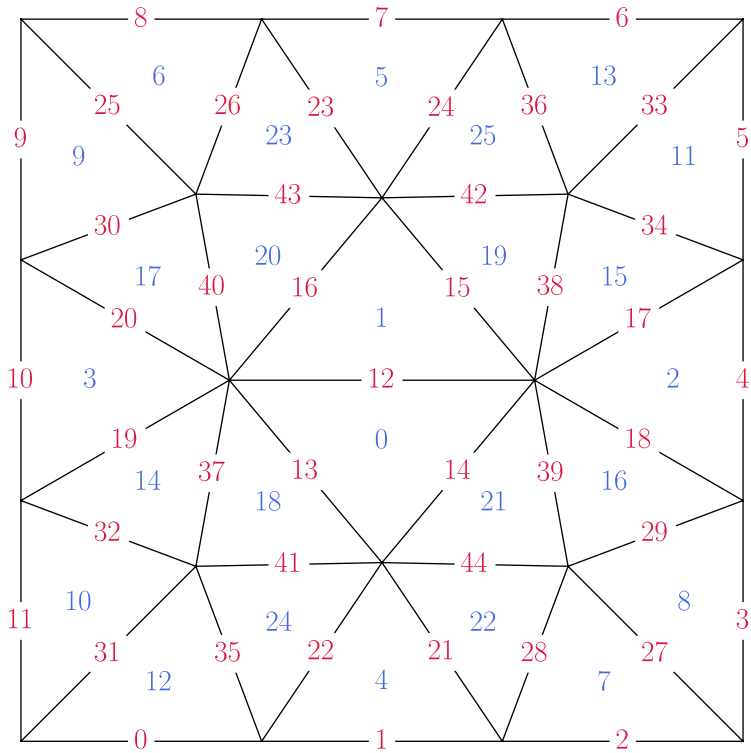
5.3.2 Weaknesses & strengths

5.3.2.1 Memory latencies

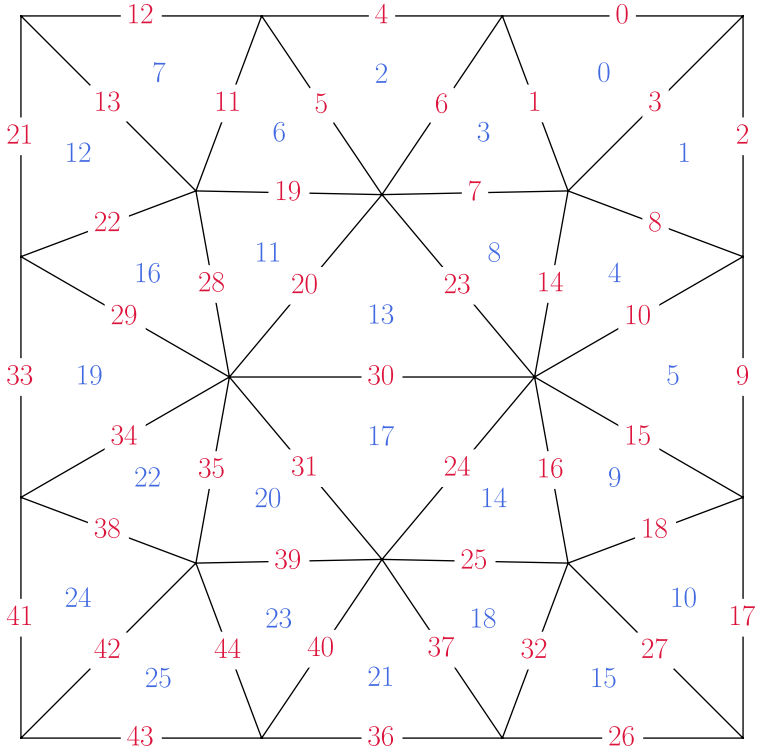
5.3.2.2 Occupancy

5.3.3 Proof of Concept

To choose the most appropriate solution for our project, it was necessary to first clearly define the objectives of the future GPU port. The most important are listed below.

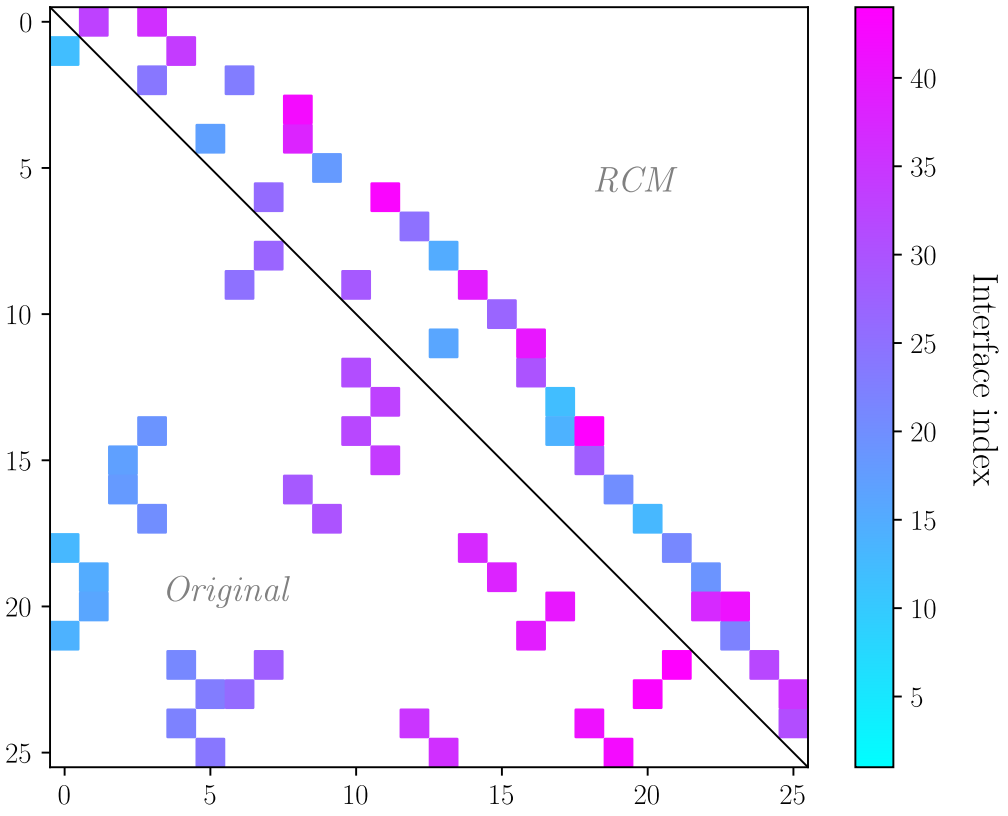


(a) Before reordering

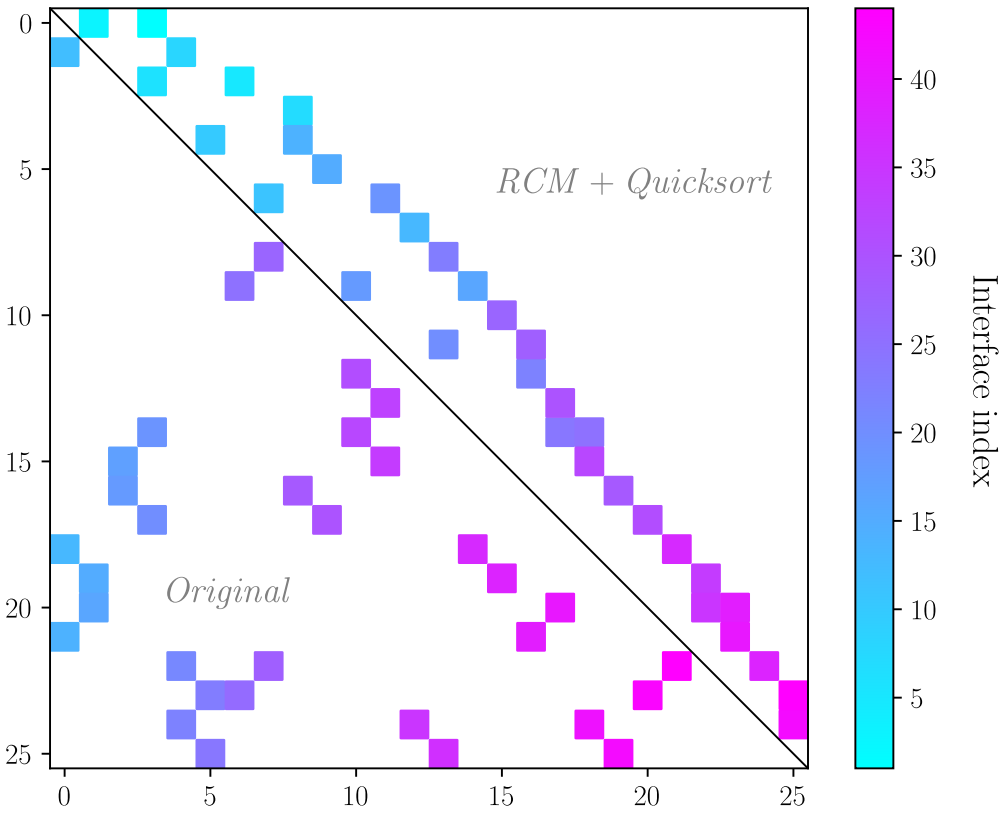


(b) After reordering

Figure 9 - 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 10 - Adjacency matrices related to the square mesh

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 [15], [16], [22], [78].

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 11.

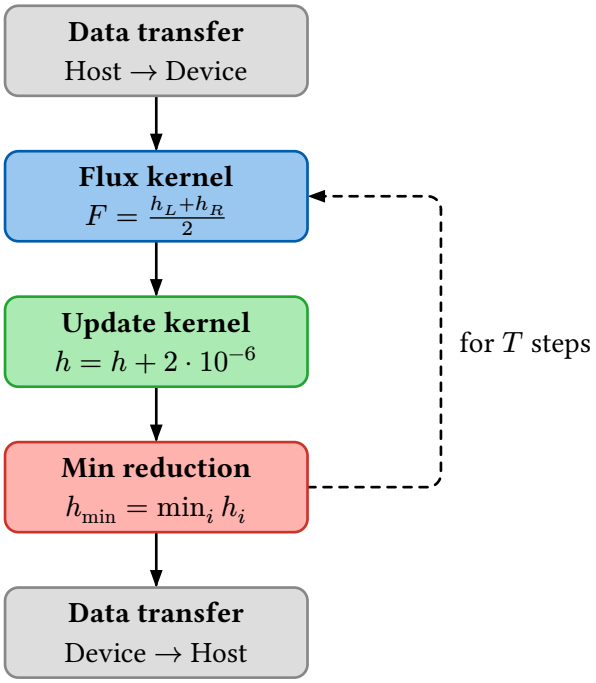


Figure 11 - *Proof of Concept* operating diagram

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 [79] 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 [71], [80], [81]. AdaptiveCpp is also the only 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 [15]. Finally, the key property that led us to choose AdaptiveCpp is its novel single-source, single-compiler-pass (SSCP) design [82].

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 NVIDIA architectures (Turing, Volta, Ampere, etc.) to produce a fat binary compatible with them. 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.[4]

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 device-independent. At runtime, the architecture is detected Just-in-Time, and the code is translated accordingly to meet driver expectations. Naturally, this approach introduces a runtime overhead, whose impact will be assessed in the next section.

5.3.3.1 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. 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.

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

5.3.4 Optimizations

5.3.4.1 RCM

5.3.4.2 Edge reordering

The benefits of reordering the edges are twofold. First, it increases the probability of accessing nearby or contiguous edges when summing the flux balance within a cell. On the other hand, two adjacent cells are likely to have nearby edges in memory, further increasing memory coalescence. To illustrate and evaluate the quality of edge reordering, we propose the following metrics;

- the maximum intra-cell distance between two indices of the edges that form its boundaries, that we denote by η_i;
- N = N ?

5.3.4.3 SoA ?

5.3.4.4 Streams boundary / inner edges

5.3.5 Case studies results

6 Perspectives

7 Conclusion

8 Acknowledgements

Bibliography

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

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

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

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

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

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

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

[8] 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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] 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

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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).

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2007, pp. 89–100. doi: 10.1145/1250734.1250746.

[75] 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.

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

[77] 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.

[78] 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.

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

[80] 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.

[81] 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.

[82] 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.