UCLouvain

École polytechnique de Louvain

Simulating the ocean on the GPU

Author: **Miguel De Le Court**
Supervisors: **Jonathan Lambrechts, Vincent Legat, Emmanuel Hanert**
Readers: **Laurent White, Jean-François Remacle**
Academic year 2021–2022
Master [120] in Mathematical Engineering

Acknowledgements

I would like to thank heartily my supervisors Jonathan Lambrechts, Vincent Legat and Emmanuel Hanert for their guidance throughout this year.

I am grateful to Vincent Legat for having sparked my interest in numerical methods and performance optimization, which lead me to chose this topic for a thesis.

I thank Jonathan Lambrechts for his help in getting me started with my thesis and his valuable answers to my questions throughout the year, especially relating to the math behind SLIM.

I am thankful Emmanuel Hanert for his continued guidance trough our weekly meetings, and for his push for me to start writing.

Thanks should also go to Ange Ishimwe for his availability and the time he took to help me to understand the code.

Finally, I would also like to thanks all members of the SLIM-dev team for the pleasant meetings on Thursday mornings and the regular advice they gave me.

Acronyms

Contents

1 Introduction

1.1 Importance of Ocean

Oceans cover approximately 70.8% of the Earth surface. They play a crucial role in the global climate and are home to numerous ecosystems, many of which have direct or indirect influence on humanity [1, 2]. Oceans play the role of a large heat buffer for the earth, acting to balance the excesses from rapidly rising temperatures. It is estimated that 90% of the excess heat induced by greenhouse gases has been absorbed by oceans, contributing to rising water temperatures for the past 40 years [3, 4]. They are also one of the largest carbon sinks on the planet. It is estimated that oceans produce between 50 and 80% of the oxygen in the atmosphere [5], most of it coming from phytoplankton. On top of the photosynthesis, the oceans regulate atmospheric CO_2 through dissolution, although this has the consequence of acidifying seawater.

These characteristics already make a case for the study of the ocean, but there are additional reasons to pay a special interest in coastal regions. While these only cover a small percentage of the surface area, they are home to a significant part of the marine biodiversity [6]. Coastal areas provide a natural habitat outside of the influence of marine currents. They are regions where the low water depth allows the sun to reach the ocean floor, where nutrients abide due to the deposition of dead organisms and feces. This results in rich areas that are also exempt of the dangers of deep ocean, making them in ideal nesting zones for many species of birds, turtles and fish. A significant part of humanity also lives close to the shore, and fishing is estimated to feed about $1/8^{\text{th}}$ of earth's population[7]. Coastal ecosystems further help communities during extreme events such as hurricanes, by slowing down and absorbing energy from the incoming waves.

Coastal areas are also very important for capturing CO_2 from the atmosphere, as they are especially good at storing carbon. The abundant nutrients present in the shallow waters can sustain a large number of plants, and marine flora is better suited to capturing carbon than forests and land-based ecosystems. Indeed, when organisms die a forest, most of the carbon is released back to the atmosphere, with only a small fraction being kept in the ground. Contrasting that, in the water, organisms don't fully break down, and a larger part of the carbon is trapped in the soil. Compared to forests, mangroves store 3 to 5 times the amount of carbon per surface area[8]. Coastal ecosystems further help by filtering carbon-rich sediments coming from other ecosystems.

1.2 Threats due to human activity and climate change

While coastal zones and wetlands greatly contribute against climate change, they are also under the threat of climate change, as they generally constitute vulnerable areas. The increasing acidity and temperature of the water can affect sensitive species, reducing their habitats. The seal level rise is another source of stress for coastal environments, as it pushes the shore towards the land. This can be problematic for mangroves, if for example, the land is too steep to allow the environment to grow inland. The speed at which the seas rise can also be problematic, as some species might not have the time to adapt. More direct human activities also pose a risk for coastal biology. The continued urbanisation of coastal areas often negatively impacts marine ecosystems. Furthermore, urbanisation inevitably leads to more marine, in the form of fertilizers, emissions from cars and ships, and other chemicals. Over-fishing is another threat to coastal zones,

disrupting and further weakening these ecosystems and by extension threatening the very people that depend on it.

1.3 The need for coastal models

With the global temperature rising and the ongoing climate change, extreme events such as storms and hurricanes are predicted to become more and more frequent. Coupled with disruptions made to coastal areas, we can expect significant[1] changes to these ecosystems. Given the huge biodiversity that is present on these areas and that directly depends on them, understanding and being able to predict the evolution of coastal ecosystems has become a necessity to evaluate and mitigate the risks to both the ecosystems and the people that depend on it.

The sheer size of both oceans and coastal regions makes studying them a complex task, requiring a lot of data which is hardly available. Scientists have to work with only a limited set of measurements, which are often not enough for the kind studies being done. Things such as the transport of a pollutant, the movement of coral larvae, and the transport of sediments in river plumes require a detailed knowledge of the flow at any point in time. In this context, numerical models offer a great tool to "bridge the gaps between the measurements". The equations governing the thermodynamics and hydrodynamics of the ocean[9] have been known for a long time, and the advances in computers made over the past 30 years have made numerical simulations of the ocean a necessary reality.

Traditional numerical models are not perfect though. The Navier-Stokes equations governing the oceans flows and the thermodynamic equations cannot be discretized directly, as small scale features cannot be resolved with current technologies. Most numerical models therefore rely on parameterizations for unresolved processes. The resulting quality of a model can greatly depend on such parameterization. Many ocean models also make further simplifications in the fluid dynamics computation, using the Boussinesq and hydrostatic approximations to accelerate the computations.

On top of these shortcomings, coastal regions have the additional feature of being highly irregular. Coastlines are fractal-shaped, meaning that their shapes feature a wide variety of scales that directly influence our circulation patterns. This makes using traditional models based on structured grids cumbersome. One obvious solution is to have a very fine grid, but this comes at significant increase in the computational cost, since the resolution is increased everywhere. Some models also support nested grids, allowing some flexibility in the spatial resolution. The downfall to this approach is that nested grids tend to introduce spurious diffusion. On top of that, it has been show that representing a smooth coastline using a "staircase-like" structure can also reduce the quality of the simulation[10]. This has lead over recent years to the creation of a variety unstructured models [11, 12, 13, 14]. Unstructured models mostly solve the geometry problem around coastlines, but introduce some new challenges, one of which is an added layer of complexity on top of already complicated models, making them expensive in terms of computing power.

1.4 Current limits in computer hardware

Numerical models have been a great tool for, among other fields, ocean modelling. The exponential growth in computing power has allowed the amount of details in simulations

[1] Meaning "mesurable", not large.

and the accuracy of models to grow tremendously over time. Each new generation of processors enabled finer, more accurate, and more realistic simulations. The traditional approach to increase performance was to reduce the transistors size, and increase the transistor count and frequency. Until the early 2000s, that trend worked remarkably well (Fig. 1). However, this exponential growth in performance and frequency also came with an exponential growth in power density. This inevitably lead to a physical barrier. At some point, the power density becomes too high[15], and it is not possible to continue scaling up the performance of a single core processor. The quest for power did however

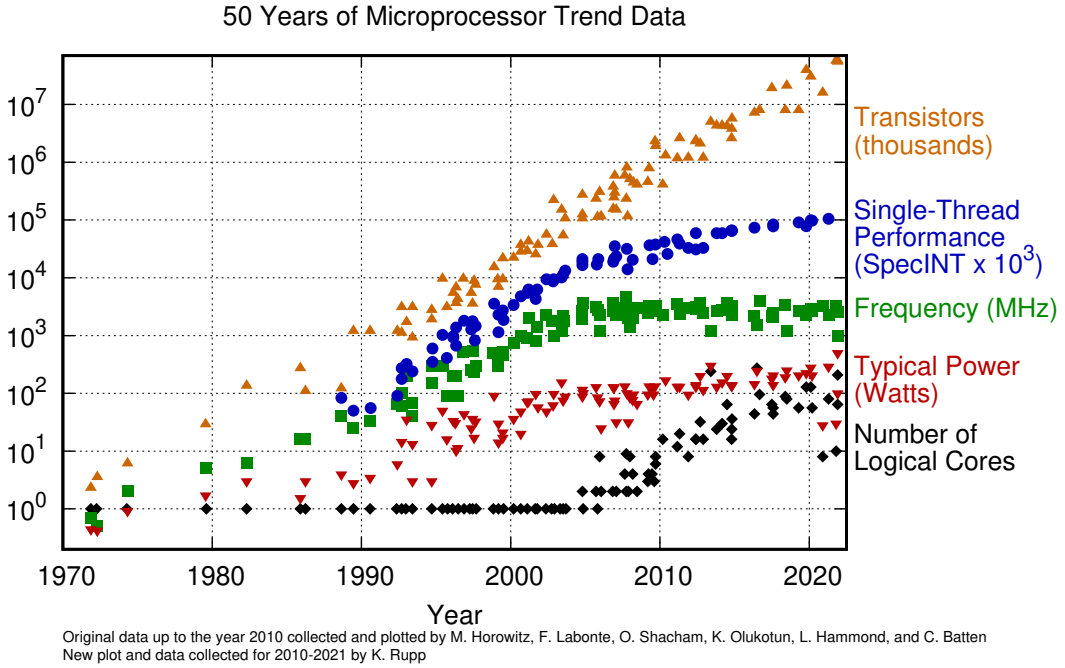


Figure 1: 50 Years of Microprocessor Trend Data

not stop, the industry started using increasingly parallel processors. This change in paradigm began in 2001[16] when IBM launched the first multi-core processor. A multi-core processor is a chip with two or more computation cores, that can work on different tasks at the same time. Since then, multi-core processors have become the norm, with most modern consumer Central Processing Units (CPUs) having a least 4 cores, with 6, 8 and more cores becoming an increasingly popular choice. In the data-center, server and High Performance Computing (HPC) market, typical core counts range from 16 to 64, with higher core count CPUs scheduled to be released in the coming months or years.

Multi-core processors, however, are limited in how fast they can keep improving. Adding cores also adds a lot of complexity, which limits the raw performance of multi-core processors. On top of that, as core counts increase, the memory bandwidth required to keep the CPU active with enough data also increases and starts to become the bottleneck. This further limits the effectiveness of adding cores. Multi-core CPUs, however, are not where the limit is today in terms of performance. One key limitation of traditional CPUs, is that they are fundamentally designed for sequential operations. This means that they try to reduce latency as much as possible, and predict what the future operations are going to be. CPUs are built by engraving nano-circuits on a piece of silicon called a *die* or

a *chip* (Figure 2). Aside from the cores, which are the parts of die that actually execute the operations, CPUs have dedicated mechanisms to reduce the latency of operations. The most two common are the caches and the branch predictor. A cache is small amount

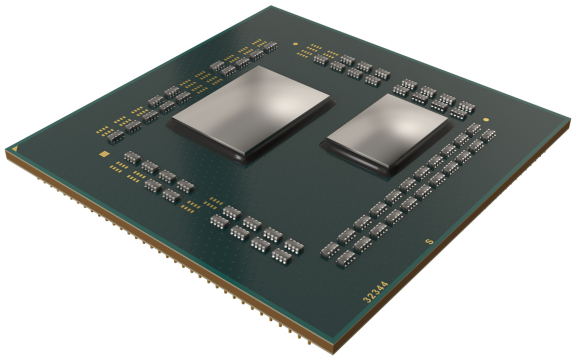


Figure 2: 3D render of an AMD CPU with two dies. The small die contains the cores and the caches, and the large die handles the communications with system memory and other buses.

of very fast memory located on the chip. Its role is to keep recently accessed data close to the core in case it is needed later. The branch predictor is a piece of logic that tries as best as possible to predict the outcome of "if" statements. This allows the cores to continue to run before the result of the "if" statement is available. If it turns out that the predictor was correct, the next instructions have already been executed. If not, whatever result was computed incorrectly is discarded and the execution continues.

In CPUs, a significant amount of the die space is dedicated to different levels of cache and the branch predictor. Multi-core CPUs behave like a collection of very good sequential units, which is needed in some scenarios, but not in fundamentally parallel computations, where the raw throughput is what matters most.

1.5 Modern massively parallel architectures

One of these areas where the raw performance throughput is very important is in rendering 3D objects to a screen. Rendering is a task that is fundamentally parallel, where each pixel has to be rendered with the same set of operations being performed. This lead NVIDIA to release in 1999 the first ever Graphical Processing Unit (GPU) to accelerate rendering tasks. GPUs were initially designed to perform a fixed sequence of operations used for rendering video game frames. Over time, they became increasingly flexible, evolving into general highly parallel devices. Compared to multi-core CPUs, GPUs were excellent at tasks involving a lot of floating point computations where the same operations are applied to different pieces of data. This lead NVIDIA to release their proprietary Compute Unified Device Architecture (CUDA) platform in 2007 [17]. Later, in 2009, the Khronos Group released the Open Computing Language (OpenCL) framework, as an open source alternative to CUDA[18].

CUDA and OpenCL allowed programmers to use the massive power of GPUs for general processing (GPGPU). One of the areas where GPU computing enables massive gains is in numerical simulations and HPC. As early as 2012, Chen et al. [19] found a single GPU to be worth around 10 CPUs in hydrodynamics simulations. Since then,

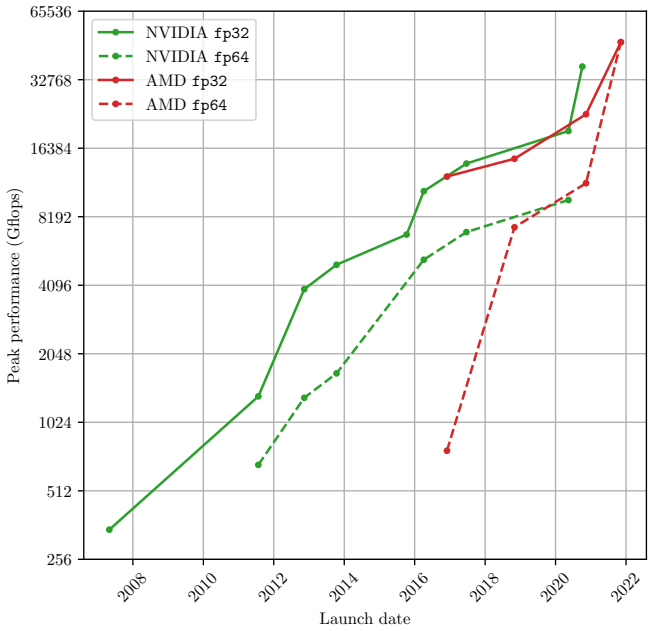


Figure 3: Performance of datacenter GPUs from AMD and NVIDIA. The data includes Telsa series GPUs from NVIDIA and Instinct GPUs from AMD

the advantage of GPUs has only increased. GPUs have experienced and maintained an exponential growth in computing power, with their peak performance roughly doubling every two years (Fig. 3). Today, high end workstation GPUs such as the NVIDIA A100 or the AMD Instinct MI200 are on the order of 30 to 60 times faster than high end CPUs for massively parallel workloads. The advantage is even bigger in some tasks that have dedicated hardware units, such as matrix multiplication and artificial intelligence[20]. The benefits of using GPUs are also not limited to computing power. Compared to traditional processors, GPUs are much more energy efficient, with the cost of a floating point operation being on the order of 10×10^{-11} Joules for GPUs versus $0.5\text{–}1 \times 10^{-9}$ Joules for CPUs. This is particularly relevant in an era where energy consumption and climate change are global issues.

Unfortunately and unsurprisingly, GPUs come with their own set of restrictions, and not all workloads can benefit from their enormous computing power. The basic idea behind a GPU is to trade memory latency for memory bandwidth and computing power. The latency introduced is then hidden by the massively parallel nature of the workload. GPU programs are made of thousands or millions of small independent tasks. To hide the latency, when the data is unavailable for one task, the GPU will switch to another for which the data is available, and come back later to the first task. This means that if a function or program cannot be split into thousands of independent tasks, there will be very little gains in using a GPU. By design, GPUs have a much larger memory bandwidth than CPUs, which is necessary to feed the large processing power of the GPU. Accessing that bandwidth, however requires regular and predictable access patterns that do not map with all applications.

On top of these restrictions, the programming model is also very different compared to a single core processor, or even a multi core processor. Rewriting existing code to

efficiently use the GPU is not an easy task, and a significant effort is needed to do so. This is likely one of the main reasons why most programs are still based on the CPU, even in the scientific community.

1.6 State of the art in Ocean modelling

Traditional ocean models are based on finite difference methods on regular grids in 3 dimensions. Examples of these models include ROMS [21], NEMO-AGRIF [22], POM... The use of finite differences makes these models relatively easy to implement, and very efficient numerically[23]. The memory access patterns are all regular, and easily predictable. On the other hand, this approach lacks flexibility when dealing with the complex coastal topography and bathymetry. It makes the study of complex topographies and localised processes cumbersome, since regular meshes cannot be locally adapted. This can be partially mitigated with the use of nested grids and meshes [24], but the inherent "staircase-like" of the boundaries remains a significant source of error[10].

Theses limitations, coupled with the growing need for coastal models allowing a variety of different scales, lead to the emergence of horizontally unstructured models[9]. The meshes used by these models are constructed by creating a 2D unstructured mesh of the ocean, and then extruding the mesh in the vertical direction in a number of layers. The resulting meshes are thus structured in the vertical direction, and unstructured horizontally. Unstructured meshes are generally preferred for the study of coastal regions, due to their natural ability to represent a complex topography.

Numerical models relying on unstructured meshes can generally be labelled in three categories. The first and most common category is finite volume models. Examples include FVCOM[25], MITgcm[26], FESOM2 and MPAS[14]. Finite volume methods are well adapted for advection-dominated processes, such as the ocean dynamics, but they are inherently first-order accurate. It is possible to build higher order numerical schemes, but those require information from multiple neighbours at each time step, which can complicate the parallelisation of such models.

Another class of unstructured models one based on finite elements. Oceanic models using finite elements include ADCIRC [27], SELFE [13] and SCHISM [12]. Although optimal for elliptic problems, the finite elements method is poorly suited to advection-dominated equation and it requires the use of stabilisation terms. Another problem of finite element methods, is that they require to solve a linear system that couples all degrees of freedom, which comes from the non-diagonal mass matrix of the problem. This makes parallelisation of the models very challenging, and greatly reduces the theoretical potential benefits of a parallel implementation, following to Amdahl's law.

The third approach is to use discontinuous finite elements, which can be thought of as a middle ground between finite elements and finite volumes. Two models that use discontinuous finite elements are Thetis [11] and SLIM [28]. Discontinuous finite elements methods are, like finite volumes, locally conservative, they don't require solving a global linear system. Another benefit over finite elements and finite volumes, is their low numerical diffusion [29, 30, 31]. This makes them particularly adapted for ocean modelling, where advection processes are largely dominant and where the diffusion is very low. Indeed, it has been shown that in general, the numerical diffusion largely dominates the physical mixing [32, 33]. Two other strengths of discontinuous finite element methods, is their natural extension to higher orders, and their relatively easy parallelisation.

Among the models cited above, a handful of then has been partly or completely

ported to GPU. A GPU implementation ROMS [34] showed it was possible to match the performance of distributed systems using a GPU on a single machine. This comparison was, however, not done with multiple machines, as the MPI component of ROMS has not been ported. POM, on the other hand, has benefited from a complete distributed GPU implementation called gpuPOM[35]. gpuPOM allowed the efficient use of multiple GPUs simultaneously for a fraction of the energy and financial costs compared to POM. Recently, tools such as JAX [36] have allowed for the fast development of higher language libraries. One of them, Veros [37] is an ocean model written in python that can use multiple GPUs and distributed parallelism.

To our knowledge at the time of writing, only two of the unstructured ocean models benefited from a GPU implementation. One of these is GPU-FVCOM [38], although it does not implement distributed parallelism. The other is MPAS (Model for Prediction Across Scales), which has been partly ported to GPU [39]. Since MPAS is an initiative from the US department of energy, we can assume that it will end up being ported to GPU in order to run on the incoming supercomputers Frontier and El Capitan, that will mostly be powered by GPUs.

1.7 Objectives and scope of this work

Over the last two decades, the SLIM team at UCLouvain has been developing a non-structured ocean model. SLIM has been specially designed for coastal regions, where its unstructured nature becomes an essential feature. SLIM can represent both fine coastal topographical details, as well as the large-scale ocean circulation. As mentioned previously, SLIM is based on the discontinuous finite element method, making it well suited for ocean modelling due to its low numerical diffusivity. The discontinuous finite elements method also makes it well suited for massive parallelism. The mesh, being 2D or 3D, contains thousands of triangles or prisms, which can all be processed independently in parallel.

The goal of this thesis is to upgrade the SLIM ocean model to be able to run on multiple GPUs across many nodes. Doing so will SLIM to fully scale across modern and upcoming supercomputers that get their processing power through massive parallelism. It will unlock previously unreachable levels of precision in ocean modelling that will be increasingly demanded in light of the current climate crisis and the role of coastal ecosystems.

SLIM is a model with 2D and 3D modules that are detailed below. Since the 3D module uses the 2D module internally, the latter must be completed before the former. This thesis is divided in 4 steps:

1. SLIM 2D : Implementing the 2D model to run efficiently on one GPU.

2. 2D + MPI : Adding multi-GPU support across multiple nodes in an efficient fashion. Since the GPU code is much faster than the CPU code, the communication step become very significant if not the dominant bottleneck, and some effort is needed to optimize it.

3. SLIM 3D : Implementing the 3D model on the GPU.

4. 3D + MPI : If enough time is left, we will be adding multi-GPU support to the 3D kernels. Since the 3D functions are significantly slower than the 2D functions,

optimising the communication of the 3D information across GPUs should yield very little benefits compared to in 2D.

2 The SLIM ocean model

Notation and conventions

For the entirety of this document, scalar variables will be written in regular math fonts while vector components will use bold fonts. When writing equations involving the gradient of a vector field, the resulting tensor will be understood as

$$[\nabla \boldsymbol{u}]_{ij} = \frac{\partial \boldsymbol{u}_j}{\partial x_i} = \partial_i \boldsymbol{u}_j$$

ie. the variable along which the derivative is computed is indicated by the first index of the resulting tensor. We also assume that the reader is familiar with the Einstein index notation for tensor contractions. To simplify notations later, we will make a distinction between the 2D and 3D gradients. The nabla operator[2] written as $\nabla \triangleq [\partial_x, \partial_y, \partial_z]$ will refer to the 3D gradient, while the 2D gradient will be written $\nabla_h \triangleq [\partial_x, \partial_y]$.

When writing integrals, the domain of integration will be denoted by Ω when we integrate over the whole mesh, or Ω_e when the integration is restricted to element e. $\partial\Omega$ will denote the boundary of the domain. To simplify notations, integrals on a domain Ω_e will be denoted by simple brackets

$$\int_{\Omega_e} f \, \mathrm{d}\Omega = \langle f \rangle_e$$

while integrals on the interfaces $\partial\Omega_e$ will use the double brackets.

$$\int_{\partial\Omega_e} f \, \mathrm{d}\sigma(\Omega) = \langle\!\langle f \rangle\!\rangle_e$$

In 2D, $\partial\Omega_e$ represents edges, while in 3D $\partial\Omega_e$ can either be the lateral (vertical) faces, or the top and bottom (horizontal) faces. Anytime a normal vector denoted by $\boldsymbol{n}$ is used, it represents the *outer* normal.

2.1 peculiarities of Ocean modelling

Ocean flows are characterised by two modes with widely different speeds. The external or fast mode is driven surface gravity waves. This mode is mainly 2-dimensional, since waves are a surface phenomenon. The internal or slow mode on the other hand is made of the actual 3D motion of the water, and it is in general orders of magnitude slower than the gravity waves. This difference in speeds is the main motivation to splitting the numerical into a 2D and 3D mode.

The fast speed of the gravity waves gives a strict restriction on the time step for explicit methods. However, since they are mostly 2-dimensional, they can be simulated well enough with a cheap 2D model. In contrast, the 3D motion of water is much more expensive computationally, but also much slower. It is thus simulated by a smaller number of larger time-steps. In order to maintain the consistency and convergence of the motion, a special care has to be put in the time-stepping algorithm that couples the 2D and 3D modes.

[2] In Cartesian coordinates

2.2 2D equations and model

The mathematical model used by SLIM for the 2D dynamics is the shallow water equations (SWE). The shallow water equations are a widely used model to simulate the evolution of surface waves and the depth-averaged water velocity.

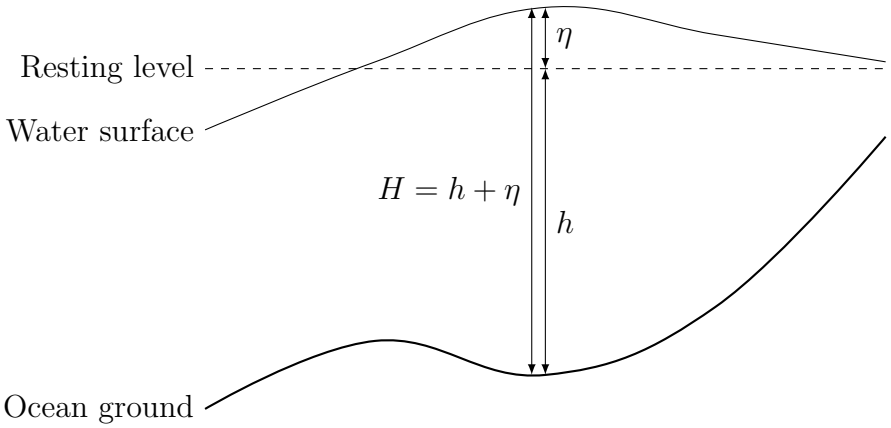


Figure 4: The water column depth H is split between the bathymetry h and the deviation from its resting level η.

Let $H = h + \eta$ be the water column height as shown in Figure 4. The bathymetry is denoted by h and the deviation from a still water level by η. Let $\bar{\boldsymbol{u}} = [\bar{u}_x, \bar{u}_y]$ be the depth-averaged water velocity. The non-dissipative, non-rotative and without forcings 2D shallow water equations written in conservative form are

$$\frac{\partial \eta}{\partial t} + \nabla_h \cdot (H\bar{\boldsymbol{u}}) = 0 \tag{1}$$

$$\frac{\partial (H\bar{\boldsymbol{u}})}{\partial t} + \nabla_h \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) = -\frac{g}{2}\nabla_h (H^2) + gH\nabla_h h = -gH\nabla_h \eta \tag{2}$$

The first equation (1) enforces the mass conservation and rightfully assumes an incompressible fluid. The second equation (2) expresses the conservation of momentum, and is derived from the the Navier-Stokes equations. This simplified for of the SWE is great as a toy model as it captures gravity wave dynamics of the motion. What it lacks, however, is the capability to simulate realistic processes, as those involve viscous diffusion, dissipation, forcings and rotations. The equation for momentum solved in SLIM2D is thus equation (3), which make up for those shortcomings.

$$\frac{\partial (H\bar{\boldsymbol{u}})}{\partial t} + \nabla_h \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) + fe_z \times H\bar{\boldsymbol{u}} = -gH\nabla_h \eta + \frac{\boldsymbol{\tau}}{\rho} + \nabla_h \cdot (\kappa H \nabla_h \bar{\boldsymbol{u}}) - \gamma H \bar{\boldsymbol{u}} \tag{3}$$

Here $\boldsymbol{\tau} = \boldsymbol{\tau}^s - \boldsymbol{\tau}^b$ is the total friction resulting from surface winds ($\boldsymbol{\tau}_s$) and bottom friction ($\boldsymbol{\tau}_b$). The Coriolis force is modelled by the term $fe_z \times H\bar{\boldsymbol{u}}$, κ is a viscosity coefficient, and γ is a friction coefficient that models dissipation. The different terms in the equation (3) can be understood as

— $-gH\nabla_h \eta$: The surface elevation gradient driving gravity waves

— $\nabla_h \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}})$: The nonlinear momentum advection due to inertial forces.

— $fe_z \times H\bar{\boldsymbol{u}}$: The Coriolis force acting on the fluid in Earth's frame of reference.

— $\nabla_h \cdot (\kappa H \nabla_h \bar{\boldsymbol{u}})$: The turbulent momentum diffusion viscosity term.

— $-\gamma H \bar{\boldsymbol{u}}$: An optional damping factor.

In the context of this thesis, we will only solve these equations on the Cartesian plane with coordinates x, y. This approximation is valid when the curvature of the domain is small enough relative to the size of the domain, which is true in the coastal studies for which SLIM is intended.

2.3 Discretization of the 2D equations

In order to solve equations (1) and (3), SLIM discretizes the domain in a 2D mesh made of triangles. The discontinuous Galerkin finite elements method (DGFEM) is then applied to the weak form of the SWE. Before engaging in the discretisations, we shall introduce the following notations for the mean value and mean difference across a boundary:

$$\{\boldsymbol{u}\} = \frac{\boldsymbol{u}^{\text{left}} + \boldsymbol{u}^{\text{right}}}{2}$$

$$[\![\boldsymbol{u}]\!] = \frac{\boldsymbol{u}^{\text{left}} - \boldsymbol{u}^{\text{right}}}{2}$$

Note that "left" and "right" are always relative to the face/edge and element considered. We also use the notation

$$\llbracket u \rrbracket = \max(u^{\text{left}}, u^{\text{right}})$$

for the maximum value between the two sides of a boundary. Finally, we introduce the notation $\boldsymbol{u}^{\text{up}}$ to refer to the "upwind" value of a discontinuous variable. Unless stated otherwise, the upwind direction is computed from the average speed at the boundary.

2.3.1 Surface height

The weak form of equation (1) can be reached by multiplying (1) by a test function φ and integrating over the domain.

$$\left\langle \varphi \frac{\partial \eta}{\partial t} \right\rangle + \langle \varphi \nabla_h \cdot (H\bar{\boldsymbol{u}}) \rangle = 0$$

Using the fact that $\nabla_h \cdot (\varphi H \bar{\boldsymbol{u}}) = H\bar{\boldsymbol{u}} \cdot \nabla_h \varphi + \varphi \nabla_h \cdot (H\bar{\boldsymbol{u}})$ and applying the divergence theorem, we get

$$\left\langle \varphi \frac{\partial \eta}{\partial t} \right\rangle = \langle \nabla_h \varphi \cdot H\bar{\boldsymbol{u}} \rangle - \langle\!\langle \varphi H(\bar{\boldsymbol{u}} \cdot \boldsymbol{n}) \rangle\!\rangle = \langle \nabla_h \varphi \cdot H\bar{\boldsymbol{u}} \rangle - \langle\!\langle \varphi H u_n \rangle\!\rangle$$

Using the discontinuous Galerkin method, we set the test function φ to be one of the basis functions φ_i, and we restrict the integration to a single element. In future developments, we will skip the previous step, jumping straight to $\varphi = \varphi_i$ and the integration is performed on a single element. Since the values on the boundaries are not uniquely defined anymore, a careful choice has to be made for the variables used at interfaces. Quantities that are

normally defined on both sides of interfaces will be denoted by an asterisk ($*$). The resulting equation solved is

$$\left\langle \varphi_i \frac{\partial \eta}{\partial t} \right\rangle_e = \langle \nabla_h \varphi_i \cdot H\bar{\boldsymbol{u}} \rangle_e - \langle\!\langle \varphi_i (Hu_n)^* \rangle\!\rangle_e \tag{4}$$

In SLIM, the variables used in this equation are either $(H, H\bar{\boldsymbol{u}})$ or $(\eta, H\bar{\boldsymbol{u}})$. In both cases, the quantity $(Hu_n)^*$ is computed based on a the Riemann solver for the SWE. Using the notation introduced above, it takes the form

$$(Hu_n)^* = \{\!\{Hu_n\}\!\} + c^* \, [\![\eta]\!]$$

The wave speed c^* is taken as the maximum wave speed between left and right: $c^* = \max(c^{\text{left}}, c^{\text{right}}) = [\![c]\!]$. On each side of the boundary, the wave speed c is computed as $c = \sqrt{gH}$, with H taking the values of H^{left} and H^{right} for the two sides. c can also be specified a-priori in the 2D model. This will turn useful when coupling the 2D and 3D models at a later time.

2.3.2 Momentum equation

The weak and discrete forms of the momentum equation (3) are obtained in a very similar process. For the sake of notations, we will develop the expressions of the different terms one by one.

Gravity waves

We start by taking the integral of the gravity term multiplied by the test function φ_i.

$$-g \langle \varphi_i H \nabla_h \eta \rangle$$

Integrating by parts and using the divergence theorem results in

$$\begin{aligned} -g \langle \varphi_i H \nabla_h \eta \rangle &= g \langle \eta \nabla_h (H\varphi_i) \rangle - g \langle \nabla_h (\varphi_i H \eta) \rangle \\ &= g \langle \eta \nabla_h (H\varphi_i) \rangle - g \langle\!\langle \boldsymbol{n} \varphi_i (H\eta)^* \rangle\!\rangle \\ &= g \langle \eta H \nabla_h \varphi_i \rangle + g \langle \eta \varphi_i \nabla_h H \rangle - g \langle\!\langle \boldsymbol{n} \varphi_i (H\eta)^* \rangle\!\rangle \end{aligned}$$

which is the weak form of the gravity term. On the boundary, $(H\eta)^*$ is computed as the arithmetic mean between the two sides : $(H\eta)^* = \{\!\{H\eta\}\!\}$. This formulation is well suited to applications where we easily have access to η, and as such is used in the GPU code. The CPU code, on the other hand, reformulates the gravity term as

$$\begin{aligned} -g \langle \varphi_i H \nabla_h \eta \rangle &= -g \langle \varphi_i H \nabla_h (H - h) \rangle \\ &= -g \left\langle \varphi_h \nabla_h H^2/2 \right\rangle + g \langle \varphi_i H \nabla_h h \rangle \end{aligned}$$

Using integration by parts on $\langle \varphi_h \nabla_h H^2/2 \rangle$ yields

$$\left\langle \varphi_h \nabla_h H^2/2 \right\rangle = \left\langle\!\!\left\langle \varphi_i \boldsymbol{n} (H^2)^*/2 \right\rangle\!\!\right\rangle - \left\langle H^2/2 \nabla_h \varphi_i \right\rangle$$

where the boundary term $(H^2)^*$ is expressed as $\{\!\{H^2\}\!\}$. While outside of the scope of this thesis, it is worth noting that this formulation is not suited to regions of discontinuous bathymetry. If that happens, a better formulation is, which is what is going to be used anyway.

$$(H^2)^* = \left\{ (\{\!\{\eta\}\!\} + h)^2 \right\}$$

figure out pourquoi le macro marche pas avec l'exposant

Advection

We again multiply the advection term by the test function φ and integrate. Note that some care has to be taken when writing these expressions, as they depend on the convention used for the gradient.

$$\begin{aligned} -\langle \varphi \nabla_h \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) \rangle &= \langle \nabla_h \varphi_i \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) \rangle - \langle \nabla_h \cdot (\varphi H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) \rangle \\ &= \langle \nabla_h \varphi_i \cdot (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}}) \rangle - \langle\!\langle \boldsymbol{n} \cdot \varphi (H\bar{\boldsymbol{u}}\bar{\boldsymbol{u}})^* \rangle\!\rangle \\ &= \langle \bar{\boldsymbol{u}} H\bar{\boldsymbol{u}} \cdot \nabla_h \varphi) \rangle - \langle\!\langle \varphi (H\bar{u}_n\bar{\boldsymbol{u}})^* \rangle\!\rangle \end{aligned}$$

On the boundaries, the term $\langle\!\langle \varphi (H\bar{u}_n\bar{\boldsymbol{u}})^* \rangle\!\rangle$ is split between its normal and tangential components. The normal component computed as the arithmetic mean from both sides:

$$(H\bar{u}_n\bar{u}_n)^* = \frac{(H\bar{u}_n\bar{u}_n)^{\text{left}} + (H\bar{u}_n\bar{u}_n)^{\text{right}}}{2} = \{\!\{ H\bar{u}_n\bar{u}_n \}\!\}$$

The tangential component $(H\bar{u}_n\bar{u}_t)^*$ on the other hand depends on the normal water speed $\bar{u}_n$. The normal speed is computed as $\bar{u}_n = \{\!\{\bar{u}_n\}\!\}$. The tangential part of the flow, on the other hand, is taken from the upwind region : $\bar{u}_t = \bar{u}_t^{\text{up}}$

Diffusion

The weak formulation of the diffusion term follows the same development, and yields

$$\begin{aligned} \langle \varphi \nabla_h \cdot (\kappa H \nabla_h \bar{\boldsymbol{u}}) \rangle &= -\langle \nabla_h \varphi \cdot \kappa H \nabla_h \bar{\boldsymbol{u}} \rangle + \langle \nabla_h \cdot (\varphi \kappa H \nabla_h \bar{\boldsymbol{u}}) \rangle \\ &= -\langle \nabla_h \varphi \cdot \kappa H \nabla_h \bar{\boldsymbol{u}} \rangle + \langle\!\langle \varphi \boldsymbol{n} \cdot (\kappa H \nabla_h \bar{\boldsymbol{u}})^* \rangle\!\rangle \end{aligned}$$

This formulation by itself is not sufficient though, as DG-FE diffusion operators require an additional stabilisation term. SLIM uses the Incomplete Interior Penality Method (IIPM) from [40, 41]. In IIPM, the interface flux is replaced by the mean flux $\{\!\{\kappa H \nabla_h \bar{\boldsymbol{u}}\}\!\}$ and an additional term $-\sigma\kappa H \, [\![\bar{\boldsymbol{u}}]\!]$. The penalty factor σ is defined in [42] as

$$\sigma_d = \frac{(o+1)(o+d)}{d} \frac{N_0}{2L_{\min}} \tag{5}$$

where d stands for the dimension of the problem and o is the order of the polynomials used for the discretization. In this work, we shall only use lineal basis functions, hence $o = 1$ everywhere. N_0 is the number of neighbours of an elements, meaning that $N_0 = 3$ in 2D and $N_0 = 5$ in 3D. $L_{\min}$ approximates the effective element length scale. In 2D, we take

$$L_{\min} = \frac{\min(A^{\text{left}}, A^{\text{right}})}{L}$$

where A stands for the triangle area and L is the length of the interface between the two triangles being considered. The complete diffusion term is thus

$$\langle \varphi \nabla_h \cdot (\kappa H \nabla_h \bar{\boldsymbol{u}}) \rangle = -\langle \nabla_h \varphi \cdot \kappa H \nabla_h \bar{\boldsymbol{u}} \rangle + \langle\!\langle \varphi \boldsymbol{n} \cdot \{\!\{\kappa H \nabla_h \bar{\boldsymbol{u}}\}\!\} \rangle\!\rangle - \langle\!\langle \sigma_2 \kappa H \, [\![\bar{\boldsymbol{u}}]\!] \rangle\!\rangle$$

Complete discrete formulation of the momentum equation

The remaining terms in (3) don't need any further treatment, they are simply added to the weak formulation. The discontinuous Galerkin formulation for this equation is

$$\begin{aligned}
\left\langle \varphi_i \frac{\partial (H\bar{\boldsymbol{u}})}{\partial t} \right\rangle_e =& - \langle \varphi_i f e_z \times H\bar{\boldsymbol{u}} \rangle_e && \text{(Coriolis)} \\
&+ g \langle \eta \nabla_h (H\varphi_i) \rangle_e - g \langle\!\langle \varphi_i \boldsymbol{n} (H\eta)^* \rangle\!\rangle_e && \text{(Gravity)} \\
&+ \langle \bar{\boldsymbol{u}} \varphi_i \nabla_h \cdot (H\bar{\boldsymbol{u}}) \rangle_e + \langle \bar{\boldsymbol{u}} H\bar{\boldsymbol{u}} \cdot \nabla_h \varphi_i) \rangle_e - \langle\!\langle \varphi_i (H\bar{u}_n \bar{\boldsymbol{u}})^* \rangle\!\rangle_e && \text{(Advection)} \\
&- \langle \nabla_h \varphi_i \cdot \kappa H \nabla_h \bar{\boldsymbol{u}} \rangle_e + \langle\!\langle \varphi_i \kappa \boldsymbol{n} \cdot (H \nabla_h \bar{\boldsymbol{u}})^* \rangle\!\rangle_e && \text{(Diffusion)} \\
&+ \langle \varphi_i \boldsymbol{\tau} / \rho \rangle_e - \gamma \langle \varphi_i H\bar{\boldsymbol{u}} \rangle_e && \text{(Friction and dampening)}
\end{aligned}$$

Current implementation

The current implementation of SLIM2D uses discontinuous polynomials of degree 1 as basis functions. When evaluating the integrals, SLIM uses a 3-point quadrature for triangles, and a 2-points quadrature for edges. The numerical computation of $\partial\eta/\partial t$ and $\partial(H\bar{\boldsymbol{u}})/\partial t$ is done in three steps. First, the integrals on the edges are computed for all elements. The above formulation of the Galerkin method ensures that any fluxes computed on the edges of the elements will be the same regardless of which element (left or right) is considered in the interior. This allows the evaluation of the integrals on the edges only once for each edges, which is twice as fast as computing them per-element. The second and third steps add the contributions from the triangle integrals and invert the mass matrix to isolate the derivatives.

The parallelisation of the 2D equation solver is done via domain decomposition. Each core computes the evolution a subset of the domain, and communication with neighbouring domains is implemented via a layer of ghosts elements. Ghosts elements are elements belonging to neighbouring subdomains that are used to compute the edge fluxes. Once the computation is done, a communication step occurs to update the ghost elements of neighbouring partitions. This step could technically be done in parallel with the computation, but since it is significantly faster its impact is negligible.

2.4 3D equations and model

The 3D model has a few fundamental differences compared to the 2D model. First of all, it is partially structured. The mesh is made of stacked layers of prisms, which makes its structure regular in the vertical direction. The second major difference is that the 3D model is also partially implicit. The regularity of the vertical direction allows us to easily separate horizontal and vertical dynamics. In general, quantities in the vertical direction will be computed implicitly, while those in the horizontal direction are done explicitly. The 3D model also has to account for the moving free surface. The main approaches to tackle this problem are the use of a coordinate system independent from the surface movement (e.g. sigma coordinates), and algorithms with a moving mesh. SLIM opted to use the latter. SLIM uses the Cartesian coordinates $\boldsymbol{x} = (x, y, z)$ and explicitly tracks the movement of the free surface by means of an Arbitrary Lagrangian-Eulerian (ALE) formulation.

2.4.1 Overview of the 3D equations

SLIM 3D operates in three steps. First, we solve for the water movement $\boldsymbol{u} = [u_x, u_y]$ horizontally and w vertically. Then we compute the mesh deformation, and by extension the vertical mesh speed w_m, and finally we compute the evolution of one or more tracer fields that is transported through advection and diffusion. For the water movement, SLIM solves the hydrostatic Boussinesq equations. In Cartesian coordinates, and accounting for the moving mesh, they are expressed as

$$\frac{\partial w}{\partial z} = -\nabla_h \boldsymbol{u} \tag{6}$$

$$\begin{aligned}\frac{\partial \boldsymbol{u}}{\partial t} + \nabla_h \cdot (\boldsymbol{u}\boldsymbol{u}) + \frac{\partial((w - w_m)\boldsymbol{u})}{\partial z} =& \nabla_h \cdot (\kappa_h \nabla_h \boldsymbol{u}) + \frac{\partial}{\partial z}\left(\kappa_v \frac{\partial \boldsymbol{u}}{\partial z}\right) \\ & - f e_z \times \boldsymbol{u} - \frac{1}{\rho_0}\nabla_h p\end{aligned} \tag{7}$$

$$\frac{\partial H}{\partial t} = \frac{\partial \eta}{\partial t} = -\nabla_h \cdot \boldsymbol{U} = -\nabla_h \cdot (H\bar{\boldsymbol{u}}) \tag{8}$$

In the equations above, κ_h and κ_v are respectively the horizontal and vertical viscosity. $\boldsymbol{U}$ is the total transport defined as

$$\boldsymbol{U} = \int_{-h}^{\eta} \boldsymbol{u}\, \mathrm{d}z = H\bar{\boldsymbol{u}}$$

with $\bar{\boldsymbol{u}}$, H, h and η as defined previously. We will also introduce the notation $\tilde{\boldsymbol{u}}$ for the deviation of $\boldsymbol{u}$ from the mean velocity : $\boldsymbol{u}(x, y, z) = \bar{\boldsymbol{u}}(x, y) + \tilde{\boldsymbol{u}}(x, y, z)$.

The boundaries of the 3D domain will be denoted by Γ_s for the free water surface, Γ_b for the bottom of the ocean, and Γ_l for lateral boundaries. Of these three, only Γ_s varies with time. When integrating on the boundaries of a prism, we will denote the top and bottom faces Γ_top and Γ_bot respectively. Together, these will be referred to as the horizontal boundaries and denoted $\Gamma_\mathrm{h} = \Gamma_\mathrm{top} \cup \Gamma_\mathrm{bot}$. The lateral boundaries of a prism are denoted by Γ_lat. The boundary conditions on the surface and bottom of the ocean are set to impermeability boundary conditions. Mathematically, they take the form :

$$w + \boldsymbol{u} \cdot \nabla_h h = 0 \qquad \text{on } \Gamma_b \tag{9}$$

$$w - \frac{\partial \eta}{\partial t} - \boldsymbol{u} \cdot \nabla_h \eta = 0 \qquad \text{on } \Gamma_s \tag{10}$$

On Γ_l, we can either have impermeability conditions or open boundary conditions. Impermeability is enforced by $\boldsymbol{n}_h \cdot \boldsymbol{u} = 0$, with $\boldsymbol{n}_h = [n_x, n_y]$ being the outgoing normal vector at the lateral boundaries[3]. Open boundaries translate to no boundary conditions.

Under the hydro-static assumption and neglecting the atmospheric pressure, the pressure p can be written in terms of the the density $\rho(\boldsymbol{x}) = \rho_0 + \rho'(\boldsymbol{x})$ integrated over the water column.

$$p(z) = g\int_z^\eta \rho(\tilde{z})\, \mathrm{d}\tilde{z} = g\rho_0(\eta - z) + g\int_z^\eta \rho'(\tilde{z})\, \mathrm{d}\tilde{z}$$

Defining the baroclinic head r as

$$r(z) = \frac{1}{\rho_0}\int_z^\eta \rho'\, \mathrm{d}\tilde{z} \tag{11}$$

[3]Lateral boundaries are vertical, so the normal lies in the horizontal plane.

we can rewrite the horizontal pressure gradient as

$$\frac{1}{\rho_0}\nabla_h p = g\nabla_h \eta + g\nabla_h r \ .$$

The horizontal gradient of r can in turn be expanded using Leibniz's formula as

$$\nabla_h r = \frac{1}{\rho_0}\left[\rho'(\eta)\nabla_h\eta - \rho'(z)\underbrace{\nabla_h(z)}_{0} + \int_z^\eta \nabla_h \rho' \,\mathrm{d}\tilde{z}\right] \ .$$

If we let

$$\boldsymbol{q} = \int_z^\eta \nabla_h \rho' \,\mathrm{d}\tilde{z} \tag{12}$$

the pressure gradient can be expressed as

$$\frac{1}{\rho_0}\nabla_h p = g\nabla_h \eta + \frac{g}{\rho_0}\boldsymbol{q} + \rho'(\eta)\nabla_h\eta \ . \tag{13}$$

In order to close this model, we need a model for the density variation ρ'. In this work, we chose a linear model in terms of the temperature T and salinity S:

$$\rho(T,S) = \rho_0 + \rho'(T,S) = \rho_0 - \alpha_T(T - T_0) + \beta_S(S - S_0) \ . \tag{14}$$

The evolution of T and S is modelled using a tracer that follows the advection-diffusion equation

$$\frac{\partial T}{\partial t} + \nabla_h \cdot (\boldsymbol{u}T) + \frac{\partial((w - w_m)T)}{\partial z} = \nabla_h \cdot (\kappa_h \nabla_h T) + \frac{\partial}{\partial z}\left(\kappa_\kappa \frac{\partial T}{\partial z}\right) \ . \tag{15}$$

The vertical mesh speed w_m only has to follow two boundary conditions which are that the mesh is static at the bottom of the ocean, and the mesh follows the water surface. These can be expressed as

$$w_m = 0 \qquad \text{on } \Gamma_b \ , \tag{16}$$

$$w_m = w \qquad \text{on } \Gamma_s \ . \tag{17}$$

This leaves a lot of freedom for the movement inside of the domain. In this work, we will use σ-like meshes, meaning that the mesh displacement will be split equally across all the layers. Other choices include z-like coordinates, where most layers remain fixed, and only the top few layers are deformed to account for the movement. This minimal movement can help with preserving the stratified nature of the ocean by limiting spurious vertical mixing.

2.4.2 Discretization of the 3D equations

Vertical velocity

We start the discretisation with the continuity equation (6), as it is the simplest and its treatment is very similar to equation (3). Multiplying by a test function φ and integrating

by parts on an element yields

$$\left\langle \varphi_i \frac{\partial w}{\partial z} \right\rangle = -\left\langle \varphi_i \nabla_h \cdot \boldsymbol{u} \right\rangle \ ,$$

$$\left\langle \frac{\partial}{\partial z}(\varphi_i w) \right\rangle - \left\langle w \frac{\partial \varphi_i}{\partial z} \right\rangle = -\left\langle \nabla_h \cdot (\varphi_i \boldsymbol{u}) \right\rangle + \left\langle \boldsymbol{u} \cdot \nabla_h \varphi_i \right\rangle \ ,$$

$$\left\langle\!\left\langle \varphi_i w^* n_z \right\rangle\!\right\rangle - \left\langle w \frac{\partial \varphi_i}{\partial z} \right\rangle = \left\langle \boldsymbol{u} \cdot \nabla_h \varphi_i \right\rangle - \left\langle\!\left\langle \boldsymbol{n}_h \cdot (\varphi_i \boldsymbol{u}^*) \right\rangle\!\right\rangle \ .$$

Since the domain of integration is a prism, the lateral boundaries are always vertical and $n_z = 0$ on Γ_{lat}. Thus, we can simplify the equation to be

$$\left\langle\!\left\langle \varphi_i w^* n_z \right\rangle\!\right\rangle_{\Gamma_\text{h}} - \left\langle w \frac{\partial \varphi_i}{\partial z} \right\rangle = \left\langle \boldsymbol{u} \cdot \nabla_h \varphi_i \right\rangle - \left\langle\!\left\langle \boldsymbol{n}_h \cdot (\varphi_i \boldsymbol{u}^*) \right\rangle\!\right\rangle \tag{18}$$

We use a Discontinuous Galerking formulation, meaning that $\boldsymbol{u}$ and w are not uniquely defined of the boundary, hence the star : "*". For this equation, we simply use the mean between the two elements: $\boldsymbol{u}^* = \{\boldsymbol{u}\}$, $w^* = \{w\}$.

2.4.3 Momentum equation

The momentum equation for $\boldsymbol{u}$ (7) is significantly more complicated, featuring 2D and 3D terms. Using the development for the pressure above, we can rewrite (7) as

$$\begin{aligned}
\frac{\partial \boldsymbol{u}}{\partial t} =& -\nabla_h \cdot (\boldsymbol{u}\boldsymbol{u}) - \frac{\partial((w - w_m)\boldsymbol{u})}{\partial z} && \text{(3D Advection)} \\
&+ \nabla_h \cdot (\kappa_h \nabla_h \boldsymbol{u}) + \frac{\partial}{\partial z}\left(\kappa_v \frac{\partial \boldsymbol{u}}{\partial z}\right) && \text{(3D Diffusion)} \\
&- f e_z \times \boldsymbol{u} && \text{(3D Coriolis)} \\
&- \frac{g}{H} \cdot H \nabla_h \eta && \text{(2D Gravity waves)} \\
&- \frac{g}{\rho_0} \boldsymbol{q} + \rho'(\eta) \nabla_h \eta && \text{(2D Baroclinic head)}
\end{aligned}$$

Advection

We start with the treatment of the advective term. The process shall not surprise the reader anymore, we integrate the product of the term and the test function φ_i by parts to reveal boundary terms :

$$\begin{aligned}
\left\langle \varphi_i \nabla_h \cdot (\boldsymbol{u}\boldsymbol{u}) \right\rangle &= \left\langle \nabla_h \cdot (\varphi_i \boldsymbol{u}\boldsymbol{u}) \right\rangle - \left\langle \nabla_h \varphi_i \cdot \boldsymbol{u}\boldsymbol{u} \right\rangle \\
&= \left\langle\!\left\langle \varphi_i (\boldsymbol{n}_h \cdot \boldsymbol{u}\boldsymbol{u})^* \right\rangle\!\right\rangle - \left\langle \nabla_h \varphi_i \cdot \boldsymbol{u}\boldsymbol{u} \right\rangle
\end{aligned} \tag{19}$$

$$\begin{aligned}
\left\langle \varphi_i \frac{\partial((w - w_m)\boldsymbol{u})}{\partial z} \right\rangle &= \left\langle \frac{\partial}{\partial z}[\varphi_i (w - w_m)\boldsymbol{u}] \right\rangle - \left\langle \frac{\partial \varphi_i}{\partial z}(w - w_m)\boldsymbol{u} \right\rangle \\
&= \left\langle\!\left\langle n_z \varphi_i (w^* - w_m)\boldsymbol{u}^* \right\rangle\!\right\rangle_{\Gamma_\text{h}} - \left\langle \frac{\partial \varphi_i}{\partial z}(w - w_m)\boldsymbol{u} \right\rangle
\end{aligned} \tag{20}$$

For equation (20), we used the same approach as for equation (18), the lateral boundaries are vertical so $n_z = 0$ on Γ_{lat}. Thus, we only integrate on the top and bottom faces. To account for the discontinuity, the boundary terms in these integrals are expressed as :

$$(\boldsymbol{n}_h \cdot \boldsymbol{u}\boldsymbol{u})^* = \boldsymbol{u}^{\text{up}} \{\boldsymbol{n}_h \cdot \boldsymbol{u}\} \ ,$$
$$(w^* - w_m)\boldsymbol{u}^* = \boldsymbol{u}^{\text{up}}(\{w\} - w_m) \ ,$$

which yields the following total contribution for the advective term

$$\begin{aligned}\text{adv} = &\langle \nabla_h \varphi_i \cdot \boldsymbol{u}\boldsymbol{u} \rangle - \langle\!\langle \varphi_i \boldsymbol{u}^{\text{up}} \{\boldsymbol{n}_h \cdot \boldsymbol{u}\} \rangle\!\rangle \\ &+ \left\langle \frac{\partial \varphi_i}{\partial z}(\{w\} - w_m)\boldsymbol{u} \right\rangle - \langle\!\langle n_z \varphi_i \boldsymbol{u}^{\text{up}}(\{w\} - w_m) \rangle\!\rangle_{\Gamma_{\text{h}}} \ .\end{aligned}$$

Diffusion

The procedure is the same as before, integrating by parts yields

$$\begin{aligned}\langle \varphi_i \nabla_h \cdot (\kappa_h \nabla_h \boldsymbol{u}) \rangle &= \langle \nabla_h \cdot (\varphi_i \kappa_h \nabla_h \boldsymbol{u}) \rangle - \langle \nabla_h \varphi_i \cdot \kappa_h \nabla_h \boldsymbol{u} \rangle \\ &= - \langle \nabla_h \varphi_i \cdot \kappa_h \nabla_h \boldsymbol{u} \rangle + \langle\!\langle \varphi_i \boldsymbol{n}_h \cdot (\kappa_h \nabla_h \boldsymbol{u})^* \rangle\!\rangle\end{aligned}$$

for the horizontal diffusion term, and

$$\begin{aligned}\left\langle \varphi_i \frac{\partial}{\partial z}\left(\kappa_v \frac{\partial \boldsymbol{u}}{\partial z}\right) \right\rangle &= \left\langle \frac{\partial}{\partial z}\left(\varphi_i \kappa_v \frac{\partial \boldsymbol{u}}{\partial z}\right) \right\rangle - \left\langle \frac{\partial \varphi_i}{\partial z} \kappa_v \frac{\partial \boldsymbol{u}}{\partial z} \right\rangle \\ &= - \left\langle \frac{\partial \varphi_i}{\partial z} \kappa_v \frac{\partial \boldsymbol{u}}{\partial z} \right\rangle + \left\langle\!\!\left\langle \varphi_i n_z \left(\kappa_v \frac{\partial \boldsymbol{u}}{\partial z}\right)^* \right\rangle\!\!\right\rangle_{\Gamma_{\text{h}}}\end{aligned}$$

for the vertical diffusion. Just like in 2D, the diffusion operators need some stabilisation. The IIPM stabilisation in 3D takes a form similar to the 2D IIPM, with fluxes being replaced by a combination of the mean gradient and an additional term encapsulating the jump $[\![\boldsymbol{u}]\!]$. The only difference is that the diffusion is no longer isotropic, so we need a more rigorous formulation of the penalty term. If we let $\boldsymbol{D}_\kappa = \text{diag}(\kappa_h, \kappa_h, \kappa_v)$ be the 3D diffusivity tensor, we can write the additional penalty term as :

$$\langle\!\langle \sigma_3 (\boldsymbol{n} \cdot \boldsymbol{D}_\kappa \cdot \boldsymbol{n}) [\![\boldsymbol{u}]\!] \rangle\!\rangle = \left\langle\!\!\left\langle \sigma_3 \kappa_h (n_x^2 + n_y^2) [\![\boldsymbol{u}]\!] \right\rangle\!\!\right\rangle + \left\langle\!\!\left\langle \sigma_3 \kappa_v n_z^2 [\![\boldsymbol{u}]\!] \right\rangle\!\!\right\rangle_{\Gamma_{\text{lat}}} \ .$$

The fluxes can thus be understood as:

$$\begin{aligned}\langle\!\langle \varphi_i \boldsymbol{n}_h \cdot (\kappa_h \nabla_h \boldsymbol{u})^* \rangle\!\rangle &= \langle\!\langle \varphi_i \kappa_h \boldsymbol{n}_h \cdot \{\nabla_h \boldsymbol{u}\} \rangle\!\rangle - \left\langle\!\!\left\langle \sigma_3 \kappa_h (n_x^2 + n_y^2) [\![\boldsymbol{u}]\!] \right\rangle\!\!\right\rangle \ , \\ \langle\!\langle \varphi_i n_z (\kappa_v \partial_z \boldsymbol{u})^* \rangle\!\rangle_{\Gamma_{\text{h}}} &= \langle\!\langle \varphi_i n_z \{\kappa_v \partial_z \boldsymbol{u}\} \rangle\!\rangle_{\Gamma_{\text{h}}} - \left\langle\!\!\left\langle \sigma_3 \kappa_v n_z^2 [\![\boldsymbol{u}]\!] \right\rangle\!\!\right\rangle_{\Gamma_{\text{h}}} \ .\end{aligned}$$

The penalty parameter σ_3 is defined by (5), with the length $L_{\min}$ chosen as

$$L_{\min} = \frac{\min(V^{\text{left}}, V^{\text{right}})}{A}$$

where V is the element volume, and A is the area of the interface between the two prisms.

Gravity waves

The gravity waves term is essentially 2D, and its weak formulation has already been done in section 2.3.2. The only difference compared to equation (3) is that we only keep surface gradient term, and we divide the result by H. Later, when we integrate in time, the evolution of the 2D terms like this one uses a different time step than the rest of the 3D equations. Therefore, in order to be consistent, the wave speed c of the Riemann solver is set to

$$c = \sqrt{gH} + \frac{U_n}{H} = \sqrt{gH} + \bar{u}_n$$

on each side of the boundary, with the value $c^* = [\![c]\!]$ being used for the computations.

2.4.4 Discrete tracer equation

The treatment of the tracer equation (15) is almost identical to the momentum equation for $\boldsymbol{u}$. They both have a diffusive and convective term, which are treated in the exact same way, the only difference being that the tracer is not affected by 2D effects of the Coriolis force. The discrete tracer equation thus looks like :

$$\begin{aligned}
\left\langle \varphi_i \frac{\partial T}{\partial t} \right\rangle &= \langle \nabla_h \varphi_i \cdot \boldsymbol{u} T \rangle - \langle\!\langle \varphi_i T^{\text{up}} \{\boldsymbol{n}_h \cdot \boldsymbol{u}\} \rangle\!\rangle && \text{(Horizontal transport)} \\
&+ \left\langle \frac{\partial \varphi_i}{\partial z} (\{w\} - w_m) T \right\rangle - \langle\!\langle n_z \varphi_i T^{\text{up}} (\{w\} - w_m) \rangle\!\rangle_{\Gamma_h} && \text{(Vertical transport)} \\
&- \langle \nabla_h \varphi_i \cdot \kappa_h \nabla_h T \rangle + \langle\!\langle \varphi_i \kappa_h \boldsymbol{n}_h \cdot \{\nabla_h T\} \rangle\!\rangle && \text{(Horizontal diffusion)} \\
&- \left\langle \frac{\partial \varphi_i}{\partial z} \kappa_v \frac{\partial T}{\partial z} \right\rangle + \langle\!\langle \varphi_i n_z \{\kappa_v \partial_z T\} \rangle\!\rangle_{\Gamma_h} && \text{(Vertical diffusion)} \\
&- \langle\!\langle \sigma_3 \kappa_h (n_x^2 + n_y^2) [\![T]\!] \rangle\!\rangle && \text{(Horizontal stabilisation)} \\
&- \langle\!\langle \sigma_3 \kappa_v n_z^2 [\![T]\!] \rangle\!\rangle_{\Gamma_h} && \text{(Vertical stabilisation)}
\end{aligned}$$

Where the attentive reader will notice that the only change compared to the equations of $\boldsymbol{u}$ are the letters $\boldsymbol{u}$ becoming T, and κ taking the role of ν.

2.5 Temporal scheme

The 2D and 3D modes have different speeds, and are thus simulated with different time steps. To couple those together, we use a split-explicit time stepping method, as illustrated in Figure 5. The figure reads from top to bottom. We start by evolving the slow internal mode. The, the fast 2D mode runs a number of second-order Runge-Kutta steps to reach the same point as the slow mode. While doing so, the previously computed internal acts as an external forcing on the 2D dynamics. Once the fast 2D mode has reached the 3D mode, we can compute the new mesh. The mesh depends on the water elevation which is computed in 2D. Then, the second 3D step is computed, while taking into account the new mesh and the values of the first step.

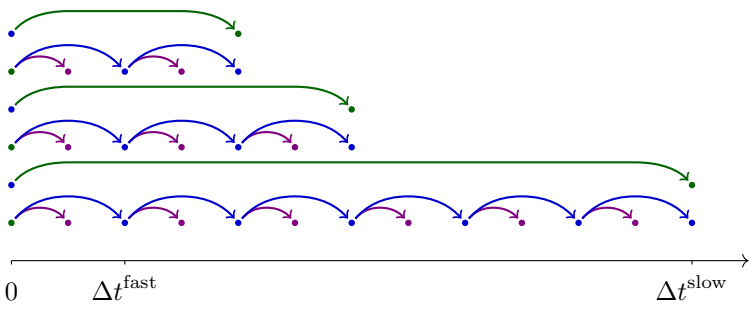


Figure 5: Temporal scheme used for coupling 2D and 3D dynamics. It reads from top to bottom. Each arrow marks the computation a derivative.

3 Deep dive in the GPU architecture

Historically, graphics cards have almost always been from NVIDIA or AMD, at least on the consumer market. This makes sense, as they are the oldest players in the GPU scene, going back to the early 2000s. INTEL has also been a quiet player in the GPU market for a long time, with their integrated graphics division. At the time of writing, the high end GPU market is still dominated by AMD and NVIDIA, both in the consumer space and in enterprise solutions. This may change soon though, as INTEL has been planning to release High-performance GPUs for quite some time now. Nevertheless, Intel's presence is sparse and this work will thus focus on AMD and NVIDIA GPUs. For historical reason, AND and NVIDIA have not been able to agree on a common nomenclature for a lot of the main features of GPUs. When the nomenclature of a feature diverges between both companies, we will mention both but then stick to the one that best self-describes.

Before diving in the software and hardware details, it is worth mentioning that this analysis is done through the prism of the HIP and CUDA programming languages. HIP and cuda are extremely similar. Both are extensions to the C/C++ language that provide access to the GPU. Their main difference is that CUDA is NVIDIA's property and is closed source while HIP is developed by AMD is open-source.

Kernel launching and synchronisation

One occurrence where both NVIDIA and AMD that the processor is called the host, and the GPU is called the device. It is always the CPU that hosts the operating system and controls the execution flow of the program, hence its name. The GPU or device is aimed at running special programs, called "kernels".

The execution model of a processor is relatively simple, instructions come in (sequentially), and are processed in order[4]. Most instructions and functions on the host are called "blocking", meaning that a function has to terminate before the next command is issued. This is in stark contrast with how most operations are done on the GPU.

In order to run code on the GPU, the host program has to launch a kernel. The main difference between kernels and normal functions is that kernels are executed asynchronously from the host's point of view : the host sends the signal to start the computation without waiting for the completion of the kernel execution.

Kernels are not independent of each other though. Kernels are always launched on a certain "stream" or "queue", and the GPU logic will make sure that all operations queued on a stream will run sequentially. This is not to say that GPUs are incapable of running more than one kernel at a time : running kernels in parallel is possible if they are launched on different streams.

Launching kernels asynchronously already brings two interesting benefits. This first is that the CPU can work on other things while a kernel is running. The other is due to the relatively long time necessary to launch a kernel. Depending on the number of arguments, it can take anywhere from a few microseconds to a few tens of microseconds to launch a kernel. This can seem negligible, but for small kernels that run in tens of microseconds, the impact is very much felt.

Asynchronous computations are great, but when comes the time to retrieve the result of the computations, it is necessary to wait for the GPU program to finish. GPUs

[4]This is not quite accurate, as modern CPUs will often reorder instructions when dependencies allow it in order to increase the instruction-level parallelism : the execution is "pipelined", meaning that the execution of consecutive instructions often overlaps.

offer 3 mechanisms for kernel synchronisation : (blocking) data transfers, stream-wide synchronisation and device-wide synchronisation. By default, copying data to and from the GPU is done synchronously : the function returns when the data transfer is complete. Just like kernels, memory copy operations are tied to a stream, and therefore, if there are any uncompleted kernels on that stream, the memory copy will wait for them to finish before beginning the data transfer. Note that there also exist ways to move data around asynchronously. The second mechanism for synchronisation is a stream-wide barrier. As its name suggests, it will block all operation on the host and wait for all the operations on a stream to finish before returning. A device-wide barrier also exists, and does the same but for the entire GPU.

3.1 Programming model

CPUs and GPUs both strive to increase performances, but follow two very distinct approaches with different trade-offs. CPUs try as much as possible to reduce *latency* of both memory operations and instructions. Latency refers to ay period of time where the CPU or GPU is not doing active work. This makes CPUs very powerful tool to execute serial codes, but maps poorly to massively parallel problems. GPUs on the other hand trade latency for massive performances, provided that they are tasked with a parallel enough problem. This fundamental difference in the physical design is accompanied with very different programming models.

Programming for the GPU follows the assumption that the same set of operations will be executed thousands or millions of times on different chunks of data. Following this premise, GPUs can spawn thousands of "threads", which are sequences of instructions that only depend on each other and that are executed sequentially. Each thread is then responsible for one piece of the workload. Although the execution inside of a thread is sequential, the execution order of different threads cannot be assumed a-priori. Consequently, communications across threads is generally not possible, with one exception detailed below.

3.1.1 Grids and blocs

During the execution of a kernel, the GPU threads are grouped by "blocs", and these blocs are arranged on a "grid". A bloc is a group of threads that can have either 1, 2 or 3 dimensions. The size of a bloc is the product of its dimensions. In order for a thread to identify which part of work it has to do, a thread has access to the dimensions of its bloc, as well as its index in each dimension of the bloc. Similarly, the grid of blocs can also have 1, 2 or 3 dimensions, and the thread can access the grid size, and the index of its bloc in each dimension of the grid. Depending on the hardware used, the maximum bloc size can vary slightly, with, typical values around 1024. Inside of a bloc, threads are arranged in a column-major order : the first index varies the fasts when iterating through the threads (Figure 6). Knowing how threads are ordered will become relevant in section 3.3 when talking about coalescent memory accesses.

3.1.2 Synchronisation

Aside from the benefits of having a multi-dimensional index[5], blocs provide the only way for threads to communicate and synchronise work. Different threads of the same bloc can

[5]Which can be useful for some problems (ex: matrix computations)

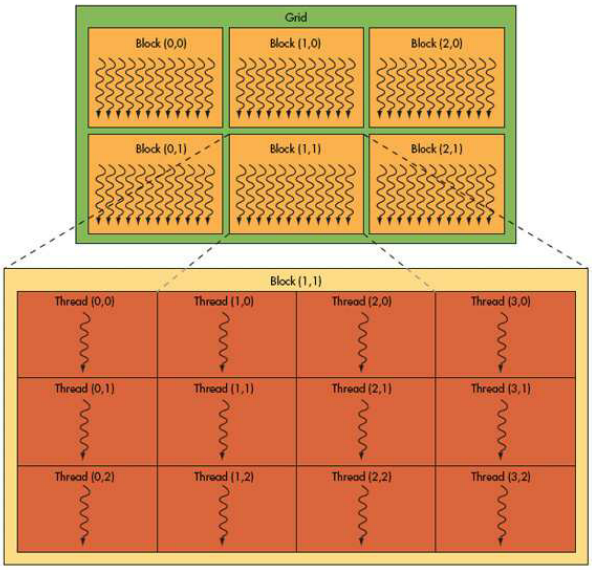


Figure 6: Example of a 2D bloc hierarchy, in a 2D grid. The order in which threads should bear is from left to first, then top to bottom. Thread (0,0) has aglobal ID of 0, thread (1,0) a global ID of 1, and thread (0,1) an ID of 4. Figure from [43].

access a special memory called "shared memory". Shared memory is unique to a bloc and can only be accessed by the threads of the bloc. It is also extremely fast. The threads from a bloc have access to a special function to "synchronise" all the threads from the bloc. When it is called, all the previously issued memory operations are guaranteed to be complete, which coupled with the use of shared memory can allow threads to communicate. Note that the cost of synchronisation can be quite high, and doing it frequently will likely degrade the performance.

3.2 Hardware model and performance implications

Knowing how to program with CUDA or HIP is very useful tool for GPU programming, but it does not guarantee that the programs will perform well. Getting the maximum out of the GPU requires a good understanding of the hardware, much more so that for a CPU.

3.2.1 SIMD model

GPUs are designed to work by applying the same operations to a large chunk of data. To do so, they use the paradigm of Single Instruction Mutiple Data (SIMD). A GPU is made of a fixed number of Compute Units (CUs) or Streaming Multiprocessors (SMs). Compute Units and Streaming Multiprocessors are very much synonymous, with Compute Units being AMD's terminology and Streaming Multiprocessors being NVIDIA's terminology. Figure 7 shows the repartition of CUs in a block diagram of AMD's CDNA2 architecture.

To achieve SIMD parallelism, each Compute Unit is made of typically 64 cores that are capable of executing a scalar instruction. AMD calls cores "shading units", but again, these terms are interchangeable. The cores a a Compute Unit are not independent. They are grouped in "SIMD units". Cores from the same SIMD unit run together as one wide core, executing the same operation for a bunch of threads at a once. Mimicking the

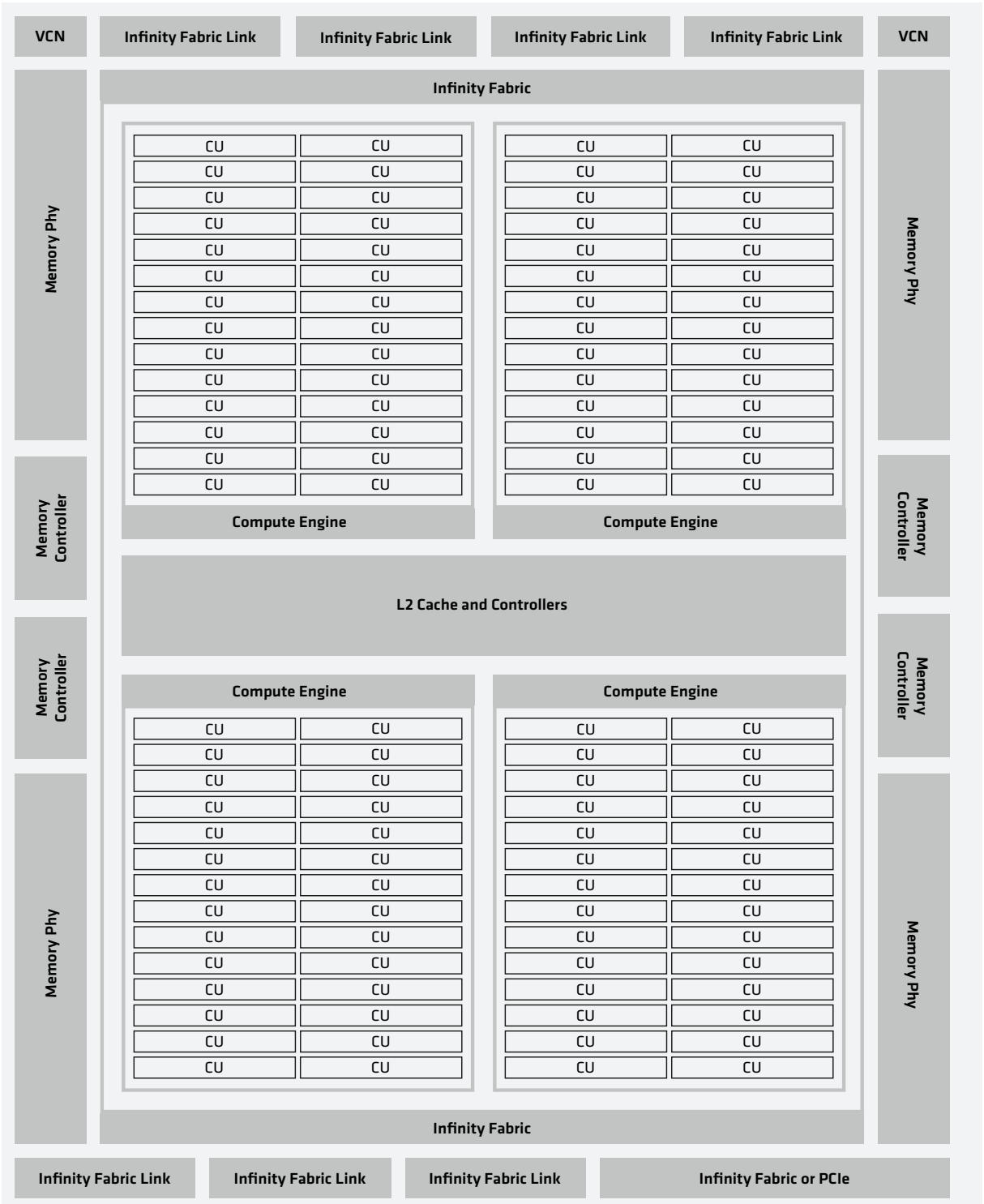


Figure 7: Block diagram of the compute die of AMD's CDNA2 architecture [20]. Each Compute Unit (CU) is made of 4 SIMD units, and each SIMD unit is made of 16 cores.

hardware, threads from a bloc are not independent, they are grouped in warps (NVIDIA) or wave-fronts (AMD). Warps are the software equivalent of SIMD units. Threads from the same warp always execute the exact same sequence of instructions. In fact, instruction are not issued per thread, they are issued warp-wide. When an instruction is issued to a warp, the warp is assigned to a physical SIMD unit to execute the instruction. If the SIMD unit and the warp have the same size, the instruction is completed in one cycle. Sometimes the warp size is a multiple (typically 2 or 4) of the SIMD size. Then the the instruction completes in 2 or 4 cycles.

Recent NVIDIA GPUs have a warp size of 32, and one Compute Unit is made of two 32-wide SIMDs. Therefore, it takes 1 clock cycle to execute a warp, and 2 warps can be executed simultaneously. It also means that we need at least two warps to completely fill the Compute Unit. On the other hand, AMD GPUs based on the GCN architecture such as their MI100 and MI200 series of accelerators have a warp size of 64 and a SIMD size of 16. Thus, issuing an instruction takes 4 clock cycles[20]. Since there are 4 SIMD units, 4 warps can be executed concurrently and at least 4 warps are needed to fill the CU.

3.2.2 warps

During the execution, a thread bloc is assigned to a Compute Unit (CU). Threads of the bloc will then be divided in a number of warps. If the size of the bloc is not a multiple of warp size, some threads will be masked. While only a handful of warps can be executed simultaneously[6], the number of warps that can reside in a CU is often much larger. The maximum number of warps residing in a CU depends on the hardware, and is on the order of 32-64. In practice, however, it is often restricted by the number of registers and shared memory used for the kernel. Programs with lots of variables will use more registers, and thus only a smaller number of warps will be able to reside in the registers of the CU. The same is true for shared memory. This mechanism is also what limits the maximum size of a bloc most of the time. To give orders of magnitude, Compute Unit of the Turing architectures from NVIDIA have 64K 32-bit registers. Tye also limit the maximum number of registers per thread to 255.

During the execution of a kernel, a warp is said to be active if it hasn't yet reached the end of the kernel. The warps can be in one of the following three states.

— **Eligible** : A warp is said to be eligible if it is active and ready to be executed. It means that all the variables needed for the next instruction are already in the registers.

— **Stalled** : A warp is stalled if it is active, but not ready for the next instruction. This can happen for tree reasons : fixed and variable latency instructions, and synchronisation. Of these three, variable-latency instructions are often the most prominent. They happen when data needs to be loaded into the registers from the caches or from the main memory. This can take anywhere from tens to hundreds of clock cycles[44].

  Fixed-latency instructions on the other hand are generally much faster and way less common. They mostly come from slow special functions.

  The impact of synchronisation calls can vary a lot depending on how frequent synchronisation calls are made, and the size of the bloc.

[6]Typically 2 for NVIDIA and 4 for GCN-based AMD GPUs

— **Inactive** : Onece a warp has finished its job, it becomes inactive and does nothing.

Having more than 2 or 4 warps per CU is therefore a key aspect of getting the maximum out of a GPU. In order to keep the hardware busy, the warp scheduler will execute one of the eligible warps. Due to the various sources of latency, that warp may then be marked as stalled for some cycles, and the warp scheduler will switch to another eligible warp, hiding the latency. In large kernels (meaning that there are few warps per CU) with lots of memory accesses, there may not be an eligible warp, which causes the GPU to stall. For most of this work, this is the main source of performance losses. As a result, a big part of the optimization in what follows is focused on either reducing the cost of memory accesses, or decreasing the "size" of the kernel.

3.2.3 Occupancy and impact of bloc size

To characterise how the efficiency of a kernel depends on the bloc size, we can look at a metric called the "occupancy". The occupancy is defined as the ratio between the number of active warps of a CU and the maximum number of warps that can reside in a CU. A low occupancy is obviously a source of stalling. It results on poor instruction issue efficiency as they are not enough eligible warps, and the GPU is not able to hide the memory latencies. However, maximising the occupancy is not always recommended, as it reduces the amount of resources per warp, which may start to hurt the performance. Note that it is possible to give an upper bound on the bloc size to the compiler. This can trigger further optimisations that can sometimes significantly improve the performance, especially if the kernel uses many registers.

3.2.4 Miscellaneous differences with a CPU

When programming a CPU, it is fairly well known that "branches are slow", especially when the branch prediction fails. GPUs suffer from a similar problem, but for very different reasons. Since instructions are executed warp-wide, "if" statements pose a challenge when threads from the same warp take different branches. It forces the warp to execute both branches one after the other, with some threads masked according to which branch the thread should follow. This happens even if only one thread from a warp diverges, and can cause the GPU to be severely under-utilised if branches are large. It does not mean however that "if" statements should be avoided altogether. If we can make sure that all threads from a warp follow the same branch, there are almost no penalties from branching.

Another way to under-utilise a Compute Unit (CU) is to chose an inappropriate bloc size. As stated above, blocs are dispatched to the different Compute Unit (CU) and executed in warps of 32 or 64 threads. However, the bloc size needs not to be a multiple of 32 or 64. When that happens, some threads of the warp are "masked", yielding inactive threads.

A third striking difference between GPUs and CPUs is that GPUs are bad at double-precision math. Since they were initially designed to run rendering operations that don't require a lot of precision, GPUs cut the number of FP64 units in the CUs. As a result, most consumer-grade GPUs have a FP64 throughput that is $1/16^{\text{th}}$ or $1/32^{\text{th}}$ of their FP32 performance. In contrast, professional GPUs designed as HPC accelerators don't cut as hard on the FP64 performance. Most of then have a FP64/FP32 ratio of only one

half, with one notable exception being the MI200 series of accelerators from AMD that have the same FP32 and FP64 capabilities[7].

3.3 Memory architecture of a GPU

The GPU memory is optimized for throughput and does not particularly focus on low latency. This is very different from what the RAM of CPU does. This changes the optimal access pattern and is one of the most important things to keep in mind when programming for a GPU.

3.3.1 Coalescence VS locality

The memory hardware of CPUs is designed to reduce latency through the use of increasingly faster but smaller caches. When exchanging information with the system memory, caches make transactions in chunks of 64[8] bytes. Additionally, caches operate in a Least Recently Used (LRU) manner, cache lines that were accessed recently are likely to be already present in the cache when needed. These characteristics favour codes with high temporal and spatial **locality**. It is beneficial to organise the data such that future memory accesses are located next to the previous ones in memory.

GPU memory is organised differently. Recall that instructions are executed by a SIMD unit, meaning that many threads execute instructions (including memory transactions) at once. Thus, memory transactions were designed to operate with chunks of 128 bytes at a time, with the hope that one memory transaction will be able to fulfill the requests from all the threads of a warp. If this fails, i.e. when threads from a warp access memory locations outside of a single 128-byte chunk, more transactions are required. This creates is a fundamental change in the optimal access pattern for data. In order to efficiently use the memory system, consecutive threads should access consecutive memory locations. This is called memory **coalescence**. Contrary to what locality would suggest, enabling coalescent memory accesses requires that neighbouring memory locations do not contain the data needed in a short future. Rather, neighbouring addresses should have the pieces of data that are accesses simultaneously by different threads. Even if future items are located nowhere near in the memory, the performance will still be good as long as accesses can be coalescent.

When coalescent memory accesses are not possible, it is worth remembering that GPUs also have caches which work similarly to the CPU. This means that data recently used by any thread of a CU is likely to be accessed faster if it is needed by a thread of that CU. Note that when a choice needs to be made, as a general rule, coalescence is more important than locality on the GPU.

Memory types

GPUs have a variety of different type of memories but they can generally be classified in three categories, in regards to who can access the memory and where it is located from a hardware point of view.

[7]This refers to "vector performance", i.e. each thread operating on one value (single or double). The MI200 series can also work with "packed" single precision floats, doubling the performance, but using them may require some additional work. Optimized libraries such as rocBLAS do use them however, so higher level codes can benefit from them.

[8]The cache line size for most x86-64 CPUs is 64 bytes, but that number can vary.

— DRAM : It is the slowest but largest pool of memory and it is located off-chip. Currently, its capacity varies between a few Gigabytes in consumer cards to up to 128GB in professional GPUs from AMD. Every CU can read and write from DRAM.

— Global cache : Closer to the cores, we find a an on-chip global cache (often referred to as L2 cache). This memory pool is much smaller that the DRAM, with sizes measured in megabytes or tens of megabytes. Its particularity when compared to other caches is that all CUs can access it.

— CU memory : Finally, there is memory local to a compute unit, consisting of the first level of cache and registers. These types of memory are private to each CU.

Note that this hierarchy is subject to variations. The recent consumer GPUs from AMD that use the RDNA and RDNA2 architectures have 4 levels of cache. The last two levels (L2 and L3 also referred as "infinity cache") are shared by the entire GPU. Then comes the L1 cache, shared by a group of CUs, and finally the L0 cache is individual to each CU.

It is worth taking some time understanding precisely how the different types of memory tide in together and how to use them efficiently.

3.3.2 Global memory

The simplest type of memory to detail is global memory. Global memory is located in the DRAM modules, outside of the GPU die. It can be viewed as the GPU-equivalent of RAM. It is the largest pool of memory available on the GPU where all variables reside in between kernels. Memory copies to and from the GPU are also directed to global memory when communications need to happen.

The DRAM chips of current GPUs use GDDR6 or HBM2e memory. Both of these technologies can deliver large bandwidth, but at a significant cost in latency. For recent consumer GPUs based on GDDR6, the DRAM latency is somewhere between 250 and 300 ns (Figure 8), which is hundreds of clock cycles. Thus, the global memory access pattern can dramatically impact the performance of a program, and it is critical that accesses are as coalescent as possible.

3.3.3 L2 and L1 caches

Most GPUs come with a first layer of cache that is global to all threads. It works very similarly to CPU caches, in the sense that it accelerates global memory accesses that exhibit locality. While its effects are very much felt, the L2 cache is "transparent" in the sense that the programmer has no control over it. The latency for the L2 is on the order of 100 ns. Each CU is also equipped with its own cache, which is, as expected, faster. The L1 cache the same way as the L2.

3.3.4 Registers

Registers are the fastest kind of memory available in the GPU. FPUs and ALUs of Compute Units (CUs) operate directly on the registers. Registers are bound to a thread and are not freed until the thread terminates. This is why increasing the bloc size (thus increasing the thread count) can become detrimental for performance (or just impossible

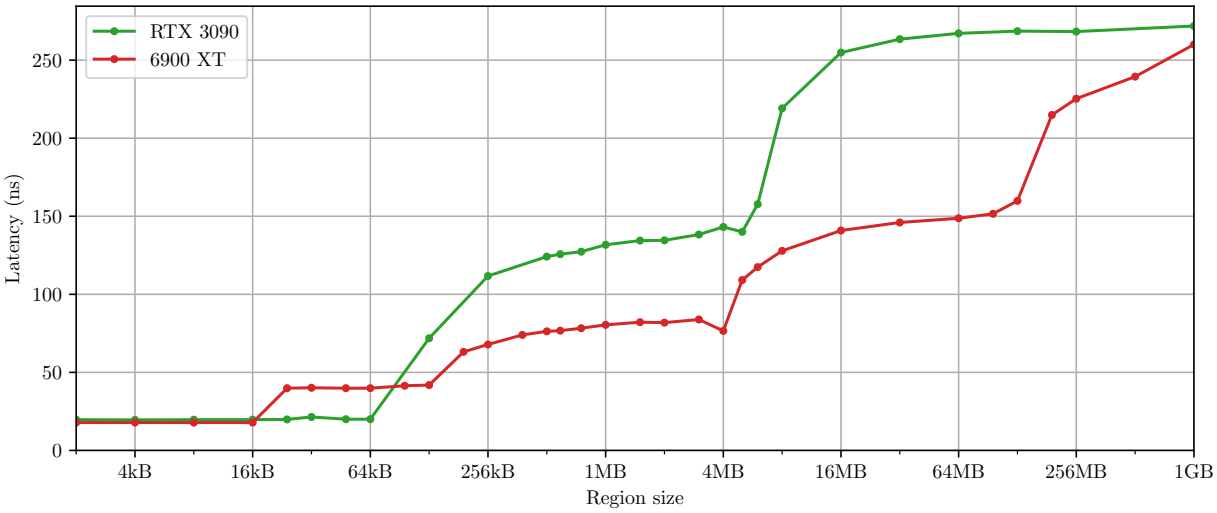


Figure 8: Random access latency of the GPU. The benchmark measures the average latency of random accesses in a region of defined size. The curve for the RTX3090 clearly shows 3 regimes that are the L1, L2 and global memory. The 6900XT is more progressive, as expected given its 4 levels of cache. Data extracted from [45]. Source code for the benchmark is available at [46].

sometimes), as it reduces the number of registers per thread. One interesting characteristic of GPUs is that they can have more register storage than L1 cache. This is explained by the need of having a lot of registers to enable the GPU to quickly switch between eligible warps.

3.3.5 Local memory

Let's begin by saying that local memory should not be called that way. Sometimes kernels would need more registers than available to store all of their variables. in these situations, rather that throwing an error, the GPU will allocate so-called local memory where exceeding data will be stored. The problem is that local memory is in fact global memory with a fancy name. Therefore, it has a extremely slow latency. Local memory benefits from the various levels of cache which somewhat mitigates its impact, but using it is still very expensive, as we trade the fastest type of memory for the slowest.

3.3.6 Shared memory/LDS

Another type of memory is the so-called shared memory (NVIDIA) or Local Data Share (LDS) (AMD). Shared memory can essentially be thought as programmable L1 cache. It is a very fast but small memory pool that is shared by the whole CU. From a programming point of view, a variable declared in shared memory can be accessed by all threads of a bloc. Shared memory is generally used for two separate functions. The first one is communications or reductions across a bloc. This should not come as a surprise, since shared memory is the only available mechanism to communicate between threads. Its other common use is as a buffer to improve the memory access pattern. A typical example of this use is transposing a matrix. Transpositions have the problem that either in loading or writing to memory, the access pattern will be strided. To mitigate this problem, one loads a a sub-matrix in shared memory, transposes the matrix in shared memory, and

then writes back the sub-matrix. Therefore, (slower) strided access happend in shared memory, and not in global memory, which massively reduces its impact.

Shared memory is made of equally-sized memory modules called "banks". In general, the building blocs of a bank are 32-bit "words"[9]. Banks have a memory bandwidth of 32 bits per clock cycle and all banks can be accessed at the same time. This makes them capable of exchanging data extremely fast, provided that there are no bank "conflicts". Bank conflicts occur when multiple threads want to access words from the same bank at the same time. If that happens, the accesses are automatically serialised and the

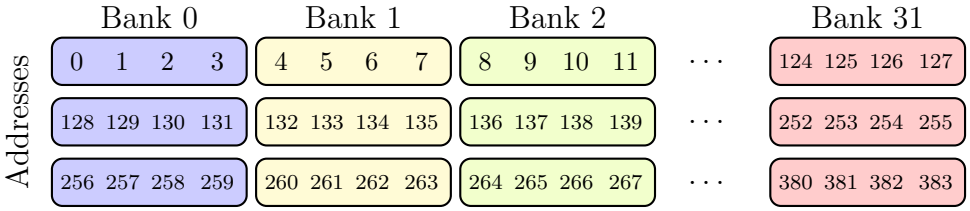


Figure 9: Allotment of shared memory addresses per banks. The figures illustrates a configuration with 32 banks and a word size of 32 bits (4 bytes).

bandwidth takes a hit. Note that most recent architectures allow multiple threads to read simultaneously one word from a bank in the form of a broadcast (all threads from the warp access the same word) or multicast (only some threads access the word). Write operations and accesses to different words within the same bank always serialize. To minimize bank conflicts, knowing how memory locations are mapped to banks is necessary. Consecutive words of shared memory are located in different banks, as illustrated by Figure 9.

3.3.7 Constant memory

GPUs provide a small amount of memory that can be used for constants. Constant memory is accessible globally by all threads. Its benefit over global memory is in its speed. Constant memory is cached on-chip, which means that on a cache hit, it is extremely fast. When threads access a small number of shared items, constant memory can be as fast as registers. The counterpart is that constant memory is rather limited to a few tens of kilobytes in most cases.

3.3.8 Texture memory

Texture memory is yet another type of memory that is read-only. Just as constant memory, texture memory is cached on-chip. The peculiarity of texture memory is that it is optimized for unpredictable but local accesses. Texture memory is accessed though Texture Mapping Units (TMUs) which are specialized hardware used for interpolating the texture data. In the context of this work, texture memory is of limited interest, as TMUs are not designed to work with 64-bit floats commonly used. One could technically still use constant memory with doubles by casting a de-referenced pointer to two 32-bit integers (of type `int2`), but this process remains janky and we don't benefit from the interpolation hardware of the TMUs.

This section concludes the general summary of the memory architecture of GPUs. Table 1 summarises the types of programmable memory and their scope [47]. Although

[9]Some architectures allow the banks to be configured to have 64-bit words.

applicable to most current GPUs, the reader should keep in mind that the model presented here is subject to variations, as illustrated by the RDNA architecture for example.

Memory	**Location on/off chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Global	Off	Yes	R/W	All threads + host	Host allocation
Register	On-chip	N/A	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On-chip	N/A	R/W	All threads in block	Block
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Table 1: Summary of the different memory types in a GPU

4 2D GPU code and optimizations

This section will be divided in three parts. We start by introducing the changes needed to move from the CPU code to the GPU code. Then we detail the optimizations made to accelerate the code on a single GPU, and finally we look the parallel performance and scaling over multiple GPUs.

4.1 Structural changes

As stated previously, computing the derivative in 2D in the original CPU code consists of 3 steps: adding the boundary integrals, adding the volume integrals, and multiplying by the inverse mass matrix. In order to recreate that process on the GPU, one would need to use two kernels. The first kernel would accumulate the contribution on the edges, while the second would add the volume terms and inverse. Both of these operations can theoretically be mapped to the GPU in a somewhat efficient way. For the computation of lateral fluxes, we can assign one thread per edge, and for the volume and mass inversion we set one thread per element. Writing two kernels, however, introduces a lot of overhead for reasons that will become apparent in the following sections. The choice was thus made to do everything in one single kernel, where one element is mapped to a thread. This requires some additional computations since the boundary fluxes are computed twice, once for element adjacent to the edge. This approach is nevertheless significantly faster than using two kernels. It is worth noting that we are only working with linear basis functions, and thus the ratio of floating point operations per second (FLOPS) per byte is relatively low. In higher order schemes where this ratio is higher, a single-kernel may not be best.

Expliquer l'overhead

When porting the code to GPU, changing the structure to a single kernel reveals a few minor challenges. Mesh elements don't usually have a list of their neighbours. Instead, lateral fluxed are tied to the edges, and moving to GPU requires the introduction of a neighbours list. One this mapping is done, we can use the three edges of a triangle as normal edges, with neighbouring elements considered to be on the right. This allows a relatively easy integration of the function computing lateral fluxed. Computing the volume integrals and inverting the mass matrix is also relatively straight-forward, as these operations are already independent from neighbouring elements.

Another peculiarity of GPUs is their low FP64 performance that makes 32 bit floats very attractive. In our case, using 32-bit floats raises a precision issue. Traditionally, we use the variables $H = h + \eta$ and $H\bar{\boldsymbol{u}}$. However, the computations involve differences between h and H, yielding values about the size of η. The problem is that η is measured in millimeters and the ocean depth H in kilometers, and the difference of scale is dangerously close to the 7 decimals of precision provided by 32-bit numbers. To remedy this problem, the GPU code was rewritten to use η and $H\bar{\boldsymbol{u}}$ as variables.

4.2 Performance metrics

Before diving into the code and the optimizations, it is worth taking some time to describe the performance metrics used to benchmark the GPU code. GPU kernels can be bottlenecked by two factors: memory or compute power. Most performance analysis will thus be centered around those two factors. The main metrics describing the efficiency of our kernel in regards to the memory and compute are respectively:

— The ratio of achieved memory bandwidth over the maximum memory bandwidth.

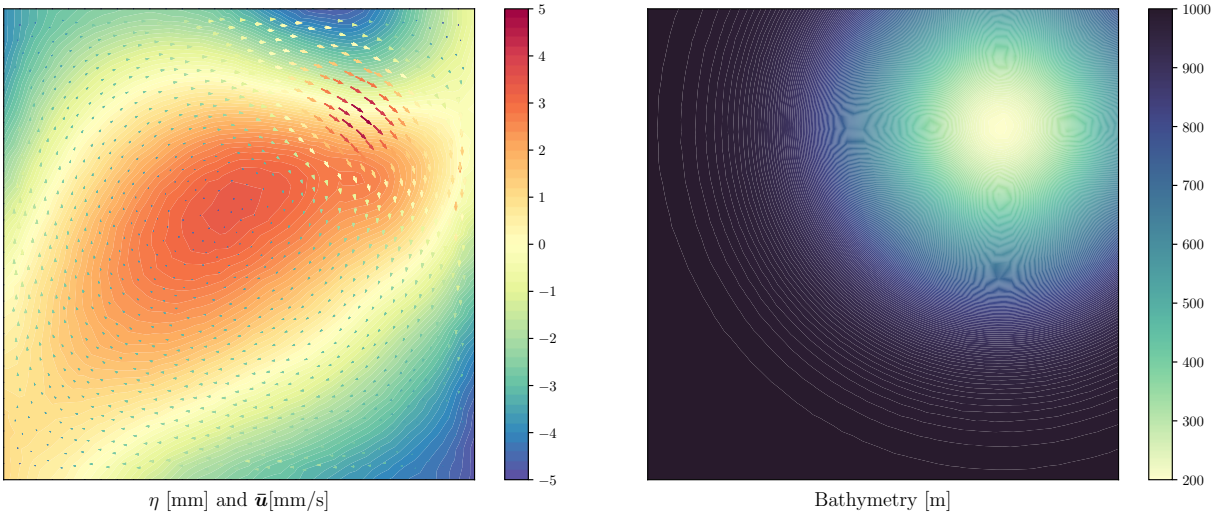
— The ratio of achieved floating point operations per second (FLOPS) over the peak FLOPS performance of the GPU.

These metrics are not the be-all and end-all of GPU benchmarks, and they can be flawed (as we sill see). In the end, what matters most is always the wall-clock time needed to complete the kernel. Nonetheless, when used appropriately, these two metrics give a good overview of how well the GPU is being used relative to its theoretical maximum.

Outside of kernels, there other aspects to keep in mind, especially when a lot of synchronisation happens. Launching a kernel takes some non negligible time. Normally, this time is hidden by the asynchronous nature of kernel launches, as following kernels are launched before the first one terminates. If, however, one waits for a kernel to finish before launching the next kernel, as is often the case with MPI synchronisation calls, the launch time can start to add up quickly. The interesting metric in this case the time in-between two kernel launches relative to the time taken by kernels.

4.3 Baseline GPU kernel and benchmark

We begin by benchmarking a single-GPU version or the code. The test case that we run uses the complete 2D model, i.e. equations (1) and (3). The mesh used for the tests has about 105k elements. That amount of triangles was necessary to hide the latency coming (mainly) from the python interface. The simulation runs 10000 time steps of 2 seconds, amounting for a bit more that 5 hours and a half. The temporal scheme used is a second-order Runge-Kutta method requiring two computations of the derivatives $\partial\eta/\partial t$ and $\partial\bar{\boldsymbol{u}}/\partial t$. Two vector additions also need to be performed. The domain for the test is square with sides of 1000 km, and the bathymetry shown in Figure 10b consists of a 800 meters high Gaussian hill on top of a 1000m-deep ocean floor. In terms of hardware, all tests were performed on a machine equipped with an 8-core AMD Ryzen 7 2700X processor and one or two NVIDIA RTX2080 GPUs.



η [mm] and $\bar{\boldsymbol{u}}$[mm/s]

(a) Expected elevation η and average speed $\bar{\boldsymbol{u}}$

Bathymetry [m]

(b) Test bathymetry

Figure 10: Overview of the test simulation. Note that figures were made with a reduced mesh size, but are visually identical at higher resolutions.

4.3.1 Comparison between CPU and GPU

The first version of the GPU code is simply a translation of the CPU adapted into a single kernel as described above. Without any particular optimizations, this code is already significantly faster than the CPU running 8 MPI processes (Figure 11). The GPU code

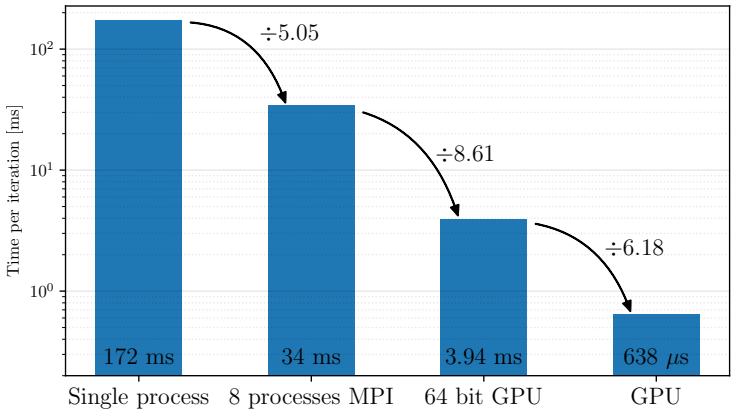


Figure 11: Comparison between the initial GPU code and the CPU code.

running with double (64-bit) precision is close to 9 times faster than the parallel CPU code. This could come as a bit of surprise, since consumer GPUs such as the RTX2080 should have a really low double-precision throughput. Looking at the specs confirms that the GPU has indeed a ratio of FP64/FP32 of $1/32^{\text{th}}$: the peak FP32 performance is around 10 TFLOPS, against 315 GFLOPS of FP64 performance. However, comparing that number to the theoretical maximum of the CPU[10] (118.4 GFLOPS) partly reveals why the GPU is still faster. A consumer GPU beating the traditional processor in this instance is a testimony of how powerful graphics cards have become.

Unfortunately, looking at the double-precision performance provides little value in regards to what the bottlenecks are in a realistic setting. HPC GPUs tend to be good at FP64 performance, thus shifting the critical parts somewhere else. For this reason, all of the performance testing and optimization following will be done with FP32 precision. To allow fair comparisons between CPU and GPU, one should thus roughly double the CPU performance, but is barely relevant in this work. The GPU code with 32-bit numbers is around 53 times faster than the CPU code, and if that number were 25, the conclusions would not change.

4.3.2 Assessment of the GPU code

Out of the 638 µs needed to complete one iterations, about 57 µs were spend in the vector additions, while the rest was spent in the kernel computing the derivative. Our focus is thus the derivative kernel. Profiling it reveals two interesting things. First, the memory seems to be used intensely. The average memory throughput during the kernel execution is at around 46% of its theoretical peak. Having a high memory throughput is a generally a good thing, as it means that the GPU is being fully utilized. We shall see, however, that it can also mean that a lot of data is being needlessly moved around. The second interesting observation is that the kernel achieves 13% of its FP32 throughput and *19%*

[10]The theoretical peak FP64 performance can be computed as #cores × IPC × Frequency. The best value possible for the IPC (Instructions Per Clock) is 4 FLOPs with 256-bit vector extension. At a frequency of 3.7GHz, we get 118.4 GFLOPS of theoretical peak performance.

it its FP64 throughput. This is indeed surprising, as all functions have been translated to their single-precision counterpart, and will be explained in section 4.5.

4.4 Mesh reordering

We begin by tackling the memory in more detail. A powerful tool for detailed memory analysis is the memory chart as shown in Figure 12. The memory chart illustrates the

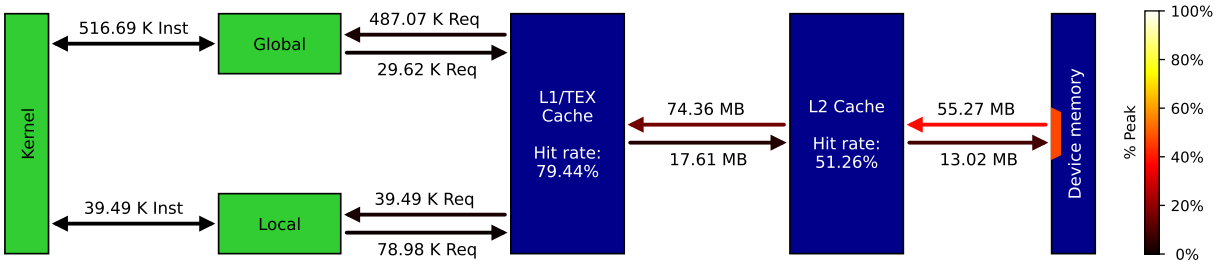
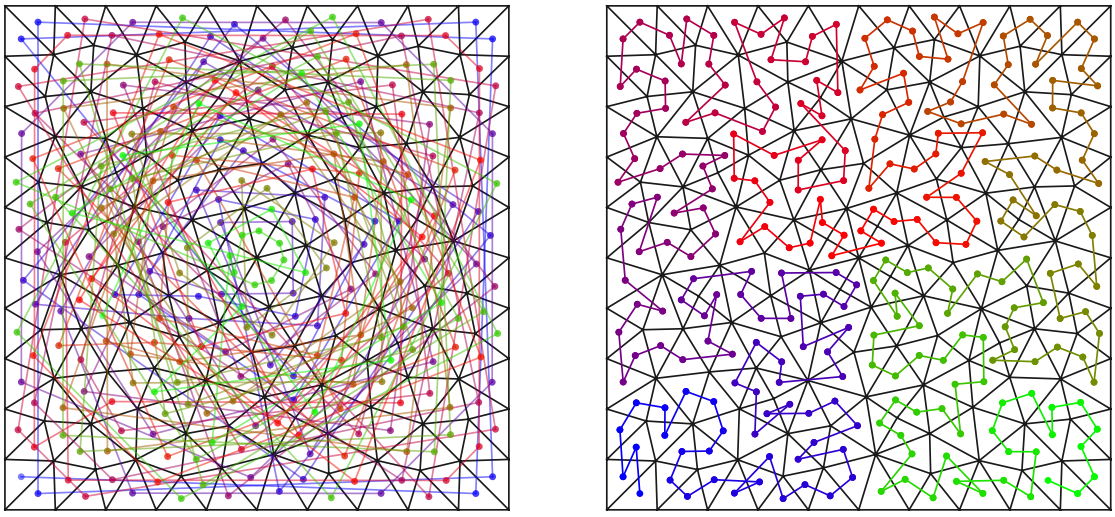


Figure 12: Memory chart of the initial unoptimized kernel.

flow of data between the various caches and the kernel. Arrows pointing left are read requests, while arrows pointing right are write requests. Not counting the two leftmost bidirectional arrows, all the arrows are coloured according to the ratio of achieved memory bandwidth over peak bandwidth.

Looking at Figure 12 reveals two problems with the kernel. First, the L2 cache hit ratio very relatively low. Secondly, and perhaps more importantly, the kernel uses a non negligible amount of local memory. Remember that local memory is in fact global memory allocated to make up for the lack of registers. Thus, using any amount of local memory will introduce a heavy overhead. Tackling one problem at a time, we first sort the cache issue. The low cache hit ratio can be explained by the lack of locality in. Remember that each



(a) Initial order (b) After reordering

Figure 13: Mesh order before and after sorting the elements following a Hilbert curve.

thread is assigned to one element, and during the computation, lateral fluxes need to be computed with neighbouring elements. Thus, if physically neighbouring elements are also close to each other in memory, we can expect the kernel to exhibit locality. Conversely,

if elements are ordered randomly, the memory access pattern will be random, and thus operate at the slowest speeds. Initially, mesh elements are not in any particular order, as see in Figure 13a. This explains the poor cache performance. The bloc nature of memory transaction also explains the high memory throughput. For each bit of data needed, a whole 128-byte long bloc is copied from global memory. This also exemplifies why the raw throughput is not enough of a metric to asses the quality of a kernel.

The solution to fix the lack of locality was to reorder the triangles along a Hilbert curve, as illustrated in Figure 13b. The Hilbert curve is member of the family of space-filling curves. I provides a way to map the unit interval to a unit square. What makes the Hilbert curve interesting is its property that two points close together in 2D space will often also be close in the unit interval. Thus, if we organise the triangles along the curve, two neighbouring elements have higher chances of being close together. Looking at the

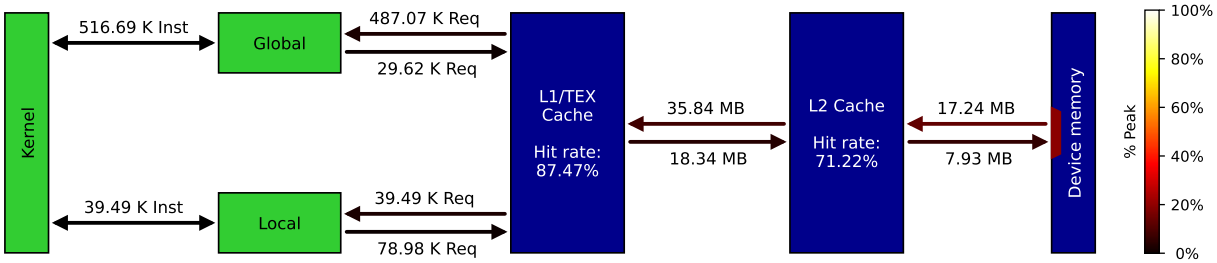


Figure 14: Memory chart after reordering elements along the Hilbert curve.

updated memory chart in Figure 14, we see a clear reduction the bandwidth going to and from global memory. It goes from around 46% of of the peak bandwidth to 23%. The cache hit ratio is also much better, at around 71% instead of 51%. These improvements also translate to a faster simulation speed, increasing by around 14%.

It is worth noting that although very useful for simple geometries, the Hilbert curve may not work with highly irregular meshes or non-planar meshes. For such applications, other heuristics such as [48] exist and tend to produce better results than space-filling curves.

4.5 Miscellaneous sources of performance degradation

Before moving on with the problem of local memory, we shall address reported 19% of FP64 utilisation mentioned in section 4.3.2. As stated previously, no variables were of type double and all the special functions used their 32-bit version. The issue faced here is in fact due to the use of double literals. Expressions such as:

```
1 float a = 12;
2 float b = 16;
3 float c = 0.5 * (a + b); // performs a double-precision operation
```

convert `(a+b)` to a double before performing a double precision multiplication and casting the result back to a float. Solving this problem was as trivial as adding an `f` at the end of floating-point literals[11]. This single change alone improved performance by 23 is the ways in which inted[tabsize=4,frame=single,fontsize=,framesep=2mm,bgcolor=codebgcolor,breaklines,linenos]cCode/f

[11]Here, a good argument could be made that the conversion should be automatically done by the compiler, since the result is casted back to a float anyway

Another surprising contributor in the poor performance of the kernel was a conditional `print` statement. IO is notoriously slow, but the peculiarity in this case was that the statement was never reached, yet removing it improved boosted the speed by an additional 20%. The change comes from the amount of is the ways in which eeded in the event where the `printf` should be executed. Removing the IO decreased the number of registers needed per thread from 201 to 129. This change is accompanied by an increase in the theoretical occupancy, jumping from 25% to 37.5%.

4.6 Using shared memory

One last problem that subsists, is the use of local memory, meaning that there are still not enough registers available. This is further confirmed by looking at the warp state statistics. Most instructions

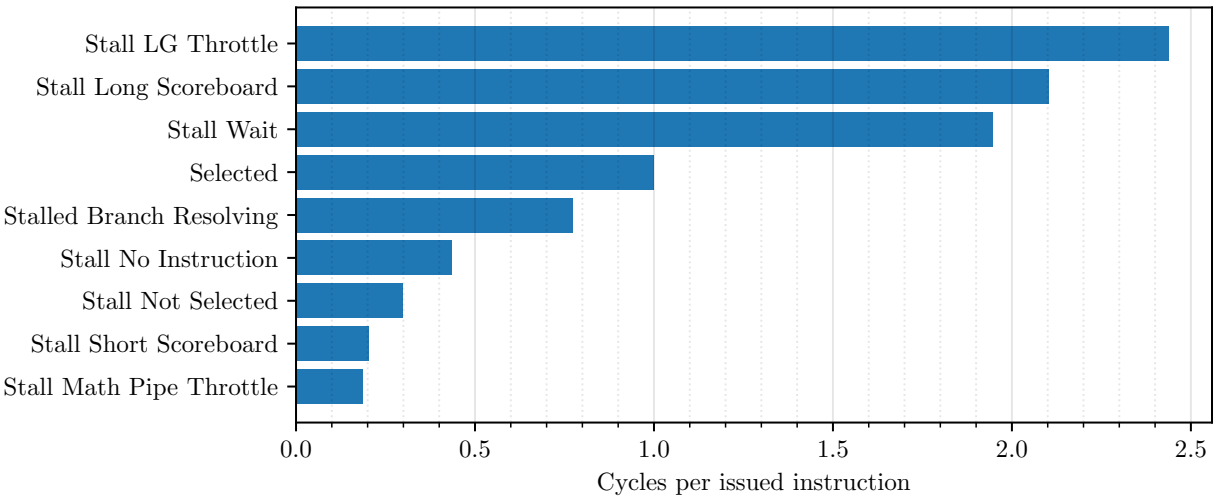


Figure 15: Warp state statistics normalized per issued instruction. This figures highlight the amount of cycles a warp spent stalling for every instruction issued to the warp. Higher values mean that the warp spends more cycles doing nothing, with the reason being specified by the row label. This figure only show the top stalling contributions.

need more than one clock cycle to complete. Consequently, many warps spend a majority of their time being but stalled. Figure 15 shows the top contributing factors that cause the warps to stall in the derivative kernel. Each bar in the figure is a different reason. Their length answers the question "For each issued instruction, how many cycles does the warp spend stalled for that one reason?" Focusing on the top three contributing factors, they can be understood as:

C'est tendu à lire cette section, non?

— Stall LG Throttle: This means that the warp would like to issue a memory request to the L1 cache of the Local or Global memory. However, it is unable to do so because the instruction queue is full. It is a result of many memory requests being sent to the cache trying to access Global or Local memory. In this kernel, the number of global accesses is fairly reduced, and this reason being in the top means that we hammer the Local memory.

— Stall Long Scoreboard: This means that the kernel is waiting for data to come back from the L1 cache. It is a fairly common cause stalling. Reducing the Long Scoreboard stall requires improving the memory access pattern to reduce the number of L1 requests, or moving some frequently used data to shared memory, skipping the L1 cache.

— Stall Wait: This means that the warp is waiting on a fixed latency instruction to terminate. It generally show up as a top contributor in already well optimized kernels. Not much can be done to reduce the amount of stall waits other that increase the occupancy, and thus hiding the stall wait by executing other warps.

To reduce the amount of local memory used, we can put part of the variables used in shared memory. To relieve the local memory as much as possible, it makes sense to put frequently used data in shared memory. In the context of the 2D kernel, there are a number of data items associated with each node that are accessed frequently and that we moved to shared memory. These are the coordinates x and y,

the bathymetry h, the density ρ, the diffusivity coefficient κ, the Coriolis term f and the external friction $\boldsymbol{\tau}$. Those make up 8 scalars fields, that all take a value on the three nodes of a triangle, yielding a total of 24 scalars. This is sufficient to reduce the pressure on registers to the point where no more data spills into local memory, as show in Figure 16. As one could expect, this significantly reduces the memory

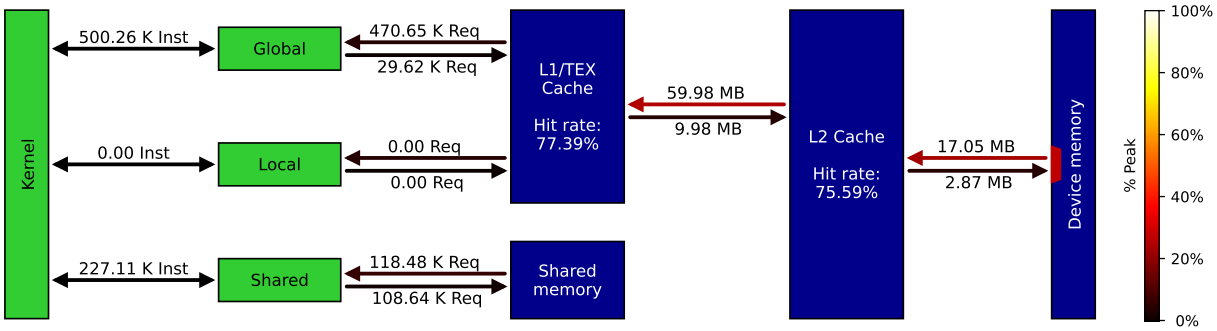


Figure 16: Memory chart of the 2D derivative kernel using share memory.

movement, especially the amount written to global memory. In turn these changes bring another 12% speed improvement.

On top of these gains, using shared memory allowed some improvements in the memory access pattern to be made. Since the impact of these changes was very minimal, they were not discussed here. Instead, the in-depth analysis of the memory access pattern is the topic of section 4.7.

4.7 Layout changes and maximizing coalescence

One important metric that we left aside until now is the efficiency of data transfers. Data is loaded from global memory in 128-byte chunks, but not all of the loaded data is part of the requests. Profiling the shared-memory version of the kernel reveals that only about 20% of the accessed data ends up being useful, while 80% is a by-product of uncoalescent memory accesses.

There are two reasons that explain this sub-optimal access pattern. The first one is the very nature of finite elements methods. Meshes are unstructured by design, to allow more flexibility, but that inevitably lower the regularity of the computation. Aside from reordering the mesh, as we already did, not much else can be done. The second reason has to do with the order in which data is stored in memory. For

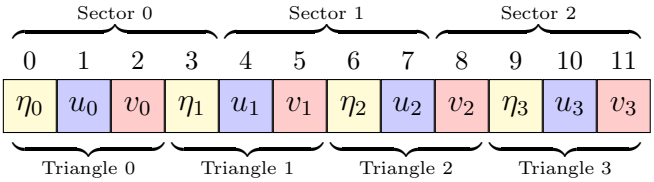


Figure 17: Simplified representation of how fields are structured on the host.

CPU codes, we favour locality, and thus it makes sense to organise the data by element first, then by component inside of the element, as illustrated in figure 17. While optimal for a sequential access, this layout is severely undermines the capabilities of the GPU hardware. Taking figure 17 as an example, imagine that we have 4 threads and that memory transactions happen in sectors of 4 numbers. Assuming that a kernel access η first, then u, then v, we would need to make 9 queries: The four values of η are spread across all three sector, thus we need 3 transactions for η. The same goes for u and v. Contrast that with the layout show in figure 18. There, we would use one transaction to load all the values of η, one for all the values of u and one for v. This phenomenon becomes worse if instead of three components, we have more, like in our case with 8 fields. Larger sectors also contribute to the degradation of performance with a CPU-oriented layout.

The next logical step in the process of optimizing the kernels is thus to transpose all of the fields so that adjacent values in memory correspond to the same feature in adjacent triangles. Figures 17 and 18 are a bit oversimplified though, as in reality arrays have 3 dimensions: elements, nodes, and fields. Taking

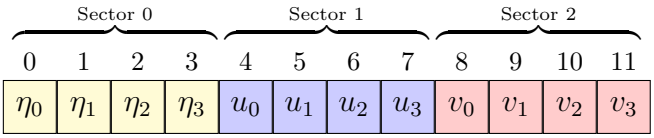


Figure 18: Simplified representation of the ideal structure of 2D fields on the GPU.

the position as an example, if we have n triangles, the position array will have a shape of $(n, 3, 2)$: n triangles, 3 nodes per triangles, and $N_F = 2$ components (x and y) per node. In the host code, accessing the field `f` of node `k` of element `e` is done via:

```
v = array[3*NF*e + NF*k + f]
```

This layout follows the hierarchy Element $\longrightarrow$Node $\longrightarrow$Field. For the GPU code, we roll this hierarchy around to get Node $\longrightarrow$Field $\longrightarrow$Element. Accessing the field `f` of node `k` of element `e` on the GPU changes to:

```
v = array[n*NF*k + n*f + e]
```

Implementing these changes yields the expected effects. The efficiency of data transfers, jumps from 20% to 55%. 55% may still seem like a small number, but in the context of unstructured we should expect these sort of limits. The run-time also drops by around 20%, and with that the achieved FP32 jumps to around 35%.

This new version of the code also brings an interesting change in the memory behaviour. Looking at figure 19, we notice that the L1 cache hit ratio has dropped dramatically to 23.5%. If we were programming for the CPU, this could be a major red flag that something is wrong. In this case, however, it means that we did a great job accessing the memory that we weed only once. This, again, illustrates the need for a good understanding of the profiling metrics and their use cases, especially in memory-bound kernels such as this one.

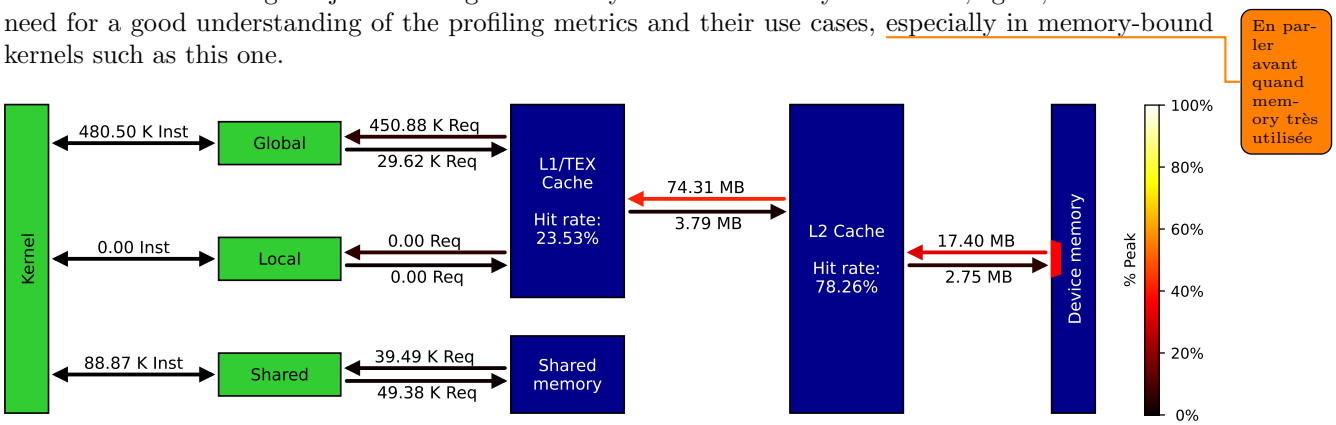


Figure 19: Memory chart of the 2D derivative kernel using a GPU-friendly data layout.

The remaining bottlenecks in after these optimizations are mostly due to the necessary transfers of data. The regions of the code that lead to the highest amount of stalling are the initial loading of the the solution (η and $\bar{\boldsymbol{u}}$) and the other fields (mesh coordinates, diffusion, Coriolis, friction). These accesses can hardly be improved, as they are already fully coalescent. The second hot spot happens when threads load data from the neighbouring elements. Due to the unstructured nature of the mesh, these accesses can not be fully coalescent, and the best we can do is hope for is that data is already loaded in the caches. Here, not much more can be easily done, as the Hilbert curve gets us pretty far already. An interesting idea to explore is to use the shared memory inside of a group when available. At this stage, shared memory is only used as an explicit cache specific to each thread. Implementing this idea is relatively easy. When threads access data from a neighbouring element, they check is that neighbour is in the group. If so, the data is retrieved from shared memory, and, if not, from global memory. We did some limited tests of this idea, but the gains were only at around 1%, which is was not enough to dedicate a

lot of time into it. The limiting factor was that while we gained some performance in memory transfers, we lost some to branch divergence. We used bloc sizes of 128 and 256 (with similar results) and the warp size is 32. 32 is fairly large compared to 128, and consequently most warps always had neighbours inside and outside of the thread block. Thus, even if 90% of the neighbours were readily accessible, the warp still has to wait for the remaining 10%. This does not mean that shared memory from the neighbours is a dead idea. To make it effective, we need to reduce the branch divergence. That can be achieved by reordering elements inside of a bloc. If we put the elements that communicate with external elements at the beginning of the bloc, we ensure that only the first warps will have to load data from outside of shared memory. In addition to that, we can "rotate" the order of the vertices of boundary triangles such that the first edge that is processed is the one facing the outside. This would mean that when computing the fluxes for the first edge, the first warps may have some branch divergence and need accesses outside from shared memory. The benefit would be that for the remaining two edges, all warps access data from shared memory.

Finally, a summary of the different optimizations is presented in Figure 20. The optimizations that

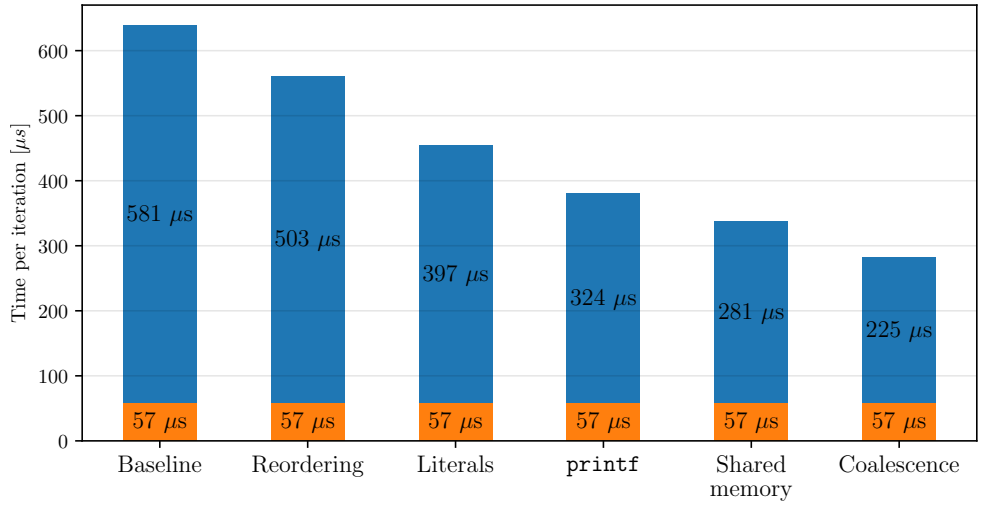


Figure 20: Overview of the different optimizations performed. The blue bars show the time spent computing the derivative and the orange bars account for the vector sums.

we applied can be classified in 4 categories that each illustrate a key aspect of the GPU. It is worth noting that the order in which they were applied is a reflection of the learning process rather than a method. In a decreasing order of importance, we have

— Using coalescent memory accesses: this is by a wide margin the most impactful metric on memory-bound kernels. In our benchmarks we only see a limited impact since our kernel was already optimized and exhibited a lot of locality. If transposing the data for coalescence was the first optimization done, we would likely have seen a stronger effect.

— Mesh reordering and locality: When coalescence is not possible, optimizing for caches is also worth it, perhaps even more than in CPU codes due to the very high global memory latency.

— Using shared memory: Shared memory can be used as a cache to reduce the number of registers and local variables, or as a cache to improve the memory access pattern. This makes shared memory a good tool to improve coalescence or manage locality explicitly.

— Miscellaneous: This category refers to the unexpected problems with the double literals and the print statements. It's worth keeping an eye open for those issues as they can be costly, but they easy to solve and avoid.

4.8 Enabling multi-GPU with MPI

Now that we have a good performance on a single GPU, the goal is to extend it to multi-GPU setups. The parallelism in SLIM is handled through domain decomposition, and communications happen with a layer of "ghost elements". Figure 21 shows an example where the domain is split in two with ghost elements

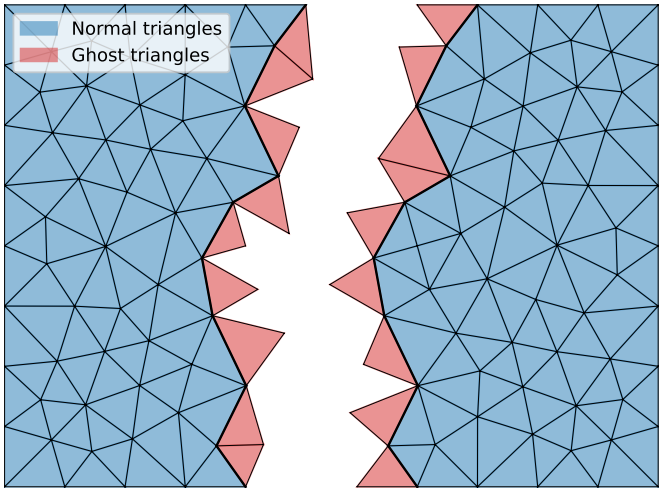


Figure 21: Example of a square domain decomposed into two partitions. Each partition has access to some of the elements of the other, aka ghost elements.

depicted in red. Ghost elements are triangles owned by another partition but that are copied at the boundary to allow computations that need the neighbours. When processes do any sort of computation that changes the values of their elements (in blue), the computation is followed by what we call a "halo exchange". Each process will send its updated values to neighbouring domain, and will receive the updated values of its ghosts edges from the other domains. The halo exchange is performed using non-blocking MPI communications to overlap the sending and receiving of data to all neighbouring processes. Each process then waits for ll the communications to terminate.

In order to communicate between GPUs, we need to build a similar process. A notable difference compared to the CPU code is that GPUs can sometimes be linked directly to each other in a system, with technologies such as NVIDIA's NVLink and AMD's Infinity Architecture. However, those are not always present, and in order to maximise compatibility, we will start by only using traditional[12] MPI.

Elements in the mesh are organised such that ghost elements are located contiguously, after normal elements. Ghost elements are also sorted by neighbouring partition, meaning that no data movement needs to be done while receiving. The GPU on the other hand required additional data movements before and after the communications occur. Since the data was transposed to improve coalescence, different fields of a given element are not contiguous in memory. Thus, we need to gather them to a sending buffer, and scatter the receiving buffer. On top of that the data has to be copied to and from main memory before in order to use MPI, and copying a single continuous buffer is always faster than multiple accesses. The complete process to compute the derivative is thus organised as follows:

1. Compute the derivative with the kernel `dudt`.
2. Gather all the values that need to be sent in a GPU buffer.
3. Copy the buffer to system memory.
4. Proceed to send and receive the data with MPI.
5. Copy the receive buffer to GPU memory.
6. Scatter the incoming ghost values from the receive buffer to the right places in global memory.

To evaluate the performance in a multi-GPU setup, we will look at two metrics. The first one is the median time per iteration. When we run some code on multiples GPUs, the chances that one GPUs is needed by another application increase greatly, introducing significant variations in the mean run time. This goes to the point where the standard deviation is regularly larger than the mean run time. The median time on the other hand is highly consistent from run to run, as it is not disturbed by large interruptions in the execution. The second source of data that we will look at is a trace of the GPU kernels and MPI calls. We are in particularly interested in the timings of the kernels and MPI. To give some context before diving into at the performance of communications, it is worth taking a look at the

[12] As opposed to GPU-aware MPI, which is the topic for section 4.11.

timings with a single GPU. Figure 22 shows a typical Runge-Kutta iteration, were we make two calls to `dudt` and add these results to the solution. As we expect, `dudt` take the majority of the time. We also

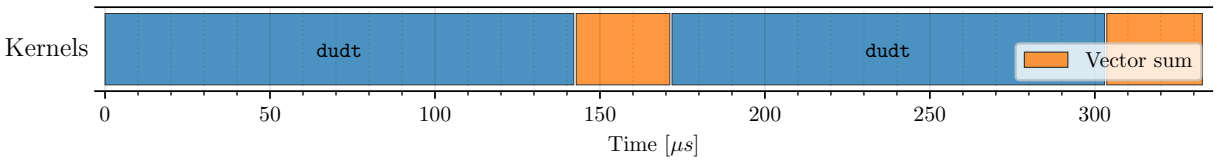


Figure 22: Timeline of a complete RK2 iteration with a single GPU.

note that almost no time is wasted in between kernels. This is because, even though launching a kernel is expensive, the asynchronous nature of kernel launches allows the GPU to en-queue many kernels in advance. The timings of Figure 22 were captured with the last version of the `dudt` kernel whiwh takes about 282 µs per Runge-Kutta iteration. The value of around 330 µs in the figure is due to the profiling overhead.

4.8.1 Performance analysis of the multi-GPU code

Running the Multi-GPU code using the approach described above, one would expect to be a bit less that two times faster, compared to a single GPU. The reality if very different, however, as the median run-time jumps from 282 µs to 663 µs. The causes for this massively increased amount of time become apparent in Figure 23. It shows the execution timeline of a complete computation of the derivative with 2 GPUs. Following the procedure described above, we start by computing the derivative in the normal

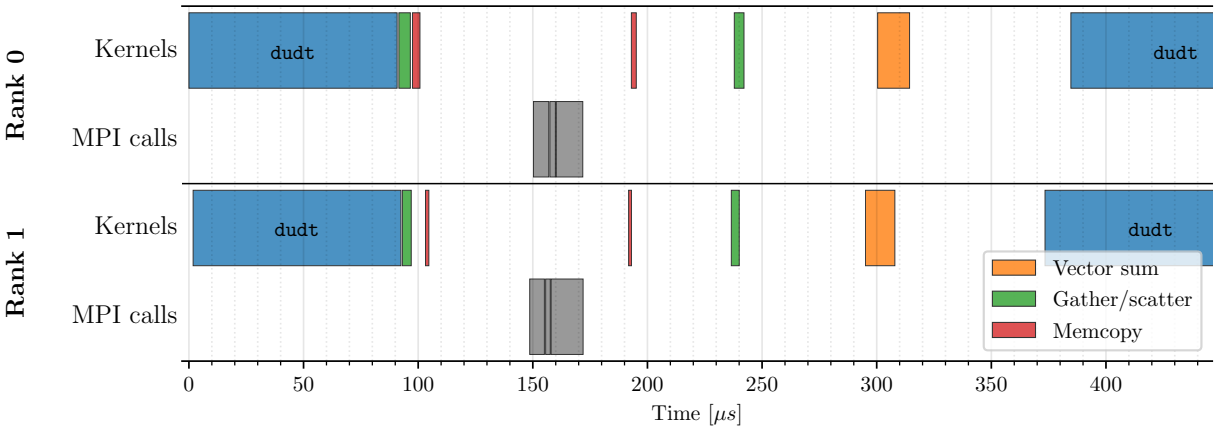


Figure 23: Typical timeline of one computation of the derivative with 2 processes.

nodes with the `dudt` kernel. We then gather the values that need to be sent to other processes in a buffer (show in green), and copy these values back to main memory (in red). Notice that compared to `dudt`, both of these operations are very fast. After that, MPI is used for the communications. Once the communications are over, the data is copied back to the GPU (in red), then scattered to the intended location in the GPU memory (in green). After that, the value of the derivative is added into the temporal scheme (shown in orange), and the process repeats.

The big difference in Figure 23 compared to Figure 22 is that there are significant periods of time where the GPU does nothing. All the idle periods can all be loosely be explained by explained by "launching a kernel takes time, python is slow". And while this was also true in the single-GPU program, the latency could be hidden since no synchronisation is needed between kernel launches. Going into more detail, what happens in Figure 23 is:

0–100 µs The `dudt` kernel en-queued previously runs on the GPU. While it runs, we en-queue the `gather` kernel (in green) and the memory copy back to system memory (in red). Remember that memory copies are synchronous by default, thus the CPU now waits until the transfer is complete.

≈100 µs The `dudt` kernel completes, the `gather` kernel (green) starts and completes quickly after, and the memory copy is initiated (in red).

110–150 µs Here two things happen that cause an idle period. First, the CPU waits for the signal that the memory copy is done. This introduces a latency of around 5 to 10 µs. The rest of the time is what is needed to make the MPI call. In theory, launching the MPI communications should be extremely quick, taking much less than a microsecond. The problem that we face is that MPI is called from python through the `ctypes` library. `ctypes` adds a non negligible amount of latency that roughly grows linearly with the number of arguments. This explains why we see nothing being done for another 30 µs.

150–175 µs The MPI communication happens (in gray).

175–195 µs This is the second stalling zone. The time is spent in the python interface and in the cost of en-queuing the memory copy operation.

≈195 µs The incoming data from neighbouring partitions is copied back to the GPU (in red).

200–245 µs The CPU waits for the memory copy be be complete. Then then `scatter` kernel in en-queued, with the now expected latency from python and the queuing process.

≈245 µs The `scatter` kernel runs.

250–300 µs The vector addition kernel in en-queued, with all the problems highlighted above.

300–320 µs The vector addition kernel runs (orange). In the mean time, the process of en-queuing the next `dudt` kernel starts.

320–380 µs The `dudt` kernel is en-queued, and we go back to the beginning of the cycle.

The clear conclusion of this analysis is that we need a way to bypass the Python latency, and ideally a way to hide the kernel launch latency.

Nevertheless, aside from the latency problems, notice that the `dudt` and sum kernels perform roughly as expected. The `dudt` kernel executes in ≈90 µs instead of ≈140 µs, and the sum runs in ≈12 µs instead of ≈21 µs. Note the multi-GPU profiling includes more metrics and the profiling overhead might be a bit higher.

4.9 Moving away from python

Launching every kernel individually from Python is great in terms of flexibility, but unfortunately not a viable option for the 2D code. To reduce Python calls as much as possible, we re-implemented the 6 steps described in section 4.8 directly in `C`. On top of that, the complete RK2 temporal scheme was also implemented in pure `C` to hide the latency in-between different calls to the derivative. This change already massively improves the communication latency, as the median time for a RK2 iteration drops to 240 µs. Looking at Figure 24, we see that latency is being severely cut severely in multiple places compared to Figure 23. Most notably, once the buffer is copied from the GPU to main memory at around the 90 µs

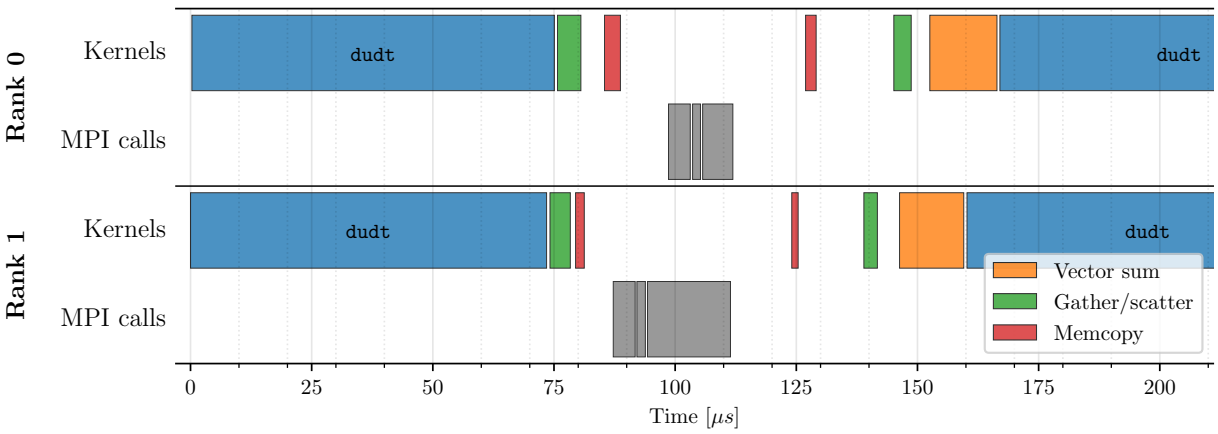


Figure 24: Timeline of the computation of the derivative directly from the `C` code.

mark (in red), the MPI calls happen almost right away. Next, we observe that the idle time between the vector sum (orange) and the following `dudt` kernel is almost inexistent. This means that the time taken

by the sum kernel is enough to hide the cost of launching the `dudt` kernel. The three remaining idle regions happen after the MPI communications, after the memory copy to the GPU (in red at ≈125 µs), and after the `scatter` (in green at ≈145 µs). In all three cases, they are caused by the high amount of time needed to launch a kernel or memory copy. After MPI, the time is spent launching the memory copy, after me memory copy it is spent launching the `scatter` kernel, and after the `scatter` kernel time is spent launching the vector sum.

Although not optimal by any metric, this implementation of GPU communications is still usable on large meshes. Figure 25 shows the scaling behaviour of both the single-GPU and 2-GPUs codes. The

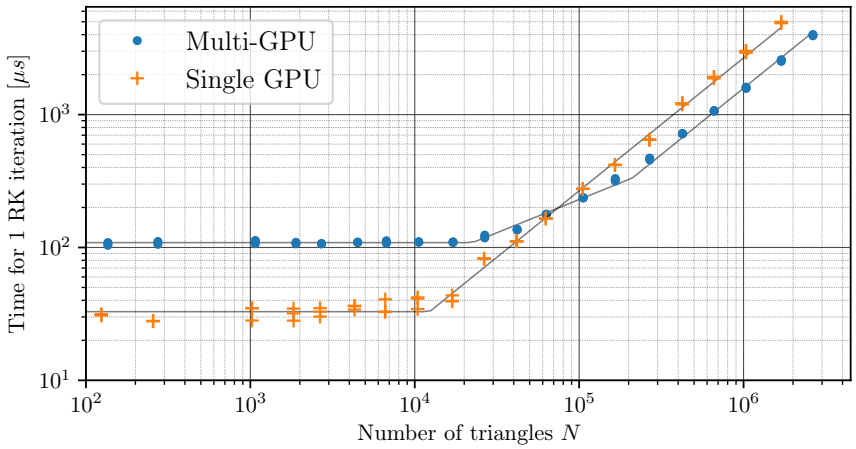


Figure 25: Comparison of the scaling with one and two GPUs

main take-away from this figure is that the general shape of the graphs is relatively similar with one or two GPUs. Up to a certain number of triangles, the run-time in constant, dominated by the cost of kernel launches. After a certain value, the behaviour changes and the time scales linearly with the number of elements. The only difference in is the transition between these two regimes. With a single GPU, there is no transition period, as we could expect. The time spent in the kernels is either below or above the launch time. In a multi-GPU setup, on the other hand, there can be a soft transition. It reflects the presence of the MPI calls. Since the size of the communications is proportional to the size of the boundary, we expect the time taken by MPI calls to be proportional to $\sqrt{N}$, with N denoting the number of elements. Consequently, at small sizes, the kernel launches dominate, at medium sizes, the MPI communication dominates, and at large sizes the kernels become the bottleneck. Therefore, good model for the run time per iteration is:

$$T = \max\left[K, \quad \sqrt{N} \cdot C_{\text{comm}}, \quad N \cdot C_{\text{per-element}}\right]$$

The solid lines in Figure 25 show the accuracy of this model. They also give good estimates on the necessary size to enjoy a linear scaling and when the single GPU will be faster. In the test system, we need around 80K triangles for the parallel to be relevant, and around 250K for it to be competitive.

4.10 Overlapping computations and communications

As it stands, the multi-GPU code exhibits decent weak scaling properties, but a relatively poor strong scaling. This is a problem, since most meshes will not have hundreds of thousands of triangles, especially when we later add the third dimension. To improve the scaling, we will need to hide the communication costs by overlapping communications with computations. To do so, we need to compute the derivatives in the elements adjacent to the ghosts first so that we can send then as soon as possible. We will refer to these elements as "fake boundary elements". "Real" boundary elements denote the elements that touch a physical boundary. The remaining elements shall be called internal elements. The hope is that once the computation is over in internal elements, the communications are done and we can move on to the next iteration. From a programming point of view, this requires two changes: the `dudt` kernel must be split in two, with one part responsible for the boundaries and one for the interior. The other change and we need to overlap communications and computations in the GPU.

Before splitting the kernel, it is also convenient to reorder the mesh so fake boundary elements are located contiguously in memory. Since we committed to reordering the elements and put fake boundaries

at the end, the decision was made to also put real boundary elements at the end. This allows the internal elements to not have to deal with boundary conditions. Therefore, it slightly lightens the interior kernel, making up for the added launch cost. Figure 26 illustrates the three types of elements as seen from the

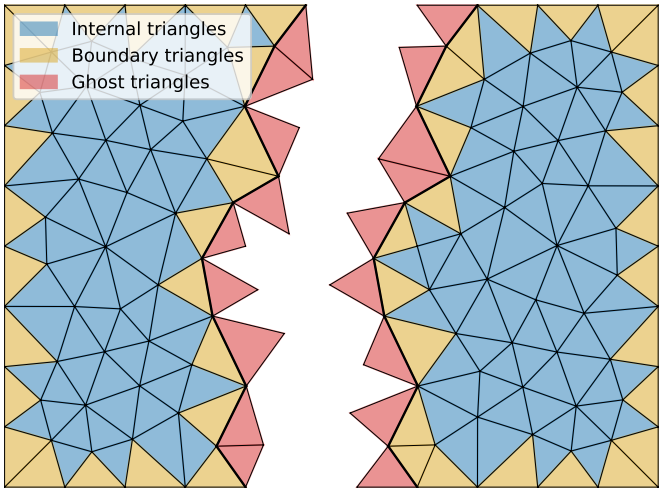


Figure 26: Illustration of the three classes of elements.

kernels: boundary elements are computed first, internal elements come next, and ghost values com from the other partitions.

The other significant modification is the use of streams. Recall that every operation on the GPU is tied to a stream, and that streams run sequentially. To run kernels and communication in parallel, we will need a second stream that will be used for communications. Creating a second stream gives rise to two new challenges. The first is that by default, we have no control over the order of execution in different streams. This sometimes leads to kernels being issued before but executed after others. To mitigate this problem, we can assign a priority to a give stream, which mostly solves the issue. The second challenge is that streams are independent by default, and we need some sort of synchronisation between them. To synchronise streams, CUDA and HIP provide functions that wait for a stream to complete, or for the whole device to finish its operations. However, these functions are used when the CPU needs to wait the GPU, and it is important to note that *these should not be used* to synchronise two streams. Instead, we can use events. Events are markers that are placed on a stream just like kernels or memory copy operations. When a stream reached an event, it will "complete the event", meaning that the event will capture various metrics relative to the stream. Their interest in this context is that other streams can wait for events to be complete. Since the event creation and assignment to a stream is done asynchronously, events allow us to express complex dependencies in the data flow in-between streams in advance.

The sequence of function calls in the computation of the derivative is shown in Table 2. We split the work in two streams, keeping default stream as the compute stream, and creating a communications stream with the maximum possible priority. Implementing these changes further hides the communication latency, dropping the time per iteration to 185 µs instead of 240 µs. Looking at Figure 27 reveals how the functions from table 2 overlap. As we expect, the boundary computations start first, in the `dudt_bnd` kernel. In the example show in Figure 27 the kernels `dudt_bnd` and `dudt_inten` barely overlap. In practice, overlapping of these kernels is fairly common. What may seem surprising is the long delay between the end of `dudt_bnd` and the start of the `gather` kernl. This is caused by `dudt` occupying the GPU, forcing `gather` to compete for the hardware. This also explaing the delay before starting the memory copy (in red at ≈60 µs). The rest of the timeline is mostly expected. There is a small bu unavoidable latency after the memory copy and before starting the MPI calls. After the communication with MPI, the cost of launching the memory copy back to the device explains the inactivity of the GPU. The launch of the `gather` and sum kernel are responsible for the two other empty regions.

4.11 GPU-aware MPI

Some implementations of MPI, such as OpenMPI, have a built-in support for GPUs. In practice, it means that we can use GPU pointers directly with MPI, and MPI will handle the memory transfers and

Communication stream	Compute stream
Launch the computation in the boundaries	
Gather all the values that need to be sent in a GPU in a buffer	
Copy the buffer to system memory asynchronously	
	Launch the computation on the internal elements
The CPU waits for the transfer of the send buffer to terminate	
Communicate with MPI	
Copy the receive buffer to the GPU asynchronously	
Scatter the incoming ghost values	
	Ensure that operation issued on the communication stream terminate before any new operation on the compute stream

Table 2: Different steps done in each stream when, computing `dudt` with multiple GPUs.

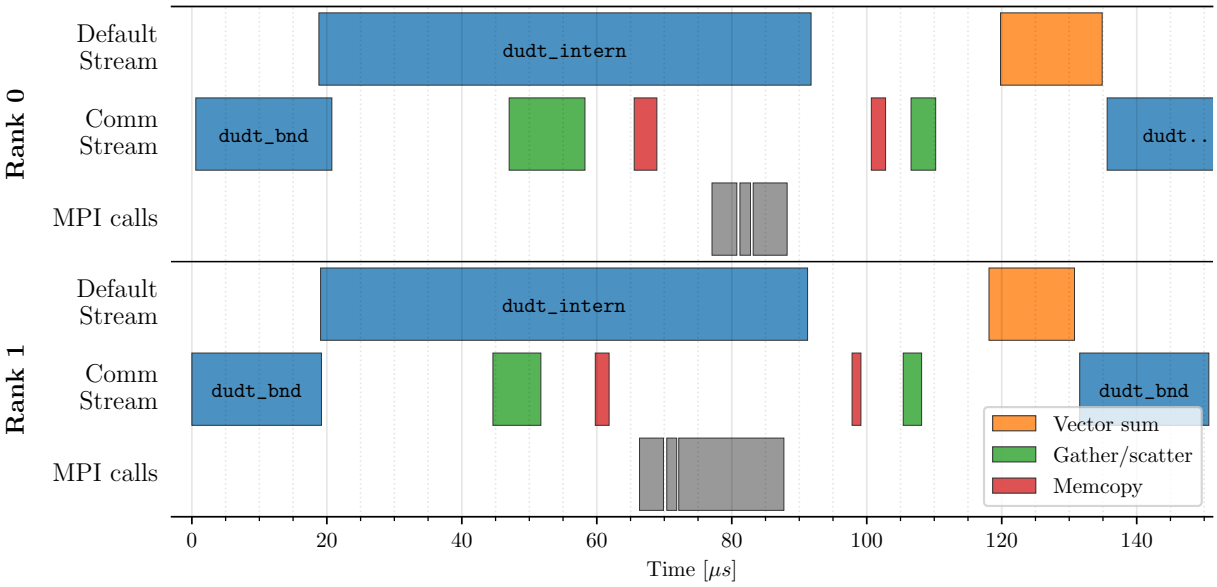


Figure 27: Timeline of the computation of the derivative with multiple streams.

synchronisations. The advantages of using GPU-aware MPI implementations are that if two GPUs happen to be connected through a specialized interface, such as NVLINK, then MPI will use that interface. The interface of GPU-aware MPI implementations is also the same as any other MPI implementation, which makes very easy to use.

Under the hood, when MPI is called with GPU pointers, communications happen in three steps. First, the MPI system will issue a device-wide synchronisation call to the GPU, waiting for all previously queued operations to terminate. Then, the communication will be issued into a dedicated stream. From there, the data will be copied using the best available hardware path from the source to the destination. Finally, the CPU will wait until the transfer is completed before moving on.

One of the strongest points of GPU-aware MPI is also one if its most restrictive characteristics: it uses the MPI interface. This means that it lacks the fine-grained control specific to GPUs. For example we can't manually select the stream MPI will use for its operation. It is also not possible to overlap memory transfers and computations. These limitations resulted in a median run-time of 190 µs per Runge-Kutta iteration, which is a bit slower than the manual asynchronous transfers detailed in section 4.10. But still, the fact that GPU-aware MPI is competitive with our best version while being completely synchronous shows that there are still possible improvements.

5 Initial benchmarks and challenges with 3D kernels

Compared to the 2D model, the complete 3D model is significantly more complex. The equations solved can be divided in three categories with increasing levels of complexity. The simplest are equations that are fundamentally 2-dimensional such as equation (8). Then we have the static equations for w (6) and $\boldsymbol{q}$ (12). For both of these fields, we know their vertical derivative, and we know the values at the top or the bottom. Getting the value of w or $\boldsymbol{q}$ is a matter of integrating known fields vertically. Finally, we have the fully 3D advection-diffusion equations. These include the horizontal momentum equation for $\boldsymbol{u}$ (7), and tracer fields T (15). The 2D equation have been covered in detail in the previous sections. We will now focus our interest on equation (6) that solves the vertical velocity. The purpose of this section is to highlight the challenges that happen during the implementation of a solver for w on the GPU from a performance point of view. We also discuss the opportunities that a regular mesh in the vertical direction may bring.

From now on, we will denote by `w3d` the kernel that assembles w based on the weak equation (18). Equation (18) is recalled below for the sake of readability.

$$\langle\!\langle \varphi_i w^* n_z \rangle\!\rangle_{\Gamma_\mathrm{h}} - \left\langle w \frac{\partial \varphi_i}{\partial z} \right\rangle = \langle \boldsymbol{u} \cdot \nabla_h \varphi_i \rangle - \langle\!\langle \boldsymbol{n}_h \cdot (\varphi_i \boldsymbol{u}^*) \rangle\!\rangle$$

The important thing to notice is this equation is the presence of $w^* = \{w\}$ in the top and bottom integrals. This introduce a dependency of w onto its neighbouring values and makes the equation implicit in the vertical direction. Consequently, we will need to solve a linear system of equations for each column of prisms. This is a serious issue because solving a linear system is inherently sequential, and thus not ideal for the GPU. Note that the kernels for $\boldsymbol{u}$ and T will also face this problem. One not-so-obvious particularity of w and $\boldsymbol{q}$ is that their mass matrix, which links elements of a column, is independent of the vertical scaling. This theoretically allows the pre-computation of the inverse of the mass matrix, but more importantly gives a recurrence formula for the solution of the linear system. The systems must still be solved sequentially, but at least we don't need to store the mass matrix. This is good news since, as we have seen in 2D, the memory is very often a bottleneck.

5.1 Thread mapping and memory layout

Let n_tri be the number of triangles in the 2D mesh, n_lay the number of layers, and $n_\mathrm{elem} = n_\mathrm{tri} \cdot n_\mathrm{lay}$ be the number of prism elements. The 3D meshes used in this work are σ-like meshes, meaning that the number of layers is the same everywhere. This opens two opportunities regarding the order in which the prisms are stored. We can either order them by triangle, and inside of each triangle by layer, or by layer first. As discussed in section 4.7, the memory layout will play a key role in the access pattern, and thus the performance of the kernels. The first ordering stacks the elements of the same column continuously in memory. The second layout can be thought of as n_lay lists of n_tri prisms.

Regarding this question, the decision was quickly made to use the first memory layout, for the two following reasons: First, having elements contiguous by column makes the most sense from a locality point of view. It could allow to store only once in shared memory the values that are constant across one column. Elements from a column will also eventually need to collaborate. They build the right-hand side of a single linear system. The second reason is that the mesh is structured vertically, and we can use this structure to have a good memory access patterns. If, for example, there are 32 layers, we can assign one complete warp per column (one thread per prism). This layout ensures that adjacent threads will access adjacent elements, even when accessing other columns. For that access patter to be useful, we also need contiguous values in memory to correspond to the same fields across elements that follow each other. This is the same layout as described in section 4.7 and illustrated in Figure 18. Aside from these reason, the chosen mesh organisation has the benefit that it continues to work with z-coordinates. Recall that z-coordinates have a different number of layers depending on the triangle, which is impossible to describe in a layer-first approach. Although not the focus of this work, z-coordinates will be the next milestone, once SLIM has been ported to GPU with σ coordinates.

5.2 Using one or two kernels

Like most 3D fields, the computation of w requires to solve a linear system. In the case of w, we happen to have an explicit recurrence formula to solve the system, but it doesn't change the fact that a system has to be solved. Solving a system is a sequential process, and thus it would make sense to launch one

thread per column. On the other hand, assembling the right-hand side is a process that can be done in parallel on each element. Thus, mapping a thread per element is best. The data layout was also chosen with the assumption that neighbouring threads would access neighbouring elements. To solve the clash in the repartition of threads, the decision was made to use two separate kernels: one to assemble the right-hand side called `w3d`, and another to solve the system denoted as `sys_solve`. `w3d` assigns one thread per prism, and `sys_solve` uses one thread per column of prisms.

Unfortunately, using two kernels does not solve all the problems. When using the memory layout described in section 4.7, solving the system has a *terrible* access pattern. Indeed, threads will start by accessing the first the node of elements in the first layer. For example, the first thread will reach into the first triangle of the first layer, and the second will access the second triangle of the same layer. There are n_{lay} unneeded values between two consecutive threads. This already ruins the potential for memory coalescence. Locality is not much better. The second access will hit the same triangles, but reach the second node. Since values from different nodes are n_{elem} items apart, this ruins the spatial locality. We would need 6 accesses to come back to the first node and have a chance of experiencing temporal locality, but this process basically loads and discards the complete vector. All in all it's a terrible idea.

The solution adopted in these kernel is to build the right-hand side (`w3d`) with a memory layout that sits somewhere in between the optimal layout for `w3d` and the optimal layout to solve the system. This intermediate layout is in fact the original way in which data is organised in the CPU. It exhibits good degrees of locality from both `w3d`'s and `sys_solve`'s point of view. Although not implemented yet, memory coalescence could also be achieved through the use of shared memory in both kernels.

5.3 Performance evaluations

Just as in 2D, the metrics used for the profiling are

— The ratio of achieved floating-point to the theoretical maximum.

— The effective memory bandwidths.

— Finally and most importantly, the measured run-time.

One difference between the benchmarks presented here and the CPU-GPU comparisons in 2D is that we use only one process for CPU benchmarks. To compare the GPU to the full potential of the CPU, we can assume that the CPU is 5 times faster[13]. If not specified otherwise, the tests are performed with a mesh made of 12K triangles and 20 layers, and the GPU uses 32-bit floats. The first version of the `w3d` and `sys_solve` kernels implement the decisions from the previous section. Beware that those choices were made a-priori, based solely on the conclusions from the 2D benchmarks and conclusions.

The wall-clock runtime reveals that the GPU is around 545 times faster than a single CPU. Running the MPI+CPU code against one GPU puts the GPU close to 90 times faster, in line with the $\times 5$ expectation. These result are impressive by themselves, but they could have been expected. The fastest 2D code on one GPU is around 120 times faster than its MPI+CPU counterpart. In 2D the main bottleneck is the uncoalescent memory accesses to the neighbours. In the `w3d` kernel, due to the regularity in the vertical, all accesses are at least somewhat coalescent. The `sys_solve` on the other hand will raise some issues.

The detailed profiling of the `w3d` kernel reveals that around 35% of its peak FP32 compute is achieved. This is already a pretty good result, and it is likely that it will not be possible to increase it by a lot. Indeed, looking at the memory throughput in Figure 28 reveals that the kernel is most likely bandwidth-limited. The L2 cache is delivering 60% of its peak throughput, which is often considered good. The only clear problem of this kernel is its use of local memory, which when possible should be avoided.

The `sys_solve` kernel on the other hand is far from optimal. In these tests, it ran in the same amount of time as the `w3d` kernel while having much less to do. This is reflected in its achieved floating point performance which is below 1%. The reasons for the poor performance are the very low coalescence achieved by the kernel. Only 13% of the memory requests ended up being useful. With larger meshes, the weight of this kernel becomes even larger relative to `w3d`. To fix this kernel, the most obvious option is to change the memory access pattern with shared memory. Caching the data explicitly in shared memory could significantly reduce the cache overhead and increase performance.

[13]The value of 5 was observed consistently with different functions. The reason why 8 processes are only 5 times faster than a simple process is due to the turbo-ing. An single core is able to run faster when the rest of the CPU is not being heavily utilized.

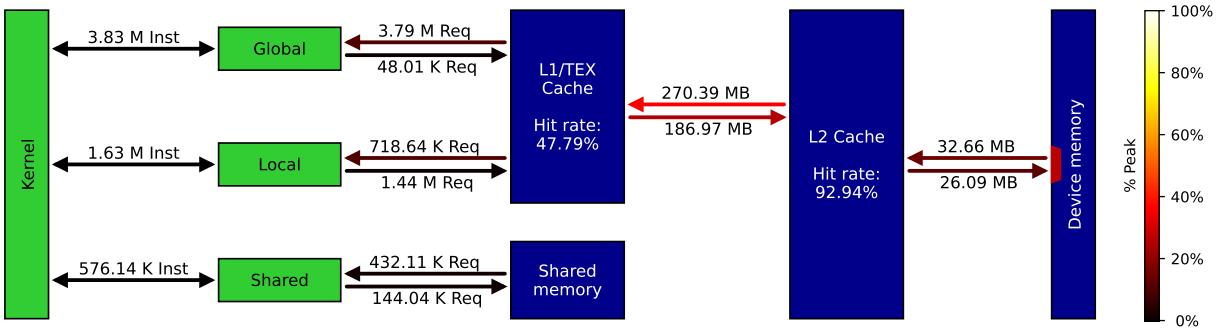


Figure 28: Memory chart of `w3d` kernel.

5.4 Tracer kernel

The next big category of function are the ones responsible for the advection-diffusion equations. Here, we will look at the tracer equation (15), since it is a bit simpler than the 3D momentum equation. The initial implementation of the `tracer` kernel is structured somewhat differently from the vertical velocity w. In `w3d`, each thread each thread to an element, and the threads grouped in blocs of size 128. For the tracer kernel, we also assign one thread per element, be we instead of using 2D blocs. Each thread of the bloc has thus 2 indices, in x and in y. The x index of a thread denotes its "depth", determines the layer on which the thread will operate. The y index on the other hand refers to a triangle. Each bloc works with a small number of triangles (typically 4) simultaneously. When there are more layers than the x-dimension of the bloc, the bloc simply moves down and processes the next layers until there are none left.

There were two key reasons to change the distribution of threads to 2D blocs. The first one is that we now have a guarantee that all the prisms in one column will be processed in the same group. Although not being used right now, this property can allow threads from the same column to avoid repeating work. In an independent threads model, all fluxes on the faces have to be computed twice, once for each adjacent prism. On the other hand, the fact that we know that all the prisms of a column will be processed in the same group greatly facilitates communications. The second reason is that the tracer kernel uses a number of 2D fields, which are thus identical over the whole column. These values can now be loaded a single time in shared memory, freeing registers of local memory. On top of that, different blocs don't require the number of layers to be constant, and this flexibility may be useful once we start adding support for z coordinates.

Compared to all the previous work, the tracer kernel introduces a new challenge in its mass matrix. The mass matrix of the tracer kernel is banded, with an upper and lower band of size 8. Unlike for w, here there is no shortcut to accelerate the resolution of the system. Consequently, the global mass matrix needs to be built explicitly. This is a challenge, because of the amount of memory needed. Each thread works on one element, a prism with 6 nodes, and its associated portion of the global mass matrix. Since its band is 8 above and below the diagonal, a thread is responsible for filling $6 \cdot (8+1+8) = 102$ elements of the the mass matrix. That many values are obviously too much to fit in the resisters or event the caches, and there will be unavoidable uses of local memory.

Just as for `w3d`, solving the system can either be done directly from the kernel building the matrices, or as a separate kernel. In this case, we chose to do everything in the same kernel. Each step of the Gaussian elimination involves the independent update of 64 entries in the matrix, and the idea is thus to use all the threads of the group to perform these operations concurrently. Looking at the performance results against the CPU reveal that our implementation is "only" 130 times faster than the single-core CPU version. Understanding why we get the low value is immediate when we look at the memory profile: There is a very high amount of local and global requests. Consequently, the GPU spends most of its time waiting for data to arrive from the caches. The average floating-point performance of the kernel is around 6.5%. However, this number hides two widely different dynamics. During about 50% of the time, the floating point performance is at roughly 13%: the matrix is being assembled. During the rest of the time, like in `w3d`, it is almost 0. In order to increase that number, the best would probably be to reduce the amount of memory needed. This can be done in several ways. The simplest one is in the interpolations functions. By default, when computing the shape functions, we always compute the values of the function and its derivative. This is done by inverting the Jacobian matrix which has a size 6×6. In many cases, however, we don't need the gradient, and thus we can skip this entire process. Note that

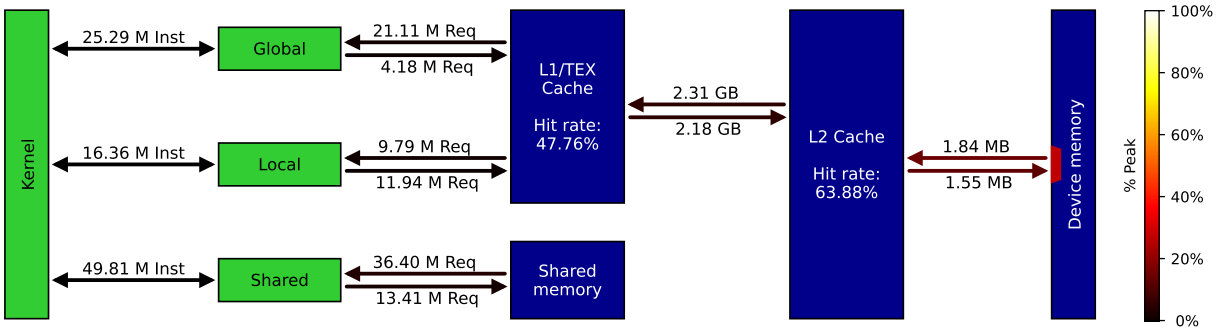


Figure 29: Memory chart of `tracer` kernel.

this is also true in the `w3d` kernel.

The second half of the kernel behaves very differently to the first. As stated above, each step of the Gaussian elimination updates 64 values. In order to be efficient, we do these in parallel in shared memory. The result is that the computations are indeed relatively fast, but a ludicrous amount of time is spent in synchronisation calls. It is worth noting that we also tryed the opposite idea in a 2-kernels approach. Instead of giving each thread its own value to work on, we assigned a thread to an entire column. The performance ended up being almost identical for different reasons. That second version spent all of its time querying global local memory. An in-between idea could be to assign a row of the matrix to each thread, in the hopes of balancing the loading of data and synchronisation. This has not be tried yet.

In 3D, a lot of resources are spent to compute the values of the shape functions, their derivatives, and the Jacobian of the change of coordinates. As it stands, the code computes all of these three values in a single function, which is wasteful when only one of them is needed. This functions is often called multiple times per kernel.

6 Conclusions

This Master's thesis focused on understanding the inner workings of the GPU to maximize the performance of SLIM 2D and SLIM 3D. In the current state of affairs, the 2D single-GPU code can mostly be considered done. The GPU code is orders of magnitude faster than the CPU, outperforming a single core by more than 500 times. This is a testament to how good GPUs have become. DG methods, like many other kinds of numerical methods, require fewer flops per byte at low orders. This tends to make them more more likely to be bandwidth-limited. With SLIM 2D, we were able to reach 3 Tflops in compute performance, which is is a third of the card's theoretical peak. Considering that we use linear polynomials on an unstructured mesh, a third of the peak performance is a solid result, in line or better than comparable results from literature [49, 50].

Extending the model to multiple GPUs reveales new challenges. The communications used to be negligible, but now that everything runs faster, they become significant. For large enough problems, we are able to hide the communication by overlapping them with computations. However, we quickly reach the strong scalabiliy limits of our code when we decrease the problem size. What we learned, however is that improvements are still possible, as demonstrated by GPU-aware MPI transfers. It showed that a fast blocking communication can rival slower but overlapping communication. Combining the flexibility of using streams and the speed of GPU-aware MPI would extend the ranges of relevant sizes in multi-GPU settings.

Finally, the analysis of two of the 3D functions reveals what we suspected would happen: solving the linear systems on the GPU is challenging. At this stage, improving the resolution of the linear systems seems like the best way to move forward, as it has a high potential reward. The routines that solve the systems heavily under-utilise the GPU due to their poor memory access patterns, and there is a lot to be gained. The 3D kernels, especially the tracer, also highlight the current limitations of the GPU hardware. These kernels use a lot of variables, that spill out to local memory, thus greatly hampering the performance.

The main take-away from this report should be that GPUs offer a massive potential that it can be hard to unlock. GPUs are bound to a different set of rules than CPUs, and a good understanding of the hardware is required to write efficient code. By far the biggest change compared to CPUs is the ways in which the memory performs and should be used. We are use to maximize locality, but coalescence turns out to be the more important metric. For GPUs, accessing memory is very slow, and often a becomes a bottleneck. In a sense, GPUs reflect the contemporary challenges faced in the HPC ans data center industry:

7 Future perspectives

Although effective as a proof of concept and a learning support, the work started is far from being over. In the short term, some time will have to be spent to figure out a better way to to solve the systems in 3D. Up to this point, only one approach was used for the analytical system in w, and two ideas were tested for the complete system for the tracer. To improve the time spent in w's system, the fist idea should be to use shared memory, and improve the coalescence of the accesses.

To solve the full system, many different ideas can be tried, with a variety of memory layouts. Transpositions are generally not recommended as they involve a lot of data movement, but since the assembly of the matrix is really slow, the cost of transpositions is negligible. For this system, we tries an approach where the active bloc of the mass matrix is in shared memory, and threads work one item at a time before synchronisations. This resulted in 90% of the time being spent for synchronisations. We already tried a approach where one threads keeps the entirety of the active part of the matrix in its private address space (thus spilling out in local memory). The next idea that should be tried is ta have a balance between the amount of work and data that each thread has. Combined with the best memory layouts, we should expect the time needed to solve this system to drop significantly.

During the building phase of the `w3d` and `ttracer` kernels, we can also reduce the amount of reg-

isters used. For example, instead of interpolating the basis functions in each element, these can be pre-computed and stored in constant memory. This frees up registers almost no cost, since constant memory is extremely fast when all threads load the same values together. Using the structure of the basis functions could also help reduce the memory used. The basis functions for prisms are the tensor product of the 2D linear P1 basis functions on a triangle by the 1D linear basis function of the reference interval. This property could be n mny places to simplify expression. For example, computing the Jacobian of the transformation reveals that $J = J_{\text{horiz}} \cdot J_{\text{vert}}$. Note that J_{horiz}, doesn't depend on h and is constant for the whole column of prisms, reducing the computation of J to a simple interpolation, as opposed to computing tthe determinant of a 3×3 matrix.

Finally, the remaining 3D kernels shall be implemented, using the knowledge acquired from the kernels and system resolutions of w and T.

Although maybe less relevant for 3D-focused applications, improve multi-GPU communications may be possible and relatively easy. A tool that we could use to accelerate data tranfers is use Collective Communication Libraries. NVIDIA released their NVIDIA Collective Communication Libraries (NCCL) [51] in 2015. More recently, AMD released the similar ROCm Communication Collectives Library (RCCL) as part of their ROCm stack. Both of these libraries provides collective and point-to-point communication primitives, similarly to MPI. Their edge over GPU-aware MPI come from being designed for GPUs first. Just as any other GPU functions, the communications primitives are tied to a stream, but now the user can manage it explicitly. In our kernels, having the possibility to launch the communications on the comm stream would probably the optimal way of doing communications. It would cut the unnecessary copies to and from system memory that make up a significant part of the time.

In the longer term, the next milestone will be to the support for other coordinate systems such as z coordinates or hybrid systems. Alongside more flexible coordinate systems, a method to include realistic boundary conditions, forcings and other external events should be set up. Finally, since the quality of a simulation is always limited by the quality of the input data, we should give the user the same level of control in the GPU-accelerated version of SLIM than in CPU version. Then, and only then shall this work be complete.

References

[1] B. A. Grantham, F. Chan, K. J. Nielsen, D. S. Fox, J. A. Barth, A. Huyer, J. Lubchenco, and B. A. Menge, "Upwelling-driven nearshore hypoxia signals ecosystem and oceanographic changes in the northeast Pacific," *Nature*, vol. 429, no. 6993, pp. 749–754, Jun. 2004, number: 6993 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/nature02605

[2] L. A. Levin, C.-L. Wei, D. C. Dunn, D. J. Amon, O. S. Ashford, W. W. L. Cheung, A. Colaço, C. Dominguez-Carrió, E. G. Escobar, H. R. Harden-Davies, J. C. Drazen, K. Ismail, D. O. B. Jones, D. E. Johnson, J. T. Le, F. Lejzerowicz, S. Mitarai, T. Morato, S. Mulsow, P. V. R. Snelgrove, A. K. Sweetman, and M. Yasuhara, "Climate change considerations are fundamental to management of deep-sea resource extraction," *Global Change Biology*, vol. 26, no. 9, pp. 4664–4678, 2020, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/gcb.15223. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/gcb.15223

[3] "Ocean Heat Content Rises," Jan. 2020. [Online]. Available: http://www.ncei.noaa.gov/news/ocean-heat-content-rises

[4] S. Levitus, J. I. Antonov, T. P. Boyer, R. A. Locarnini, H. E. Garcia, and A. V. Mishonov, "Global ocean heat content 1955-2008 in light of recently revealed instrumentation problems: GLOBAL OCEAN HEAT CONTENT," *Geophysical Research Letters*, vol. 36, no. 7, pp. n/a–n/a, Apr. 2009. [Online]. Available: http://doi.wiley.com/10.1029/2008GL037155

[5] NOAA, "How much oxygen comes from the ocean?" [Online]. Available: https://oceanservice.noaa.gov/facts/ocean-oxygen.html

[6] T. Kärnä, V. Legat, and E. Deleersnijder, "A baroclinic discontinuous Galerkin finite element model for coastal flows," *Ocean Modelling*, vol. 61, pp. 1–20, Jan. 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500312001333

[7] "Marine and coastal ecosystem services - Ocean & Climate Platform," Dec. 2016. [Online]. Available: https://ocean-climate.org/en/marine-and-coastal-ecosystem-services/

[8] D. C. Donato, J. B. Kauffman, D. Murdiyarso, S. Kurnianto, M. Stidham, and M. Kanninen, "Mangroves among the most carbon-rich forests in the tropics," *Nature Geoscience*, vol. 4, no. 5, pp. 293–297, May 2011, number: 5 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/ngeo1123

[9] B. Fox-Kemper, A. Adcroft, C. W. Böning, E. P. Chassignet, E. Curchitser, G. Danabasoglu, C. Eden, M. H. England, R. Gerdes, R. J. Greatbatch, S. M. Griffies, R. W. Hallberg, E. Hanert, P. Heimbach, H. T. Hewitt, C. N. Hill, Y. Komuro, S. Legg, J. Le Sommer, S. Masina, S. J. Marsland, S. G. Penny, F. Qiao, T. D. Ringler, A. M. Treguier, H. Tsujino, P. Uotila, and S. G. Yeager, "Challenges and Prospects in Ocean Circulation Models," *Frontiers in Marine Science*, vol. 6, 2019. [Online]. Available: https://www.frontiersin.org/article/10.3389/fmars.2019.00065

[10] A. Adcroft and D. Marshall, "How slippery are piecewise-constant coastlines in numerical ocean models?" *Tellus A: Dynamic Meteorology and Oceanography*, vol. 50, no. 1, pp. 95–108, Jan. 1998, publisher: Taylor & Francis _eprint: https://doi.org/10.3402/tellusa.v50i1.14514. [Online]. Available: https://doi.org/10.3402/tellusa.v50i1.14514

[11] T. Kärnä, S. Kramer, L. Mitchell, D. Ham, M. Piggott, and A. Baptista, "Thetis coastal ocean model: Discontinuous Galerkin discretization for the three-dimensional hydrostatic equations," *Geoscientific Model Development*, vol. 11, pp. 4359–4382, Oct. 2018.

[12] Y. J. Zhang, F. Ye, E. V. Stanev, and S. Grashorn, "Seamless cross-scale modeling with SCHISM," *Ocean Modelling*, vol. 102, pp. 64–81, Jun. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500316300312

[13] Y. Zhang and A. M. Baptista, "SELFE: A semi-implicit Eulerian–Lagrangian finite-element model for cross-scale ocean circulation," *Ocean Modelling*, vol. 21, no. 3, pp. 71–96, Jan. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500307001436

[14] T. Ringler, M. Petersen, R. L. Higdon, D. Jacobsen, P. W. Jones, and M. Maltrud, "A multi-resolution approach to global ocean modeling," *Ocean Modelling*, vol. 69, pp. 211–232, Sep. 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500313000760

[15] D. Etiemble, "45-year CPU evolution: one law and two equations," p. 6.

[16] "IBM100 - Power 4 : The First Multi-Core, 1GHz Processor," Mar. 2012, publisher: IBM Corporation. [Online]. Available: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/

[17] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[18] "The Khronos Group Releases OpenCL 1.0 Specification," Dec. 2008, section: News. [Online]. Available: https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification

[19] T.-q. Chen and Q.-h. Zhang, "GPU acceleration of a nonhydrostatic model for the internal solitary waves simulation," *Journal of Hydrodynamics*, vol. 25, no. 3, pp. 362–369, Jun. 2013. [Online]. Available: http://link.springer.com/10.1016/S1001-6058(11)60374-1

[20] "AMD: Introducing AMD CDNA 2 architecture," 2021. [Online]. Available: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

[21] A. F. Shchepetkin and J. C. McWilliams, "The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean Modelling*, vol. 9, no. 4, pp. 347–404, Jan. 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500304000484

[22] L. Debreu, C. Vouland, and E. Blayo, "AGRIF: Adaptive grid refinement in Fortran," *Computers & Geosciences*, vol. 34, no. 1, pp. 8–13, Jan. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S009830040700115X

[23] S. Danilov, Q. Wang, M. Losch, D. Sidorenko, and J. Schröter, "Modeling ocean circulation on unstructured meshes: comparison of two horizontal discretizations," *Ocean Dynamics*, vol. 58, no. 5, pp. 365–374, Dec. 2008. [Online]. Available: https://doi.org/10.1007/s10236-008-0138-5

[24] E. Deleersnijder and P. F. J. Lermusiaux, "Multi-scale modeling: nested-grid and unstructured-mesh approaches," *Ocean Dynamics*, vol. 58, no. 5, pp. 335–336, Dec. 2008. [Online]. Available: https://doi.org/10.1007/s10236-008-0170-5

[25] "HOME - FVCOM @ MEDML - Marine Ecosystem Dynamics Modeling Laboratory." [Online]. Available: http://fvcom.smast.umassd.edu/

[26] A. Adcroft, S. Dutkiewicz, D. Ferreira, P. Heimbach, O. Jahn, and G. Maze, "MITgcm User Manual," *http://mitgcm.org/public/r2_manual/latest/online_documents/manual.pdf*, pp. 1–451, Jan. 2011.

[27] J. Luettich, Richard and J. Westerink, "Formulation and Numerical Implementation of the 2D/3D ADCIRC Finite Element Model Version 44.XX," Jan. 2004.

[28] E. Deleersnijder, T. Dobbelaere, I. Draoui, E. Hanert, A. L. Hoang, A. Ishimwe, J. Lambrechts, V. Legat, A. Saint-Amand, S. Soares Frazao, V. Vallaeys, and D. Vincent, "SLIM: A multi-scale model of the land-sea continuum," 2019. [Online]. Available: https://dial.uclouvain.be/pr/boreal/object/boreal:243281

[29] V. Aizinger and C. Dawson, "The local discontinuous Galerkin method for three-dimensional shallow water flow," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 4, pp. 734–746, Jan. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782506002222

[30] B. Cockburn, "Discontinuous Galerkin Methods for Convection-Dominated Problems," in *High-Order Methods for Computational Physics*, ser. Lecture Notes in Computational Science and Engineering, T. J. Barth and H. Deconinck, Eds. Berlin, Heidelberg: Springer, 1999, pp. 69–224. [Online]. Available: https://doi.org/10.1007/978-3-662-03882-6_2

[31] E. J. Kubatko, J. J. Westerink, and C. Dawson, "hp Discontinuous Galerkin methods for advection dominated problems in shallow water flow," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 1, pp. 437–451, Dec. 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S004578250600171X

[32] H. Burchard and H. Rennau, "Comparative quantification of physically and numerically induced mixing in ocean models," *Ocean Modelling*, vol. 20, no. 3, pp. 293–311, Jan. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S146350030700128X

[33] P. Marchesiello, L. Debreu, and X. Couvelard, "Spurious diapycnal mixing in terrain-following coordinate models: The problem and a solution," *Ocean Modelling*, vol. 26, no. 3, pp. 156–169, Jan. 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1463500308001510

[34] J. Mak, P. Choboter, and C. Lupo, "Numerical ocean modeling and simulation with CUDA," in *OCEANS'11 MTS/IEEE KONA*, Sep. 2011, pp. 1–6, iSSN: 0197-7385.

[35] S. Xu, X. Huang, Y. Zhang, H. Fu, L.-Y. Oey, F. Xu, and G. Yang, "gpuPOM: a GPU-based Princeton Ocean Model," *Geoscientific Model Development Discussions*, vol. 7, pp. 7651–7691, Nov. 2014.

[36] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[37] D. Häfner, R. Nuterman, and M. Jochum, "Fast, Cheap, and Turbulent—Global Ocean Modeling With GPU Acceleration in Python," *Journal of Advances in Modeling Earth Systems*, vol. 13, no. 12, p. e2021MS002717, 2021, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2021MS002717. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1029/2021MS002717

[38] X. Zhao, S.-x. Liang, Z.-c. Sun, X. Zhao, S. Jiawen, and Z. Liu, "A GPU accelerated finite volume coastal ocean model," *Journal of Hydrodynamics, Ser. B*, vol. 29, pp. 679–690, Aug. 2017.

[39] P. Zhang, Y. Ao, C. Yang, Y. Liu, F. Liu, C. Wu, and H. Zhao, "Pattern-Driven Hybrid Multi- and Many-Core Acceleration in the MPAS Shallow-Water Model," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 71–80, iSSN: 0190-3918.

[40] B. Rivière, *Discontinuous Galerkin Methods for Solving Elliptic and Parabolic Equations*, ser. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2008. [Online]. Available: https://epubs.siam.org/doi/book/10.1137/1.9780898717440

[41] C. Dawson, S. Sun, and M. F. Wheeler, "Compatible algorithms for coupled flow and transport," *Computer Methods in Applied Mechanics and Engineering*, vol. 193, no. 23, pp. 2565–2580, Jun. 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782504001252

[42] K. Shahbazi, "An explicit expression for the penalty parameter of the interior penalty method," *Journal of Computational Physics*, vol. 205, no. 2, pp. 401–407, May 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999104004796

[43] A. Fulop and R. Forster, "Yang-Mills lattice on CUDA," *ACTA UNIVERSITATIS SAPIENTIAE*, vol. 5, p. 184, Jul. 2013.

[44] C. Lam, "Measuring GPU Memory Latency," Apr. 2021. [Online]. Available: https://chipsandcheese.com/2021/04/16/measuring-gpu-memory-latency/

[45] "Memory Latency Data," Aug. 2021. [Online]. Available: https://chipsandcheese.com/memory-latency-data/

[46] C. Lam, "Microbenchmarks/GpuMemLatency at master · clamchowder/Microbenchmarks." [Online]. Available: https://github.com/clamchowder/Microbenchmarks

[47] "CUDA C++ Best Practices Guide," archive Location: Programming Guides. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[48] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 886–893, Jul. 2005. [Online]. Available: https://doi.org/10.1145/1073204.1073278

[49] A. Modave, A. St-Cyr, and T. Warburton, "GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models," *Computers & Geosciences*, vol. 91, pp. 64–76, Jun. 2016, arXiv:1602.07997 [physics]. [Online]. Available: http://arxiv.org/abs/1602.07997

[50] A. Klöckner, T. Warburton, and J. S. Hesthaven, "High-Order Discontinuous Galerkin Methods by GPU Metaprogramming," in *GPU Solutions to Multi-scale Problems in Science and Engineering*, D. A. Yuen, L. Wang, X. Chi, L. Johnsson, W. Ge, and Y. Shi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 353–374, series Title: Lecture Notes in Earth System Sciences. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16405-7_23

[51] C. Woolley, "Nccl: Accelerated multi-gpu collective communications," 2015.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**