Eikonal CUDA solver for 3D unstructured meshes

Authors:
Cesaroni Sabrina (10677284)
Tonarelli Melanie (10787497)
Trabacchin Tommaso (10705974)

Course: Advanced Methods for Scientific Computing (058166)
Professor: Formaggia Luca
HPC Engineering Master's Degree
Academical Years: 2023/24

Contents

1 Introduction

1.1 Description of the mathematical problem

An Eikonal equation is a non-linear first-order partial differential equation that is encountered in problems of wave propagation.

An **Eikonal equation** is one of the form:

$$\begin{cases} H(x, \nabla u(x)) = 1 & x \in \Omega \subset \mathbb{R}^3 \\ u(x) = g(x) & x \in \Gamma \subset \partial\Omega \end{cases} \tag{1}$$

where

- $u(x)$ is the eikonal function, the shortest time needed to travel from the boundary $\partial\Omega$ to x inside Ω;
- $\nabla u(x)$ is the gradient of u, a vector that points in the direction of the wavefront;
- H is the Hamiltonian, which is a function of the spatial coordinates x and the gradient ∇u;
- Γ is a set smooth boundary conditions.

In most cases,

$$\begin{aligned} H(x, \nabla u(x)) = |\nabla u(x)|_M = \\ = \sqrt{(\nabla u(x))^T M(x) \nabla u(x)} \end{aligned} \tag{2}$$

where $M(x)$ is a symmetric positive definite function encoding the speed information on Ω.

In the simplest cases, $M(x) = c^2 I$ therefore the equation becomes:

$$\begin{cases} |\nabla u(x)| = \frac{1}{c} & x \in \Omega \\ u(x) = g(x) & x \in \Gamma \subset \partial\Omega \end{cases} \tag{3}$$

where c represents the celerity of the wave.

1.2 Description of classical algorithms

Fast Marching and Fast Sweeping algorithms are numerical methods commonly used in solving partial differential equations (PDEs) that govern the evolution of fronts or interfaces in various physical processes. These algorithms are particularly useful for solving Eikonal equations, which describe the propagation of wave fronts as seen above.

Here is a brief overview of each algorithm:

- **Fast Marching method** (FMM): Developed originally by J. Sethian, FMM represents a specialized form of a level-set method. It operates by "marching" through the computational domain from known points (e.g., initial conditions) to unknown points in a wavefront-like manner. At each step, it updates the values of unknown points based on the values of neighboring points, typically selecting the minimum arrival time. This approach guarantees adherence to the causality principle, ensuring that the past cannot be influenced by the future.
- **Fast Iterative methods** (FIM): FIM can be likened to a nonlinear Gauss-Seidel iteration. It iteratively "sweeps" through the computational domain, updating points in a specific order to minimize computational costs. The solution evolves iteratively until the changes between successive iterations fall below a given tolerance or meet other convergence criteria.

Given the total number N of nodes in the domain, it's important to note that the FMM has a complexity of $O(N \log N)$ due to the use of a heap, while FIM has a linear complexity of $O(N)$.

While FMM offers the advantage of computing the solution after a number of steps equal to the mesh nodes, it poses challenges in parallelization compared to FIM. This is because each update in FMM requires finding the active node with the smallest value. Consequently, for our project, we opt for Fast Iterative methods, which facilitate GPU general-purpose programming, enabling efficient parallelization and enhanced computational performance.
A detailed description of FIM can be found in [4].

1.3 Goals of the project

The project aims to enhance the Eikonal solver by harnessing the computational capabilities of GPU architectures. Additionally, it utilizes a domain decomposition strategy to tackle scenarios where the problem scale is extensive, potentially surpassing the shared memory capacity of a single host or GPU, or where the computing power of a compute unit falls short.

The project is a **CUDA library** designed for computing the Eikonal equation based on the FIM algorithm on 3D unstructured tetrahedrical meshes. It is designed to be highly extensible, and easy to integrate in different applications. The library comprises three primary classes

1. `LocalSolver`, responsible for resolving the local problem detailed in Section 2.
2. `Mesh`, representing a 3D mesh as discussed in Section 3.
3. `Solver`, implementing the CUDA solver, elaborated upon in Section 4.

All the information to compile and run some examples can be found here.

2 Local Solver

The FIM is based on a local solver which, in each tetrahedron, solves (1) with the simplification introduced in (2), specifically it solves the following equation:

$$\sqrt{(\nabla u(x))^T M \nabla u(x)} = 1 \qquad \forall x \in \Omega \tag{4}$$

Note that the above problem is equivalent to a constrained optimization problem, as seen in [2], which can also be solved explicitly with the approach introduced in [5].

An upwind scheme is used to compute an unknown solution ϕ_4, assuming that the given values ϕ_1, ϕ_2 and ϕ_3 comply with the causality property of the Eikonal solutions. Since we assume a constant speed function within each tetrahedron, the arrival time ϕ_4 is determined by the time associated with that segment from x_5 to x_4 (see figure 1) that minimizes the solution value at x_4. Therefore we have to determine the coordinates of that auxiliary point x_5 where the wave-front intersects first the plane defined by the vertices x_1, x_2 and x_3.

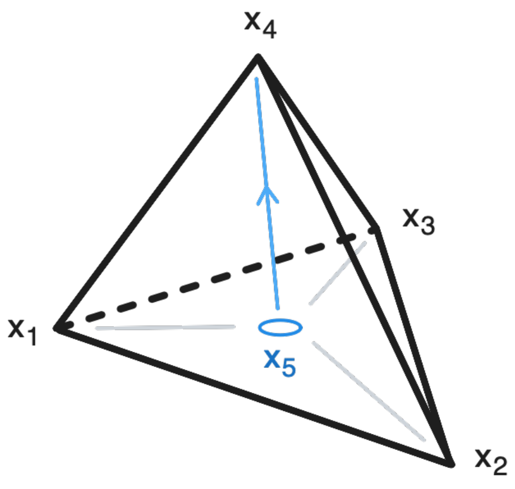


Figure 1: Notations in the tetrahedron.

As demonstrated in [5], the optimization problem transforms into the non-linear system of equations for λ_1 and λ_2:

$$\begin{cases} \phi_{1,3}\sqrt{\lambda^T M' \lambda} = \lambda^T \underline{\alpha} \\ \phi_{2,3}\lambda^T \underline{\alpha} = \phi_{1,3}\lambda^T \underline{\beta} \end{cases} \tag{5}$$

where

$$\phi_{x,y} = \phi_y - \phi_x \quad \forall x, y \in \{1,2,3,4\}, \quad x \neq y$$
$$e_{i,j} = x_j - x_i \; \forall i, j \in 1, ..., 5$$

$$\begin{aligned}
&\underline{\alpha} = [\alpha_1\ \alpha_2\ \alpha_3]^T = [e_{1,3}^T M e_{1,3},\ e_{2,3}^T M e_{1,3},\ e_{3,4}^T M e_{1,3}]^T\\
&\underline{\beta} = [\beta_1\ \beta_2\ \beta_3]^T = [e_{1,3}^T M e_{2,3},\ e_{2,3}^T M e_{2,3},\ e_{3,4}^T M e_{2,3}]^T\\
&\underline{\gamma} = [\gamma_1\ \gamma_2\ \gamma_3]^T = [e_{1,3}^T M e_{3,4},\ e_{2,3}^T M e_{3,4},\ e_{3,4}^T M e_{3,4}]^T\\
&M' = [\underline{\alpha}\ \underline{\beta}\ \underline{\gamma}] \quad \text{being a symmetric matrix}\\
&\lambda = [\lambda_1,\ \lambda_2,\ 1-\lambda_1-\lambda_2]^T
\end{aligned}$$

In this way, the solution ϕ_4 can be computed as follows:

$$\phi_4 = \lambda_1\phi_1 + \lambda_2\phi_2 + (1-\lambda_1-\lambda_2)\phi_3 + \sqrt{e_{5,4}^T M e_{5,4}} \tag{6}$$

with $e_{5,4} = [e_{1,3}\ e_{2,3}\ e_{3,4}]\lambda$.

To address (5), a direct approach was selected over an iterative method, considering the poor performance of the latter on a GPU architecture. Nevertheless, leveraging GPUs presents its own challenges. To mitigate memory consumption, inspired by recommendations from [5], we decided against fully storing matrix M, a decision elaborated in Section 2.3. However, this decision introduces additional complications, which we discuss in Section 2.2.

2.1 Algebraic direct method

The non-linear system in (5) is solved directly by firstly simplifying it with the above system of equations:

$$\begin{cases} a\lambda_1 + b\lambda_2 + c = 0 & \text{(7a)}\\ d\lambda_1^2 + e\lambda_2^2 + f\lambda_1\lambda_2 + g\lambda_1 + h\lambda_2 + k = 0 & \text{(7b)} \end{cases}$$

where

$$\begin{aligned}
&a = \phi_{1,3}(\beta_3-\beta_1) + \phi_{2,3}(\alpha_1-\alpha_3)\\
&b = \phi_{1,3}(\beta_3-\beta_2) + \phi_{2,3}(\alpha_2-\alpha_3)\\
&c = -\phi_{1,3}\beta_3 + \phi_{2,3}\alpha_3\\
&d = \phi_{1,3}^2(\alpha_1-\alpha_3-\gamma_1+\gamma_3) - (\alpha_1-\alpha_3)^2\\
&e = \phi_{1,3}^2(\beta_2-\beta_3-\gamma_2+\gamma_3) - (\alpha_2-\alpha_3)^2\\
&f = \phi_{1,3}^2(\alpha_2-\alpha_3+\beta_1-\beta_3-\gamma_1-\gamma_2+2\gamma_3) - 2(\alpha_1-\alpha_3)(\alpha_2-\alpha_3)\\
&g = \phi_{1,3}^2(\alpha_3+\gamma_1-2\gamma_3) - 2\alpha_3(\alpha_1-\alpha_3)\\
&h = \phi_{1,3}^2(\beta_3+\gamma_2-2\gamma_3) - 2\alpha_3(\alpha_2-\alpha_3)\\
&k = \phi_{1,3}^2\gamma_3 - \alpha_3^2
\end{aligned}$$

By isolating λ_1 in (7a) and substituting it in (7b), a quadratic equation for λ_2 is obtained:

$$\hat{a}\lambda_2^2 + \hat{b}\lambda_2 + \hat{c} = 0$$

where

$$\begin{aligned}
\hat{a} &= e + \frac{b^2 d}{a^2} - \frac{bf}{a} \\
\hat{b} &= h + 2\frac{bcd}{a^2} - \frac{bg - cf}{a} \\
\hat{c} &= k + \frac{c^2 d}{a^2} - \frac{cg}{a}
\end{aligned}$$

and whose solutions is computed with the numerically stable method jokingly called the "citardauq" formula ("quadratic" spelled backwards).

However, due to the constraints $\lambda_i \in [0,1] \forall i = 1, 2$, a solution to the local problem may not always exist. In such cases, we fall back on the minimum solution obtained by evaluating the problem at the following three values of $(\lambda_1, \lambda_2) = (1,0)$, $(0,1)$ or $(0,0)$.

2.2 Gray-code indexing

Throughout the local solver, a gray-code indexing system, as described in [6], is employed.

Specifically, a local Gray-code of 4 bits length is used to identify uniquely all possible vertices and edges in a tetrahedron. Each vertex $k \in 1, 2, 3, 4$ is represented by a 4-bit number with exactly one bit set to 1 and the Gray index of the connecting edge $e_{k,l}^{\text{gray}} = \text{OR}(k^{\text{gray}}, l^{\text{gray}})$ contains exactly two bits set to 1.

x_4	x_3	x_2	x_1	**edge**
0	0	1	1	(0)
0	1	0	1	(1)
0	1	1	0	(2)
1	0	0	1	(3)
1	0	1	0	(4)
1	1	0	0	(5)

Table 1: Local Gray-code numbering of edges, following the convention shown in figure 2.

2.3 Matrix footprint

To solve system (5) within each tetrahedron, the initial step involves storing the 3x3 matrix M alongside the vertex coordinate data to compute all the necessary information. Afterward, all data needs to be transferred from main memory to GPU global memory. However, efficient utilization of global memory and bandwidth is crucial for exploiting GPU capabilities. This necessitates organized and coalesced memory access aligned with DRAM bursts, as depicted

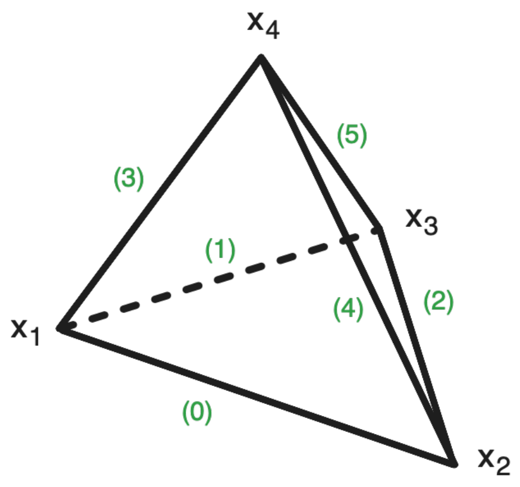


Figure 2: Notations of edges in the tetrahedron.

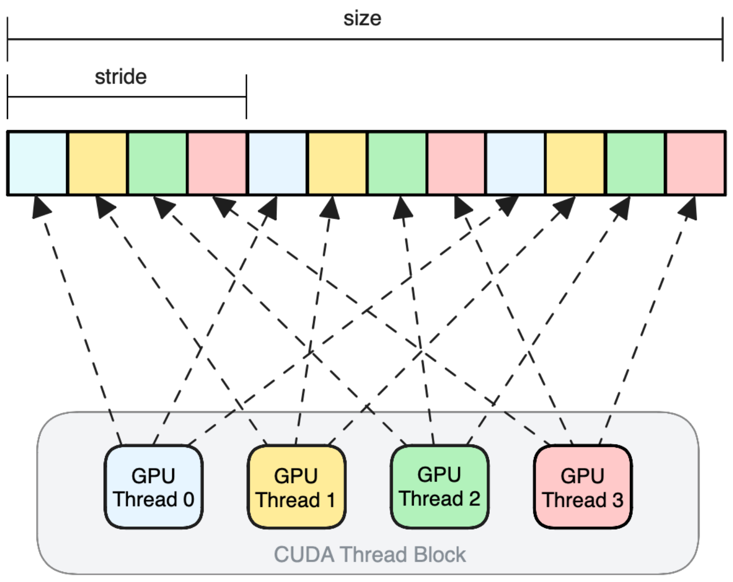


Figure 3: Coalesced Memory Access Pattern.

in Figure 3. To enhance memory access coherence and efficiency, the guidelines outlined in [5] are adhered to.

Assuming ϕ_4 as a reference configuration (indicating the solution computation for x_4 within the specified tetrahedron), all 18 potential inner products listed in Table 2 should be precomputed and stored.

0	**1**	**2**	**3**	**4**	**5**
$e_{1,2}^T M e_{1,2}$	$e_{1,2}^T M e_{1,3}$	$e_{1,2}^T M e_{2,3}$	$e_{1,2}^T M e_{1,4}$	$e_{1,2}^T M e_{2,4}$	~~$e_{1,2}^T M e_{3,4}$~~
	$e_{1,3}^T M e_{1,3}$	$e_{1,3}^T M e_{2,3}$	$e_{1,3}^T M e_{1,4}$	~~$e_{1,3}^T M e_{2,4}$~~	$e_{1,3}^T M e_{3,4}$
		$e_{2,3}^T M e_{2,3}$	~~$e_{2,3}^T M e_{1,4}$~~	$e_{2,3}^T M e_{2,4}$	$e_{2,3}^T M e_{3,4}$
			$e_{1,4}^T M e_{1,4}$	$e_{1,4}^T M e_{2,4}$	$e_{1,4}^T M e_{3,4}$
				$e_{2,4}^T M e_{2,4}$	$e_{2,4}^T M e_{3,4}$
					$e_{3,4}^T M e_{3,4}$

Table 2: Initial storage requirement for 18 scalar products. Note that three scalar products have been removed from the dataset as they lack physical significance.

However, upon closer examination, it becomes clear that all the scalar products mentioned above can be computed solely using the six scalar products of the main diagonal. Therefore, it is possible to achieve reduced memory usage. Consequently, only the six scalar products listed in Table 3 need to be stored.

0	**1**	**2**	**3**	**4**	**5**
$e_{1,2}^T M e_{1,2}$	$e_{1,3}^T M e_{1,3}$	$e_{2,3}^T M e_{2,3}$	$e_{1,4}^T M e_{1,4}$	$e_{2,4}^T M e_{2,4}$	$e_{3,4}^T M e_{3,4}$

Table 3: Reduced number of scalar products

Indeed, all mixed scalar products are computed using the formula:

$$e_{k_1,k_2}^T M e_{l_1,l_2} = esgn \times \frac{1}{2}\big(e_{k_1,k_2}^T M e_{k_1,k_2} + e_{l_1,l_2}^T M e_{l_1,l_2} - e_{s_1,s_2}^T M e_{s_1,s_2}\big) \tag{8}$$

Here, the indices s_1, s_2 crucial for the computation, are derived from the edges' representation in gray code, described in section 2.2. Specifically, if k^{gray} and l^{gray} denote the gray code associated with edges $\{k_1, k_2\}$ and $\{l_1, l_2\}$ respectively, then the indices $\{s_1, s_2\}$ are recovered through the associated gray code which is computed as $s^{\text{gray}} = \text{XOR}(k^{\text{gray}}, l^{\text{gray}})$.

Furthermore, the determination of the sign involves to calculate the value denoted as $esgn$, which is derived from comparisons:

$$esgn = \big(2(s^{\text{gray}} > k^{\text{gray}}) - 1\big) \times \big(2(s^{\text{gray}} > l^{\text{gray}}) - 1\big)$$

If the wave front hits a tetrahedron differently, meaning that the computation of the solution is required in a different configuration (ϕ_k for $k \in 1, 2, 3$), then a rotation of the tetrahedron is needed as described in [5]. An example of rotation is shown in figure 4.

However, thanks to the use of the Gray-code edge numbers, this problem is solved. In fact, the use of the gray code is justified as it allows to perform the tetrahedron rotation without using branching code which badly affects the GPUs performance and without storing the above six scalar products for each possible configuration of the tetrahedron. This rotation is implemented by a bit-wise circular shift of the gray code and a sign change of the scalar product in 8, resulting from redirected edges in the different configuration. Some examples are shown in picture 5.

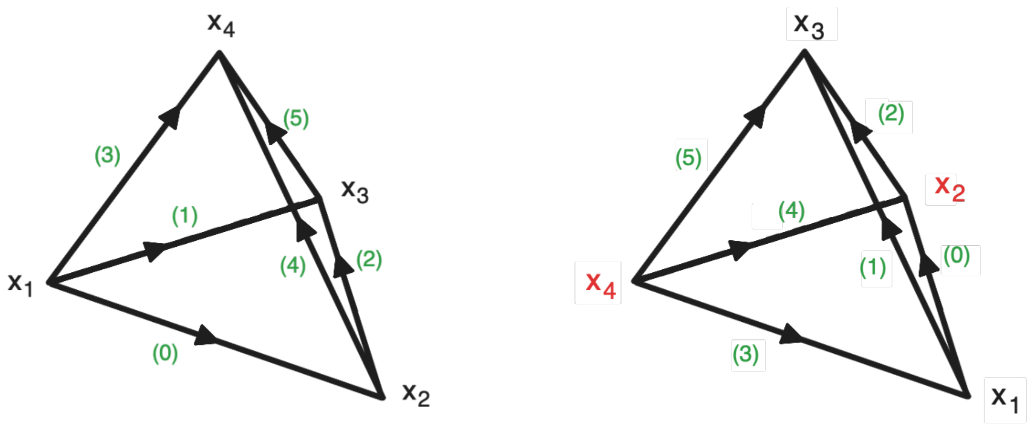


Figure 4: Rotation by one position on the right, passing from reference configuration ϕ_4 to ϕ_3. Note that edges (3), (4) and (5) change directions.

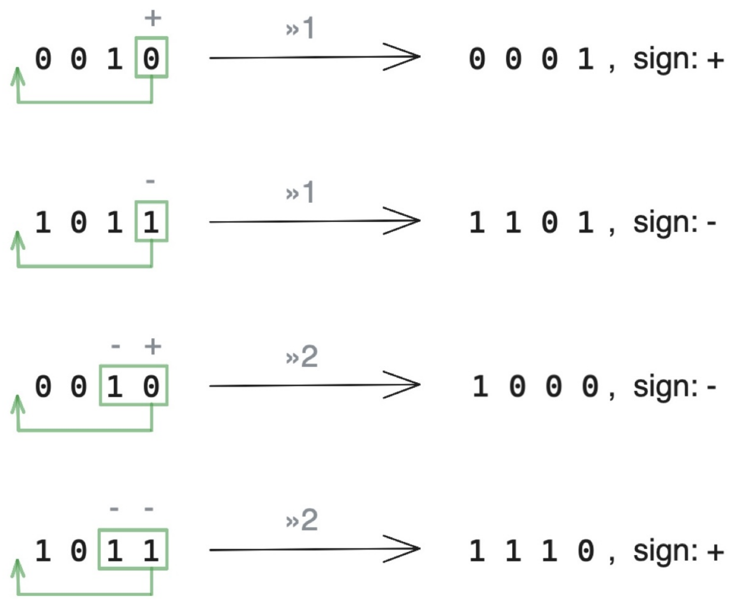


Figure 5: Bit-wise circular shift of gray code and associated sign. It involves shifting a specific bit from right to left. The sign of the resulting number is determined by whether the count of 1s shifted is even (resulting in a positive sign) or odd (resulting in a negative sign).

When rotating the tetrahedron from its reference configuration to another, the mixed scalar products outlined in equation (8 must be scaled by the signs obtained from the bit-wise circular shift.

3 Data Structures

The data structures used to describe the mesh are detailed in section 3.1. Designing an efficient CUDA Eikonal solver necessitates a focus on optimizing throughput rather than latency, leveraging the GPU's streaming capabilities. Central to this optimization is the principle of coalesced memory access, which maximizes memory bandwidth utilization. Our approach hinges on a critical design decision: permitting data redundancy to enhance coalesced memory access patterns, throughput, and occupancy. Although this approach introduces redundancy in the data, it significantly enhances performance by improving memory access patterns, increasing the number of active threads, and boosting overall bandwidth. These enhancements are critical for achieving a fast and efficient GPU algorithm.

3.1 Mesh

The `Mesh` class, defined as `template <typename Float> class Mesh`, encapsulates all geometric information about the domain $\Omega \subset \mathbb{R}^3$ within its private members.

Displayed in Figure 6, the class comprises various data structures:

- `geo` is a vector of type `Float` designed to hold node coordinates within the mesh. These coordinates are logically arranged in sets of three consecutive elements, each set representing the position of a node. For instance, for a given node index i, the coordinates (x, y, z) of the node can be retrieved as (`geo[i]`, `geo[i+1]`, `geo[i+2]`).
- `tetra` is a vector of type `int` designed to store all tetrahedra within the domain. These tetrahedra are logically organized in groups of four consecutive elements, each group representing the vertices of a tetrahedron. Specifically, the elements correspond to indices of the nodes stored in `geo`. For a particular tetrahedron t, its vertices have node indices `tetra[t]`, `tetra[t+1]`, `tetra[t+2]`, and `tetra[t+3]`. Note that the vertex `tetra[t+3]` is the one associated with the reference configuration ϕ_4, while the vertex `tetra[t+i]` with $i \in 1, 2, 3$ is associated to the configuration ϕ_i (as discussed in section 2).
- `partitions_number` is an integer indicating the number of subdomains dividing the domain.
- `partitions_vertices`, a vector of `int` with size `partitions_number`, delineates subdomain boundaries according to vertex indices.
- `M` is a vector of type `Float` holding six scalar products for each tetrahedron, as detailed in Table 3.
- `shapes` is a vector of `TetraConfig`, where each element holds the tetrahedra indices containing a particular node i. Here, `TetraConfig` is a `struct` including the tetrahedron index and the associated configuration ϕ_j for i in the tetrahedron.

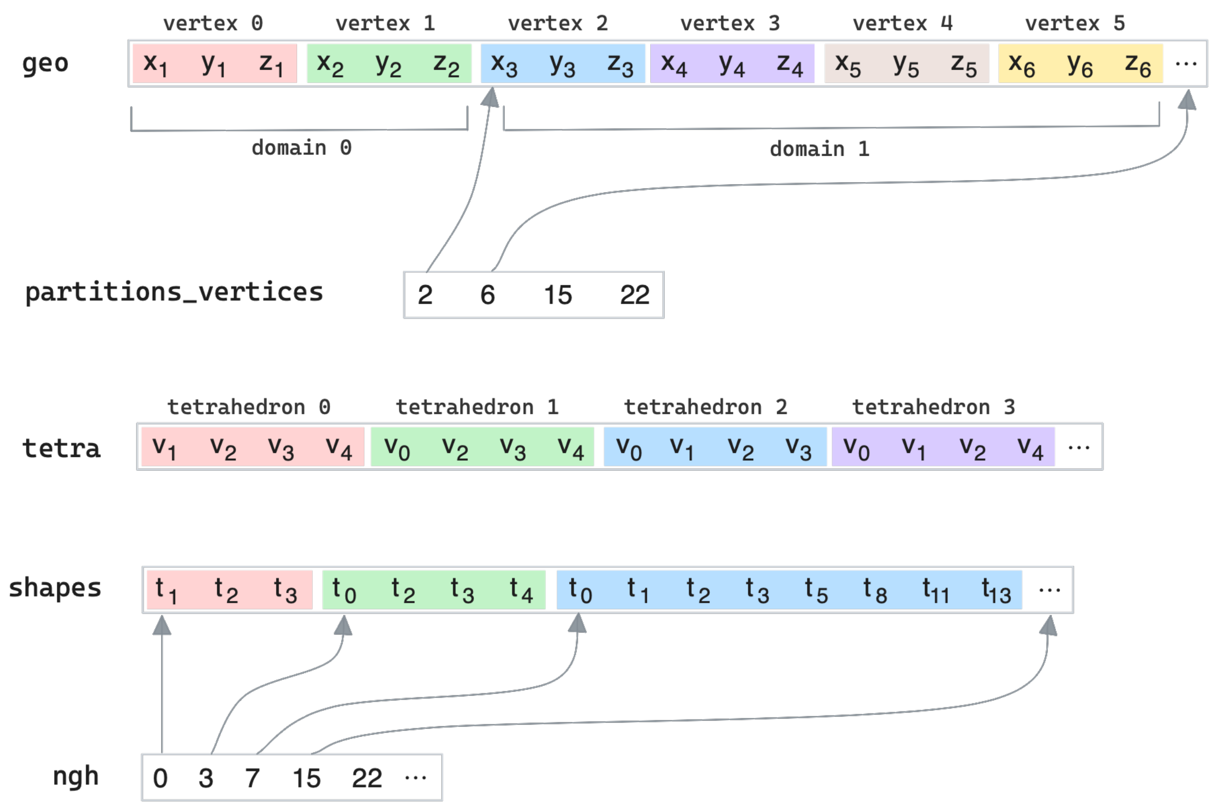


Figure 6: Data Structures in Mesh.cuh

- `ngh` is a vector of integers indicating the grouping boundaries in `shapes`. Typically, the tetrahedra linked to node i in `geo` are confined within the sub-array of `shapes` bordered by indices `ngh[i]` and `ngh[i+1]-1`.

3.2 Construction of the `Mesh`

The mesh is constructed by providing a `vtk` file and specifying the number of subdomains. Additionally, users need to supply either a 3x3 velocity matrix (as described by matrix M in 2) applicable to all tetrahedra, or alternatively, a file containing a distinct 3x3 velocity matrix for each tetrahedron within the mesh. Furthermore, domain decomposition is carried out using the METIS software [1], which partitions both nodes and tetrahedra into the specified number of subdomains.

The construction process consists of the following steps:

1. Read node coordinates from the `vtk` file and store them in `geo`.
2. Eliminate duplicate nodes.

3. Read tetrahedral information from the `vtk` file and store it in a temporary set of sets, where each element set represents a tetrahedron.
4. Retrieve matrices from the designated `vtk` file or duplicate the provided velocity matrix for each tetrahedron, storing them in a temporary vector of `Matrix` objects, `tempM`.
5. Perform domain partitioning using METIS.
6. Read METIS files containing node partition information, then reorder nodes in `geo` accordingly; `partitions_vertices` is concurrently generated.
7. Read METIS files containing tetrahedra partition information, then reorder both tetrahedra in `tetra` and `tempM` based on METIS results;
`partitions_tetrahedra` and `M` (derived from reordered `tempM`) are constructed simultaneously.
8. Create `shapes` and `ngh` by leveraging the set of sets containing tetrahedra.

It's worth noting that reordering nodes and tetrahedra ensures coalesced memory access and maximizes memory locality.

4 CUDA Solver

The class `Solver`, a templated class defined as `template <typename Float> class Solver`, is designed for solving the Eikonal equation using CUDA. It consists of functionalities for transferring data between the host and device, and executing the Eikonal equation solver on the GPU. Taking a pointer to a Mesh object as input, which encapsulates essential geometry and data, this class leverages CUDA kernels defined in an external file to execute parallel computations efficiently on the GPU, thereby efficiently solving the Eikonal equation.

The `solve` method serves as the chief conductor for the solution process of the Eikonal equation. It initializes CUDA streams for each subdomain and iteratively performs domain processing (according to 2) until convergence or a maximum number of iterations is reached. During each iteration, it checks the activity status of each subdomain and launches CUDA kernels to compute the solution over active subdomains. After convergence, it copies the solutions from the device to the host and generates a `vtk` file for visualization. Finally, it deallocates memory and destroys CUDA streams to clean up resources.

4.1 Data Structures

Before the `solve` method is invoked, all relevant attributes in the `Mesh` class are transferred from the host to the device's global memory.

Within the `solve` method, several arrays are utilized — one for each domain, as illustrated in 1. For each domain, all of these arrays, except for the `predicate` array, have a length corresponding to the number of nodes within their respective domains. The `predicate` array, however, has a length equal to the total number of nodes in the entire mesh, encompassing nodes from all domains.

It is important to understand the roles of the `predicate` and `activeList` arrays. The `predicate` array is used to indicate whether a node requires further processing in the `processNodes` kernel, as will be detailed in section 4.3. For each domain, only the `predicate[domain]` entries associated with nodes within that domain are considered if set to 1; entries corresponding to nodes in other domains are used to trigger the activation of nodes across domains when the `propagatePredicateBetweenDomains` kernel is executed, as explained below in section 4.3.

For each domain, the entries in the `activeList` array can only have values of 0, 1, or 2. Specifically, if `activeList[node]` is set to 0, it indicates that the `node` is not part of the active list (as described in [4]). If the value is 1, the `node` is included in the active list. Finally, if the value is 2, the `node` has been removed from the active list and will not be processed by the `processNodes` kernel, even if its corresponding `predicate` entry is set to 1.

Algorithm 1 Eikonal Solver Scheme

```
while not converged do
    for each domain do
        while countActiveNodes(activeList[domain]) > 0 do
            sAddr[domain] ← exclusiveScan(activeList[domain])
            cList[domain] ← compact(sAddr[domain],activeList[domain])
            nbhNr[domain] ← countNbhs(cList[domain])
            sAd[domain] ← exclusiveScan(nbhNr[domain])
            elemList[domain] ← gatherElements(sAd[domain], cList[domain])
            predicate[domain] ← constructPredicate(elemList[domain],
                                    sAddr[domain], activeList[domain])
            if countActiveNodes(predicate[domain]) != 0 then
                s ← exclusiveScan(predicate[domain])
                conList[domain] ← compact(s, predicate[domain])
                nbhNr[domain] ← countNbhs(conList[domain])
                sAd[domain] ← exclusiveScan(nbhNr[domain])
                ptList[domain] ← gatherElements(sAd[domain], conList[domain])
                processNodes()
            end if
            activeList[domain] ← removeConvergedNodes(activeList[domain])
            predicate[domain] ← setToZero()
        end while
    end for
    propagatePredicateBetweenDomains()
    check for convergence
end while
Copy solutions from device to host
Write solution in vtk file
```

4.2 Detailed description of `solve` method

The algorithm continues running until convergence is achieved, meaning all nodes in each domain have converged. During each iteration, all domains are processed in parallel using CUDA streams. For each domain, the following steps are executed:

- Lines 4-8 of 1 collect all neighboring tetrahedra of active nodes into a contiguous array called `elemList[domain]`.
- In line 9, the `predicate` array is constructed as described in section 4.3 and shown in 3.
- Lines from 12 to 17 are executed only if one active neighbor is found in `predicate`, indicating further processing is required.In this case, the operations from lines 4-8 are repeated based on the `predicate` array, and the gathered elements are processed using the `processNodes()` kernel, detailed in section 4.3 and algorithm 2.
- In line 19, all entries in `activeList[domain]` set to 2 are reset to 0.
- In line 20, all `predicate[domain]` entries corresponding to nodes within the domain are reset to 0.
- Line 23 calls the `propagatePredicateBetweenDomains()` kernel, which propagates the activation of nodes across domains, as explained in section 4.3.
- In line 24, the iteration stops only if convergence is reached, meaning no active nodes remain in any domain.
- Lines 26-27 handle copying and writing the final solution to a file.

Notably, there is no data transfer between the host and device in this algorithm, except for the result of the `countActiveNodes` kernel (which is needed for conditional statements on the host) and for writing the final solution to an output file.

4.3 Kernels

There are many kernels used in the `solve` method:

- The so-called `compact` kernel basically filters elements based on an array `predicated`, in particular the thread with global id `tid` within the CUDA grid will write `tid + offset` in position `map[tid]` of an `output` array if and only if `predicate[tid]` is 1. Essentially, it compacts the input indices based on the predicate, storing results in specific mapped locations in the output array.
- The kernel `countNbhs` calculates the number of neighbors for a set of active nodes and stores the result in an output array.
- `gatherElements` is a CUDA kernel that basically gathers neighbours of active nodes and stores them into a contiguous output array.

Algorithm 2 Process Nodes Scheme

```
1: procedure PROCESSNODES(...)
2:     tetraIndex ← compute tetra index associated to the CUDA thread
3:     if tetraIndex is not out-of-bound then
4:         tetra ← retrieve information about the tetraIndex tetrahedron
5:         node ← retrieve index of the node associated with tetra.config
6:         if activeList[node] != 0 then
7:             return
8:         end if
9:         old_sol ← solution[node]
10:        if this thread is the first then
11:            shared_sol ← old_sol
12:        end if
13:        coords ← retrieve coordinates of all nodes in tetra
14:        values ← retrieve solutions of all nodes in tetra
15:        M ← retrieve velocity matrix associated to tetra
16:        sol ← localSolver(coords, values, M)
17:        shared_sol ← min(shared_sol, sol)
18:        if this thread is the first and shared_sol < old_sol then
19:            solution[node] ← shared_sol
20:            activeList[node] ← 1
21:        end if
22:    end if
23: end procedure
```

- `constructPredicate` This kernel's purpose is to determine which nodes have converged and which ones require further processing. In particular, the kernel is started with the following configuration: each active node is assigned to a block, and each node's neighbour tetrahedron is assigned to a block's thread. All threads within a block collectively solve the local problem for the tetrahedron it is assigned to, with respect to the block's node. If, according to the found solution, the node is deemed to have converged, then all its neighbour nodes are required to undergo further processing. Therefore, such nodes' entries in the predicate array are set to 1. However, it is necessary to ensure that converged nodes are not set for further processing by other nodes' threads within the same subdomain. Therefore, all converged nodes' entries in the active_list are set to 2. As the last step, the solution vector for the block's node is updated with the newly computed solution.

- `processNode` This kernel's purpose's is to process all nodes set for further processing by the `constructPredicate` kernel. This kernel is started with the following configuration: each node set for further processing is assigned to a block, and each node's neighbour tetrahedron is assigned to a block's thread. If the block's node's entry in the active_list array is not set to 0 the kernel for that block returns immediately. Otherwise, all threads within a block collectively solve the local problem for the tetrahedron it is assigned to, with respect to the block's node. If the newly found solution is better

Algorithm 3 Construct Predicate Scheme

```
procedure CONSTRUCTPREDICATE(...)
    tetraIndex ← compute tetra index associated to the CUDA thread
    if tetraIndex is not out-of-bound then
        tetra ← retrieve information about the tetraIndex tetrahedron
        node ← retrieve index of the node associated with tetra.config
        old_sol ← solution[node]
        if this thread is the first then
            shared_sol ← old_sol
        end if
        coords ← retrieve coordinates of all nodes in tetra
        values ← retrieve solutions of all nodes in tetra
        M ← retrieve velocity matrix associated to tetra
        sol ← localSolver(coords, values, M)
        shared_sol ← min(shared_sol, sol)
        pred ← ‖ shared_sol-old_sol ‖ tol*(1+0-5*(shared_sol+old_sol))
        if pred == 1 then
            for each node n in tetra except for node do
                if solution[n]-shared_sol > 0 then
                    predicate[n] ← 1
                end if
            end for
            if this thread is the first then
                activeList[node] ← 2
            end if
        end if
        if this thread is the first then
            solution[node] ← min(solution[node],shared_sol)
        end if
    end if
end procedure
```

than the old one, then the solution array is updated and the block's node's entry in the active_list array is set to 1.

- `removeConvergedNodes` This kernel's purpose is to map all 2 in active_list array to 0, with the naive configuration of one thread per node.

- `propagatePredicateBetweenDomains` When the `constructPredicate` kernel sets to 1 `predicate[domain][node]`, with `node` not belonging to `domain`, then such node is not processed by the `processNodes` kernel. However, such node needs to be processed at the following iteration by the domain `domain_node` it belongs to.
  To allow this, the `propagatePredicateBetweenDomains` kernel ensures that `active_list[domain_node][node]` is set to 1.

Additionally, for other kernels, for example `exclusiveScan` and `countActiveNodes`, we made use of the `thrust` [3] CUDA library to guarantee optimal performance.

4.4 Optimisations

To optimize data transfer, all data members of the Mesh class are collectively allocated and transferred to the GPU, thereby maximizing bandwidth utilization.

To fully exploit the available computational resources, we implemented several strategies for workload management and inter-block communication. Each subdomain is assigned to a separate CUDA stream, with these streams managed using a thread-based approach to prevent blocking operations within a stream from stalling the entire execution. To minimize the impact of thread creation, all threads are instantiated at the start of the algorithm and reused as needed.

For the most computationally intensive kernels, each block is mapped to an active node, with each CUDA thread within a block assigned to one of the tetrahedra connected to that node. While ideally, we would compute the exact number of tetrahedra adjacent to each node, to avoid this overhead, we instead assign each block a number of threads equal to the maximum number of adjacent tetrahedra.

Furthermore, to reduce communication overhead between subdomains, the algorithm is structured so that communication occurs only after all subdomains have completed their computations. At this stage, each subdomain is notified by the others of which of its nodes need to be activated.

5 Performance Analysis

5.1 Experimental Setup

CPU	**Accelerator**	**RAM**
Intel Core i7 - 1065G7 @ 1.3GHz	Nvidia GeForce MX250 - 4GB	16GB

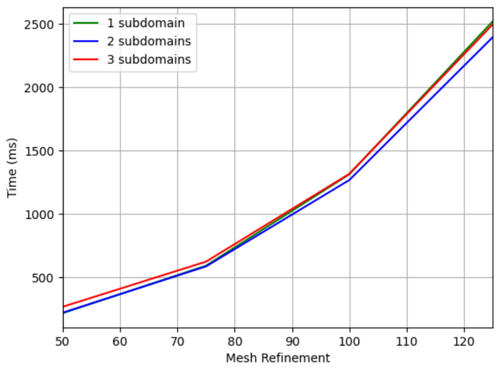
Table 4: System Overview.

The benchmark was conducted on the system described in table 4.

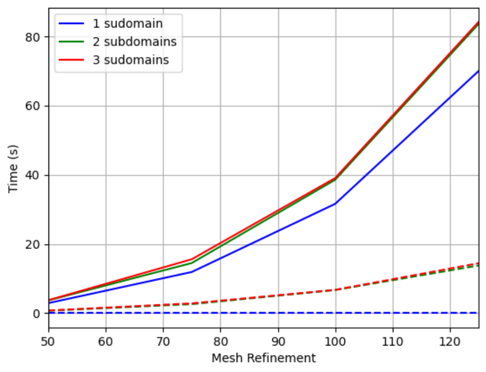
5.2 Experimental Results

To evaluate our implementation, we conducted a series of tests using various mesh refinements and subdomain configurations. Specifically, we used cubic meshes with spatial steps of 0.02m (1/50), 0.013m (1/75), 0.01m (1/100), and 0.008m (1/125).

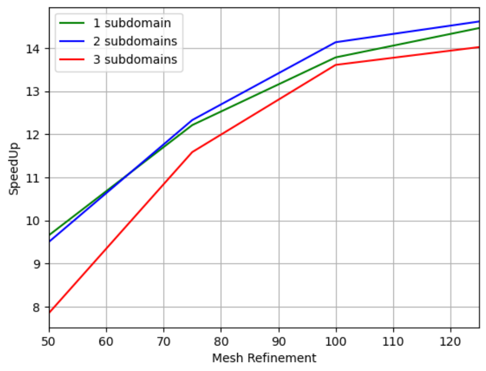
Additionally, we performed tests to compare the speed-up of our implementation against a previous version based on OpenMP. The results are presented in the following graphs and images:



(a) Solver time (ms) as a function of mesh refinement for different number of subdomains.

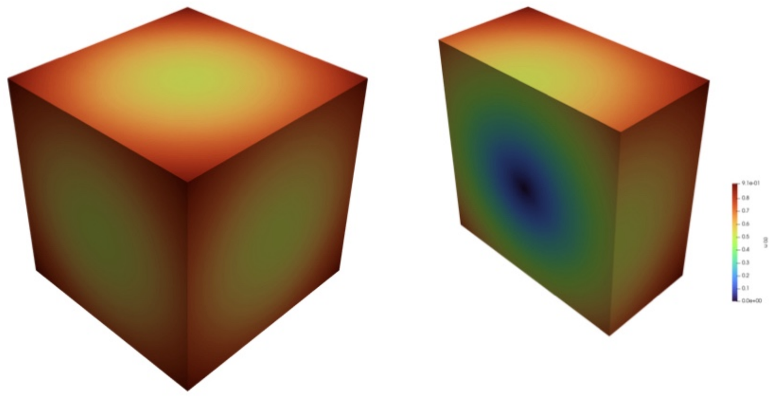


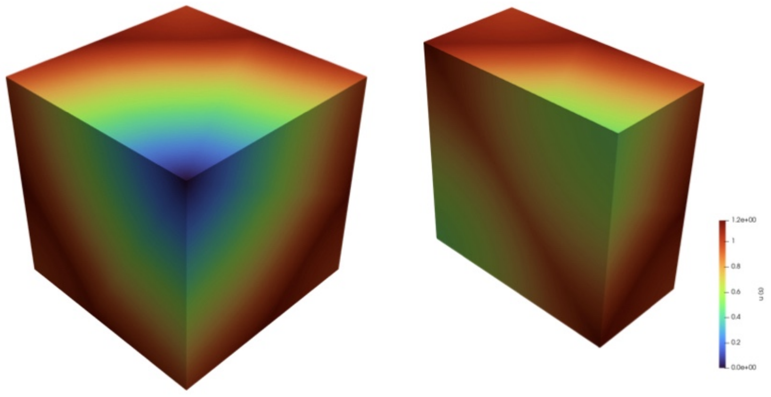
(b) The thick line represents the total time to construct the mesh, while the dashed line shows the time required by METIS for partitioning, plotted as a function of mesh refinement for different numbers of subdomains.



(c) Speedup compared to our previous OpenMP parallel version using 4 threads.

Figure 7: Performance plots showing the speed-up and execution time for different mesh refinements and subdomain configurations.


(a) Simulation with one wave source point corresponding to the center of the cube.


(b) Simulation with two wave sources corresponding to the antipodal nodes of the cube.

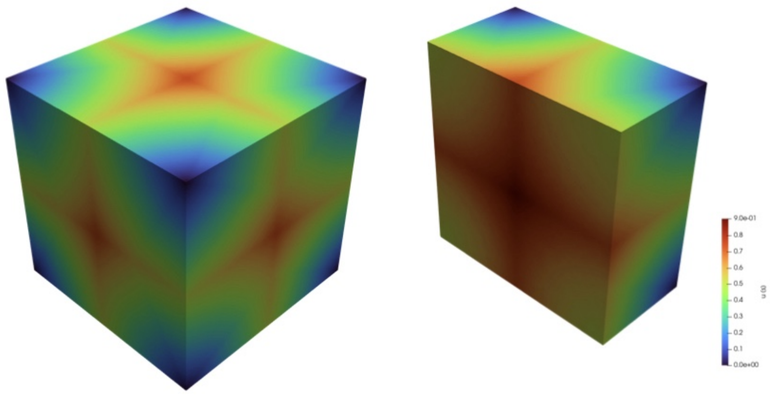

(c) Simulation with eight wave sources positioned at all the vertices of the cube.

Figure 8: Various simulation using a cuboid mesh with dimensions 1m×1m×1m with mesh refinement ratio of 1:75.

5.3 Analysis

Firstly, as shown in Figure 7(c), our implementation achieves a significant speed-up compared to the thread-based version. While this result might be expected, it's important to note that this case presents a considerable challenge. The implemented kernels are highly memory-intensive, which is atypical for GPU applications, which are generally more compute-intensive.

Additionally, Figure 7(a) illustrates how our application scales with varying numbers of partitions. The results, however, are not entirely satisfactory. Specifically, using two subdomains yields only a slight improvement over the single-partition case, while using three subdomains leads to the same or even worse performance. This behaviour likely stems from the limited computational resources available. A multi-domain approach typically proves beneficial only when processing the entire domain without partitioning fully utilizes the available resources, necessitating scaling to a multi-GPU scenario. Since that was not possible in our case, adding more domains merely increases overhead without offering significant benefits.

Although the primary focus of this project was optimizing the code for solving the Eikonal equation, it is also worth examining the impact of mesh construction and partitioning. As shown in Figure 7(b), our implementation consistently takes more time to execute these tasks than to solve the equation itself. This can be attributed to the use of basic algorithms for mesh handling, which are clearly not state-of-the-art. Notably, the time required for mesh partitioning becomes increasingly negligible as the mesh resolution improves.

6 Conclusions and future works

In summary, our implementation has demonstrated significant improvement over the existing thread-based approach. While the multi-domain strategy did not provide substantial benefits with the currently available computational resources, it is likely that further performance gains could be realized with enhanced resources.

Moreover, although mesh construction and partitioning were not the primary focus of our project, they emerged as the key bottlenecks in the application. Future efforts should consider parallelizing these processes to fully exploit parallel computing capabilities. Additionally, exploring alternative workload management strategies could be crucial in unlocking further performance improvements.

References

[1] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[2] Notes on local problem by optimization.

[3] Thrust: The c++ parallel algorithms library, 2024. `https://nvidia.github.io/cccl/thrust/index.html#thrust-module`.

[4] Zhisong Fu, Robert Kirby, and Ross Whitaker. A fast iterative method for solving the eikonal equation on tetrahedral domains. *SIAM journal on scientific computing : a publication of the Society for Industrial and Applied Mathematics*, 35:c473–c494, 01 2013.

[5] Daniel Ganellari, Gundolf Haase, and Gerhard Zumbusch. A massively parallel eikonal solver on unstructured meshes. *Comput. Vis. Sci.*, 19(5–6):3–18, dec 2018.

[6] Eric W. Weisstein. “Gray Code.” From MathWorld–A Wolfram Web Resource. `https://mathworld.wolfram.com/GrayCode.html`.