HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Capstone project report

Parallel Programming for 2D Shallow Water Equation (MPI-CUDA)

Submitted by

Nguyen Nhat Minh - 20225510

Ngo Duy Dat - 20225480

Vu Tuan Truong - 20225535

Nguyen Hoang Son Tung - 20225536

Doi Sy Thang - 20225528

Guided by

Ph.D. Vu Van Thieu

Abstract

Parallel programming is a key approach for speeding up numerical algorithms. This paper focuses on parallelizing a program designed to solve the **two-dimensional shallow water equation**, which serves as an example of a conservation law. The equation is tackled using finite difference and finite volume techniques, applying schemes such as the Lax-Friedrichs and Lax-Wendroff methods (we exploit **3 algorithms**). The parallel implementation employs **both MPI and CUDA**. Performance evaluation is primarily conducted on shared memory systems, and due to the availability of a sufficient number of processors, results are also presented for MPI and CUDA.

Contents

Part I
Problem Formulation

Physical simulation is a prominent task in parallel computing, as it often requires solving systems of partial differential equations. In this project, we focus on the 2D shallow water equations—an important model in physical simulations—and explore how to solve them efficiently using parallel computing techniques [1][3]. In our problem, we specifically consider the two-dimensional (2D) shallow water equations, which are a set of hyperbolic partial differential equations that describe the flow below a pressure surface in a fluid. These equations are widely used in oceanography, meteorology, and hydraulic engineering to model phenomena such as tides, storm surges, and tsunamis.

1 Conservation Law

In the simulation of physical systems, especially in fluid dynamics, conservation laws form the fundamental basis for modeling the behavior of fluids over time and space. These laws describe how physical quantities such as mass, momentum, and energy are preserved in a closed system, and are typically expressed as partial differential equations (PDEs). Two of the most important conservation principles are mass conservation and momentum conservation, which are captured in the Euler equations for inviscid flows.

1.1 Mass Conservation

The law of mass conservation, or the *continuity equation*, states that mass cannot be created or destroyed. In differential form, it is written as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{V}) = 0$$

Where:

- ρ is the fluid **density**,
- $\mathbf{V}$ is the fluid **velocity vector**,
- $\nabla\cdot$ denotes the divergence operator.

This equation ensures that any change in density within a control volume is balanced by the net mass flux across its boundaries.

1.2 Momentum Conservation

The conservation of momentum is derived from Newton's second law and expresses how the momentum of a fluid element changes due to external forces. It is expressed as:

$$\frac{\partial (\rho \mathbf{V})}{\partial t} + \nabla \cdot (\rho \mathbf{V} \otimes \mathbf{V}) + \nabla p = 0$$

Where:

- $\rho\mathbf{V}$ is the **momentum density**,
- $\rho\mathbf{V} \otimes \mathbf{V}$ is the **momentum flux tensor**,
- p is the **pressure**,
- $\mathbf{g}$ is the **gravitational acceleration vector**,
- ∇p represents the **pressure gradient**.

This equation models the transport of momentum in the fluid, accounting for both internal pressure forces and external body forces such as gravity.

These conservation laws are fundamental in describing real-world phenomena such as weather systems, ocean currents, and hydraulic flows. They serve as the foundation for simplified models like the 2D shallow water equations, which we focus on in this project. Efficient numerical solutions to these equations, especially in a parallel computing environment, enable accurate and scalable physical simulations.

2 Physical Modeling of the Two-Dimensional Shallow Water Equation

To model the behavior of shallow water in two spatial dimensions, we begin with the conservation laws of mass and momentum. Assuming a fluid domain that is shallow relative to its horizontal dimensions and neglecting external forces such as friction or Coriolis effects, the shallow water equations can be derived from the depth-integrated Euler equations.

Let the fluid flow be defined in the x- and y-directions, and assume a hydrostatic pressure distribution. The resulting system of partial differential equations describes the time evolution of the water height and horizontal velocities.

The two-dimensional shallow water equations are given by:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial hv}{\partial y} + \dfrac{\partial hu}{\partial x} = 0 & (1) \\ \dfrac{\partial (hv)}{\partial t} + \dfrac{\partial \left(hv^2 + \frac{1}{2}gh^2\right)}{\partial y} + \dfrac{\partial huv}{\partial x} = 0 & (2) \\ \dfrac{\partial (hu)}{\partial t} + \dfrac{\partial huv}{\partial y} + \dfrac{\partial \left(hu^2 + \frac{1}{2}gh^2\right)}{\partial x} = 0 & (3) \end{cases}$$

where:

- $h(x, y, t)$ is the water depth at position (x, y) and time t,
- $u(x, y, t)$ is the horizontal velocity component in the x-direction,
- $v(x, y, t)$ is the horizontal velocity component in the y-direction,
- g is the gravitational acceleration constant.

2.1 Vector Form of the Shallow Water Equations

For compactness and computational convenience, the system can be written in vector form using the conserved variables and flux functions. Let the state vector U and flux vectors $F(U)$ and $G(U)$ be defined as:

$$U = \begin{pmatrix} h \\ hv \\ hu \end{pmatrix}, \quad F(U) = \begin{pmatrix} hv \\ hv^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix}, \quad G(U) = \begin{pmatrix} hu \\ huv \\ hu^2 + \frac{1}{2}gh^2 \end{pmatrix}$$

2.2 Conservation Form

With these definitions, the two-dimensional shallow water equations take the conservative form:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} = 0$$

This formulation highlights the conservation of mass and momentum and provides a foundation for numerical discretization methods, such as finite volume or discontinuous Galerkin methods, for solving fluid flow problems in shallow domains.

2.3 Boundary Conditions of 2D Shallow Water Equations

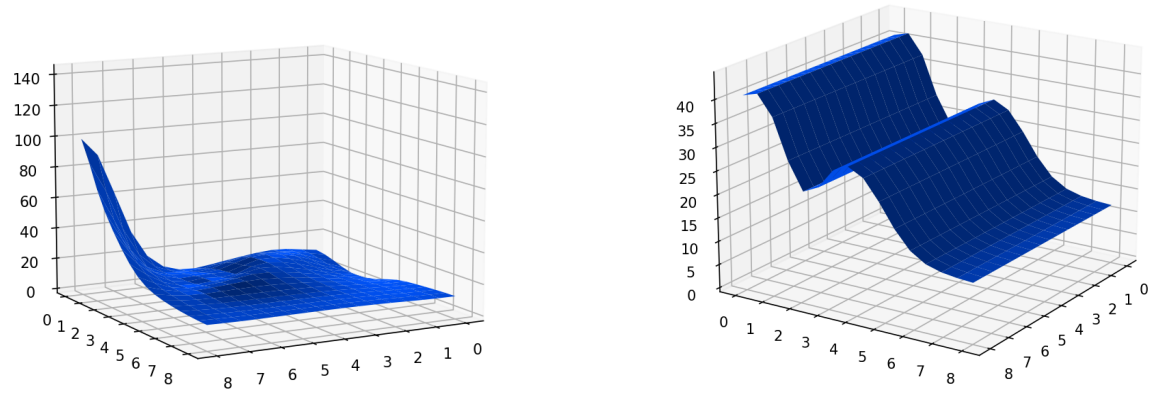
To numerically solve the two-dimensional shallow water equations, it is necessary to define appropriate boundary conditions. The choice of boundary conditions has a significant impact on the flow behavior near domain edges and the overall accuracy and stability of the simulation. In our work, we mention two common types of boundary conditions for the 2D shallow water model, including reflective and transmissive.

Reflective Boundary Condition

Reflective boundary conditions are used to model impermeable solid walls, where the normal velocity component at the boundary is zero. The flow is reflected back into the domain, mimicking the presence of a rigid barrier. For a boundary aligned with the x-axis (e.g., vertical wall), the conditions can be written as:

$$\begin{aligned} u_0 &= u_{n-1} = -u_{ghost} \quad \text{(reverse vertical velocity)} \\ v_0 &= v_{n-1} = v_{ghost} \quad \text{(preserve horizontal velocity)} \\ h &= h \quad \text{(no change in water height)} \end{aligned}$$

This condition is typically applied by setting ghost cells such that the vertical component of the flow is mirrored and the horizontal and depth values are preserved. In contrast, the reflective boundary condition in the y-axis was preserved vertical velocity and depth, while the horizontal velocity reflected into the domain.



(a) Reflective Condition Bound

(b) Transitive Condition Bound

Figure 1 . Simulation of 2D shallow water equation with reflective and transmissive condition bound.

Transmissive Boundary Condition

Transmissive boundary conditions allow waves and disturbances to exit or enter the computational domain without artificial reflection. These boundaries are used when the flow at the boundary is not constrained, simulating open boundaries. The simplest form assumes zero gradient across the boundary:

$$\begin{aligned} u_0 &= u_{n-1} = u_{ghost} \quad \text{(preserve vertical velocity)} \\ v_0 &= v_{n-1} = v_{ghost} \quad \text{(preserve horizontal velocity)} \\ h &= h \quad \text{(no change in water height)} \end{aligned}$$

In practical implementations, this is achieved by copying the values from the interior adjacent cell to the ghost cells along the boundary, effectively imposing a Neumann condition with zero derivative.

Part II
Numerical Methods

1 Finite Different Method

The finite difference method (FDM) is one of the most widely used numerical techniques for solving partial differential equations, including the 2D shallow water equations. It works by discretizing the spatial and temporal domains into a grid and approximating derivatives using differences between neighboring grid points. In this section, we introduce the Lax-friedrichs and Lax-Wendroff schemes for the finite difference method.

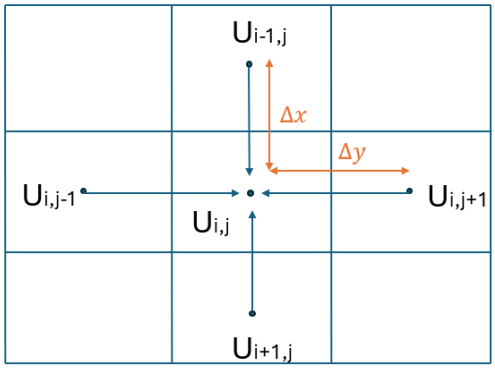


Figure 2 . Finite difference computation matrix

Lax-Friedrichs scheme

The Lax-Friedrichs scheme is a simple and robust first-order finite difference method for solving hyperbolic partial differential equations such as the 2D shallow water equations. It introduces numerical dissipation, which helps to stabilize the solution, especially in the presence of discontinuities or sharp gradients.

$$\begin{cases} \dfrac{\partial U}{\partial t} \approx \dfrac{U_{i,j}^{t+1} - U_{i,j}^{t}}{\Delta t}, \\ \dfrac{\partial F(U)}{\partial x} \approx \dfrac{F(U_{i+1,j}^{t}) - F(U_{i-1,j}^{t})}{2\Delta x}, \\ \dfrac{\partial G(U)}{\partial y} \approx \dfrac{G(U_{i,j+1}^{t}) - G(U_{i,j-1}^{t})}{2\Delta y}. \end{cases}$$

Approximate different equation with difference value:

$$\frac{U_{i,j}^{t+1} - U_{i,j}^{t}}{\Delta t} + \frac{F(U_{i+1,j}^{t}) - F(U_{i-1,j}^{t})}{2\Delta x} + \frac{G(U_{i,j+1}^{t}) - G(U_{i,j-1}^{t})}{2\Delta y} = 0$$

$$U_{i,j}^{t+1} = U_{i,j}^{t} - \frac{\Delta t}{2\Delta x}(F(U_{i+1,j}^{t}) - F(U_{i-1,j}^{t})) - \frac{\Delta t}{2\Delta y}(G(U_{i,j+1}^{t}) - G(U_{i,j-1}^{t}))$$

$$\boxed{U_{i,j}^{t+1} = \tfrac{1}{4}(U_{i+1,j}^{t} + U_{i-1,j}^{t} + U_{i,j+1}^{t} + U_{i,j-1}^{t}) - \tfrac{\Delta t}{2\Delta x}(F(U_{i+1,j}^{t}) - F(U_{i-1,j}^{t})) - \tfrac{\Delta t}{2\Delta y}(G(U_{i,j+1}^{t}) - G(U_{i,j-1}^{t}))} \quad (1)$$

Lax-Wendroff scheme

The Lax-Wendroff scheme is a second-order accurate finite difference method for solving hyperbolic conservation laws, such as the 2D shallow water equations. It improves upon the Lax-Friedrichs

method by reducing numerical dissipation and achieving higher accuracy, particularly for smooth solutions. Because the Lax-Wendroff scheme employs a second-order time discretization, it produces smoother and sharper wave propagation compared to the Lax-Friedrichs scheme. However, this can lead to numerical instabilities and potential overflow during simulation. To mitigate this issue, an entropy-based diffusion term is added to the partial differential equations to stabilize the system when using the Lax-Wendroff method.

2D Shallow Water system with entropy noise:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}$$

Using finite different method and taylor expansion for second order of time, then the final formula show below:

$$\begin{aligned}
U_{i,j}^{t+1} &= U_{i,j}^{t} - \frac{\Delta t}{\Delta x}\left(F(U_{i+1,j}^{t+\frac{1}{2}}) - F(U_{i-1,j}^{t+\frac{1}{2}})\right) \\
&\quad - \frac{\Delta t}{\Delta y}\left(G(U_{i,j+1}^{t+\frac{1}{2}}) - G(U_{i,j-1}^{t+\frac{1}{2}})\right) + \alpha_x \cdot U_x^{\text{entropy}} + \alpha_y \cdot U_y^{\text{entropy}} \\
U_{i,j}^{t} &= \frac{1}{4}\left(U_{i+1,j}^{t} + U_{i-1,j}^{t} + U_{i,j-1}^{t} + U_{i,j+1}^{t}\right) \\
U_x^{\text{entropy}} &= U_{i+1,j}^{t} - 2U_{i,j}^{t} + U_{i-1,j}^{t} \\
U_y^{\text{entropy}} &= U_{i,j+1}^{t} - 2U_{i,j}^{t} + U_{i,j-1}^{t}
\end{aligned}$$

2 Finite Volume Method

The finite volume method (FVM) is a robust numerical technique commonly used to solve conservation laws, including the two-dimensional shallow water equations. Unlike the finite difference method (FDM), which approximates derivatives at discrete points using neighboring point values, the FVM operates by integrating the governing equations over finite control volumes (cells). This formulation directly enforces the conservation of physical quantities such as mass and momentum within each cell. While the FDM is often simpler to implement and analyze, especially on structured grids, it does not guarantee conservation across cell interfaces unless specifically designed to do so. In contrast, the FVM naturally ensures local and global conservation, making it especially suitable for fluid dynamics problems where preserving integral quantities is essential.

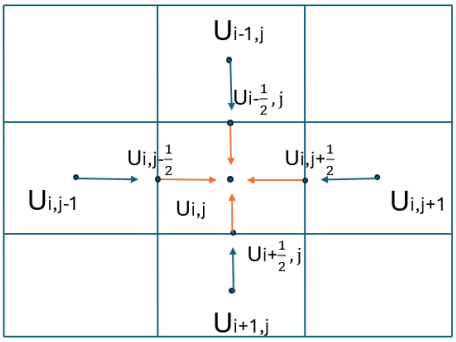


Figure 3 . Finite volume computation matrix

Lax-Friedrich scheme

The Lax-Friedrichs scheme is a foundational numerical method for solving hyperbolic conservation laws, known for its simplicity, robustness, and diffusive stability. When adapted to the finite volume method, it operates by updating the cell-averaged conserved variables through flux computations at control volume interfaces. The scheme achieves first-order accuracy in both space and time and is particularly valued for its ability to handle discontinuities and shocks without introducing spurious oscillations.

$$U_{i,j}^{t+1} = U_{i,j}^{t} - \frac{\Delta t}{2\Delta x}\left(F(U_{i+\frac{1}{2},j}^{t}) - F(U_{i-\frac{1}{2},j}^{t})\right) - \frac{\Delta t}{2\Delta y}\left(G(U_{i,j+\frac{1}{2}}^{t}) - G(U_{i,j-\frac{1}{2}}^{t})\right)$$

$$U_{i,j}^{t} = \frac{1}{4}\left(U_{i+1,j}^{t} + U_{i-1,j}^{t} + U_{i,j+1}^{t} + U_{i,j-1}^{t}\right)$$

$$F(U_{i+\frac{1}{2},j}^{t}) = \frac{1}{2}\left(F(U_{i,j}^{t}) - F(U_{i+1,j}^{t})\right), \quad F(U_{i-\frac{1}{2},j}^{t}) = \frac{1}{2}\left(-F(U_{i,j}^{t}) + F(U_{i-1,j}^{t})\right)$$

$$G(U_{i,j+\frac{1}{2}}^{t}) = \frac{1}{2}\left(G(U_{i,j}^{t}) - G(U_{i,j+1}^{t})\right), \quad G(U_{i,j-\frac{1}{2}}^{t}) = \frac{1}{2}\left(-G(U_{i,j}^{t}) + G(U_{i,j-1}^{t})\right)$$

3 CFL Stability Condition

To ensure numerical method stability in time-dependent simulations of hyperbolic partial differential equations such as the 2D shallow water equations, it is essential to satisfy the Courant–Friedrichs–Lewy (CFL) condition. This condition restricts the size of the time step Δt relative to the spatial grid sizes Δx and Δy, based on the maximum wave propagation speed in the domain. In the context of the shallow water equations, the wave speeds are governed by the fluid velocity components u, v, and the gravitational wave speed $\sqrt{gh}$, where g is the gravitational constant and h is the local water depth. The CFL condition can be expressed as:

$$\Delta t \leq \mathbf{C} \cdot \min\left(\frac{\Delta x}{|u| + \sqrt{gh}}, \frac{\Delta y}{|v| + \sqrt{gh}}\right)$$

Here, C $\in (0, 1]$ is a user-defined safety coefficient, typically less than or equal to 1, which controls how aggressively the time step approaches the stability limit.

This condition is critical for both finite difference and finite volume methods, though in the finite volume framework, it also ensures flux conservation across cell interfaces, especially when using explicit schemes like Lax-Friedrichs or Lax-Wendroff.

4 General Pseudocode

This pseudocode represents the overall structure shared by all three algorithms. They have the same main function and update order; the only difference lies in the specific formulas used within each function, which follows the rule that we mentioned in the previous section. Therefore, we will present a generalized version that applies to all of them.

Algorithm 1 General pseudocode

Initialize: `sizex` $\leftarrow X/\Delta x$, `sizey` $\leftarrow Y/\Delta y$
Allocate memory: `U`, `FU`, `GU` of size $3 \times$ `sizex` $\times$ `sizey`
Set initial condition: **InitialConditionBound**(U, FU, GU, sizex, sizey)
`step` $\leftarrow 0$, `count_time` $\leftarrow 0$
$\Delta t \leftarrow$ **CNF_Condition**(U, sizex, sizey)
while `count_time` $<$ `time` **do**
    Allocate temporary array `U_new`
    Update_U(U_new, U, FU, GU, Δt, sizex, sizey)
    for all i, j, k in domain **do**
        $U[i][j][k] \leftarrow U\_new[i][j][k]$
    end for
    Update_FluxF(U, FU, sizex, sizey)
    Update_FluxG(U, GU, sizex, sizey)
    `step` $\leftarrow$ `step` $+ 1$
    `count_time` $\leftarrow$ `count_time` $+ \Delta t$
    $\Delta t \leftarrow$ **CNF_Condition**(U, sizex, sizey)
end while
Free arrays: `U`, `FU`, `GU`

Part III
Parallel Design

In this project, we implement three different algorithms using parallel programming. Although each algorithm differs in its specific computation and update logic (which are handled in the sequential

part), they all share the same parallelization strategy and communication structure. To ensure clarity and avoid redundancy, we present only the general idea and parallel code structure that applies to all three algorithms. This unified approach simplifies the explanation while still capturing the essential design of the parallel implementation.[2]

1 MPI

1.1 General Idea

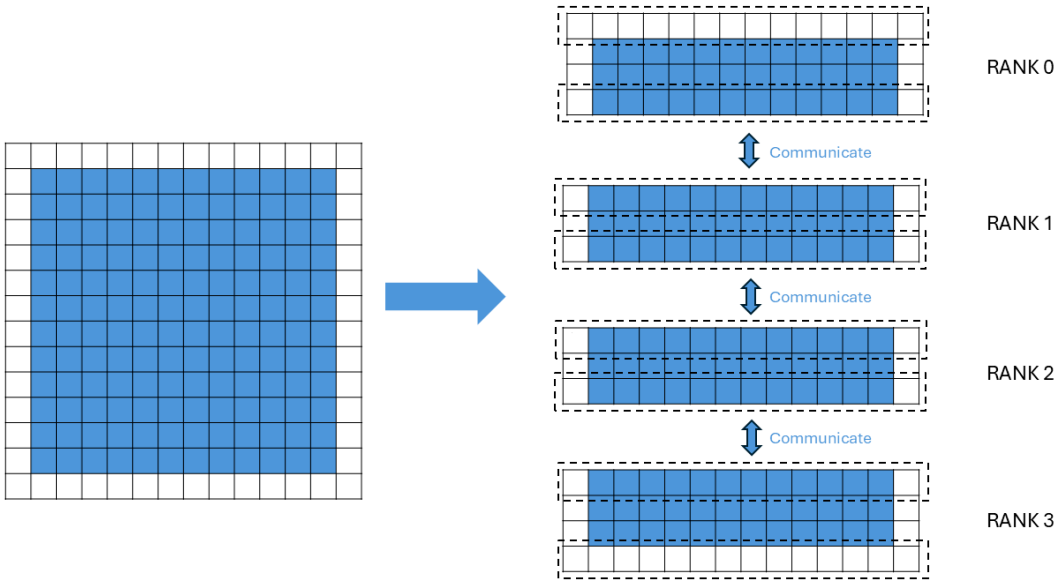


Figure 4 . General idea of MPI programming

The figure4 illustrates the decomposition of a 2D computational grid into horizontal subdomains, each assigned to a separate MPI rank (RANK 0–3). Each rank handles a subset of rows and includes ghost cells to facilitate data exchange with neighboring ranks. Communication occurs at the shared boundaries to ensure consistency across subdomains. This approach supports parallel computation of grid-based simulations like the shallow water equations.

- **Split the grid**.To enable parallel computation using MPI, the simulation domain is first split horizontally into equal subdomains, each assigned to a separate process. This allows each process to independently compute a portion of the grid while minimizing inter-process dependency.

- **Exchange boundary rows**. According to equation of sequential programming, computing the next pressure value at position (i, j) requires values from neighboring points: (i + 1, j), (i − 1, j), (i, j + 1), and (i, j − 1). Since the domain is split horizontally, the top and bottom neighbors — (i + 1, j) and (i−1, j) — may belong to adjacent ranks.Therefore, boundary rows are exchanged between neighboring processes. This step ensures that each subdomain has

access to adjacent data required for computing numerical stencils, especially near the domain boundaries.

- **Bulk computation**: Following data exchange, each process performs the bulk computation on its local subdomain. Using the updated ghost rows, processes compute new values independently, applying the same update rules as in the serial case.
- **Reduction**: Furthermore, our problem also includes a section on reduction, which is used to obtain the minimum delta value from each process. Specifically, we determine the smallest delta among the four processes and use process 0 to broadcast this information to the remaining three processes in each iteration of loop.
- **Gather data**: Finally, after all computations are complete, processes gather the data into a single process . This step reconstructs the global solution from local subdomain results for output, visualization, or further processing.

1.2 General Code

MPI Initialization. The code begins by initializing MPI and querying the total number of processes (`NP`) and the current rank (`Rank`):

```
// Step 1: Initialize MPI
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &NP);
MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
```

Domain Decomposition: A simple decomposition is applied by dividing the domain along the x-axis, particularly from the matrix size sizex $\times$ sizey into sizexc $\times$ sizey submatrices for each process. Then the initial condition is scattered from rank 0 to all processes using `MPI_Scatter` :

```
// Domain decomposition
int sizexc;
double *Uc, *FUc, *GUc;
sizexc = (int)(sizex / NP); // number of rows per process

Uc  = (double*)malloc(sizeof(double) * 3 * sizexc * sizey);
FUc = (double*)malloc(sizeof(double) * 3 * sizexc * sizey);
GUc = (double*)malloc(sizeof(double) * 3 * sizexc * sizey);

// Scatter global arrays to each process
MPI_Scatter(U,  3 * sizexc * sizey, MPI_DOUBLE, Uc,  3 * sizexc
    * sizey, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(FU, 3 * sizexc * sizey, MPI_DOUBLE, FUc, 3 * sizexc
    * sizey, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(GU, 3 * sizexc * sizey, MPI_DOUBLE, GUc, 3 * sizexc
    * sizey, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Then initialize the ghost rows: Ud(down), Up(up) for each process to communicate with the others.

```
double *Up, *Ud;
Up = (double*)malloc(sizeof(double) * 3 * sizey);
Ud = (double*)malloc(sizeof(double) * 3 * sizey);
```

Ghost Row Exchange.Before each update step, processes exchange boundary rows with their neighbors to obtain the necessary halo data, they need to communicate with up and down neighborhood to get enough data. For the border process(process 0 and process NP-1) do not need the neighbors in one side, we initialize the ghost condition through `Update_Border` condition.

```
 // Communicate from down to up
if (RANK == 0){
    Update_Border(Ud, Uc, 0, sizey);
    MPI_Send(Uc+3*sizey*(sizexc-1), sizey*3, MPI_DOUBLE,RANK+1,
        RANK, MPI_COMM_WORLD);
}else if(RANK == NP-1){
    MPI_Recv(Ud, sizey*3, MPI_DOUBLE, RANK-1, RANK-1,
        MPI_COMM_WORLD,&state);
}
else{
    MPI_Send(Uc+3*sizey*(sizexc-1), sizey*3, MPI_DOUBLE,RANK+1,
        RANK, MPI_COMM_WORLD);
    MPI_Recv(Ud, sizey*3, MPI_DOUBLE, RANK-1, RANK-1,
        MPI_COMM_WORLD,&state);
}
```

```
//Communicate from up to down
if (RANK == 0){
    MPI_Recv(Up, sizey*3, MPI_DOUBLE, RANK+1, RANK+1,
        MPI_COMM_WORLD,&state);
}else if(RANK == NP-1){
    Update_Border(Up, Uc, sizexc-1, sizey);
    MPI_Send(Uc, sizey*3, MPI_DOUBLE,RANK-1, RANK,
        MPI_COMM_WORLD);
}else{
    MPI_Send(Uc, sizey*3, MPI_DOUBLE, RANK-1, RANK,
        MPI_COMM_WORLD);
    MPI_Recv(Up, sizey*3, MPI_DOUBLE, RANK+1, RANK+1,
        MPI_COMM_WORLD,&state);
}
```

Parallel computation: With ghost rows added, each process updates its local subdomain using the same computation kernel as in the serial version. The boundary rows rely on ghost values exchanged with neighboring processes.

Reduction: After parallel computation, then computes a local time step `delta_t` for each MPI process and gathers all values to process 0 using `MPI_Gather`. Process 0 determines the global minimum `delta_t`, then broadcasts it to all other processes using `MPI_Send`. Non-root processes receive the global value with `MPI_Recv`, ensuring synchronization of the time step across the domain.

```
// Compute local delta_t in each process
delta_t = CNF_Condition(Uc, sizexc, sizey);
// Gather all delta_t to process 0
MPI_Gather(&delta_t, 1, MPI_DOUBLE, deltas, 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
if (RANK == 0) {
    min_delta = deltas[0];
    for (int p = 1; p < NP; ++p)
        if (deltas[p] < min_delta)
            min_delta = deltas[p];
    delta_t = min_delta;
    // Send minimum delta_t to all other processes from process 0
    for (int p = 1; p < NP; ++p)
        MPI_Send(&delta_t, 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
} else {
    // Receive global delta_t from process 0
    MPI_Recv(&delta_t, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}
```

Gathering the Result. After the final time step, all subdomains are gathered back to rank 0:

```
MPI_Gather(Uc, 3*sizexc*sizey, MPI_DOUBLE, U, 3*sizexc*sizey,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

2 CUDA

2.1 General Idea

The figure 5 illustrates the general concept of CUDA programming applied to grid-based computations. In CUDA, the global data domain is divided into multiple blocks, each consisting of a group of threads. Every block is responsible for handling a subregion of the entire computational grid, and each thread within a block processes a specific cell or element within that subregion.

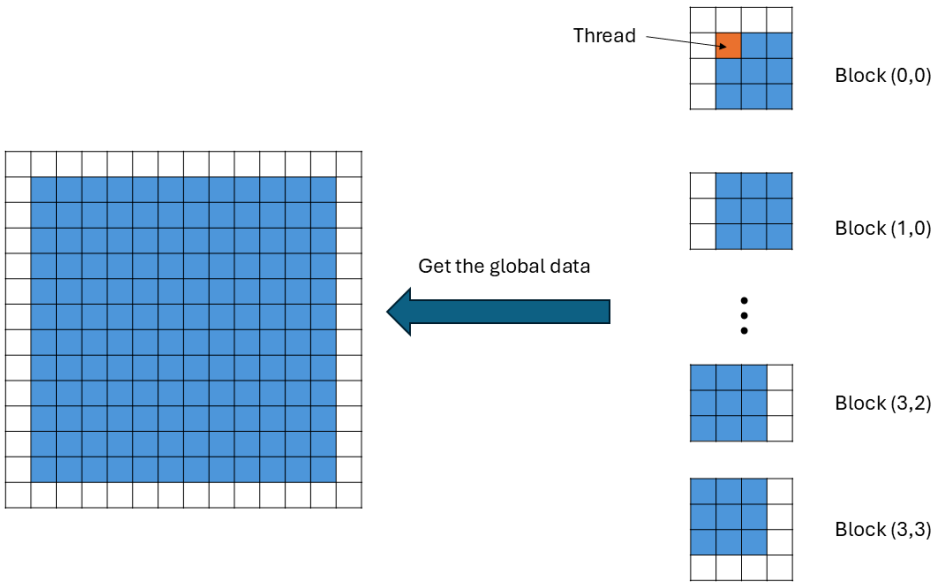


Figure 5 . General idea of CUDA programming

In this example, the entire grid is decomposed into 3×3 blocks, and each block contains a fixed number of threads. Threads execute the same kernel function but operate on different data elements based on their thread and block indices. This hierarchical execution model enables parallel processing of large datasets on the GPU for each unit, significantly accelerating computation compared to MPI programming.

- **Declare variables on CPU, GPU and copy inputs from CPU host and GPU device**: Simulation arrays for water height, velocities, and bathymetry are first declared on the CPU. Corresponding device pointers are declared on the GPU, and memory is allocated for all necessary data. Initial conditions and static data are copied from host to device. This ensures the GPU starts with a consistent physical setup before any computation occurs.
- **Launch kernels on GPU with pre-determined grid, block dimension and execute kernel on GPU**: Grid and block dimensions are defined based on the domain size and hardware capacity. These configurations control how threads are distributed for parallel execution. CUDA kernels update water height and velocities using finite volume methods. Each thread handles part of the domain, reading neighboring cells and applying update rules in parallel.
- **Transfer outputs from GPU to CPU and cleanup memory on both GPU and CPU**: Once results are ready, selected arrays are transferred back to the CPU for visualization, analysis, or saving. All allocated memory is freed on the device and appropriate methods on the host, ensuring no memory leaks occur after simulation ends.

2.2 General Code

Step 1: Declare variables on CPU and GPU: The simulation grid is initialized by computing `sizex` and `sizey` based on the domain size and spatial resolution. CUDA `dim3` types define the

block and grid sizes for kernel launches. Memory is allocated on both CPU (using `malloc`) and GPU (using `cudaMalloc`) for the main state variables `U`, `FU`, `GU`, and the temporary buffer `UGpuNew`.

```
// Define grid and block sizes
int sizex = (int)(X / delta_x);
int sizey = (int)(Y / delta_y);
dim3 dimGrid(GridSize_h, GridSize_w);
dim3 dimBlock(BlockSize_h, BlockSize_w);

// Host allocation
double *Ucpu = (double*)malloc(sizeof(double) *3*sizex*sizey);
// Device allocation
double *Ugpu, *UgpuNew, *FU, *GU;
cudaMalloc((void**)&UgpuNew, sizeof(double) * 3 * sizex * sizey);
cudaMalloc((void**)&Ugpu, sizeof(double) * 3 * sizex * sizey);
cudaMalloc((void**)&FU, sizeof(double) * 3 * sizex * sizey);
cudaMalloc((void**)&GU, sizeof(double) * 3 * sizex * sizey);
```

Step 2: : Copy inputs from CPU host and GPU device: The initial condition is set on the host using `InitialConditionBound`, after which data is transferred to the GPU using `cudaMemcpy`. Once on the device, CUDA kernels `Update_FluxF` and `Update_FluxG` are launched to compute the initial fluxes `FU` and `GU` from the input state `UGpu`.

```
// Initialize and copy data to device
InitialConditionBound(Ucpu, sizex, sizey);
cudaMemcpy(Ugpu, Ucpu, sizeof(double) * 3 * sizex * sizey,
    cudaMemcpyHostToDevice);
// Compute fluxes using CUDA kernels
Update_FluxF<<<dimGrid, dimBlock>>>(Ugpu, FU, sizex, sizey);
Update_FluxG<<<dimGrid, dimBlock>>>(Ugpu, GU, sizex, sizey);
```

Step 3: Launch kernels on GPU with pre-determined grid and block dimension: In this section, each of our algorithms has different functions and performs different tasks. For simplicity, I will only present one method — the Finite Difference Lax-Friedrichs method — as an example, since the others follow a similar structure.

```
void InitialConditionBound(double* U, int sizex, int sizey)
{
// Initialize the border ghost bound.
}
__global__ void Update_FluxF(double* U, double* FU, int sizex,
    int sizey)
{
// Update the value of F based on value of U for each unit.
}
__global__ void Update_FluxG(double* U, double* GU, int sizex,
    int sizey)
{
// Update the value of G based on value of U for each unit.
}
__global__ void CNF_Condition(double* U,double* local_max_u,
    double* local_max_v,double sizex, double sizey)
{
// Find the speed_u and speed_v for each unit as input
to find the delta_t in global
}
__global__ void Update_U(double* U_new, double* U, double* FU ,
    double* GU,double delta_t, int sizex , int sizey)
{
// Update U for each unit based on the previous step.
}
```

The `ReductionMax` kernel computes the maximum value in a given input array using shared memory and a tree-based reduction strategy. Each thread block loads its portion of the data into shared memory. A binary tree reduction is performed iteratively to find the local maximum. Finally, the result is written to the global memory by the thread with `tid = 0`:

- Shared memory `sdata[]` is used to store thread-local values.
- Threads synchronize after loading and after each reduction stage.
- The result for each block is written to `maxOut`.

```
__global__ void ReductionMax(double *maxIn, double *maxOut){
    // synchronize all the data
    extern __shared__ double sdata[];
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tid = threadIdx.x * blockDim.y + threadIdx.y;
    int idx = tx * gridDim.y*blockDim.y + ty;
    sdata[tid] = maxIn[idx];
    __syncthreads();
```

```
    // find the max value based on tree search
    for (int i = 1; i < blockDim.x*blockDim.y; i *= 2){
        if (tid % (2*i) == 0 && tid + i < blockDim.x*blockDim.y){
            sdata[tid] = fmax(sdata[tid] , sdata[tid + i]);
        }
        __syncthreads();
    }
    // save the max values
    if (tid == 0)
        maxOut[blockIdx.x* gridDim.y + blockIdx.y] = sdata[0];
}
```

Step 4: Execute kernel on GPU:

```
while (count_time < time){
        CNF_Condition<<<dimGrid, dimBlock>>>(Ugpu, localMaxu,
            localMaxv, sizex, sizey);
        cudaDeviceSynchronize();
        /*
        Reduction Code
        */
        Update_U<<<dimGrid, dimBlock>>>(UgpuNew, Ugpu, FU, GU,
            delta_t, sizex, sizey);
        std::swap(Ugpu, UgpuNew);
        Update_FluxF<<<dimGrid,dimBlock>>>(Ugpu,FU, sizex,sizey);
        Update_FluxG<<<dimGrid,dimBlock>>>(Ugpu,GU, sizex,sizey);
    }
```

Reduction to find min value: The CUDA code performs a parallel reduction to find global maximum values. It computes local maxima on the GPU, reduces them to block-level, then to global maxima, and copies the results back to the host. A time step is then calculated using the CFL condition to ensure simulation stability.

```
// Allocate device memory for local and block maxima
double *localMaxu, *localMaxv;
double *blockMaxu, *blockMaxv;
double max_u, max_v;

cudaMalloc((void**)&localMaxu, M * sizeof(double));
cudaMalloc((void**)&localMaxv, M * sizeof(double));
cudaMalloc((void**)&globalMaxu, sizeof(double));
cudaMalloc((void**)&globalMaxv, sizeof(double));
cudaMalloc((void**)&blockMaxu, BlockSize_h * BlockSize_w *
    sizeof(double));
cudaMalloc((void**)&blockMaxv, BlockSize_h * BlockSize_w *
    sizeof(double));

```

```
//Reduction Code in loop
while (count_time < time){
    //Reduction Code
    // Compute local maximums using CNF condition
    CNF_Condition<<<dimGrid, dimBlock>>>(Ugpu, localMaxu,
        localMaxv,sizex, sizey);
    cudaDeviceSynchronize();
    // Reduce from local maxima to block maxima
    ReductionMax<<<dimGrid, dimBlock, BlockSize_h * BlockSize_w
        *sizeof(double)>>>(localMaxu, blockMaxu);
    ReductionMax<<<dimGrid, dimBlock, BlockSize_h * BlockSize_w *
        sizeof(double)>>>(localMaxv, blockMaxv);
    // Reduce from block maxima to global maxima
    ReductionMax<<<1, GridSize_h * GridSize_w, GridSize_h *
        GridSize_w * sizeof(double)>>>(blockMaxu, globalMaxu);
    ReductionMax<<<1, GridSize_h * GridSize_w, GridSize_h *
        GridSize_w * sizeof(double)>>>(blockMaxv, globalMaxv);
    // Copy final results to host
    cudaMemcpy(&max_u, globalMaxu, sizeof(double),
        cudaMemcpyDeviceToHost);
    cudaMemcpy(&max_v, globalMaxv, sizeof(double),
        cudaMemcpyDeviceToHost);
    // Time step based on CFL condition
    double delta_t = C * fmin(delta_x / max_u, delta_y / max_v);
}
```

Step 5: Transfer outputs from GPU to CPU

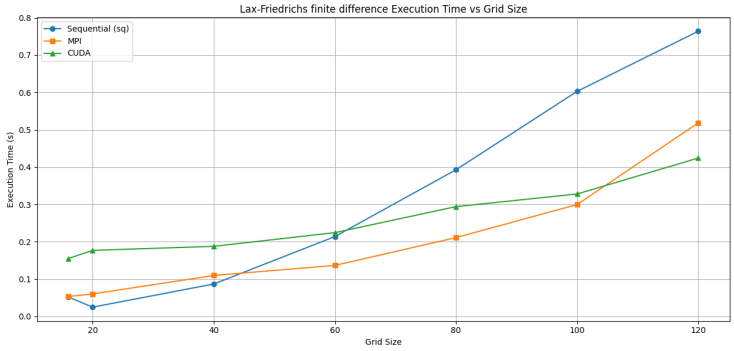
```
cudaMemcpy(Ucpu, Ugpu, sizeof(double) * 3 * sizex * sizey,
    cudaMemcpyDeviceToHost);
```

Step 6: Cleanup memory on both GPU and CPU

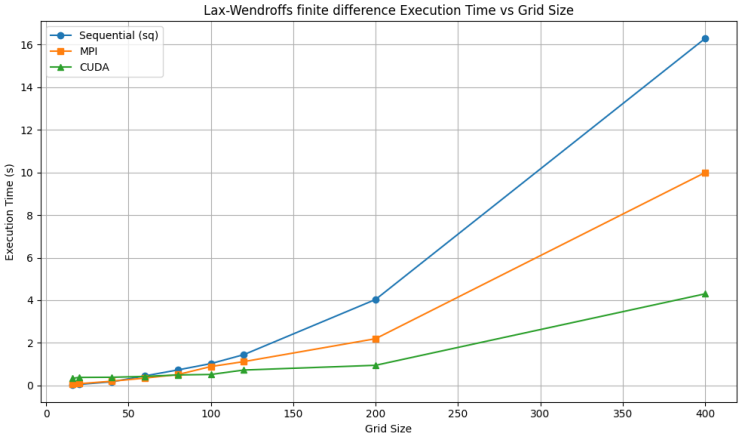
```
cudaFree(Ugpu);
cudaFree(FU);
cudaFree(GU);
cudaFree(UgpuNew);
free(Ucpu);
```

Part IV
Results

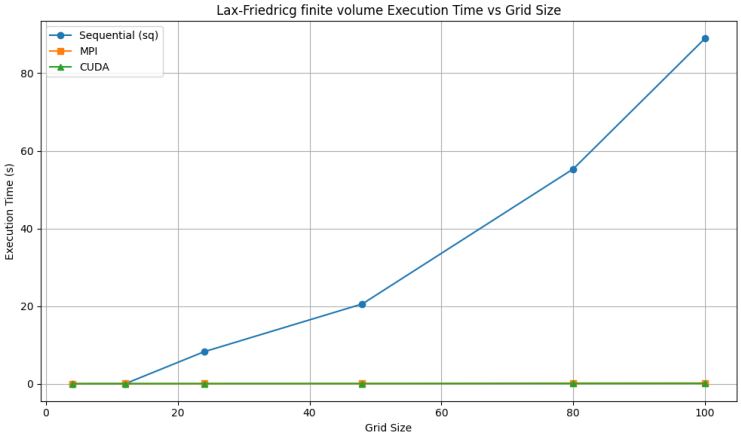
1 Empirical Result



(a) Lax-Friedrichs finite difference method



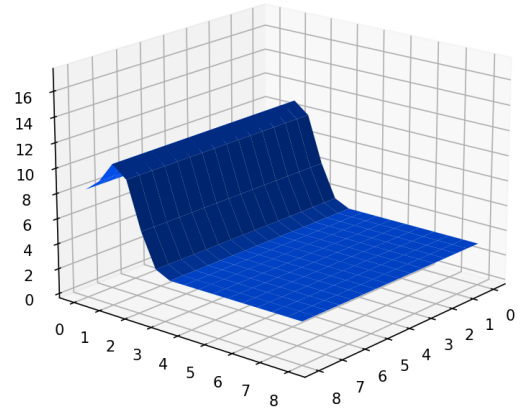
(b) Lax-Wendroff finite difference method



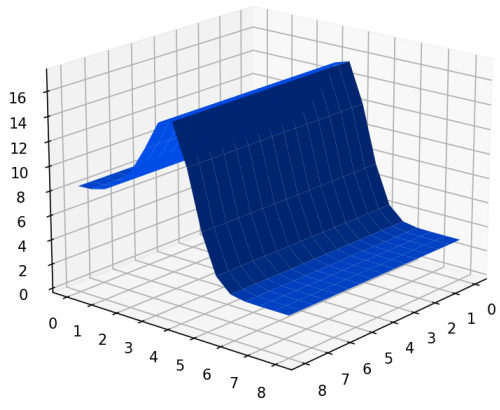
(c) Lax-Friedrich finite volume method

Figure 6 . Empirical result runtime of three algorithms in CUDA and MPI

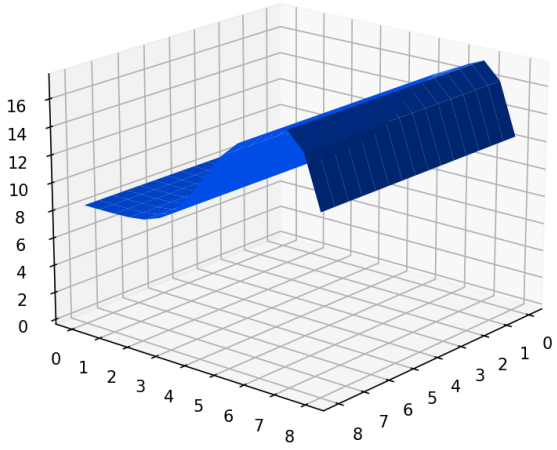
2 Simulation Result



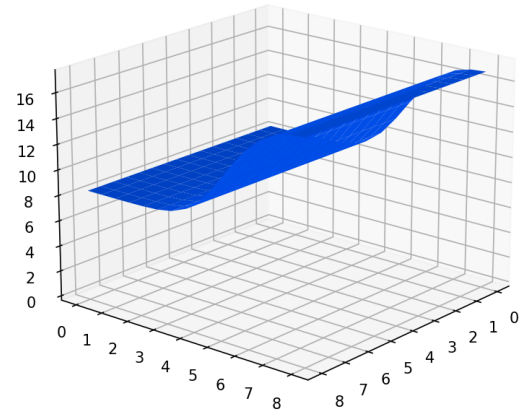
(a) t = 20



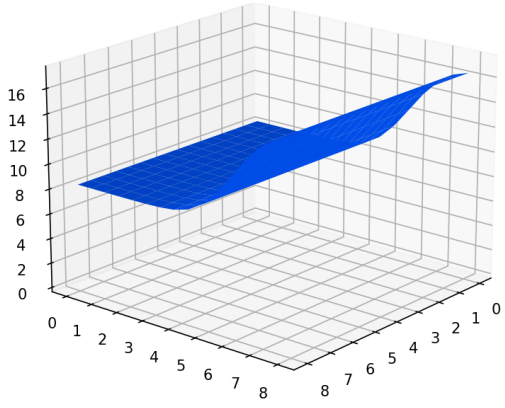
(b) t = 40



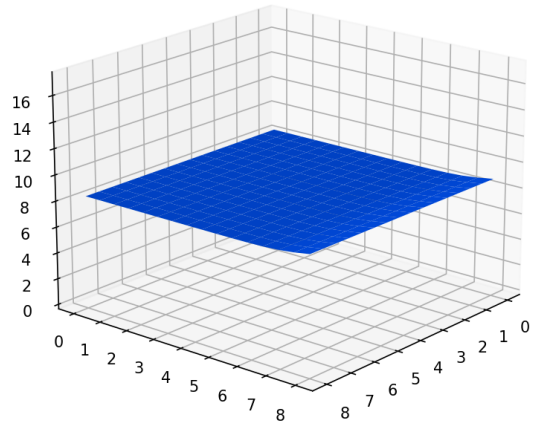
(c) t = 60



(d) t = 80

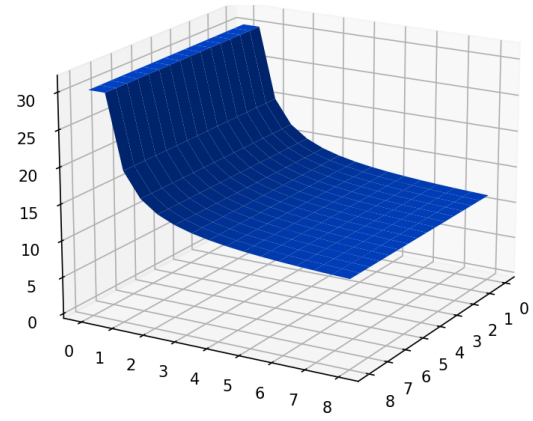


(e) t = 100

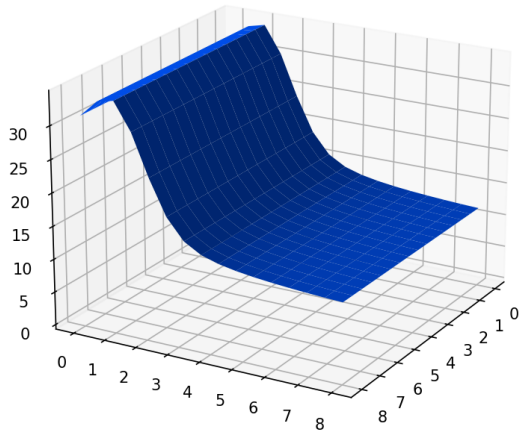


(f) t = 120

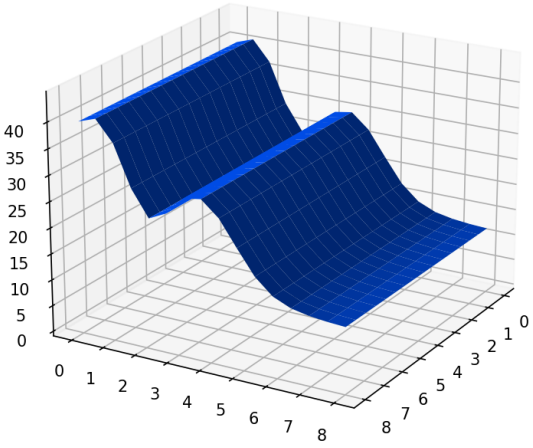
Figure 7 . Lax-Friedrich finite difference method with transitive boundary condition.



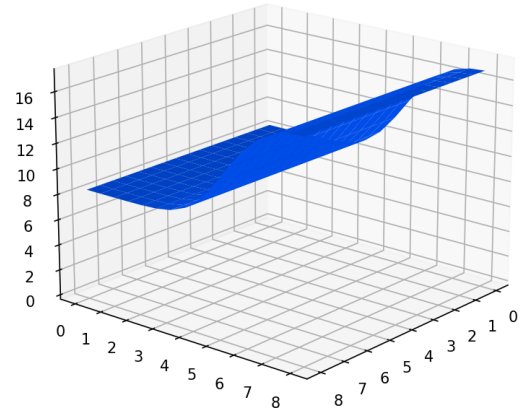
(a) t = 20



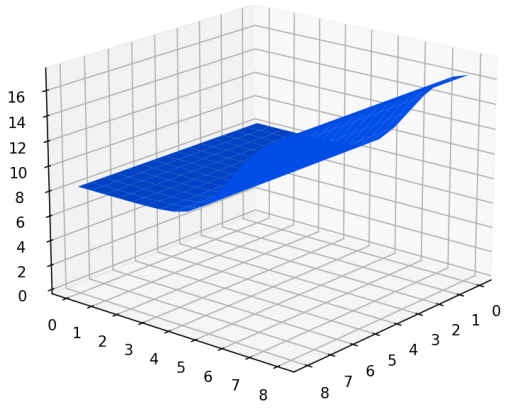
(b) t = 40



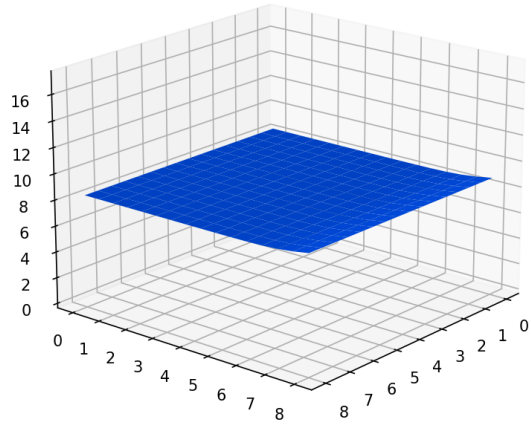
(c) t = 60



(d) t = 80

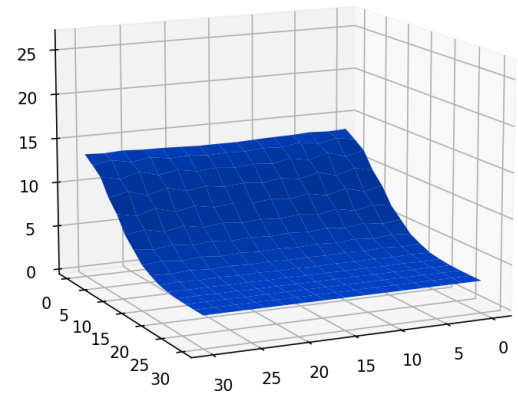


(e) t = 100

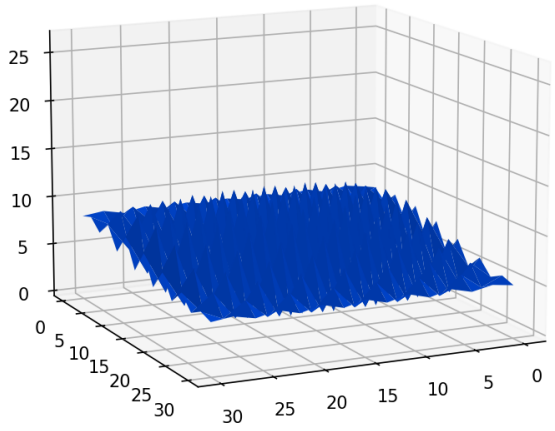


(f) t = 120

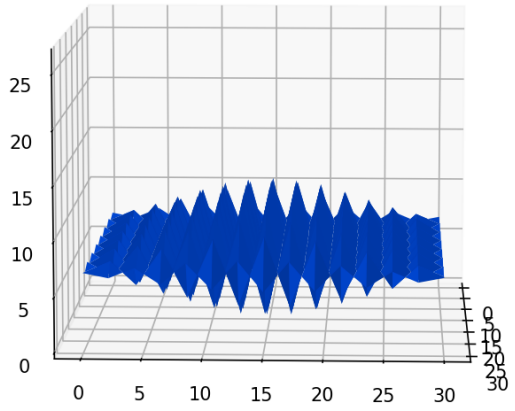
Figure 8 . Lax-Wendroff finite different method with transitive boundary condition.



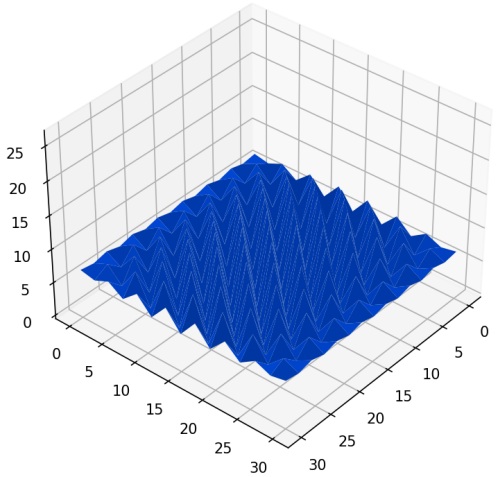
(a) t = 10



(b) t = 20

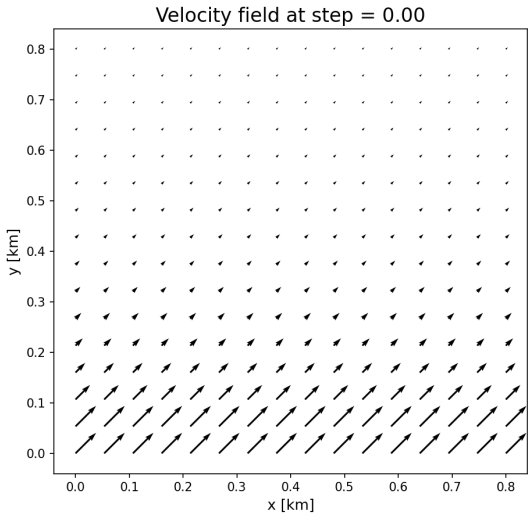


(c) t = 30

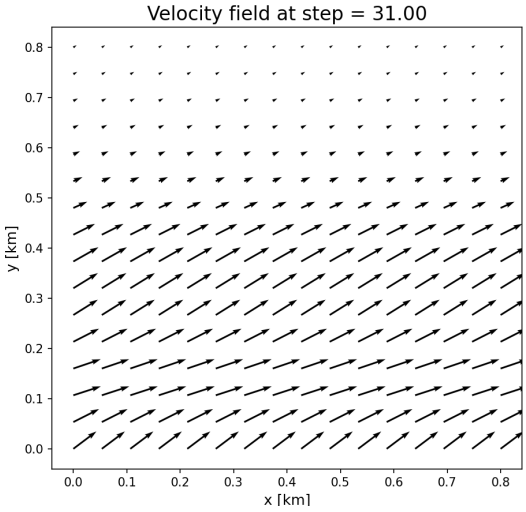


(d) t = 40

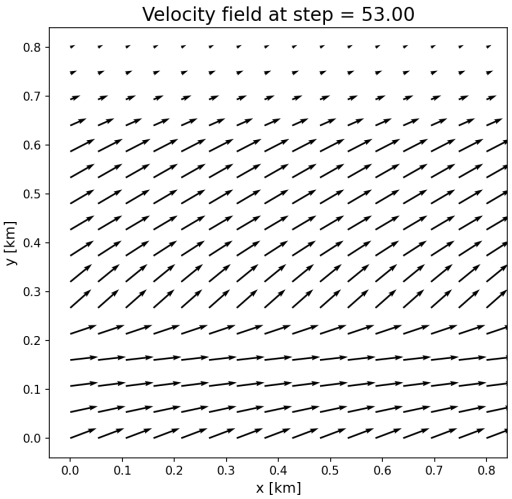
Figure 9 . Lax-Friedrichs finite volume method with reflective boundary condition.



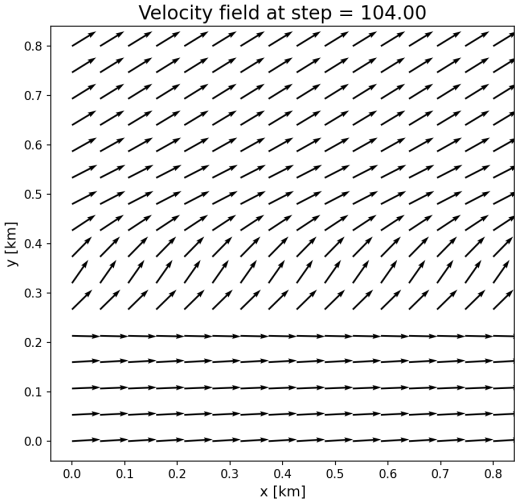
(a)



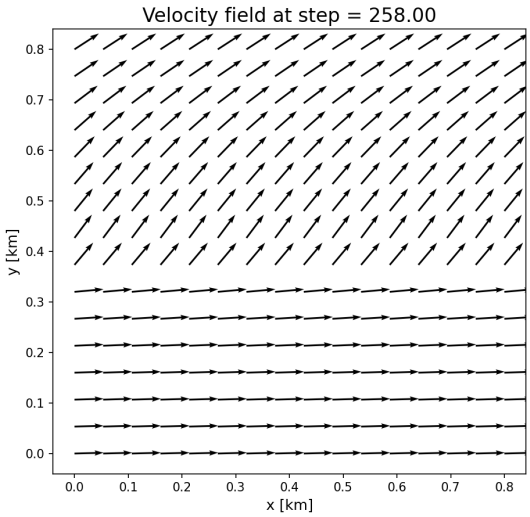
(b)



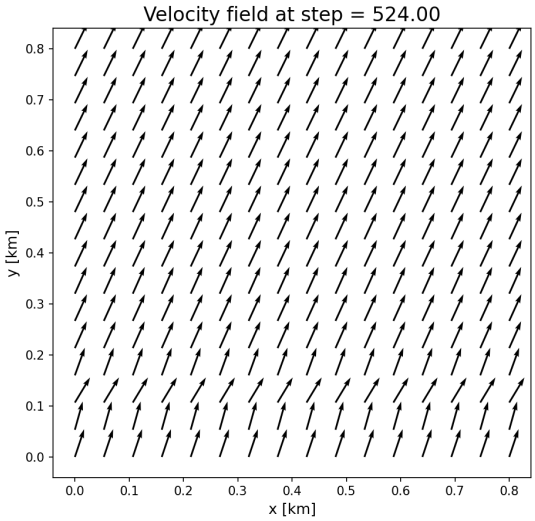
(c)



(d)



(d)



(d)

Figure 10 . Vector field simulation with Lax-Wendroff finite difference method in reflective condition.

Part V
Conclusion

This project successfully applied parallel programming techniques to accelerate the numerical solution of the two-dimensional shallow water equation using finite difference and finite volume methods. By implementing the Lax-Friedrichs, Lax-Wendroff, we were able to compare the performance of different schemes. The use of MPI enabled distributed memory parallelism through domain decomposition and inter-process communication, while CUDA was employed to offload intensive computations to the GPU, enhancing local performance.

Acknowledgment

Overall, our project remains at the basic level expected for project in a course; we do not delve deeply into research on this topic, but rather use it as a means to explore Shallow Water Equation problem and Parallel Programming. However, given each team member's divergent future research interests, we have carefully allocated tasks and the scope of investigation to best suit individual strengths and objectives.

We would like to express our sincere gratitude to Ph.D.Vu Van Thieu, our instructor for this final semester project, for giving us invaluable knowledge through the Computer Vision course. Besides, we also give respect to community research in field of Parallel Distributed and Programming, whose public research material in the conference and article about this field. Finally, this report is heard by all members of our team.

References

[1] S. Brand. Parallel algorithm for numerical solution of the shallow water equation. In *Proceedings of the Czech–Japanese Seminar in Applied Mathematics*, pages 25–36, Prague, September 2006.

[2] Alexander Breuer and Michael Bader. Teaching parallel programming models on a shallow-water code. In *2012 11th International Symposium on Parallel and Distributed Computing*, pages 301–308. IEEE, 2012.

[3] B. M. Ginting, R. P. Mundani, and E. Rank. Parallel simulations of shallow water solvers for modelling overland flows. In *Proceedings of the 13th International Conference on Hydroinformatics (HIC 2018)*, EPiC Series in Engineering, pages 1–6, Palermo, Italy, July 2018.