Chalmers University of Technology

# GPU-accelerated Computational Methods Using Python and CUDA

Wave Propagation

Adyuth Hisham
Max Fredsdal
Isak Ulin
Hampus Johansson

January 30, 2026

**Abstract**

This report presents the results of an investigation into the performance improvements of a Wave Propagation Solver designed using the Finite Difference Time Domain (FDTD) method. While wave propagation problems can be solved on a CPU, increasing computational intensity due to factors such as larger grid sizes or smaller spatial and temporal steps makes the use of GPUs increasingly attractive. Given that the problem can be parallelized across the spatial domain, GPUs can significantly reduce computational time compared to CPUs.

This study compares the performance of a CPU implementation with GPU-accelerated implementations using CuPy, Numba, and PyTorch across different GPUs (A40, A100, and H100). The focus is on evaluating these libraries in terms of computational performance, scalability, and efficiency for solving two-dimensional wave propagation problems.

A general CPU implementation was first developed and then individually optimized for GPUs using each respective package. The objective of the study was to evaluate the impact of GPU type, package library, and floating-point precision on computational performance, using GPUs available on the *Vera* computing cluster.

The final analysis indicates that Numba provided the best performance for large grids, outperforming the CPU implementation by a substantial margin. For medium-sized grids, both PyTorch and CuPy showed the highest speedups, while for smaller grids, the CPU implementation remained competitive, with the GPU libraries offering modest improvements.

# Contents

# 1 Introduction

The wave equation is a second-order linear partial differential equation that describes the propagation of waves through a medium. In classical physics, it is commonly used to model mechanical and electromagnetic wave phenomena. Numerical methods are required to solve the wave equation for most practical applications, where analytical solutions are not feasible.

$$\frac{\partial^2 u}{\partial t^2}(x,y,t) - \alpha^2 \left( \frac{\partial^2 u}{\partial x^2}(x,y,t) + \frac{\partial^2 u}{\partial y^2}(x,y,t) \right) = 0 \tag{1}$$

Equation (1) represents the two-dimensional wave equation for the field $u(x, y, t)$, where $\alpha$ denotes the wave propagation speed.

Among these methods, the finite-difference time-domain (FDTD) method is widely used for modelling and simulation purposes. FDTD is a numerical technique used to model systems governed by time-dependent partial differential equations. It is widely employed in computational science and engineering to simulate the evolution of physical fields over space and time within a bounded domain. In FDTD, continuous spatial and temporal derivatives are approximated using finite-difference expressions, transforming the governing equations into a set of explicit update equations.

The computational domain is discretized into a grid of spatial points and advanced in discrete time steps. At each time step, the field variables at each grid point are updated using values from neighbouring points and previous time levels. This explicit time-marching scheme enables the direct simulation of wave propagation from specified initial conditions, while requiring careful consideration of numerical stability and accuracy.

# 2 Problem Description

The main goal of this project is to investigate efficient computational methods for solving the wave equation using FDTD, with a focus on reducing computation time. By discretizing the wave equation, the problem becomes solvable on a conventional CPU. However, as the spatial and temporal resolution of the grid increases, the computational cost grows rapidly, making parallelized methods increasingly relevant for larger simulations. The FDTD update scheme is well suited for parallel computation since the value at each grid point only depends on values from neighbouring points at previous time steps. This means that each spatial point can be updated independently, making the problem embarrassingly parallel. Such problems are well suited for execution on graphics processing units (GPUs), where a large number of threads can perform computations simultaneously.

## 2.1 Python based GPU libraries

To enable GPU acceleration, this project uses NVIDIA CUDA through several Python-based libraries that provide different levels of abstraction for GPU

programming. Three different GPU computing libraries are considered in this work: Numba, CuPy, and PyTorch.

Numba allows Python functions to be compiled just-in-time into custom CUDA kernels, giving the user a high degree of control over GPU execution at the cost of increased implementation effort. CuPy provides a NumPy-like interface for GPU arrays, which allows existing array-based code to be transferred to the GPU with minimal changes, but with less direct control over kernel execution. PyTorch, although primarily designed for machine learning applications, provides a CUDA-enabled tensor framework that supports element-wise operations and tensor slicing. In this project, PyTorch is used as a general-purpose GPU array library in order to evaluate the performance overhead associated with using a high-level framework for stencil-based numerical simulations.

### 2.2 Benchmarking Criteria

Since these libraries differ in their approach to GPU execution and level of abstraction, they also introduce different overhead costs. For this reason, a wave propagation solver based on the FDTD method is implemented using each library and benchmarked under identical conditions.

The benchmarking focuses on computational performance and scalability, specifically measuring computation time as the number of grid points increases. To assess how performance varies with hardware, the solvers are tested on three NVIDIA GPUs with different architectures: the A40, A100, and H100. This allows evaluation of how each library performs on GPUs with varying compute capabilities and memory bandwidth, and helps to determine in which scenarios a more ready-to-use library is sufficient versus when a more optimized, low-level approach is necessary.

### 2.3 Aim

The aims of this project are to:

1. Implement a 2D wave propagation solver using the finite-difference time-domain (FDTD) method and optimize it for CPU performance.
2. Develop GPU-accelerated versions of the solver using CuPy, Numba, and PyTorch, targeting NVIDIA GPUs.
3. Benchmark and compare the CPU and GPU solvers on NVIDIA A40, A100, and H100 GPUs, using both single-precision (float32) and double-precision (float64), in order to identify the most efficient approach for simulations of varying scale.

## 3 Method

This section presents the methods used to solve the two-dimensional wave equation and to evaluate computational performance. A CPU-based implementation using NumPy serves as the performance baseline, while GPU-accelerated implementations are developed using CuPy and PyTorch, both of which leverage

tensor-based parallel computation. In addition, a Numba-based implementation employing just-in-time compilation is included to enable further performance optimization. The section also describes the procedures used to verify the correctness and stability of the implementations, as well as the methods for collecting and analyzing performance data.

## 3.1 Discretization using FDTD

The propagation of waves in a two-dimensional domain can be described by the classical wave equation for a scalar field $u(x, y, t)$:

$$\frac{\partial^2 u}{\partial t^2}(x, y, t) - \alpha^2 \left( \frac{\partial^2 u}{\partial x^2}(x, y, t) + \frac{\partial^2 u}{\partial y^2}(x, y, t) \right) = 0, \tag{2}$$

where $\alpha$ is the wave propagation speed. This can also be written compactly using the Laplace operator $\Delta$ as

$$u_{tt} - \alpha^2 \Delta u = 0. \tag{3}$$

Equation (2) governs the evolution of the wave field over space and time and forms the basis for the numerical solver developed in this project.

To solve the wave equation numerically, the domain is discretized into a grid with spatial step size $h$ in both the $x$ and $y$ directions, and time step $t$ in the temporal dimension. For the temporal discretization, three successive time levels are maintained: $u^{(0)}$, $u^{(1)}$, and $u^{(2)}$, corresponding to the future, current, and previous states of the field, respectively.

The second derivative in time at a grid node $(i, j)$ is approximated using a central finite difference scheme:

$$u^{(1)}_{tt_{i,j}} = \frac{u^{(0)}_{i,j} - 2u^{(1)}_{i,j} + u^{(2)}_{i,j}}{t^2}. \tag{4}$$

Spatial derivatives are approximated using a standard five-point stencil for the Laplace operator:

$$\nabla^2 u_{i,j} \approx \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \tag{5}$$

leading to

$$\Delta u^{(1)}_{i,j} = \frac{u^{(1)}_{i-1,j} + u^{(1)}_{i+1,j} + u^{(1)}_{i,j-1} + u^{(1)}_{i,j+1} - 4u^{(1)}_{i,j}}{h^2}. \tag{6}$$

Combining the temporal and spatial discretizations, the explicit update formula for advancing the solution by one time step becomes:

$$u^{(0)}_{i,j} = \left( \frac{t\alpha}{h} \right)^2 \left( u^{(1)}_{i-1,j} + u^{(1)}_{i+1,j} + u^{(1)}_{i,j-1} + u^{(1)}_{i,j+1} - 4u^{(1)}_{i,j} \right) + 2u^{(1)}_{i,j} - u^{(2)}_{i,j}. \tag{7}$$

This explicit time-marching scheme forms the core of the FDTD solver, allowing the wave field to be propagated forward in time from given initial conditions while maintaining second-order accuracy in both space and time.

Finally, an external driving term is applied to initiate wave propagation within the computational domain. The temporal dependence of the driver is chosen as a sinusoidal function, while its spatial distribution is defined by a two-dimensional Gaussian. Applying the source over a finite spatial region rather than at a single grid point reduces numerical artifacts and improves stability, particularly when varying the spatial resolution of the grid. The Gaussian profile provides a smooth, localized excitation that remains effectively scale-invariant under grid refinement.

The driving term is defined as

$$f(x, y, t) = A \sin(2\pi f t) \, \exp\left( -\frac{(x - x_0)^2 + (y - y_0)^2}{2\sigma^2} \right), \tag{8}$$

where $A$ is the source amplitude, $f$ is the driving frequency, $(x_0, y_0)$ denotes the center of the excitation, and $\sigma$ controls its spatial extent.

In discrete form, the driver evaluated at grid point $(i, j)$ and time step $n$ is given by

$$f_{i,j}^{(n)} = A \sin(2\pi f n \, t) \, \exp\left( -\frac{(ih - x_0)^2 + (jh - y_0)^2}{2\sigma^2} \right). \tag{9}$$

The driving term is added to the field update equation during a prescribed time interval in order to inject energy into the system and initiate wave propagation.

## 3.2 Baseline CPU Implementation

A CPU implementation of the 2D wave propagation solver was first developed to serve both as a performance baseline and as the foundation for the GPU implementations. The solver is implemented using NumPy for efficient computations on the large arrays that represent the discretized spatial domain.

The main update loop uses vectorized array operations to compute the discrete Laplace operator and advance the wave field in time. The update at each grid point is computed simultaneously across the grid using array slicing. In addition to the main wave update, a spatial driver can be applied, and boundary values are updated explicitly to maintain correct edge behavior. After each timestep, the three time-level arrays representing the previous, current, and next states of the field $(u_{\text{prev}}, u_{\text{curr}}, u_{\text{next}})$ are rotated to prepare for the next iteration.

Validation utilities, including energy evaluation and frame storage for visualization, were incorporated into the solver to support correctness and stability checks during development. These utilities were disabled during benchmarking to avoid influencing performance measurements. The implementation was further optimized using profiling tools to identify bottlenecks, ensuring fair comparisons with GPU-based implementations. The fully vectorized NumPy code for the CPU solver can be found in Appendix B.

## 3.3 GPU implementations

Refactoring the NumPy implementation to use CuPy was straightforward due to the similarity between their APIs. Most NumPy functions were directly replaced with their CuPy equivalents, with the addition of explicit data transfers

between the CPU and GPU. A similar approach was used for the PyTorch implementation. The main addition was that the three used arrays needed to be transfered to the GPU before the computing phase, which was done with the corresponding transfer function for the different libraries.

The Numba implementation required more extensive modifications. In contrast to the other approaches, Numba necessitates that each computation intended to run on the GPU be defined as a separate kernel function, decorated with `@cuda.jit`. This decorator triggers just-in-time (JIT) compilation, meaning that the Python function is compiled to CUDA machine code at runtime when first invoked. Unlike CuPy or PyTorch, which operate at a higher level of abstraction and automatically manage kernel launches internally, Numba requires explicit control over the execution configuration. Each kernel must be launched with a specified number of thread blocks and threads per block, defining how computations are distributed across the GPU. Within the kernels, thread indices are obtained using `cuda.grid(2)`, allowing each thread to operate on a specific grid point in the spatial domain. The wave update, spatial driver, and boundary handling were therefore implemented as separate CUDA kernels, each executed sequentially within a timestep.

Since CUDA kernel launches are asynchronous by default, the CPU does not wait for a launched kernel to complete before continuing execution. This behavior affects performance measurements, as timing a kernel launch alone would not capture the actual computation time on the GPU. Therefore, `synchronize` was used, from `numba.cuda`, after each timestep to ensure that all previously launched kernels had finished executing before recording timing results. This guarantees that the measured runtime reflects the full cost of the GPU computations.

### 3.3.1 Multi-GPU Numba Implementation

To extend the computational capability of the GPU-accelerated wave propagation solver beyond a single device, a multi-GPU implementation was developed. The primary goal was to enable simulation of larger spatial domains and longer time spans that exceed the memory or performance limits of a single GPU, while building directly on the Numba based single GPU kernels described earlier.

An MPI-based multi-GPU extension was implemented using the `mpi4py` package to coordinate processes and perform inter GPU communication. For the 2D wave propagation solver, the global grid of size $xn \times yn$ is decomposed across $n$ processes (each mapped to one GPU). The decomposition is performed in two dimensions, resulting in a process grid of dimensions $px \times py$ and the factors $px$ and $py$ are chosen to balance subdomain sizes as evenly as possible. Each process handles a local subdomain of approximate dimensions $\lceil xn/px \rceil \times \lceil yn/py \rceil$ (with slight adjustments at domain edges to ensure exact coverage).

Since the FDTD stencil at each interior point depends on its four nearest neighbors, boundary consistency across subdomains requires the use of ghost (halo) cells thereby one layer of ghost cells is allocated on all four sides of each local grid.

Halo exchanges are performed after each relevant field update to maintain nu-

merical consistency at subdomain interfaces. The exchange routine uses non-blocking-aware `MPI_Sendrecv` operations directly on views of Numba `DeviceNDArray` objects. Horizontal exchanges transfer interior columns to the neighboring left or right ghost columns, while vertical exchanges transfer interior rows to the neighboring up or down ghost rows.

**Current Status**

A working prototype has been implemented following the structure outlined above. The multi-GPU version successfully handles larger grid sizes than the single-GPU Numba implementation, demonstrating correct domain decomposition and halo exchange. However, the overall runtime remains significantly higher than expected. The multi-GPU implementation requires further debugging to be compared with the single GPU implementations.

## 3.4 Profiling and verification

To identify performance bottlenecks, we employed `line_profiler` and subsequently optimized the sections of code with the highest impact. To measure execution times for comparison, we used the `perf_counter` function from Python's `time` module. The function was called immediately before and after each timestep, and the elapsed time ($\Delta t$) was computed as the difference between these two points. These values were accumulated to obtain the total execution time. From this, we calculated both the total and average time per simulation, enabling a quantitative comparison of the performance across different implementations. The initial code used to set up the source, data transfer to GPU and related components was excluded from the timing measurements.

The correctness and stability of the solver were verified using two complementary methods: visualization of the wave propagation and monitoring of total energy. For visualization, the Python library Matplotlib was used to generate heat maps of the wave over time, enabling identification of unexpected behaviour and early detection of implementation errors. For each simulation, 200 time frames were saved. In GPU-based implementations, the data were transferred back to the CPU before visualization, and the stored frames were then passed to Matplotlib's `FuncAnimation` function to generate GIF animations for inspection.

In addition, the numerical stability of the simulations was evaluated by monitoring the total energy over time. Due to the use of a single sinusoidal source and the fact that the simulations were terminated before the wave reached the domain boundaries, the total energy was expected to remain approximately constant after an initial transient phase. The energy was recorded every five time steps as the sum of the total kinetic and potential energy, providing a discrete approximation of the continuous energy associated with the wave equation.

## 3.5 Data collection

To ensure a fair comparison between implementations, all benchmarking cases were run with identical parameter settings. The wave solver executes 200 time steps on the same wave function and spatial domain. The primary variables considered are the number of spatial points and the floating-point precision. It

is worth noting that fixing the number of time steps while increasing the spatial resolution reduces the time step size, due to the stability criteria of the wave equation. Practically, this means that simulating the same total time at higher resolution requires more time steps than at lower resolution. Since the focus of this project is on runtime comparison, keeping the number of time steps fixed was deemed appropriate.

The number of spatial points ranges from $100^2$ to $25,000^2$ with 12 samples for the A40 and A100 GPUs, and up to $35,000^2$ with 14 samples for the H100. Both float32 and float64 precisions were used. All benchmarking was performed on the VERA cluster. For a fair CPU baseline, the CPU implementation was also benchmarked on the Icelake and Zen4 processors available on the cluster. CPU benchmarks were limited to 8 samples per grid size to reduce total benchmarking time, as CPUs are substantially slower than GPUs. Each implementation (CPU, CuPy, PyTorch, and Numba) was benchmarked 10 times per test case to minimize the impact of outliers, such as cold-start variability or faulty hardware.

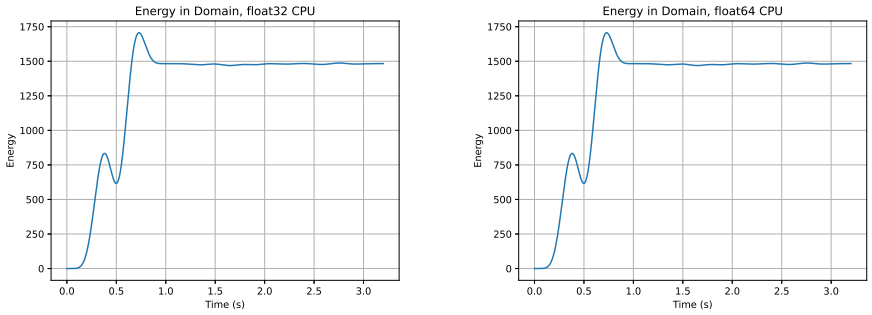
The benchmarking process was automated using scripts that executed each wave propagation implementation for the parameter combinations described above and recorded the runtime required to compute 200 time steps. All benchmarks were run as `sbatch` jobs on the VERA cluster, ensuring a consistent execution environment across CPU and GPU measurements. Different runtime environments were used depending on the implementation, reflecting the software requirements of each framework.

In particular, execution of the Numba implementation on the VERA cluster required the use of the `Numba-CUDA` module, which provides CUDA support for the python JIT compiler on the commercial GPUs. This requirement was specific to the cluster environment and was not necessary during development on a lower-end desktop GPU. The PyTorch and CuPy implementations were executed in apptainer container using a PyTorch container image which had all dependecies necessary for both PyTorch and CuPy to function.

# 4 Results

The data was collected in two sessions, first the stability analysis, then the benchmarking. All results displayed come from the VERA cluster.

## 4.1 Numerical stability of solver




(a) CPU float32 (b) CPU float64

Figure 1: The total energy on the grid over simulated for 2000 time steps with $1000^2$ points on the grid. (a) CPU implementation using float32 precision. (b) CPU implementation using float64 precision.

## 4.2 CPU Benchmarks

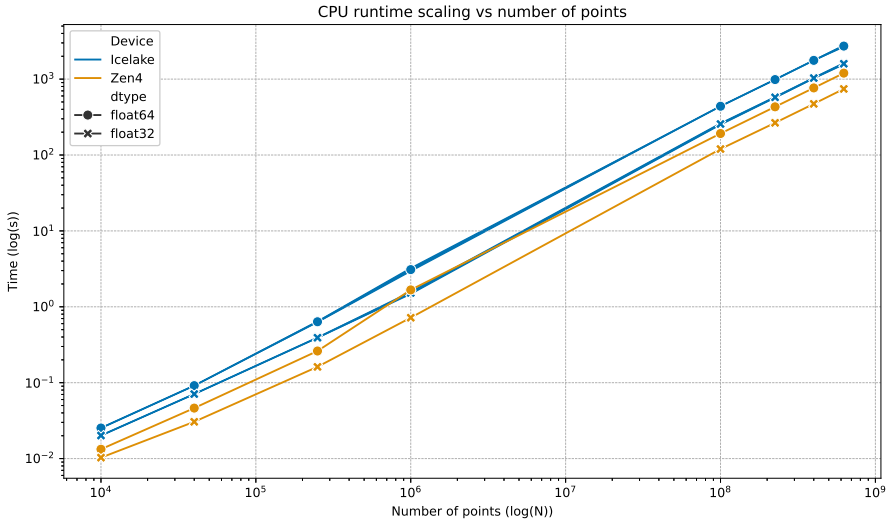



Figure 2: CPU benchmarking results (Icelake and Zen4) across all implementations, averaged over 10 runs per sample. Runtime for 200 time steps is shown versus grid size, both on logarithmic scales. float32 and float64 results are distinguished by marker shape.

## 4.3 GPU benchmarks

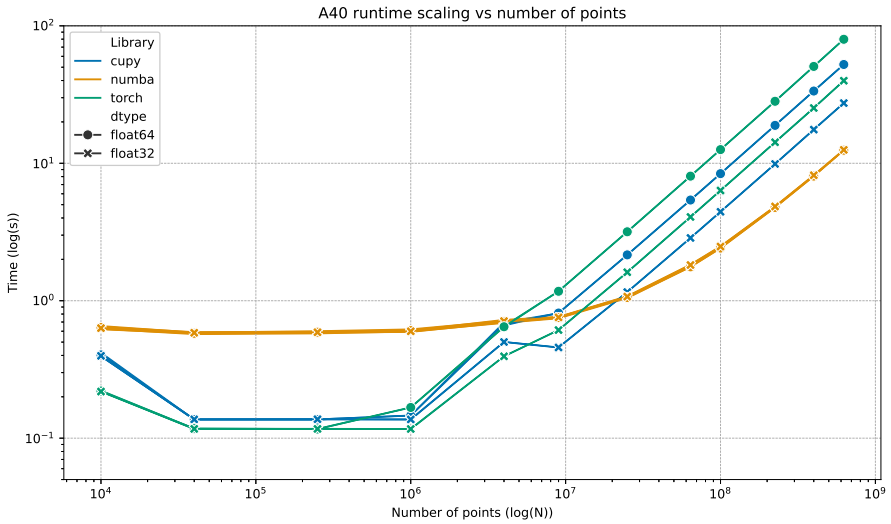



Figure 3: GPU benchmarking results on the A40 across all implementations, averaged over 10 runs per sample. Runtime for 200 time steps is shown versus grid size, both on logarithmic scales. Float32 and float64 results are distinguished by marker shape. Libraries are distinguished by colour.

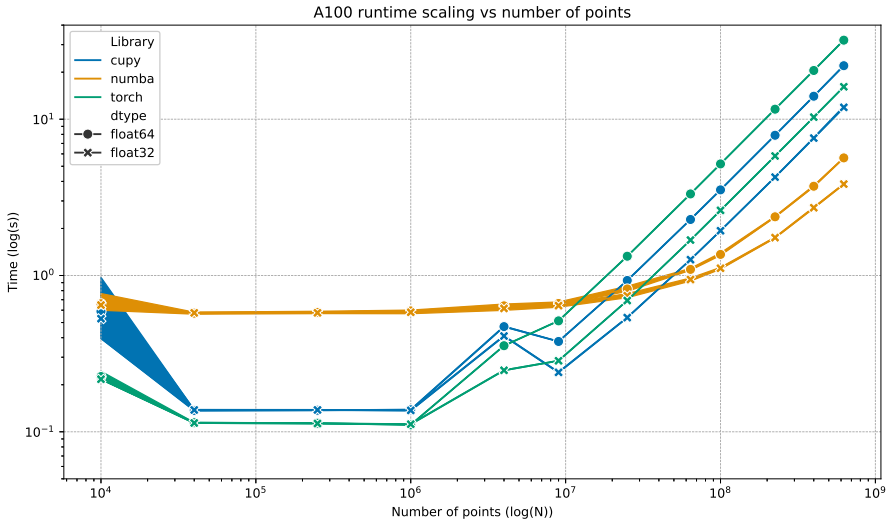



Figure 4: GPU benchmarking results on the A100 across all implementations, averaged over 10 runs per sample. Runtime for 200 time steps is shown versus grid size, both on logarithmic scales. Float32 and float64 results are distinguished by marker shape. Libraries are distinguished by colour.

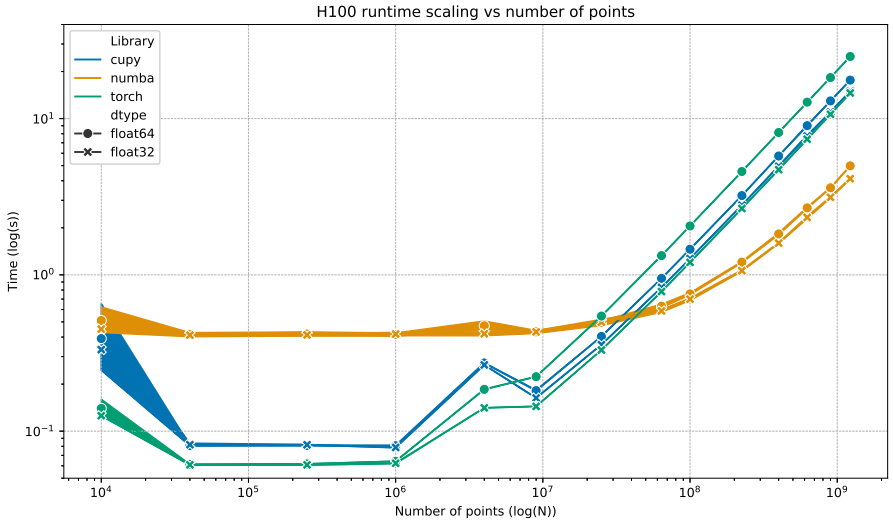



Figure 5: GPU benchmarking results on the H100 across all implementations, averaged over 10 runs per sample. Runtime for 200 time steps is shown versus grid size, both on logarithmic scales. Float32 and float64 results are distinguished by marker shape. Libraries are distinguished by colour.

## 4.4 Summary

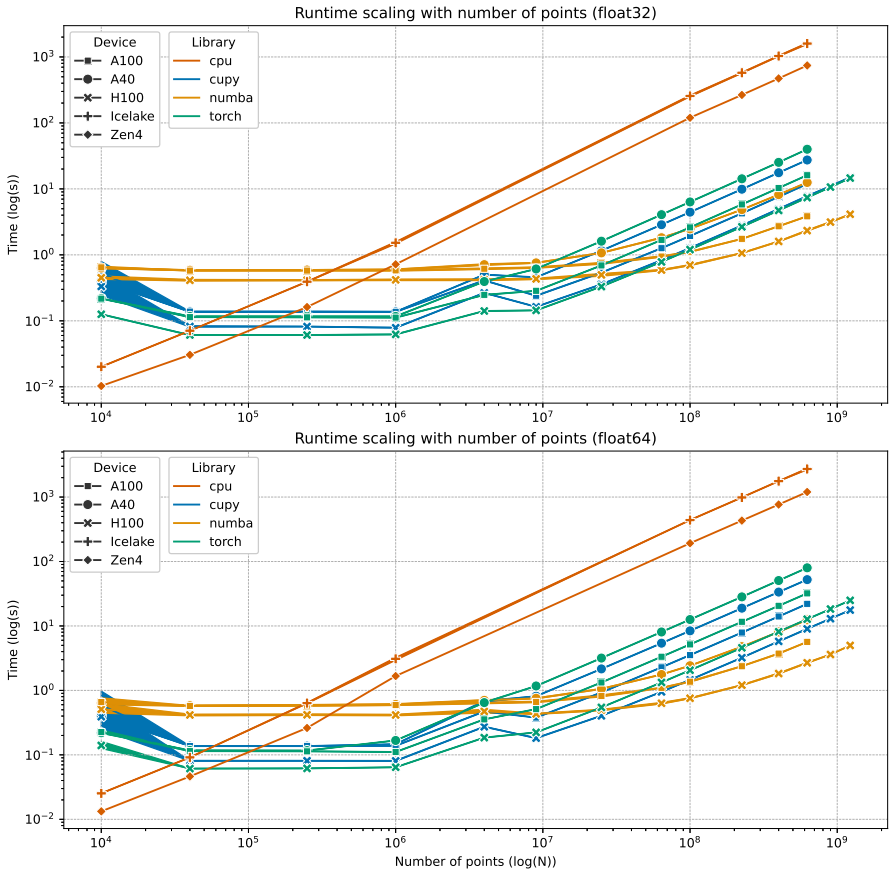



Figure 6: Side-by-side comparison of benchmarking results for CPU (Icelake, Zen4) and GPU (A40, A100, H100) devices across all implementations, averaged over 10 runs per sample. The top panel shows float32 results and the bottom panel shows float64 results. Runtime for 200 time steps (seconds) is plotted on the y-axis and grid size on the x-axis, both on logarithmic scales.

| | $1 \times 10^4$ | $4 \times 10^4$ | $2 \times 10^5$ | $1 \times 10^6$ | $1 \times 10^8$ | $2 \times 10^8$ | $4 \times 10^8$ | $6 \times 10^8$ |
|---|---|---|---|---|---|---|---|---|
| **A40** | | | | | | | | |
| cupy | 0.05 | 0.52 | 2.85 | 11.14 | 57.7 | 58.22 | 58.72 | 58.18 |
| numba | 0.03 | 0.12 | 0.66 | 2.54 | 103.35 | 118.67 | 126.17 | 127.37 |
| torch | 0.09 | 0.61 | 3.35 | 13.06 | 40.41 | 40.47 | 40.9 | 39.98 |
| **A100** | | | | | | | | |
| cupy | 0.04 | 0.51 | 2.84 | 11.12 | 132.39 | 135.15 | 136.38 | 133.92 |
| numba | 0.03 | 0.12 | 0.68 | 2.62 | 230.37 | 329.48 | 380.34 | 414.82 |
| torch | 0.09 | 0.62 | 3.47 | 13.63 | 98.15 | 98.98 | 100.16 | 98.87 |
| **H100** | | | | | | | | |
| cupy | 0.03 | 0.37 | 1.98 | 9.14 | 94.4 | 95.17 | 95.39 | 96.0 |
| numba | 0.02 | 0.07 | 0.39 | 1.72 | 171.51 | 249.65 | 295.65 | 318.59 |
| torch | 0.08 | 0.5 | 2.66 | 11.54 | 99.7 | 99.86 | 100.21 | 100.62 |

Table 1: Relative runtime between GPUs and their correpsonding CPU baseline (Icelake for A40, A100 and Zen4 for H100). Featured is data from benchmarking using float32.

# 5 Discussion

This section analyses the benchmarking results and discusses their implications for implementation design choices across CPU and GPU platforms. Additionally some unexpected results and technical considerations are highlighted.

## 5.1 CPU Performance Analysis

As can be seen in Figure 1, regardless of the data type employed, the simulations exhibited very similar stability and energy behavior. This indicates that using `float32` is not impacting the numerical stability of the simulation and depending on the floating point accuracy needs float32 is preferable in this case, as it consistently outperforms `float64` in computational performance, see Figure 2, while requiring only half the memory. The superior speed of `float32` is likely due to its lower memory bandwidth requirements, smaller cache footprint, and simpler, more efficient arithmetic operations which is all achieved without compromising stability in our simulations.

The baseline CPU benchmarking shows that both Icelake and Zen4 exhibit similar performance trends, with Zen4 achieving slightly lower runtimes due to its higher computational capability. As shown in Figure 2, the runtime increases rapidly with grid size, which is expected given the algorithm's time complexity of $\mathbf{O}(T \cdot N^2)$, where $T$ is the number of time steps and $N$ is the number of points in each spatial dimension. While this complexity remains unchanged when executing the algorithm on a GPU, parallel execution allows multiple grid points to be computed simultaneously rather than sequentially, reducing the overall runtime.

## 5.2 Benchmark Comparison

Comparing the relative runtime between the CPU and GPU benchmarks, see Table 1, the CPU is the most efficient option for smaller grid sizes. This is

likely due to the fixed overheads associated with GPU execution, such as kernel launch latency and data transfer costs, which dominate the runtime for small workloads. As the problem size increases, these overheads become negligible relative to the available parallelism on the GPU, allowing the CuPy, PyTorch, and Numba implementations to outperform the CPU.

Looking close at the GPU benchamrkings, see Figures 3, 4 and 5, two common patterns are very clear. Firstly CuPy and PyTorch have very similar runtime patterns, secondly Numba has a different runtime pattern. For all GPU models Numba is the slowest until around $N^2 = 4 \cdot 10^6$ on the A40 and $N^2 = [1, 5] \cdot 10^7$ on the A100 and H100, where all libraries' runtimes intersect.

For large $N$ Numba seems to perform the best on all devices, however it seems that the margin between Numba and the other two libraries is different depending on device. In Figure 6 all devices are compared with each other, and since the GPUs have different specifications it is not a fair comparison. What is interesting about this result is that the H100, which has the most powerful hardware, does outperforme all other devices regardless of library used except for the A40 and A100 using numba which are comparable to the H100 performance using PyTorch or CuPy.

The impact of floating point precision was not very big, see Figures 3, 4 and 5. The conclusion is that there is a minor performane improvement when using float32 compared to using float64 for all implementations at potential cost of accuracy. However as seen in the stability analysis, the impact on numerical stability is negligible. Therefore float32 is seen as the superior data type for a small but consistent improvement on runtime. The bigger gains is in memory management which allows for larger scale simulations on the same device compared to float64.

## 5.3 Observations

As mentioned in the previous section, both CuPy and PyTorch have similar runtime patterns, see Figure 6. Most notably there is consistently, across 10 runs, a local maximum in execution time at approximately three million points. One possible explanation for this behavior is a hardware-related effect. Specifically, the GPUs used in this study have L1 and L2 caches, and the observed performance variation may correspond to a problem size where the working data exceeds the cache capacity. This can lead to cache misses, forcing slower global memory accesses and resulting in a temporary increase in execution time. Further study could clarify this behaviour.

For small $N$, particularly on the A100 and H100 (Figures 4 and 5), the runtime exhibits noticeable variation between runs, even when averaged over 10 repetitions. One plausible explanation is GPU underutilization: these high-end GPUs have thousands of cores, and for small problem sizes there may not be enough parallel work to fully occupy the hardware. As a result, fixed overheads such as kernel launch latency, memory allocation, and data transfer dominate the runtime, causing greater relative fluctuations. In contrast, larger problem sizes provide sufficient work to fully utilize the GPU, reducing variability and allowing the expected scaling behavior to emerge.

### 5.4 Instrumentation Overhead and Synchronization

Performance was measured using time.perf_counter(). While this function provides high-resolution timing, it introduces a small but non-zero measurement overhead. As a result, the reported execution times include both the algorithm runtime and the timing instrumentation itself, which may slightly affect absolute timing values. To obtain a more accurate estimate of the actual runtime, it would be necessary to quantify the overhead introduced by the perf_counter() function itself and subtract this from the measured time.

Another potential problem with the way we are measuring time is that the GPU implementations executes GPU operations asynchronously, meaning that kernel launches may return before execution completes. As a result, the measured times may underestimate actual GPU computation time unless explicit synchronization is enforced. Additionally, calls to .get() introduce implicit synchronization and data transfer overhead, which may shift some GPU execution time outside the measured region.

Consequently, the reported timings should be interpreted as approximate and primarily suitable for relative comparisons rather than precise absolute performance measurements.

### 5.5 Energy Consumption

An improvement to this project would be to record energy consumption. While execution time is reported as a measure of performance, faster computation does not necessarily correspond to lower energy usage. In particular, GPU computations can achieve higher throughput but often at the cost of increased power draw, which may offset energy savings gained from reduced runtime.

Additionally, numerical precision can impact energy use: `float32` operations generally require less memory bandwidth and simpler arithmetic than `float64`, which could lead to lower energy consumption. Hardware-specific factors, such as cache behavior, memory hierarchy, and dynamic frequency scaling, may also influence power usage in complex ways. For example, the execution-time peaks observed around three million points (Section 4.3) might correspond to temporary increases in energy consumption due to cache misses and slower memory accesses.

A more comprehensive assessment would involve direct measurement of power draw during execution, using tools such as NVIDIA's `nvidia-smi`, Intel RAPL, or external power meters. Such measurements would allow evaluation of the trade-offs between runtime performance and energy efficiency and could inform more energy-conscious choices in algorithm and hardware selection.

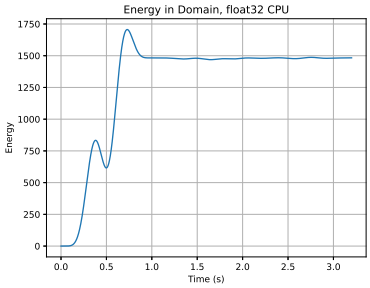
## 6 Conclusion

A two-dimensional wave propagation solver was successfully implemented using the finite-difference time-domain (FDTD) method. The solver was shown to be numerically stable and produced physically reasonable wave propagation behaviour across all tested configurations. This confirms the suitability of the chosen discretization and implementation for both CPU and GPU execution.

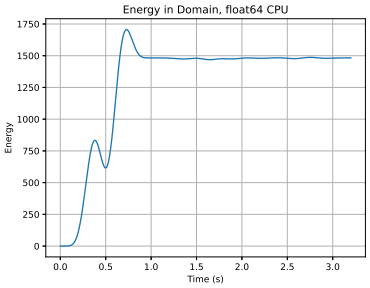
Benchmarking results demonstrate that performance is influenced by GPU architecture, software library, and floating-point precision, although to differing extents. The most significant factor is the choice of GPU, with the H100 consistently outperforming the A40 and A100 due to its superior computational and memory capabilities. The choice of GPU programming library also has a substantial impact: Numba achieves the best performance for large-scale simulations, while CuPy and PyTorch provide comparable and more consistent performance across problem sizes. Floating point precision has the smallest effect on runtime, with `float32` offering a modest but consistent performance advantage over `float64` while maintaining numerical stability.

Overall, the results indicate that CPUs are preferable for small problem sizes due to lower overheads, whereas GPUs offer substantial performance benefits for large-scale simulations. Among GPU-based approaches, `float32` implementations on modern GPUs—particularly using Numba for large workloads—provide the most efficient balance between performance, memory usage, and numerical stability.

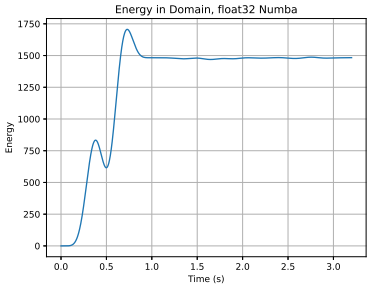
# A Additional Figures



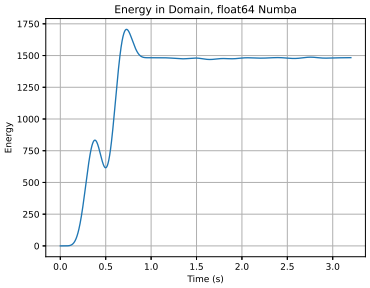

(a) CPU float32



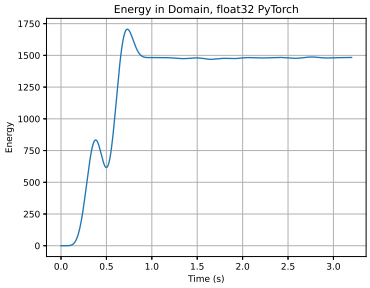

(b) CPU float64



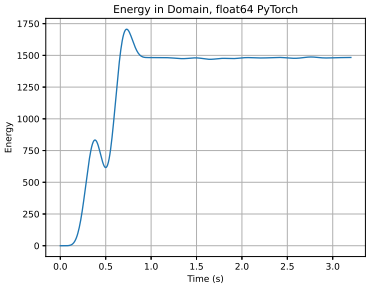

(c) Numba float32



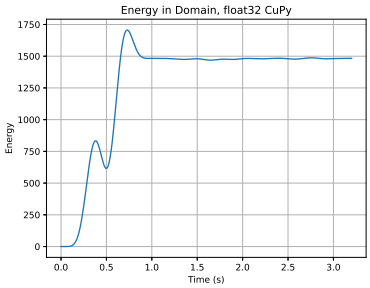

(d) Numba float64



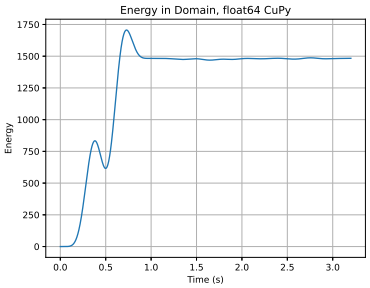

(e) PyTorch float32




(f) PyTorch float64




(g) CuPy float32




(h) CuPy float64

Figure 7: Energy over time. Comparison across implementations and data types to verify similar simulations.

# B Python Source Code

## 2D Wave Propagation Solver (CPU version)

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time

from common import *

def wave_solver_2D_timestep(n):
    global u_prev, u_curr, u_next
    lap = (
        Cy*(u_curr[2:,1:-1] - 2*u_curr[1:-1,1:-1] + u_curr
            [:-2,1:-1]) +
        Cx*(u_curr[1:-1,2:] - 2*u_curr[1:-1,1:-1] + u_curr
            [1:-1,:-2])
    )

    u_next[1:-1,1:-1] = 2*u_curr[1:-1,1:-1] - u_prev
        [1:-1,1:-1] + lap

    # Apply spatial driver
    if n*dt <= T:
        u_next += spatial_gauss * driver(n, dt, T, freq)

    # Boundaries
    u_next[0,:]  = u_curr[0,:]
    u_next[-1,:] = u_curr[-1,:]
    u_next[:,0]  = u_curr[:,0]
    u_next[:,-1] = u_curr[:,-1]

    # Rotate
    u_prev, u_curr, u_next = u_curr, u_next, u_prev


# Main solver
def wave_solver_2D():

    # ------- MAIN LOOP ---------
    total = 0
    for n in range(1, tn):
        start = time.perf_counter()
        wave_solver_2D_timestep(n)
        end = time.perf_counter()
        total += end - start

        # Energy
        if energy_analysis:
            E_domain = calc_total_energy(u_curr, u_prev, c,
                dx, dy, dt)
            energy_domain.append(E_domain)
```

```
        if plotting and n % save_every == 0:
            frames.append(u_prev.copy())

        # if n % 100 == 0:
            # print(f"Step {n}/{tn}")


    print(f"{(total):.6f}")
    # print(f"Total: {(total):.6f} seconds, Average: {(total
        /(tn-1)):.6f} seconds")
    visualize(frames)

# Run
wave_solver_2D()
```

## 2D Wave Propagation Solver (PyTorch GPU version)

```
import torch
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time

from common import *

if dtype == np.float32:
    dtype = torch.float32
else:
    dtype = torch.float64

# Check device
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
# print(f"Running on device: {device}")

# Driver function
def driver_gpu(n, dt, T, freq=1.0, device="cpu"):
    t = n * dt
    if t < 0 or t > T:
        return torch.tensor(0.0, device=device)
    # Hann window to smoothly turn on/off
    env = 0.5 * (1.0 - torch.cos(torch.tensor(2 * math.pi *
        t / T, device=device)))
    return env * torch.sin(torch.tensor(2 * math.pi * freq *
        t, device=device))


spatial_gaussian_gpu = torch.tensor(spatial_gauss,device="
    cuda",dtype=dtype)
u = torch.stack([
    torch.tensor(u_prev,dtype=dtype),
    torch.tensor(u_curr,dtype=dtype),
    torch.tensor(u_next,dtype=dtype)
```

```
]).to("cuda",dtype=dtype)

def wave_solver_2D_timestep(n):
    global u
    lap = (
        Cy*(u[1, 2:,1:-1] - 2*u[1, 1:-1,1:-1] + u[1,
            :-2,1:-1]) +
        Cx*(u[1, 1:-1,2:] - 2*u[1, 1:-1,1:-1] + u[1,
            1:-1,:-2])
    )

    u[2, 1:-1,1:-1] = 2*u[1, 1:-1,1:-1] - u[0, 1:-1,1:-1] +
        lap

    # Apply spatial driver
    if n*dt <= T:
        #u 2, += spatial_gauss * driver(n, dt, T, freq)
        u[2, 1:-1,1:-1] += spatial_gaussian_gpu[1:-1,1:-1] *
             driver_gpu(n, dt, T, freq)

    # Boundaries
    u[2, 0,:]  = u[1, 0,:]
    u[2, -1,:] = u[1, -1,:]
    u[2, :,0]  = u[1, :,0]
    u[2, :,-1] = u[1, :,-1]

    u[0], u[1], u[2] = u[1], u[2], u[0]

# Main solver
def wave_solver_2D():


    # torch.tensor(np.array([u_prev,u_curr,u_next]), device
        ="cuda")

    # ------- MAIN LOOP ---------
    total = 0
    for n in range(1, tn):
        start = time.perf_counter()
        wave_solver_2D_timestep(n)
        torch.cuda.synchronize(device="cuda")
        end = time.perf_counter()
        total += end - start


        # Energy
        if energy_analysis and n % 5 == 0:
            E_domain = calc_total_energy(u[1].cpu().numpy(),
                 u[0].cpu().numpy(), c, dx, dy, dt)
            energy_domain.append(E_domain)

        if plotting and n % save_every == 0:
            frames.append(u[0].cpu().numpy().copy())
```

```
        # if n % 100 == 0:
        #     print(f"Step {n}/{tn}",end="\r")

    # print(f"Total: {(total):.6f} seconds, Average: {(total
        /(tn-1)):.6f} seconds")
    print(f"{(total):.6f}")
    visualize(frames)

# Run
wave_solver_2D()
```

## 2D Wave Propagation Solver (CuPy version)

```
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time

from common import *

u_prev = cp.array(u_prev,dtype=dtype)
u_curr = cp.array(u_curr,dtype=dtype)
u_next = cp.array(u_next,dtype=dtype)

spatial_gaussian_gpu = cp.array(spatial_gauss,dtype=dtype)

# Driver function
def driver_gpu(n, dt, T, freq=1.0):
    t = n * dt
    if t < 0 or t > T:
        return 0.0
    # Hann window to smoothly turn on/off
    env = 0.5 * (1 - cp.cos(2*cp.pi*t/T))
    return env * cp.sin(2*cp.pi*freq*t)

def wave_solver_2D_timestep(n):
    global u_prev, u_curr, u_next
    lap = (
        Cy*(u_curr[2:,1:-1] - 2*u_curr[1:-1,1:-1] + u_curr
            [:-2,1:-1]) +
        Cx*(u_curr[1:-1,2:] - 2*u_curr[1:-1,1:-1] + u_curr
            [1:-1,:-2])
    )

    u_next[1:-1,1:-1] = 2*u_curr[1:-1,1:-1] - u_prev
        [1:-1,1:-1] + lap

    # Apply spatial driver
    if n*dt <= T:
        u_next += spatial_gaussian_gpu * driver_gpu(n, dt, T
            , freq)

    # Boundaries
```

```
    u_next[0,:]  = u_curr[0,:]
    u_next[-1,:] = u_curr[-1,:]
    u_next[:,0]  = u_curr[:,0]
    u_next[:,-1] = u_curr[:,-1]

    # Rotate
    u_prev, u_curr, u_next = u_curr, u_next, u_prev


# Main solver
def wave_solver_2D():

    # ------- MAIN LOOP ---------
    total = 0
    for n in range(1, tn):
        start = time.perf_counter()
        wave_solver_2D_timestep(n)
        end = time.perf_counter()
        total += end - start

        # Energy
        if energy_analysis:
            E_domain = calc_total_energy(u_curr.get(),
                u_prev.get(), c, dx, dy, dt)
            energy_domain.append(E_domain)

        if plotting and n % save_every == 0:
            frames.append(u_prev.get())

        # if n % 100 == 0:
        #     print(f"Step {n}/{tn}",end="\r")


    print(f"{(total):.6f}")
    visualize(frames)

# Run
wave_solver_2D()
```

## 2D Wave Propagation Solver (Numba version)

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time
from numba import cuda

from common import *

u = [
    cuda.to_device(u_prev),
    cuda.to_device(u_curr),
    cuda.to_device(u_next),
]
```

```
spatial_gaussian_gpu_ = cuda.to_device(spatial_gauss)

@cuda.jit
def calculate(u_previous, u_current, u_next, Cx, Cy):
    x,y = cuda.grid(2)
    lap = ( Cy * (u_current[y+2, x+1] - 2*u_current[y+1, x
        +1] + u_current[y, x+1]) +
             Cx * (u_current[y+1, x+2] - 2*u_current[y+1, x
                 +1] + u_current[y+1, x]) )
    u_next[y+1, x+1] = ( 2*u_current[y+1, x+1] - u_previous[
        y+1, x+1] + lap )

@cuda.jit
def apply_spatial_driver(u_next, spatial_gaussian_gpu,
    driver_scalar):
    x, y = cuda.grid(2)
    u_next[y, x] += spatial_gaussian_gpu[y, x] *
        driver_scalar

@cuda.jit
def apply_boundaries(u_current, u_next):
    x, y = cuda.grid(2)
    u_next[0,x]  = u_current[0,x]
    u_next[-1,x] = u_current[-1,x]
    u_next[y,0]  = u_current[y,0]
    u_next[y,-1] = u_current[y,-1]

def wave_solver_2D_timestep(n):
    global u, driver_scalar

    threads_per_block = (16,16)
    blocks =  ((xn + 15) // 16, (yn + 15) // 16)

    calculate[blocks, threads_per_block](u[0], u[1], u[2],
        Cx, Cy)

    if n*dt <= T:
        driver_scalar = driver(n, dt, T, freq)
        apply_spatial_driver[blocks, threads_per_block](u
            [2], spatial_gaussian_gpu_, driver_scalar)

    apply_boundaries[blocks, threads_per_block](u[1], u[2])

    # Rotate
    u[0], u[1], u[2] = u[1], u[2], u[0]

# Main solver
def wave_solver_2D():

    # ------- MAIN LOOP ---------
    total = 0
    for n in range(1, tn):
        start = time.perf_counter()
```

```
            wave_solver_2D_timestep(n)
            cuda.synchronize()
            end = time.perf_counter()
            total += end - start

            # Energy
            if energy_analysis:
                E_domain = calc_total_energy(u[1].copy_to_host()
                    , u[0].copy_to_host(), c, dx, dy, dt)
                energy_domain.append(E_domain)

            if plotting and n % save_every == 0:
                frames.append(u[0].copy_to_host().copy())

            # if n % 100 == 0:
            #     print(f"Step {n}/{tn}")

        print(f"{(total):.6f}")
        # print(f"Total: {(total):.6f} seconds, Average: {(total
            /(tn-1)):.6f} seconds")
        visualize(frames)

    # Run
    wave_solver_2D()
```