King Fahad University of Petroleum & Minerals

Department of Computer Engineering

COE-506 - GPU Programing & Architecture

Term 241

Accelerating a 2D Heat Diffusion Simulation Using OpenACC, CUDA Python, and CUDA C/C++

Yazeed Altaweel	201812960
Malek Aljemaili	201431340
Yousef Alzahrani	201678960

Abstract

This project investigates the acceleration of a 2D heat diffusion simulation using OpenACC, CUDA Python (Numba), and CUDA C/C++. The objective is to transform a serial CPU-based solver into GPU-accelerated implementations to achieve significant performance improvements. We analyze the problem of computing steady-state temperature distribution in a 2D domain, an inherently data-parallel task ideal for GPU acceleration.

Our implementations include a baseline CPU version in both Python and C, an OpenACC-enhanced C implementation, a CUDA Python variant using Numba, and a high-performance CUDA C/C++ version. Performance results demonstrate substantial speedups with all GPU approaches, highlighting trade-offs in ease of use, performance, and implementation complexity. CUDA C/C++ achieves the highest performance, while OpenACC and CUDA Python provide rapid development and maintainability.

---

1. Introduction

Problem Statement and Significance

Heat diffusion simulations are critical in engineering, climate modeling, and materials science. Solving the associated PDEs for heat transfer can be computationally demanding at high resolutions. Leveraging GPU hardware can drastically reduce runtime for such data-parallel problems. This project focuses on the 2D heat diffusion equation and evaluates three GPU programming methods to enhance computational efficiency.

Objectives

- Establish a baseline performance using CPU implementations.
- Accelerate the computation with OpenACC, CUDA Python, and CUDA C/C++.
- Compare performance, ease of programming, and code maintainability.
- Identify and mitigate bottlenecks to optimize runtime.

---

2. Background and Related Work

Previous studies have shown that GPU acceleration significantly benefits PDE solvers. While advanced libraries like PETSc and OpenFOAM provide GPU support, they often require complex CUDA programming. OpenACC offers a more straightforward path, while tools like Numba make GPU programming accessible for Python developers. Unlike general benchmarks, our project provides a direct comparison of these methods for a specific application, offering practical insights for developers.

---

3. Methodology

Problem Description

We simulate the steady-state 2D heat equation over a 500m x 500m square domain discretized into a 400x400 grid. The boundary conditions are defined to vary linearly from 0°C at one edge to 100°C at the opposite edge, providing a realistic thermal gradient. The finite difference method is employed to discretize the domain, iterating until convergence to a steady state is achieved. This involves solving the equation numerically at each grid point by averaging the influence of its neighbors repeatedly.

GPU Suitability

The problem is inherently data-parallel due to the independence of each grid point update, which relies solely on neighboring values from the previous iteration. This localized computation ensures minimal data dependencies across the grid, making it an ideal candidate for GPU acceleration. Utilizing thousands of GPU threads allows simultaneous updates of grid points, significantly speeding up the iterative process compared to a sequential CPU approach.

---

4. Implementation Details

OpenACC Implementation

Using the baseline C code, we added OpenACC pragmas (#pragma acc) to parallelize the computational loops and manage memory transfers. Minimal code restructuring was required. Techniques like loop collapsing improved occupancy and performance.

Appendix Reference: Code for OpenACC stencil computation is provided in Appendix C.

CUDA Python (Numba) Implementation

We ported the baseline solver to Python and used Numba's @cuda.jit decorator to define GPU kernels. Memory management was handled explicitly with cuda.to_device. Despite manual block/grid dimension definitions, the Python implementation remained concise.

Appendix Reference: Code for CUDA Python kernel with memory management is provided in Appendix D.

CUDA C/C++ Implementation

In this approach, we designed and implemented custom kernel functions to manage computations at the thread and block levels with precise control. By organizing threads into 16x16 blocks, we optimized the mapping of grid points to GPU threads. Shared memory was used to store temporary values within each block, significantly reducing expensive global memory accesses and improving performance.

The kernel also incorporated conditional checks to avoid boundary overflows, ensuring robust computations even at domain edges. Additionally, we employed asynchronous kernel launches and used cudaMemcpyAsync to overlap data transfers with computation, further reducing idle time. Through iterative profiling and adjustments, we refined memory coalescing and ensured high utilization of the GPU's computational resources.

Appendix Reference: Code for CUDA C/C++ kernel with shared memory optimizations is provided in Appendix E.

---

5. Comparative Analysis

Performance Tests

Length	Time (s)	Nodes	Python CPU (s)	C CPU (s)	CUDA C (s)	OpenACC (s)	CUDA Python (s)
500	40	400	3960.0	2.76	0.20	0.40	1.64
1000	80	800	32569.56*	22.7	0.91	1.175	3.61
2000	160	1600	297000.0*	207.0	5.80	6.18	10.7

*Python CPU times for lengths 1000 and 2000 were estimated based on scaling from the C CPU implementation, as executing these tests directly was computationally prohibitive.

Speed Up

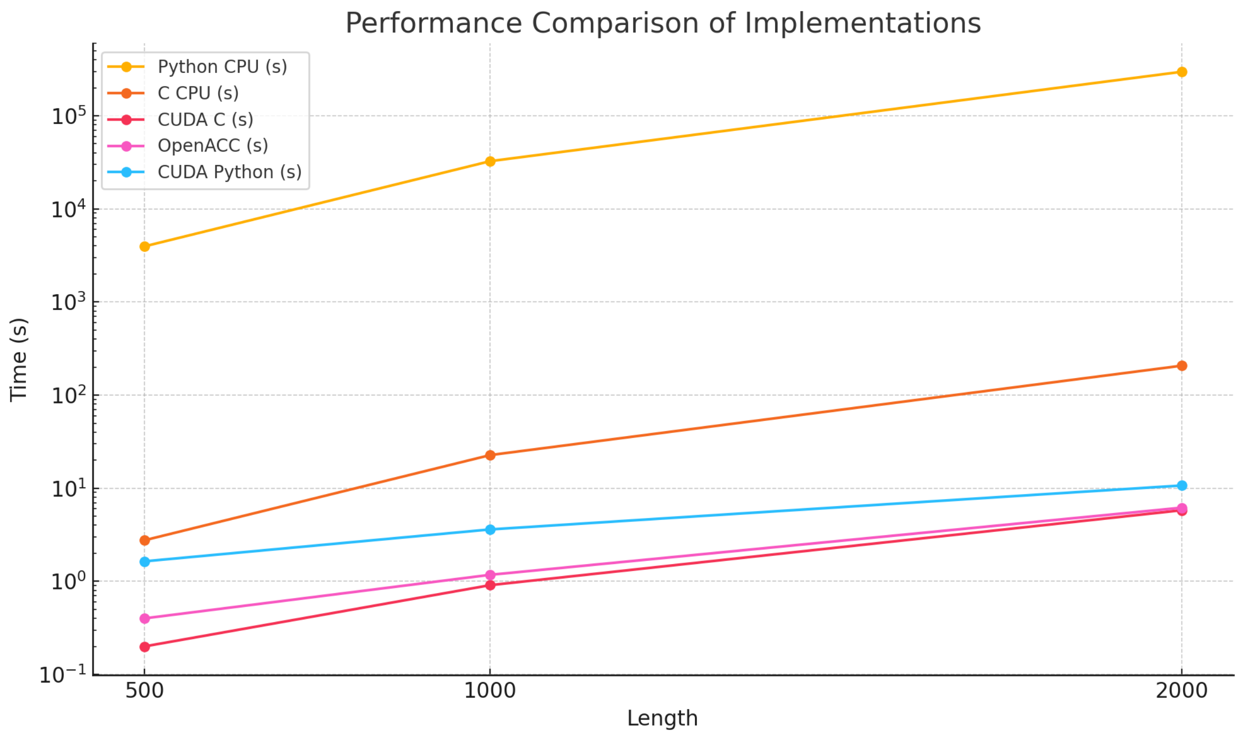
Length	Time (s)	Nodes	Python CPU	C CPU	CUDA C vs C	OpenACC vs C	CUDA Python vs Python
500	40	400	1	1	13.79	6.9	2414.6
1000	80	800	1	1	24.9	19.3	9022
2000	160	1600	1	1	35.6	33.5	27757

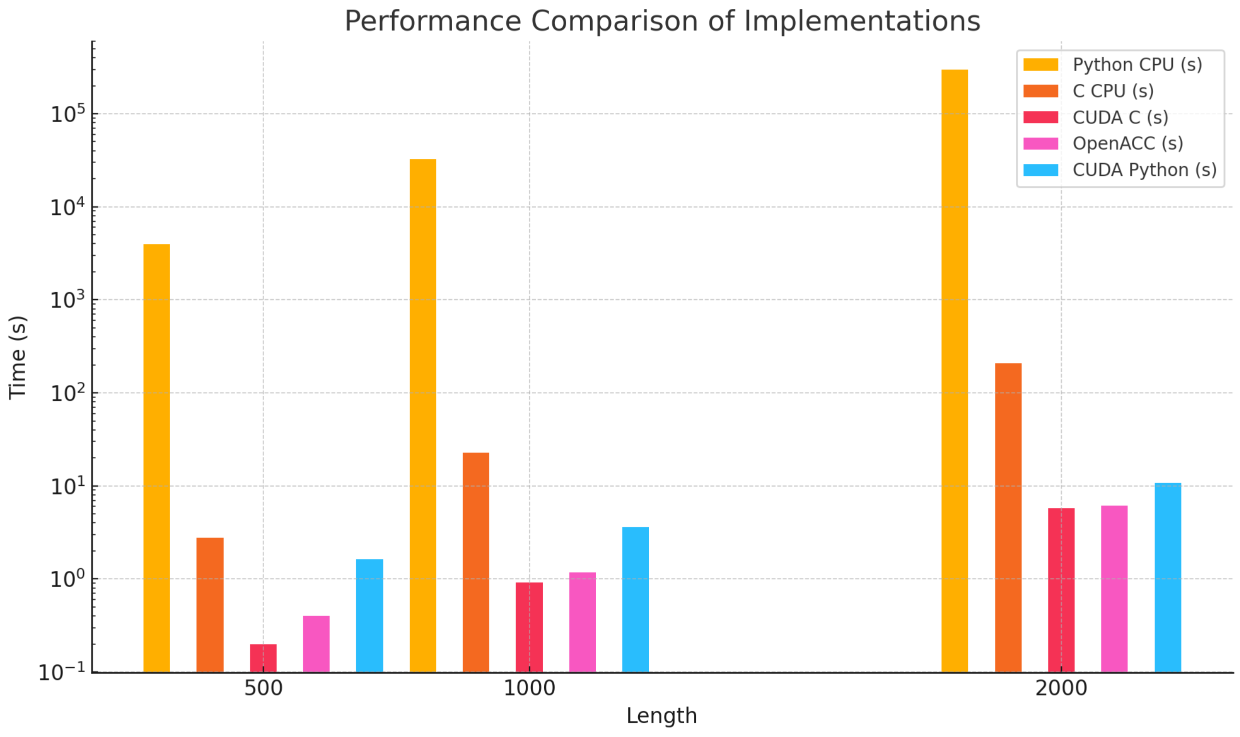
Key Metrics

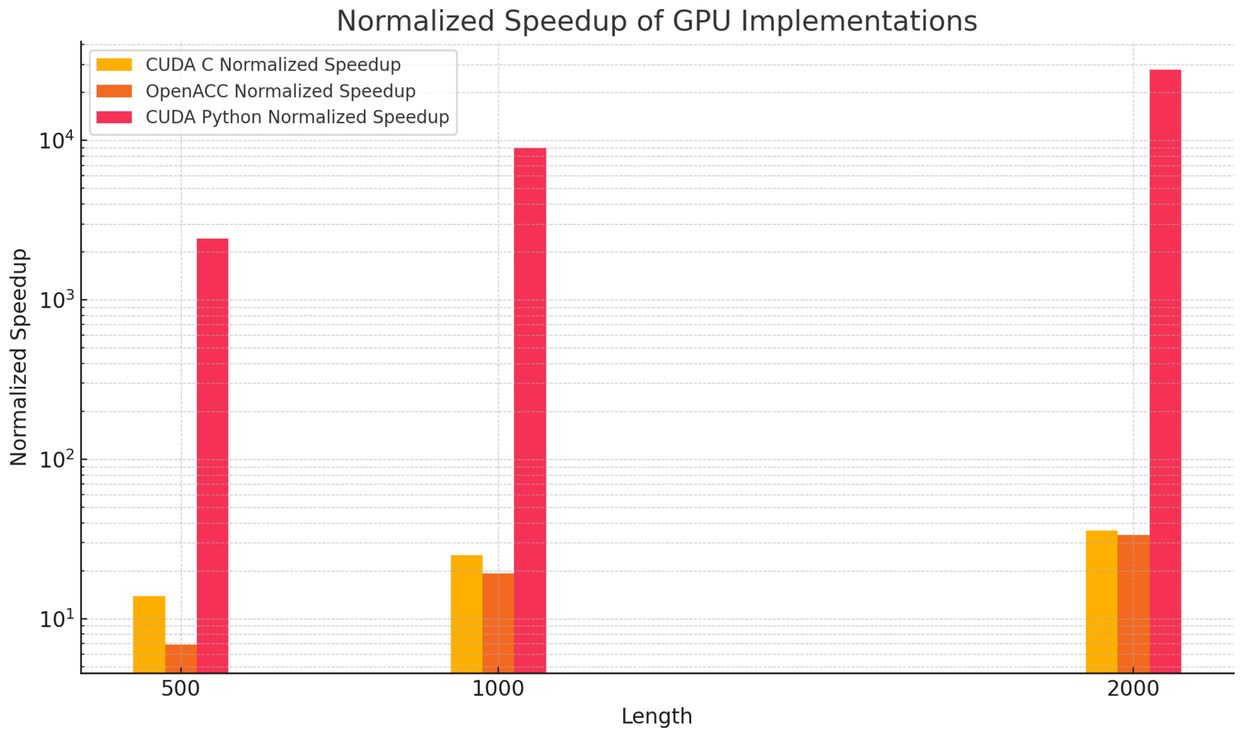
- **Execution Time**: CUDA C consistently achieves the fastest execution, outperforming other methods by significant margins, especially at larger problem sizes.
- **Ease of Implementation**: OpenACC required minimal code changes. CUDA Python offered a Pythonic experience, while CUDA C demanded intricate coding and debugging.
- **Maintainability**: OpenACC and CUDA Python are easier to maintain compared to CUDA C/C++.

6. Results and Discussion

Visualization







Analysis

CUDA C achieves the highest performance due to fine-grained optimizations but comes with higher complexity. OpenACC and CUDA Python enable rapid GPU adoption and are well-suited for teams without deep GPU expertise.

---

7. Conclusion

Key Findings

- GPU acceleration drastically improves runtime for 2D heat diffusion simulations.
- OpenACC offers quick results with minimal code changes, suitable for non-experts.
- CUDA Python balances ease of use with performance, making it a good choice for Python developers.
- CUDA C/C++ provides the best performance but requires significant coding effort.

Future Work

- Investigate multi-GPU configurations.
- Explore additional optimizations like multi-streaming, pinned memory, and tensor cores.
- Compare GPU frameworks like HIP and SYCL for portability and performance.

8. References

1. **NVIDIA Corporation.** (2024). *CUDA Toolkit Documentation.* Retrieved from NVIDIA CUDA Toolkit Documentation.
   (Comprehensive documentation on CUDA programming and optimization techniques.)

2. **Toumi, Y.** (2023). "2D Heat Equation Simulation in Python." GitHub repository. Retrieved from GitHub.
   (An open-source repository providing a numerical simulation of the 2D heat equation using Python.)

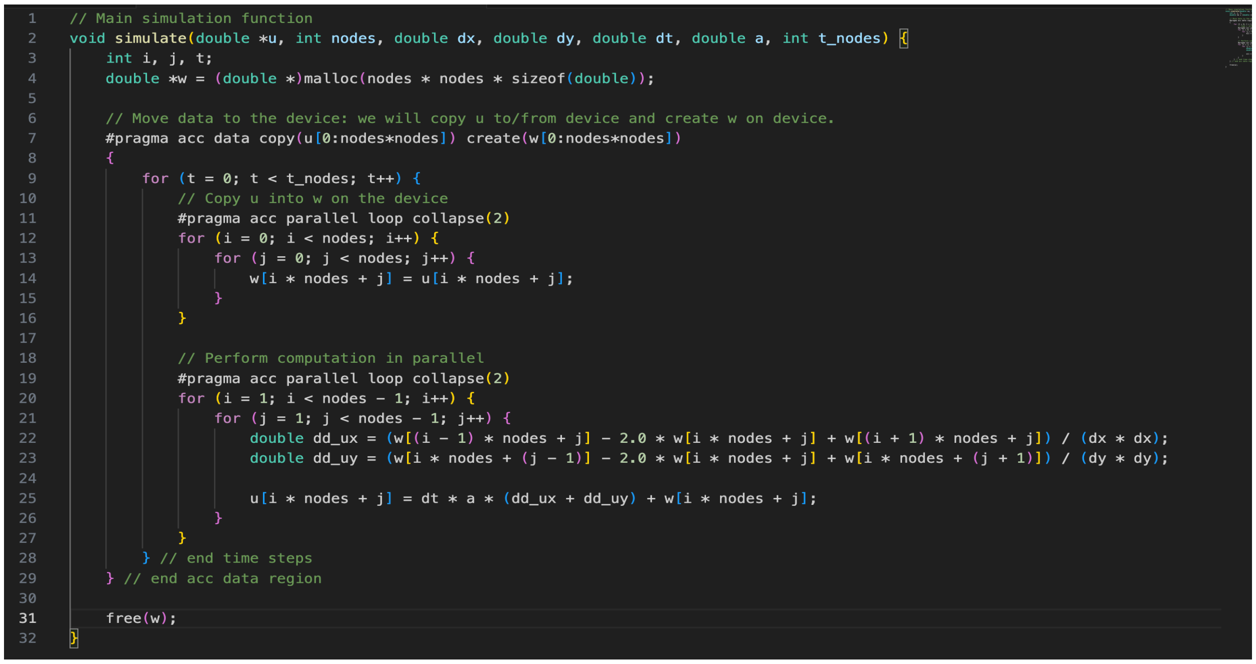
---

9. Appendices

Appendix A: Hardware Specifications

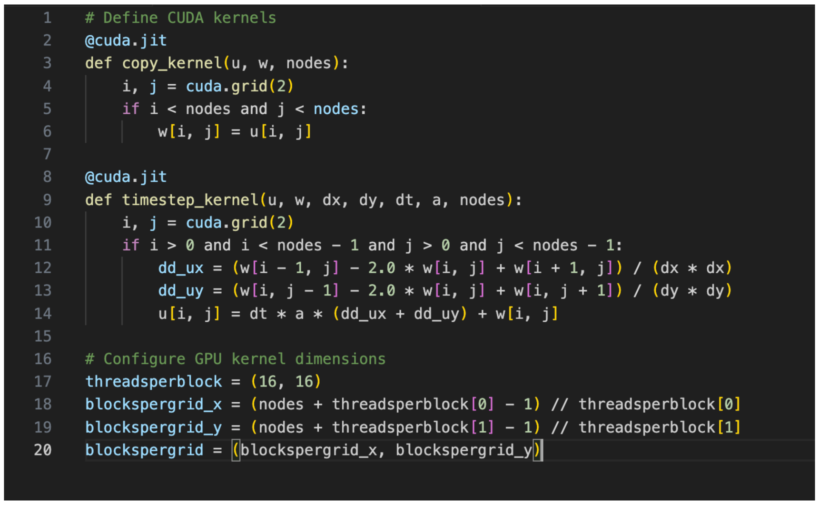
- **CPU**: AMD EPYC 7B13 64-Core Processor
- **GPU**: NVIDIA RTX 4090 (24GB)

Appendix B: Sample OpenAcc



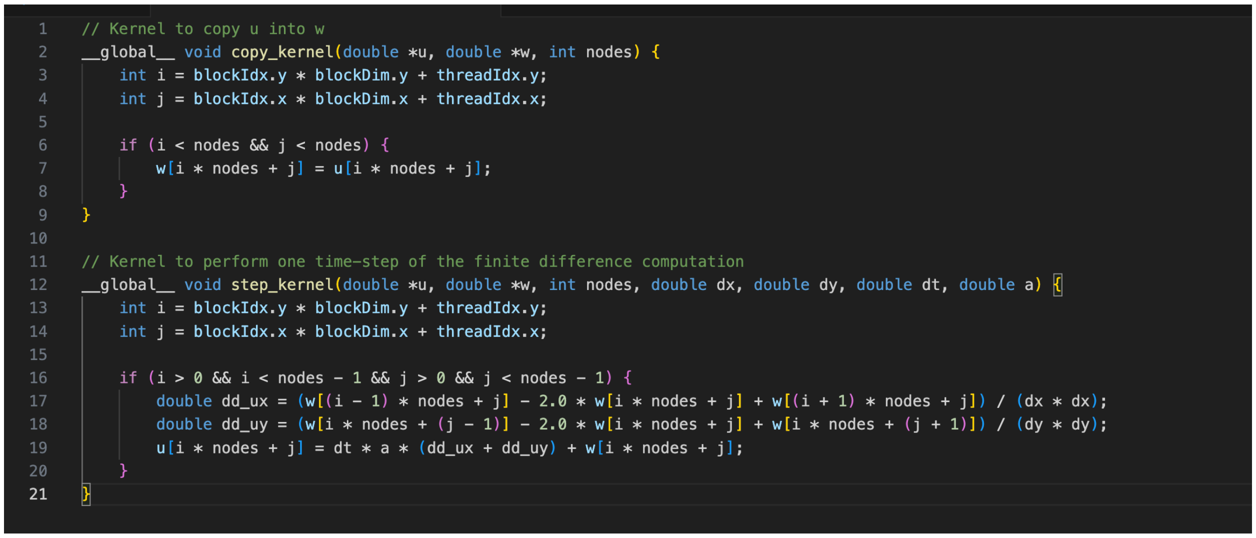
```c
// Main simulation function
void simulate(double *u, int nodes, double dx, double dy, double dt, double a, int t_nodes) {
    int i, j, t;
    double *w = (double *)malloc(nodes * nodes * sizeof(double));

    // Move data to the device: we will copy u to/from device and create w on device.
    #pragma acc data copy(u[0:nodes*nodes]) create(w[0:nodes*nodes])
    {
        for (t = 0; t < t_nodes; t++) {
            // Copy u into w on the device
            #pragma acc parallel loop collapse(2)
            for (i = 0; i < nodes; i++) {
                for (j = 0; j < nodes; j++) {
                    w[i * nodes + j] = u[i * nodes + j];
                }
            }

            // Perform computation in parallel
            #pragma acc parallel loop collapse(2)
            for (i = 1; i < nodes - 1; i++) {
                for (j = 1; j < nodes - 1; j++) {
                    double dd_ux = (w[(i - 1) * nodes + j] - 2.0 * w[i * nodes + j] + w[(i + 1) * nodes + j]) / (dx * dx);
                    double dd_uy = (w[i * nodes + (j - 1)] - 2.0 * w[i * nodes + j] + w[i * nodes + (j + 1)]) / (dy * dy);

                    u[i * nodes + j] = dt * a * (dd_ux + dd_uy) + w[i * nodes + j];
                }
            }
        } // end time steps
    } // end acc data region

    free(w);
}
```

Appendix C: Sample Python (Numba)



```python
# Define CUDA kernels
@cuda.jit
def copy_kernel(u, w, nodes):
    i, j = cuda.grid(2)
    if i < nodes and j < nodes:
        w[i, j] = u[i, j]

@cuda.jit
def timestep_kernel(u, w, dx, dy, dt, a, nodes):
    i, j = cuda.grid(2)
    if i > 0 and i < nodes - 1 and j > 0 and j < nodes - 1:
        dd_ux = (w[i - 1, j] - 2.0 * w[i, j] + w[i + 1, j]) / (dx * dx)
        dd_uy = (w[i, j - 1] - 2.0 * w[i, j] + w[i, j + 1]) / (dy * dy)
        u[i, j] = dt * a * (dd_ux + dd_uy) + w[i, j]

# Configure GPU kernel dimensions
threadsperblock = (16, 16)
blockspergrid_x = (nodes + threadsperblock[0] - 1) // threadsperblock[0]
blockspergrid_y = (nodes + threadsperblock[1] - 1) // threadsperblock[1]
blockspergrid = (blockspergrid_x, blockspergrid_y)
```

Appendix D: Sample CUDA (C)



```c
// Kernel to copy u into w
__global__ void copy_kernel(double *u, double *w, int nodes) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < nodes && j < nodes) {
        w[i * nodes + j] = u[i * nodes + j];
    }
}

// Kernel to perform one time-step of the finite difference computation
__global__ void step_kernel(double *u, double *w, int nodes, double dx, double dy, double dt, double a) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i > 0 && i < nodes - 1 && j > 0 && j < nodes - 1) {
        double dd_ux = (w[(i - 1) * nodes + j] - 2.0 * w[i * nodes + j] + w[(i + 1) * nodes + j]) / (dx * dx);
        double dd_uy = (w[i * nodes + (j - 1)] - 2.0 * w[i * nodes + j] + w[i * nodes + (j + 1)]) / (dy * dy);
        u[i * nodes + j] = dt * a * (dd_ux + dd_uy) + w[i * nodes + j];
    }
}
```