Hanoi University of Science and Technology

Semester: 2024.2

Course: Parallel and Distributed Programming

Supervisor: Vu Van Thieu

2D Heat Equation

May 25, 2025

GROUP 18

Group 18:	VU HOANG NHAT ANH	20225471
	NGUYEN HUU CONG	20225476
	DANG KIEU TRINH	20214933
	TRAN THANH VINH	20225539

Contents

1 Problem

1.1 Problem description

In this report, we implement a solution to solve a 2-dimensional differential equation using parallel algorithms using a concrete example: *Given a square metal plate of dimensions M×N with the initial temperature of 25°C, the center of the plate is in contact with a heat source at 100°C. Our goal is to mathematically model the way* ***thermal energy moves through the plate.***

1.2 Mathematical Modelling

Let

- M, N is the size of the plate. In our problem assume M=N.
- x, y is the x and y coordinate of a position in the plate.
- t is the amount of time passed since the start of the heating process.
- $u(x, y, t)$ be temperature distribution of the page at position (x, y) and time t.

The heat equation is:

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

where:

- D is the thermal diffusivity.
- $\frac{\partial^2 u}{\partial x^2}$ is the second partial derivatives of u with respect to x.
- $\frac{\partial^2 u}{\partial y^2}$ is the second partial derivatives of u with respect to y.
- $\frac{\partial u}{\partial t}$ represents the rate of changes of temperature with respect to time t, describing how the temperature evolves over time.

2 Algorithms

2.1 Numerical Approach

Assuming that M and N are integers of a given unit (E.g., if M is 1.09 meters, it needs to be converted to 109 centimeters), we split the plate into NxM squares, with the length of each side being 1 unit.

2.2 Forward Euler Method

The Forward Euler Method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value.

Consider the (1) equation, we use the finite difference method to transform its right-hand side. Afterwards, we obtain the following equation:

$$\frac{\partial u(i,j,t)}{\partial t} = D\left(\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{dx^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{dy^2}\right) \tag{2}$$

Let the right-hand side equals to $FD(i,j)$, (2) becomes:

$$\frac{\partial u(i,j,t)}{\partial t} = FD(i,j) \tag{3}$$

Recall that we have to compute the value in each (i,j) point on the MxN grid with respect to time, so for each step dt, we consider it as a step in a loop run for a fixed amount of time. In mathematical notation, it is:

$$u^t(i,j) = dt * FD^t(i,j) + u^{t-1}(i,j) \tag{4}$$

The last component $u^{t-1}(i,j)$ can be consider as a number determined by the initial value $u^0(i,j)$, which is the initial values of the problem $\forall(i,j) \in [0,M] \times [0,N]$.

2.3 Implementation

2.3.1 Initial values

We suppose that:

$$u^0(i,j) = 100^oC$$

$\forall(i,j) \in [\frac{M}{2} - 4 * (\frac{M}{64}), \frac{M}{2} + 4 * (\frac{M}{64})] \times [\frac{N}{2} - 4 * (\frac{N}{64}), \frac{N}{2} + 4 * (\frac{N}{64})]$

This setting is to keep the square proportional to the grid size (for e.g. 64x64 => 8x8 square but for 512x512 => 64x64 square). This settings only works for the grid size 64x64 and above, and the grid does not have to be a square ($M \neq N$). The other points at t=0 have the value 25^oC.

At each step t, the formula will compute $u^t(i,j)$ using the 4 equation.

2.3.2 Boundary conditions

$$u^t(0,j) = u^t(i,0) = u^t(M-1,j) = u^t(i,N-1) = 25^oC$$

$\forall (i, j) \in [0, M] \times [0, N]$

$\forall t \in [0, Ntime - 1]$

3 Parallel Design

3.1 Serial implementation (non-parallel)

For this implementation, we use 2 for loops to iterate through the grid and calculate the values.

Algorithm 1 2D Heat Equation Solver (Serial)

```
procedure MAIN
    Allocate arrays T and dT of size M × N
    Initialize T with boundary and initial conditions
    for t = 0 to N_time − 1 do
        DERIVATIVE(T, dT)
        SOLVINGODE(T, dT)
    end for
    Free T, dT
end procedure
```

From this we improve this implementation by letting the Derivative() and SolvingODE() execute in parallel.

3.2 Parallel implementation

3.2.1 Variables and Initialization

- M, N: Dimensions of the metal plate
- dx, dy: Spatial step size in x and y directions
- *GridSize*: Number of thread blocks in each dimension
- *BlockSize*: Number of threads in each block
- dt: Time step size
- D: Diffusion coefficient

The metal plate is discretized into a 64×64 grid by default. We also try changing the size of the grid into 256×256, 512×512, 2048×2048, with a spatial step size of 0.1 in both x and y directions. The CUDA kernel is launched with a 4×4 grid of thread blocks, where each block consists of $16 \times 16 = 256$ threads. This is only true for the 64×64 grid. On larger grids, we increase the number of blocks in a grid (GridSize) to ensure there are fewer than 1024 threads in a thread block.

The time step size (dt) is set to 0.1, and the diffusion coefficient (D) is specified as 0.01.

3.2.2 Pseudocode

Algorithm 2 CUDA Implementation of 2D Heat Equation

procedure DERIVATIVE_KERNEL(T, dT)
  $i \leftarrow$ global row index from block and thread indices
  $j \leftarrow$ global column index from block and thread indices
  if $1 \leq i < M-1$ and $1 \leq j < N-1$ **then**
    $dT_{i,j} \leftarrow D \cdot \left(\frac{T_{i-1,j} - 2 \cdot T_{i,j} + T_{i+1,j}}{dx^2} + \frac{T_{i,j-1} - 2 \cdot T_{i,j} + T_{i,j+1}}{dy^2} \right)$
  end if
end procedure
procedure SOLVINGODE_KERNEL(T, dT)
  $i \leftarrow$ global row index from block and thread indices
  $j \leftarrow$ global column index from block and thread indices
  if $i < M$ and $j < N$ **then**
    $T_{i,j} \leftarrow T_{i,j} + dt \cdot dT_{i,j}$
  end if
end procedure
procedure MAIN
  Allocate T_{cpu} on host
  Initialize T_{cpu}
  Allocate T_{gpu} and dT_{gpu} on device
  Copy $T_{cpu} \rightarrow T_{gpu}$
  Define grid: *dimGrid* $\leftarrow$ ($GridSizeX$, $GridSizeY$)
  Define block: *dimBlock* $\leftarrow$ ($BlockSizeX$, $BlockSizeY$)
  for $t = 0$ to $N_{\text{time}} - 1$ **do**
    Launch DERIVATIVE_KERNEL(T_{gpu}, dT_{gpu}) with *dimGrid*, *dimBlock*
    Launch SOLVINGODE_KERNEL(T_{gpu}, dT_{gpu}) with *dimGrid*, *dimBlock*
  end for
  Copy $T_{gpu} \rightarrow T_{cpu}$
  Free T_{cpu}, T_{gpu}, dT_{gpu}
end procedure

4 Results

4.1 CUDA and Serial

We recorded runtime of the 2 methods using (ran on one Kaggle session, GPU T4 x2):

- **Serial method**: Using <time.h> package and recorded the runtime.
- **CUDA method**: Using a number of functions provided in the <cuda.h> package.

Below is the code to record the runtime of the serial method:

```
clock_t start = clock();
for (int t=0;t<Ntime;t++)
{
    derivative(Tcpu,dTcpu);
    solvingODE(Tcpu,dTcpu);
}
clock_t end = clock();
double cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("CPU Time: %f seconds\n", cpu_time);
clock_t end = clock();
```

Below is the code to record the runtime of the CUDA method:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

for (int t=0; t<Ntime; t++) {
    Derivative<<<dimGrid,dimBlock>>>(Tgpu,dTgpu);
    SolvingODE<<<dimGrid,dimBlock>>>(Tgpu,dTgpu);
}
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("GPU Time: %f ms\n", milliseconds);
```

After recording runtimes of 4 different grid sizes, we obtained the following data:

Mesh grid	Serial (ms)	Parallel (CUDA) (ms)	Difference (ms)
64×64	3.113	1.108	2.005
256×256	46.538	4.799	41.739
512×512	201.180	16.153	185.027
2048×2048	3115.403	247.434	2867.969

Table 1: Execution time comparison between serial and CUDA implementations across different mesh grid sizes.

In the 64x64 grid, the CUDA implementation is approximately 3 times faster than that of the Serial. On larger grids, the CUDA implementation proves to be much more effective than the Serial implementation.

4.2 Visualization

To enable visualization, we have to store the states of the grid into a tensor of NtimexMxN size. Below is an image of the last state in the Tensor(64x64) of the Serial method:

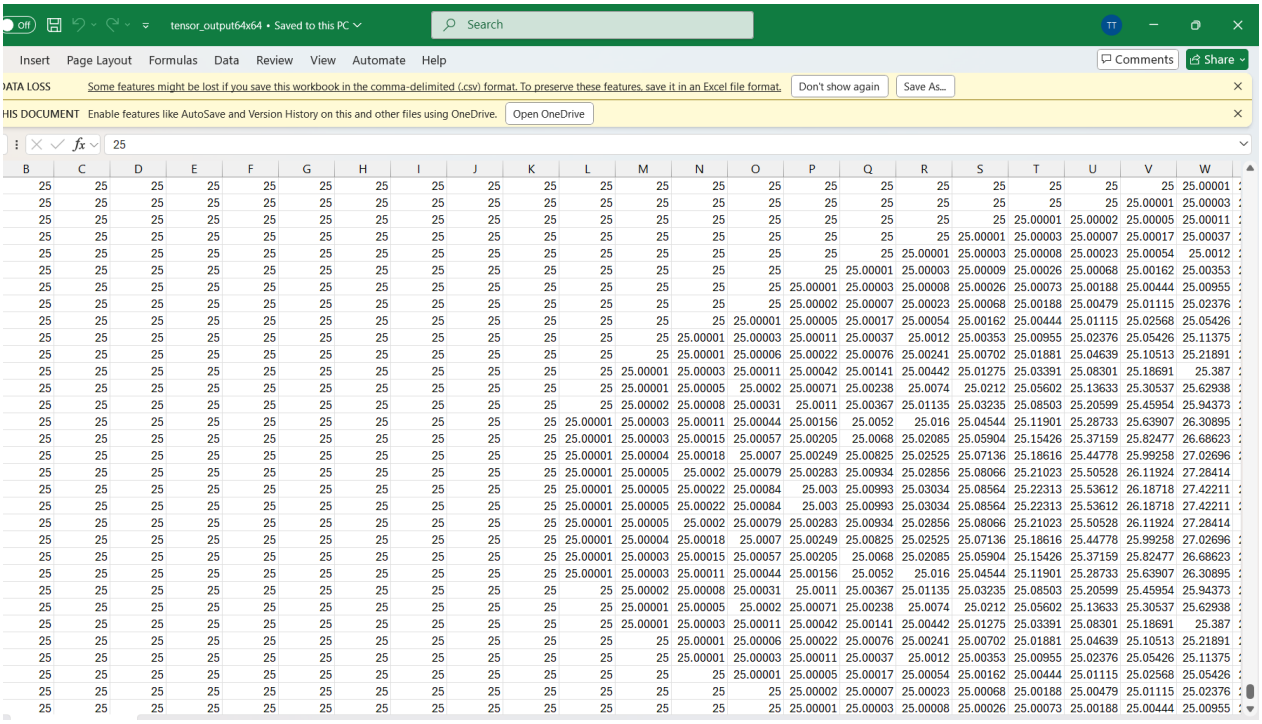

Figure 1: *A section of the last state of the environment (Serial)*

Below is an image of the last state in the Tensor(64x64) of the CUDA method:

Here is a visualization made by Python libraries, such as matplotlib:

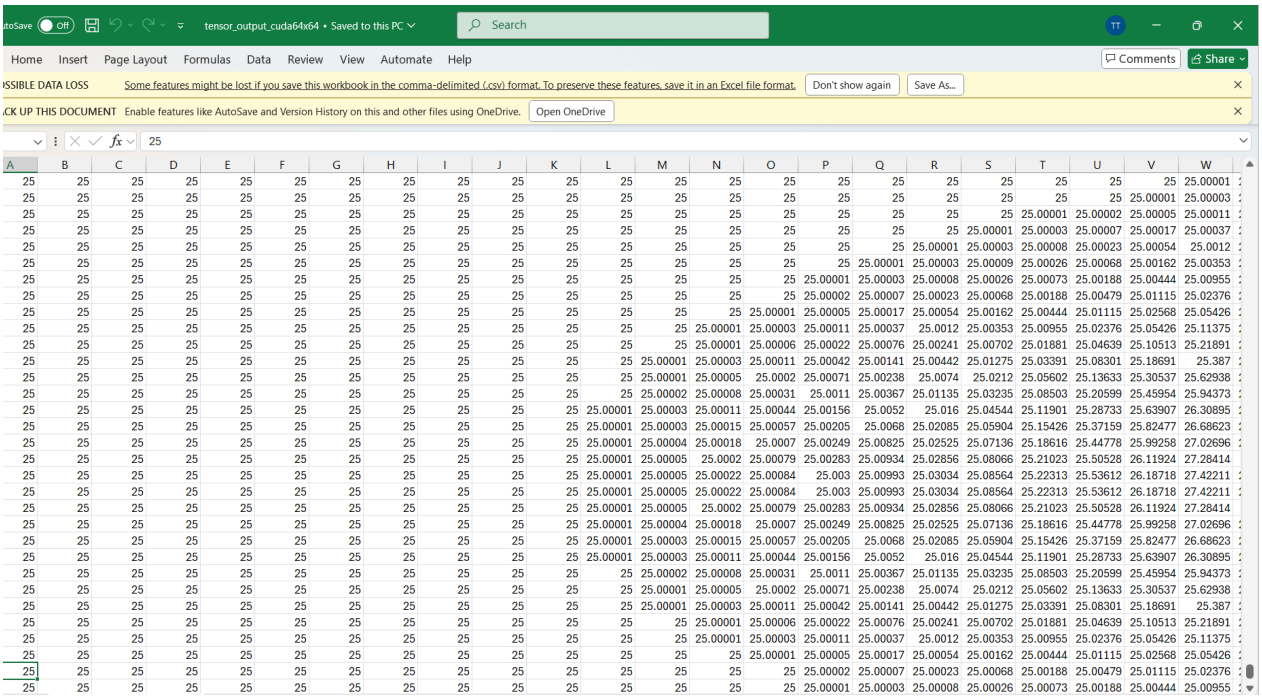

Figure 2: *A section of the last state of the environment (CUDA)*

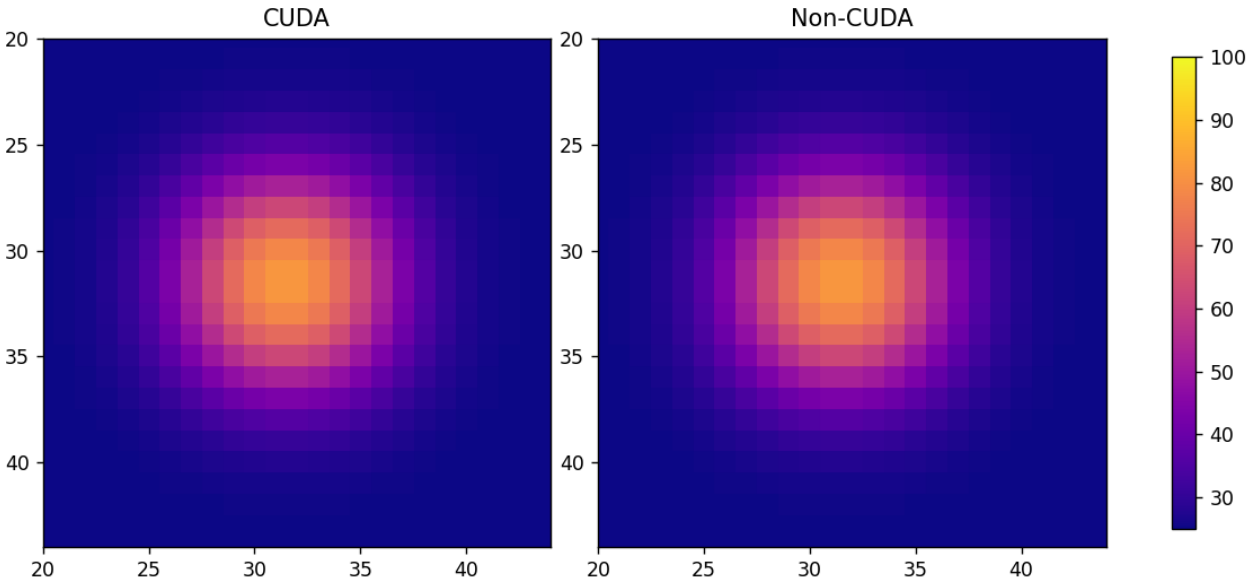

Figure 3: *Heatmap comparison between Serial and CUDA*

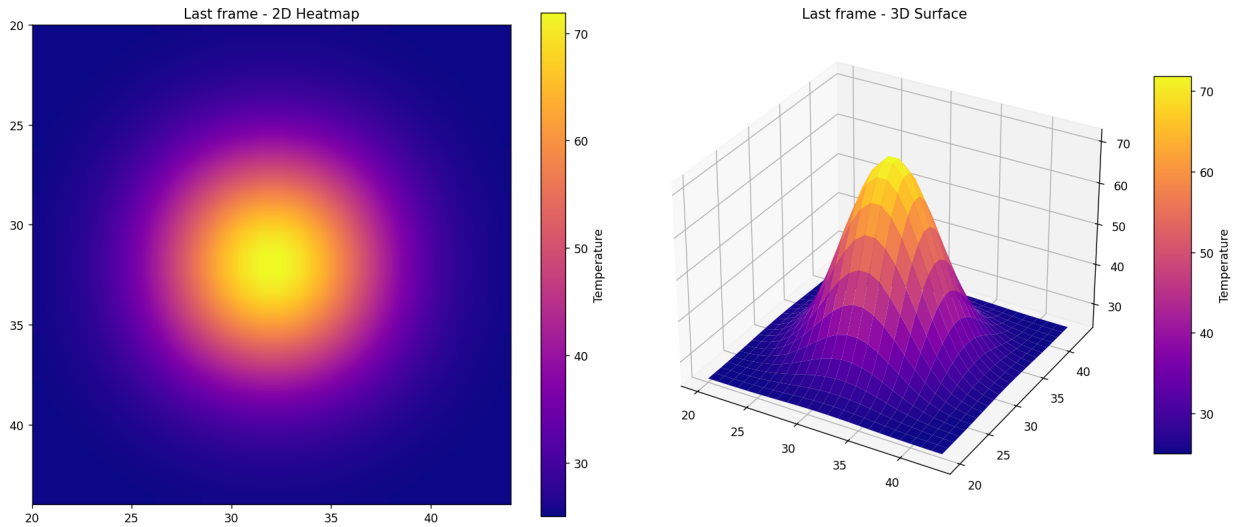

Figure 4: *Visualization of the last state/frame of the environment*