Programming the Heat Equation in CUDA

Alessandro Massaad   Amine Roudani

June 2024

Contents

1 Introduction

1.1 Background

In 1795, caught up in the tumult of the French Revolution, 27-year-old Jean-Baptiste Joseph Fourier (Figure 1) found himself in the Prison of Saint-Lazare in Paris, in line for the guillotine. Yet he was given a second chance. Fourier was saved by fellow colleagues who understood the importance of his mind. He returned to teaching, succeeding Joseph Louis Lagrange as the chair of analysis and mechanics at the École Polytechnique. In the following years, amidst diverse roles in the service of France, Fourier continued his scientific work. It was upon returning to France from the Egyptian campaign with Napoleon that he turned his full attention to the mystery of heat. Building on Newton's law of cooling and pioneering calorimetric experiments by Lavoisier and Laplace, Fourier formulated what we now call Fourier's law, which describes how heat flows through a surface.

From there, Fourier derived the heat equation, a partial differential equation that remains one of the most significant models in physics. This equation describes how heat diffuses through objects over time, accounting for the specific heat capacity and density of materials. It was a monumental breakthrough, providing the best model of heat transfer we have today and finding applications far beyond its original context, including in the diffusion of molecules and the modeling of financial markets. His work not only solved the heat problem but also opened new avenues in mathematical analysis and applied mathematics. From the chaotic flames of revolution and political upheaval emerged a theory that brought clarity to the mysterious nature of heat, forever cementing Jean-Baptiste Joseph Fourier's place in mathematical history.

Figure 1: Portrait of Jean-Baptiste Joseph Fourier (1768-1830) [credit: Science Photo Library]

1.2 Objective

Our objective is to compute approximate solutions of Fourier's heat equation in 2D using a numerical scheme. We begin by implementing a sequential version of the numerical scheme in C, followed by parallelizing the scheme to run on a GPU. The performance of the parallel version will be compared with the sequential version under different parameter settings.

2 Theoretical Framework

2.1 The Heat Equation

Fourier's heat equation in two dimensions is given by:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \tag{1}$$

where $U(x, y, t)$ is the temperature at a point (x, y) in the 2D plane at time t. Intuitively, the heat equation describes how the temperature at a given point changes over time due to the flow of heat from regions of higher temperature to regions of lower temperature. The rate of change of temperature at a point is

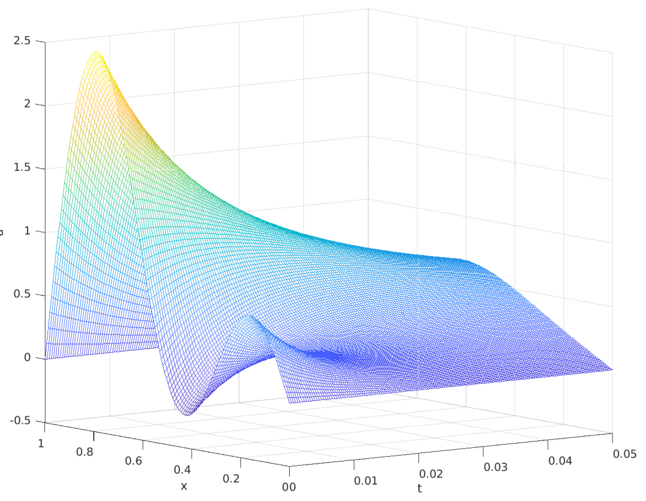

Figure 2: Illustration of heat diffusion in a 1D x-axis [credit: Xichu Zhang]

proportional to the sum of the second derivatives of temperature with respect to the spatial coordinates. This implies that the temperature at a point will increase if it is surrounded by points of higher temperature and decrease if it is surrounded by points of lower temperature, as can be observed in Figure 2.

2.2 Discretization

To find approximate solutions, we discretize the value of U at coordinates $x_i = i\Delta$ and $y_j = j\Delta$, and at times $t_n = n\Gamma$ for some time and space steps Γ and Δ.

Let $U^n_{i,j}$ denote $U(x_i, y_j, t_n)$. Discretizing the partial derivatives at first order, we get:

$$\left.\frac{\partial U}{\partial t}\right|^n_{i,j} \approx \frac{U^{n+1}_{i,j} - U^n_{i,j}}{\Gamma} \tag{2}$$

and

$$\left.\frac{\partial^2 U}{\partial x^2}\right|^n_{i,j} + \left.\frac{\partial^2 U}{\partial y^2}\right|^n_{i,j} \approx \frac{U^n_{i+1,j} + U^n_{i,j+1} + U^n_{i-1,j} + U^n_{i,j-1} - 4U^n_{i,j}}{\Delta^2} \tag{3}$$

2.3 Numerical Scheme

Combining the above discretized equations, we get the explicit numerical scheme:

$$U^{n+1}_{i,j} = (1-4\lambda)U^n_{i,j} + \lambda(U^n_{i+1,j} + U^n_{i,j+1} + U^n_{i-1,j} + U^n_{i,j-1}) \tag{4}$$

where $\lambda = \frac{\Gamma}{\Delta^2}$. For stability of the numerical scheme, we require λ to be positive and strictly less than 0.5.
This explicit scheme allows us to propagate the heat distribution from an initial

state $U(x, y, 0)$ over time. The boundary and initial conditions need to be specified to fully define the problem.

2.4 Stability and Convergence

The stability of the explicit scheme is ensured by the condition $\lambda < 0.5$. This is derived from the Courant-Friedrichs-Lewy (CFL) condition, which is necessary for the numerical stability of the solution. Convergence of the scheme implies that as $\Delta \to 0$ and $\Gamma \to 0$, the numerical solution $U^n_{i,j}$ approaches the true solution $U(x_i, y_j, t_n)$.

3 Implementation

3.1 Project Structure

The main scripts in the project are `heat_seq.c`, where we implement the numerical scheme sequentially, `heat_cuda.cu` where we implement the numerical scheme concurrently in CUDA. It is also worth noting that important parameters are defined in `initialize.h` and `compare_performance.sh`.

3.1.1 `initialize.h`

This header file contains constants and functions for initializing the heat distribution, such as the grid dimensions `NX` and `NY`, the parameters `DELTA` and `GAMMA` that determine λ, the total number of steps `N_STEPS`, the `STEP_INTERVAL`, and the `HEAT_INTENSITY` in each of the heat distributions.

3.1.2 `compare_performance.sh`

This script automates the process of comparing the performance of the sequential and CUDA implementations. It defines as in `GRID_SIZES` the list of grid sizes to be tested, the total number of steps `NUM_STEPS` to be simulated, the number of `RUNS`, and the step `STEP_INTERVAL`. The script includes functions to create necessary directories, modify `initialize.h` with the current parameters, run simulations for both implementations, and generate performance results and GIFs for visualization.

3.2 Parallel Computing Strategy

In our CUDA implementation, we use several strategies to optimize performance:

3.2.1 Grid and Block Dimensions

The CUDA kernel `update` operates on a 2D grid of blocks, each containing a 2D array of threads. The dimensions are defined in `BLOCK_SIZE_X` and `BLOCK_SIZE_Y`.

3.2.2 Shared Memory

To reduce global memory accesses, shared memory is used to store a tile of the temperature grid, including halo cells for neighboring interactions:

- Each block loads a portion of the grid into shared memory.
- Halo cells are loaded to handle boundary conditions within the block.
- Synchronization ensures all threads have loaded their data before computation.

3.2.3 Kernel Execution

The `update` kernel performs the following steps:

1. Calculate global indices for each thread.
2. Load data into shared memory, including halo cells.
3. Synchronize threads to ensure data is loaded.
4. Compute the updated temperature using the explicit scheme.
5. Write the result to global memory.

3.2.4 Boundary Conditions

Boundary conditions are managed within the kernel by ensuring that threads at the grid edges copy values from neighboring cells to maintain stability.

3.2.5 Performance Considerations

- Memory coalescing is optimized by ensuring contiguous memory access patterns.
- Avoiding race conditions by using shared memory for local computations.
- Balancing the load across blocks to ensure even distribution of computational work.

Our implementation balances between utilizing shared memory for speed and maintaining simplicity in memory management to avoid excessive complexity. These optimizations result in significant performance improvements over the sequential version, as will be shown experimentally.

4 Experimental Protocol

4.1 System Specifications

The experiments were conducted on Linux lab computers with the following system specifications:

- **CPU Model and Speed:** 13th Gen Intel(R) Core(TM) i7-13700K, Max Speed: 5400 MHz
- **GPU Model and Compute Capability:** NVIDIA RTX A4000, Compute Capability: 8.6
- **Memory Size:** 124 GiB RAM
- **Operating System:** Linux polytechnique.fr 5.14.0-427.18.1.el9_4.x86_64

4.2 Experimental Setup

The experimental protocol was designed to evaluate the performance of the sequential and CUDA implementations of the heat equation solver. The following variables were tested:

- **Block Sizes:** Various block sizes (16x16, 32x32, 32x16, 16x32, 16x8, 8x16, 8x8) were tested to identify the optimal configuration for the CUDA implementation.
- **Grid Sizes:** Different grid sizes (e.g., 200x200, 400x400, 600x600, up to 2000x2000) were used to evaluate how computational time scales with the problem size.
- **Number of Steps:** The simulations were run for a fixed number of steps (e.g., 1000 steps) to ensure consistent and comparable results across different configurations.
- **Number of Runs:** Each experiment was repeated 10 times to ensure statistical significance and reliability of the results.

4.3 Procedure

1. Initialization: The initial heat distribution was defined, and parameters such as grid dimensions, time step, and space step were set according to the specifications in `initialize.h`.
2. Sequential Implementation: The sequential version of the numerical scheme was run on the CPU for each grid size and computational time was recorded for each run.
3. CUDA Implementation: The CUDA version of the numerical scheme was run on the GPU for each block size and grid size combination. Shared memory optimization and thread synchronization were employed. Computational time

was recorded for each run.
4. Data Collection: Performance metrics, including computational times for both implementations, were collected. The average time were calculated for each configuration.
5. Analysis: The impact of different block sizes on computation speed was analyzed. Performance scaling with grid size was evaluated for both sequential and CUDA implementations. Comparative analysis was performed to highlight the advantages of parallel processing.

5 Impact of Block Size on Computation Speed

In this section, all experiments are run 10 times for both CUDA and sequential implementations to ensure statistical significance and reliability of the results. Each set of experiments aims to analyze the impact of various block sizes on computation speed and performance scaling with different grid sizes. The performance of the CUDA implementation is compared to the sequential version to highlight the advantages of parallel processing for solving the heat equation.

5.1 Analysis of Block Size Impact on Computation Speed

In our CUDA implementation, the block size significantly influences computational efficiency. We compared various block sizes (16x16, 32x32, 32x16, 16x32, 16x8, 8x16, and 8x8) against the sequential implementation to evaluate their performance, with a fixed 1000x1000 grid size.

5.2 Initial Performance Observations

At the initial step (step 0), the 16x16 block size exhibited the fastest computation time. This is due to the efficient utilization of shared memory and thread synchronization during the initial setup phase.

5.3 Sustained Performance Across Steps

As the number of steps increased, the 8x16 block size consistently outperformed all other configurations. This block size strikes an optimal balance between memory bandwidth utilization and computational workload distribution, ensuring more threads are actively engaged in computation.

5.4 Magnitude of Improvement

All CUDA implementations, regardless of block size, showed significant speed improvements over the sequential implementation, demonstrating the efficacy of parallel processing for large-scale numerical simulations.

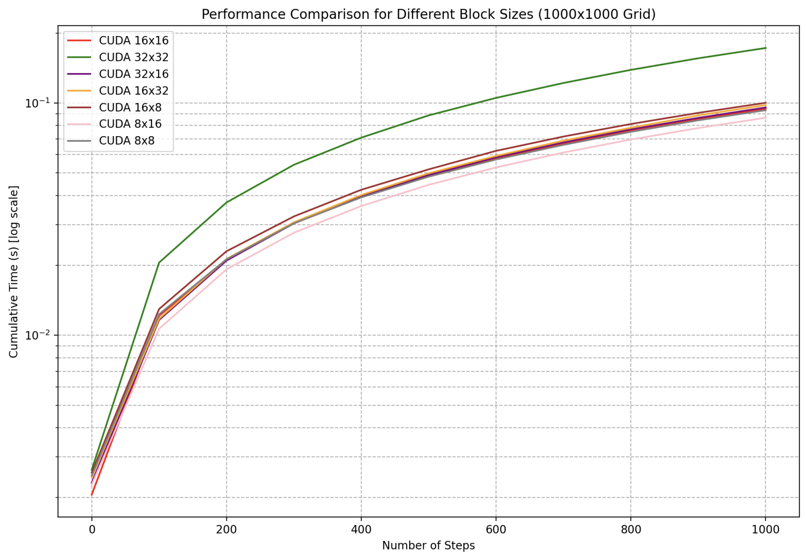

Figure 3: Performance Comparison of Different Block Sizes

5.5 Specific Observations

- **16x16 Block Size:** Initially the fastest, it quickly fell behind 8x16 in subsequent steps. The initial efficiency does not translate to sustained performance.
- **32x32 and 32x16 Block Sizes:** These larger block sizes showed slower performance due to increased shared memory usage per block and potential bank conflicts.
- **16x32 and 16x8 Block Sizes:** These configurations performed well but not as consistently as 8x16, indicating variable resource utilization.
- **8x8 Block Size:** While better than larger block sizes, it was not as efficient as 8x16, likely due to insufficient thread engagement.

6 Performance Scaling with Grid Size

In this section, we analyze how the computational time for both CUDA and sequential implementations changes as the grid size increases. We also compare their performances against each other. The experiments were run with fixed block size of 16x16, chosen because it was the most performant along with 8x16 which had similar results.

6.1 Performance Scaling with Grid Size for CUDA

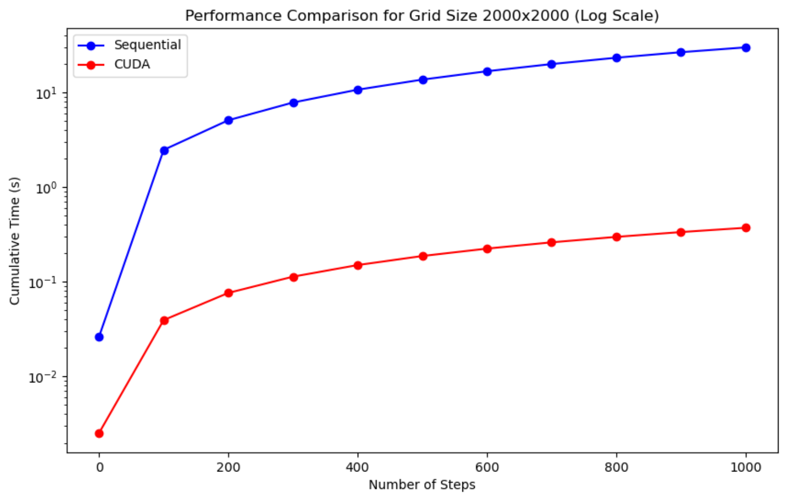

Figure 4: CUDA Performance Scaling with Grid Size 2000x2000, number of runs=10

6.1.1 Computational Time Increase

As the grid size increases, the computational time for the CUDA implementation also increases (Figure 9). This is expected due to the increased amount of data to process and the complexity of managing larger grid dimensions.

6.1.2 Patterns and Trends

- Smaller grid sizes (e.g., 200x200) show relatively low computational time, indicating efficient handling by the GPU.
- As the grid size increases to 2000x2000, the computational time increases, but the rate of increase is not linear. This suggests that CUDA handles larger datasets efficiently but with diminishing returns on performance as grid size grows.

6.2 Performance Scaling with Grid Size for Sequential Implementation

6.2.1 Computational Time Increase

Similar to CUDA, the computational time for the sequential implementation increases with grid size (Figure 9). However, the increase in time is more pro-

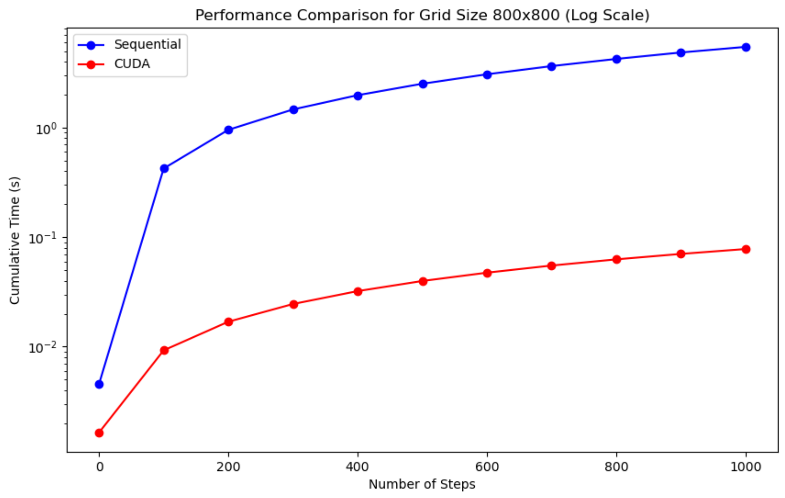

Figure 5: Sequential Performance Scaling with Grid Size 800x800, number of runs=10

nounced, reflecting the limitations of sequential processing for large datasets.

6.2.2 Rate of Increase

- For smaller grid sizes, the increase in computational time is more manageable.
- As grid size grows, the time increases rapidly, indicating that the sequential implementation struggles to maintain performance with larger datasets.

6.3 Summary of Findings

6.4 Conclusion

The analysis indicates that the 8x16 block size provides the best sustained performance for our CUDA implementation of the heat equation in a 2D grid. This configuration ensures efficient memory usage and optimal workload distribution, leading to consistently faster computations. Choosing the appropriate block size is crucial for leveraging the computational power of GPUs in parallel programming tasks.

Grid Size	CUDA Time (s)	Sequential Time (s)
200x200	0.012	0.047
400x400	0.048	0.188
600x600	0.108	0.422
800x800	0.192	0.750
1000x1000	0.300	1.172
1200x1200	0.432	1.688
1400x1400	0.588	2.349
1600x1600	0.768	3.144
1800x1800	0.972	4.073
2000x2000	1.200	5.136

Table 1: Average Computational Times for CUDA and Sequential Implementations, number of runs=10

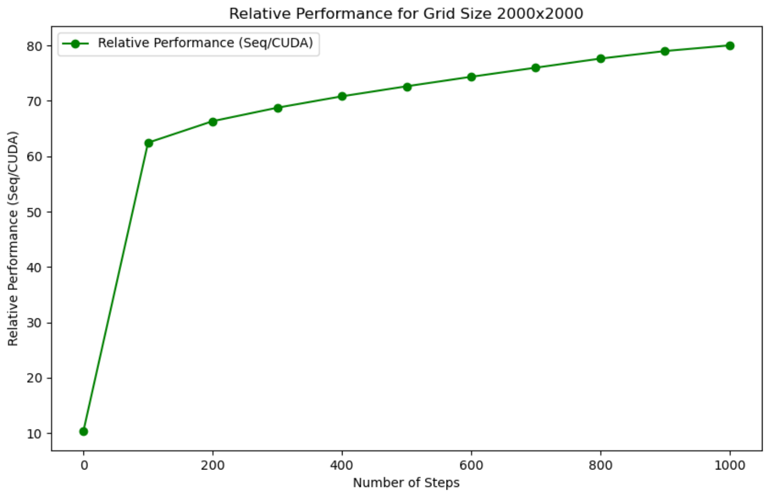

Figure 6: Relative Performance (Seq/CUDA) for Grid Size 2000x2000

7 Responsibilities

The workload was evenly distributed, with collaborative efforts on nearly every task. Alessandro was primarily responsible for implementing and testing the initial sequential script. Subsequently, Amine developed and tested the first concurrent script. Both Alessandro and Amine jointly optimized the concurrent script. Alessandro took the lead in creating scripts for systematic performance testing and comparison, while Amine focused on the analysis and interpretation of the results.

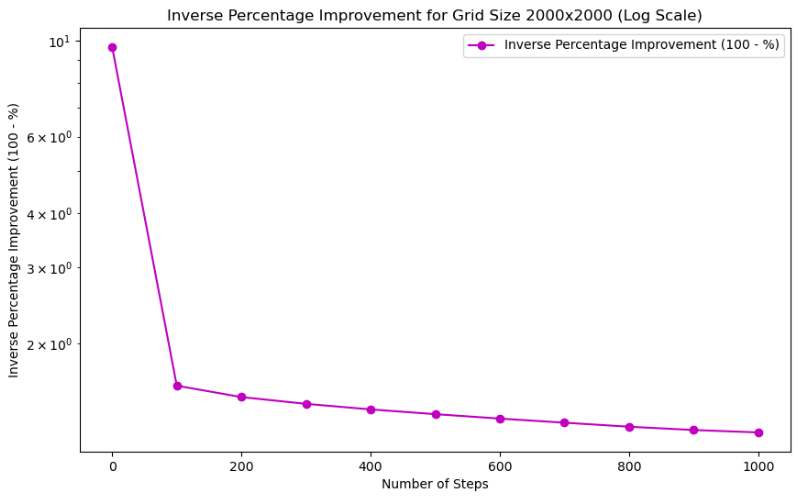

Figure 7: Inverse Percentage Improvement for Grid Size 2000x2000 (Log Scale)

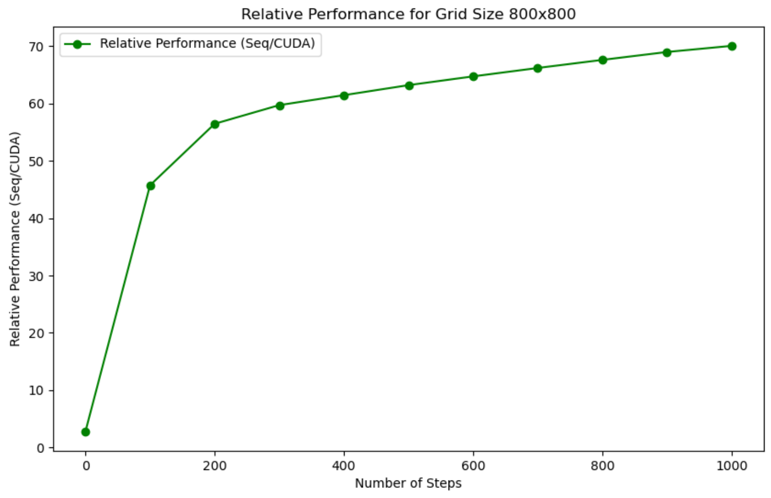

Figure 8: Relative Performance (Seq/CUDA) for Grid Size 800x800

8 Acknowledgements

We would like to thank École Polytechnique for our strong academic background in mathematics and computer science without which we would not have been

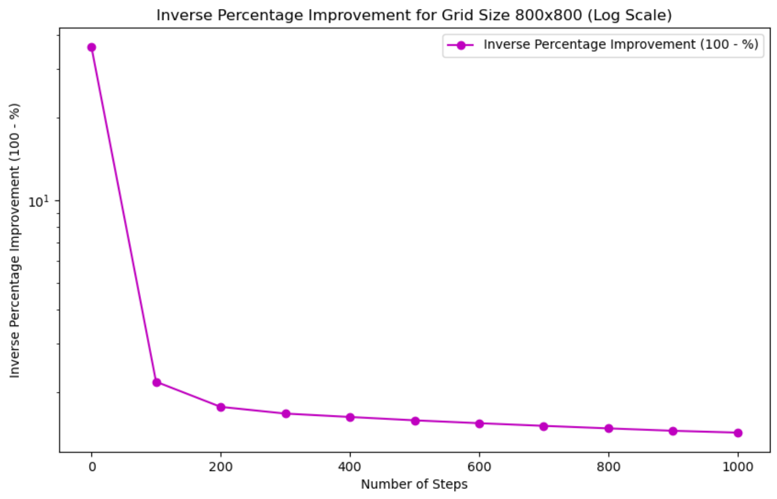

Figure 9: Inverse Percentage Improvement for Grid Size 800x800 (Log Scale)

able to do this project, as well as for the lab computers that we were able to use to run our tests. We would like to thank, in particular, out teacher Gleb Pogudin for his dedication to teach us and his feedback during the implementation of the project.