Evaluating High-Level Abstraction Frameworks for the Heat Equation

Mehmed Mulalić
University of Vienna

Supervised by:
ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

February, 2026

1 Introduction

The increased popularity of GPU computing has brought about a more efficient computing paradigm in the domain of scientific computing, enabling simulations that were previously infeasible on CPUs alone. Modern GPUs offer massive parallel processing capabilities, with thousands of cores capable of executing concurrent operations, naturally providing the ability to handle computationally intensive simulations. However, the use of such modern hardware requires extensive knowledge of parallel computing paradigms through the use of parallel computing platforms, like NVIDIA's CUDA (Compute Unified Device Architecture) or other frameworks like OpenCL (Open Computing Language), which presents a significant barrier for scientists and researchers.

To address this challenge, high-level programming frameworks have emerged that seek to abstract away from the complexities of GPU programming while still being able to compete in terms of performance. A multitude of frameworks are available across different languages, including Python libraries such as CuPy and Numba, Julia with GPU integration, and lower-level C++ libraries such as NVIDIA's Thrust. This study focuses on CuPy and Numba due to Python's popularity among researchers and its accessibility.

The most important factor to consider when discussing GPU abstraction is how well abstracted high-level code performs compared to original low-level code and what are the trade-offs between productivity and performance. Although low-level programming in CUDA provides the most control and maximum performance potential, it requires significant development time and expertise. High-level frameworks offer a less strained development stage, but their performance characteristics and usability in the context of scientific equations require investigation.

This work addresses this question by implementing and comparing multiple numerical solution methods for the 2D Heat Equation across different GPU programming frameworks. The Heat Equation is a great candidate mainly because it is solvable using multiple solution methods, but also because it represents a practical scientific scenario with thorough research and documentation. Three approaches are evaluated through high-level frameworks: 5-point stencil operations, sparse Laplace matrix computations, and spectral methods. By analyzing performance benchmarks and code complexity, this study quantifies the trade-offs between abstraction level and computational efficiency, providing practical insight for deciding the appropriate tools for GPU-accelerated scientific computing.

2 Background

Previously standard methods of solving scientific programs did not use heterogeneous hardware to accelerate program execution, causing multiple issues during development in means of execution speed and development difficulty to achieve reasonable optimizations. Such issues can be mitigated through the use of GPU acceleration, and to display the effectiveness of GPU programming through abstraction frameworks, the Heat Equation is used. The Heat Equation, as compared to other equations such as the Direct Coulomb Summation (DCS) which is primarily solved using a stencil operation, is a great candidate because it offers multiple solution methods as well as providing a practical scientific scenario, providing the ability to showcase how different API calls provide different results both in performance and in output accuracy.

2.1 The Heat Equation

The Heat Equation is a partial differential equation that describes the evolution of temperature distribution within a given spatial domain. It models how heat diffuses from regions of higher temperature to regions of lower temperature over time due to thermal conduction while maintaining the law of conservation of energy. The physical, continuous, formula in 2D space for the homogeneous heat equation is given by

$$\frac{\partial u}{\partial t} = \alpha \cdot \Delta u; \Delta u = (\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}),$$

where α is the thermal diffusivity (measured in m^2/s) which characterizes how quickly heat spreads through a material and Δu represents the Laplacian operator describing the spatial diffusion of heat [1]. There are multiple methods to discretize this equation, of which, two are the focus of this paper. The first discretization method is the naive five-point stencil operation described by the formula

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha \cdot \Delta t(\frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} - 2u_{i,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} + u_{i,j-1}^{n} - 2u_{i,j}^{n}}{\Delta y^2}),$$

where the Δu Laplacian operator is replaced by the stencil operation in which the surrounding matrix values are summed and multiplied with Δt time step and finally summed with the previous time step to calculate the value at position (i, j) for time step $t + 1$. This solution method involves using the five-point stencil operation for spatial discretization in combination with the Explicit Euler function for temporal discretization. In addition to the Explicit Euler function, there exists the Implicit Euler (Backward Euler) time stepping function which offers better optimization through API calls compared to the former method. The two functions are respectively defined with the following formulas

$$y_{i+1} = y_i + h \cdot f(t_i, y_i); \qquad y_{i+1} = y_i + h \cdot f(t_{n+1}, y_{n+1}),$$

where h is the step size (Δt) and the function f is the stencil operation [2] [3]. Lastly, the second discretization method is achieved through spectral analysis, where the temperature field is expressed using a set of basis functions, such as Fourier modes. Instead of approximating spatial derivatives directly, the heat equation is transformed into the frequency domain through Discrete Cosine Transform (DCT). After applying the Fourier transform, the spectral output can be defined by

$$\hat{u}_k^{n+1} = \hat{u}_k^n \cdot e^{\alpha \lambda_k \Delta t},$$

where λ determines how fast the spectral component decays and $\hat{u}$ is the spectral output after DCT [4]. After utilizing the inverse DCT (iDCT), the output is transformed from the spectral space back to the spatial domain to obtain the final result at time step n.

2.2 Boundary Conditions

When discussing PDEs, there exist boundary conditions which define how the solution behaves on the boundary of the spatial domain. Two most common conditions are the Dirichlet and the Neumann boundary conditions. The Dirichlet boundary condition specify the value of the solution on the boundary

and can be described by $y(x) = f(x) \; \forall \; x \; \epsilon \; \partial\Omega$ where f is a known function defined on the boundary $\partial\Omega$ [5]. The Neumann boundary condition specify the derivative of the solution on the boundary, described by $\frac{\partial u}{\partial n} = f(x) \; \forall \; x \; \epsilon \; \partial\Omega$ where n denotes the normal to the boundary $\partial\Omega$ [6]. For this experiment, the Neumann condition was strictly used to define $\frac{\partial u}{\partial n} = 0$, meaning that there is no heat transfer at the boundaries, imitating a perfectly insulated domain.

2.3 GPU Programming Frameworks

CUDA provides a multitude of API calls each tailored for specific usage and are highly optimized to run as efficient as possible on NVIDIA GPUs. Out of the various API possibilities, three API calls are widely used: cuBLAS (Basic Linear Algebra Subprograms), an API highly accustomed to solving basic algebraic operations), cuSPARSE, an API for solving large sparse matrices efficiently, and cuFFT (Fast Fourier Transforms), an API tailored to solving Fourier transforms efficiently.

In order to take advantage of such highly optimized APIs, frameworks must be utilized. The standard method for developing heterogeneous programs is mainly through the C/C++ programming language in combination with parallel frameworks such as CUDA or OpenCL. Such frameworks are considered low-level and, although they offer highest optimization possibility, they require the developer to be well-versed in the language and consider every aspect during development, such as memory management, optimal thread blocks, memory transfer, etc. Despite the standard development method, other frameworks exist, in various programming languages, which strive to abstract away most difficulties of heterogeneous programming. The main language for this project is Python which provides several heterogeneous abstraction frameworks such as CuPy and Numba.

CuPy is an open-source library with NumPy syntax designed for GPU-accelerated computing using the Python programming language [7]. The main focus of this library is to create a set of tools for the developer to create code which will run on the GPU while maintaining simplicity by using syntax similar to NumPy. CuPy's operations offer quick development by reducing boilerplate coding required for heterogeneous programming while retaining simple code; in most cases, CuPy can be used as a drop-in replacement with NumPy code [7]. Numba is an open-source JIT (Just-In-Time) compiler designed to translate Python functions into optimized machine code [11]. Numba includes both CPU and GPU parallelization, declared through keywords. Numba also reduced the amount of boilerplate code and it also provides abstraction but on a lower level. Development with Numba allows more control over GPU kernels while maintaining a high enough abstraction, providing easier development as well as high control. Although this framework provides a lower abstraction level, it provides exceptional performance, making it ideal for scenarios which require optimized custom kernels. While this study focuses on NVIDIA hardware via CUDA, the frameworks investigated also offer varying levels of AMD ROCm support. CuPy provides a separate ROCm build which does not natively support all functionalities, such as an API equivalent to NVIDIA's cuSPARSE. Numba's ROCm support is currently experimental and is not a good candidate for this study. Taking these limitations into consideration, despite hardware availability, the experiments were restricted to run exclusively on NVIDIA hardware due to limited AMD support.

These frameworks achieve their abstraction level by the way how the higher-level framework code is compiled to low-level GPU code. The frameworks append additional steps to the usual translation procedure from platform-specific code (CUDA/ROCm/oneAPI) to GPU code by including a conversion from Python code to LLVM code (an assembly language used for developing a frontend for any programming language [10]). CuPy and Numba both utilize JIT compilation where the code is dynamically compiled because a bridge is required to connect Python's dynamic nature with the platform's static requirements. Despite the common JIT compilation, the two slightly differ in the way they implement the aforementioned additional steps, namely that Numba exclusively uses JIT compilation compared to CuPy where it can use JIT explicitly or implicitly. Implicit JIT compilation implies that there are already existing templates in CuPy which do not require compilation but are instead called upon, thus effectively skipping JIT compilation while explicit JIT compilation requires the kernel to be compiled first before use, creating a one-time overhead. CuPy's translation procedure executes JIT compilation for any custom kernels by utilizing the GPU's compiler (NVCC for NVIDIA, HIPCC for AMD, and DPC++ for Intel) to translate the framework's code to platform-specific code. Numba, on the other hand, handles this differently, where, instead of translating to platform-specific code, it utilizes its own IR (intermediate representation) to translate its code directly to platform-specific code, without using any GPU compiler.

3 Methodology

The Heat Equation was solved using three distinct numerical approaches, implemented across frameworks of varying abstraction levels. Each method was implemented across different frameworks to provide a comprehensive performance comparison.

3.1 Stability Analysis

When discretizing a continuous PDE, stability analysis is essential because numerical errors, such as truncation or round-offs, are introduced at every iteration inside the discretized space. Stability analysis ensures that errors do not grow unbounded as the computation proceeds. In numerical analysis, and in the case of the Heat Equation, the Von Neumann stability analysis can be implemented to determine the stability of numerical schemes [8].

For the 1D Heat Equation discretized with the Forward-Time Central-Space (FTCS) scheme which corresponds to explicit Euler in time, the discretized formula becomes

$$u_j^{n+1} = u_j^n + r(u_{j+1}^n - 2u_j^n + u_{j-1}^n),$$

where $r = \frac{\alpha \Delta t}{(\Delta x)^2}$ describes the ratio of time step to diffusion time [13]. The Von Neumann stability analysis shows that the amplification factor must satisfy $|G| \leq 1$ for the error to be bounded, leading to the condition $r \leq \frac{1}{2}$ in 1D and $r \leq \frac{1}{4}$ in 2D [8]. Thus, the stability condition for the 1D Heat Equation becomes $r = \frac{\alpha \Delta t}{(\Delta x)^2} \leq \frac{1}{2}$. For the 2D Heat Equation, for equal spacing $\Delta x = \Delta y$ using the explicit Euler method, the stability condition becomes $r = \frac{\alpha \Delta t}{(\Delta x)^2} \leq \frac{1}{4}$. Although the FTCS is conditionally stable, the Implicit Euler (Backwards Euler) as well as the spectral method are unconditionally stable, meaning that they are not conditioned on the time step value to be stable. In order to satisfy the stability condition for the FTCS scheme, the parameters $\alpha = 1.6563 \times 10^{-4} \frac{m^2}{s}$ and $\Delta t = 5 \times 10^{-3} s$ were chosen for all grid sizes in order to provide an equal comparison.

3.2 Implementation Strategy

All three numerical methods share a common code structure: an initialization phase that constructs the computational grid and applies the initial condition, a main computational loop that advances the solution, and a timing loop that executes the loop five times to obtain accurate timing measurements.

Due to the high level of abstraction provided by Cupy, sequential and GPU-accelerated implementations differ only in the imported library, effectively differentiating only between syntaxes for the imported library — np for NumPy and cp for CuPy. Memory allocation, host-to-device and device-to-host memory transfer, device synchronization, and memory deallocation is handled implicitly by the respective framework, eliminating the need for explicit code management. Numba differs substantially only in the kernel since it allows more control, there is multiple steps required to achieve the same procedure as with prior frameworks.

3.2.1 Method 1: Five-Point Stencil

The five-point stencil method begins by computing the 2D Laplacian matrix at the beginning of the computational loop by applying a convolution kernel to the temperature matrix. This kernel is a 3×3 matrix which describes the stencil operation by defining the central and neighboring values. The convolution is performed using SciPy's ndimage.convolve method with reflect mode which automatically implements the Neumann zero-flux boundary condition at domain edges. The resulting matrix is multiplied by $\alpha \Delta t$ and added to the current temperature matrix to produce the next time step through the Explicit Euler scheme. The CuPy implementation is identical except that it implements the CuPy library instead of the SciPy library wherever possible. The Numba implementation, by contrast, defines a custom GPU kernel where each thread is responsible for updating a single grid point.

3.2.2 Method 2: Sparse Laplacian Matrix

The sparse Laplacian matrix method constructs the 2D discrete Laplacian operator L as a sparse matrix in Compressed Sparse Row (CSR) format before the computational loop using Kronecker products. At each time step, the computation $u^n + \alpha\Delta t \cdot Lu^n$ is applied with a sparse matrix-vector product and is added to the flattened temperature matrix. This formulation uses Explicit Euler time-stepping rather than Implicit Euler, as the latter would require solving a linear system at every step. The sequential implementation uses SciPy's sparse module, while the CuPy implementation transfers L to GPU memory in CSR format and uses CuPy's sparse matrix-vector product, which dispatches internally to NVIDIA's cuSPARSE library.

3.2.3 Method 3: Spectral Method using Discrete Cosine Transform

For a domain with the Neumann zero-flux boundary condition, the DCT naturally handles the insulated boundaries while maintaining high efficiency, thus making the implementation simpler. The eigenvalues for pure Neumann boundary conditions in discretized 1D space is defined by $\lambda_i = -\frac{4}{h^2}\sin^2(\frac{\pi(i-1)}{2n})$ [14]. This value can be transformed into 2D space with cosines through the trigonometric identity $\sin^2(\theta/2) = \frac{(1-\cos\theta)}{2}$ which results in

$$\lambda_k = -\frac{4}{h^2}\left(\sin^2(\frac{\pi(i-1)}{2n}) + \sin^2(\frac{\pi(j-1)}{2n})\right) = -\frac{4}{h^2}(\frac{1-\cos(k_x)}{2} + \frac{1-\cos(k_y)}{2}),$$

$$\lambda_k = \frac{2}{h^2}(\cos k_x + \cos k_y - 2); k_x = \frac{\pi(i-1)}{n}, k_y = \frac{\pi(j-1)}{n}, h^2 = (\Delta x)^2.$$

The algorithm proceeds by first applying the DCT to the initial matrix and for each time step, multiplying the transformed matrix with the evolution operator — an object which progresses the system forwards in time — $e^{\alpha\lambda_k\Delta t}$. After the computational loop, the iDCT is applied to transform the matrix back to the spatial domain, completing the algorithm. The sequential implementation uses SciPy's fft package while the GPU-accelerated version implements CuPy's equivalent of SciPy's fft package which utilizes NVIDIA's cuFFT library – a highly optimized Fast Fourier Transform API.

3.3 Validation Approach

The output of each GPU-accelerated implementation was verified by comparing its output against the corresponding sequential implementation. Since discretized numerical methods utilize floating numbers, no two computations can offer exact results. Therefore, validation utilized NumPy's isclose function with an absolute tolerance of $\epsilon = 1 \times 10^{-6}$ which returns a Boolean array indicating whether the elements of two matrices are within the specified tolerance. Each numerical method and each grid size was compared and was determined valid if all elements of the isclose function passed the validity check.

4 Experimental Setup

4.1 Hardware and Software Environment

There are two university clusters available for experimentation: the UCS9 cluster, utilizing an NVIDIA GPU, and the EXA cluster which utilizes an AMD GPU. Despite hardware accessibility mentioned in subsection 2.3, all experiments were conducted on the UCS9 cluster, running on the Ubuntu 22.04 LTS (Jammy Jellyfish) operating system. The system contains an AMD EPYC 7543 processor with a max clock speed of 3.74 GHz, containing 2 threads per core, 32 cores per socket, and 2 sockets, totaling 128 logical units. Alongside the processor, the system additionally contains an NVIDIA Tesla A100 GPU, running on the Ampere architecture, with 80GB of GDDR6 VRAM, a clock speed of 1410MHz, and 6912 CUDA cores. Lastly, the system is equipped with 503GiB of RAM. All frameworks and libraries are installed locally, with Table 4.1 summarizing the complete hardware and software configuration.

Component	**Specification**
Processor	AMD EPYC 7543
System RAM	503 GiB
GPU	NVIDIA TESLA A100
OS	Ubuntu 22.04 LTS (Jammy Jellyfish)
CUDA	12.8
Python	3.12
NumPy	2.3
SciPy	1.16
CuPy	13.6
Numba	0.62

Table 4.1: Hardware and software configuration used for all experiments.

4.2 Problem Parameters and Domain Configuration

The Heat Equation is solved on a square domain $[0,1]$x$[0,1]m^2$ with uniform grid spacing. Five grid sizes are tested — $N \in \{256, 512, 1024, 1536, 2048\}$ — to display how GPU computation changes gradually from less intensive to intensive sizes, thus capturing the full behavior of each framework.

The time step $\Delta t = 5 \times 10^{-3} s$ was chosen to satisfy the Von Neumann stability condition described in subsection 3.1. At the largest grid size $[2048, 2048]$, the diffusion number evaluates to $r = \frac{\alpha \Delta t}{(\Delta x)^2} \approx 0.208$, which remains within the stability bound $r \leq 0.25$. The simulation advances for $N_t = 12,000$ time steps, covering a total simulated time of 60 seconds. The thermal diffusivity $\alpha = 1.6563 \times 10^{-4} \frac{m^2}{s}$ is a representative value for 99.9% pure silver [12]. The initial Gaussian temperature distribution is defined as [9]

$$u(x, y, t) = Ae^{-((x-x_0)^2+(y-y_0)^2) \div (2\sigma^2)}.$$

When passing in previously defined arguments, the initial condition becomes

$$u(x, y, 0) = 100 \cdot e^{-((X-0.5)^2+(Y-0.5)^2) \div (2 \cdot 0.05^2)}.$$

This represents a concentrated heat source with 100°C as the peak temperature with a standard deviation of $\sigma = 0.05m$, producing a sharp Gaussian bump that decays steeply away from the center. This initial condition is kept constant across all experiments so that the final outcome remains identical regardless of grid size or framework, resulting in a fair performance comparison. Zero-flux Neumann boundary conditions are applied at all domain boundaries, modeling a perfectly thermally insulated domain in which no heat escapes. Table 4.2 summarizes all parameters used.

Parameter	Value
Grid sizes (NxN)	256, 512, 1024, 1536, 2048
Grid spacing (h)	$1/(N-1)$
Time step (Δt)	$5 \times 10^{-3} s$
Final time (t_f)	$60s$
Time steps (N_t)	$12,000$
Thermal diffusivity (α)	$1.6563 \times 10^{-4} \frac{m^2}{s}$
Gaussian amplitude (A)	100°C
Gaussian width (σ)	$0.05m$
Boundary condition	Neumann ($\frac{\partial u}{\partial n} = 0$)
Timing runs	5 (one warm-up run)

Table 4.2: List of parameters used across all numerical methods and framework implementations.

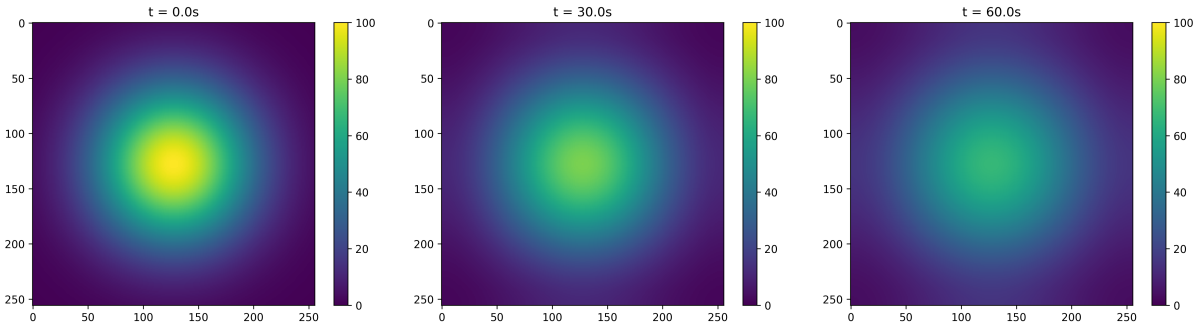
4.3 Measurement Methodology

The execution time is defined as the wall-clock duration of the computational loop. One-time preprocessings steps, such as calculating the evolution operator or the sparse matrix, are excluded from the timing measurement since they are calculated only once and are not fully part of the computational loop. Each experiment is executed five times consecutively with an additional warm-up execution for JIT compilation frameworks. To provide a more accurate reading, the reported execution time represents the mean time from the five executions excluding the warm-up execution. Timing is measured using Python's time.perf_counter() method which provides a counter to accurately measure the duration of the code. The execution time includes only the computational loop, meaning that there are no additional measurements to evaluate GPU specific timings.

5 Results

5.1 Temperature Distribution

Figure 5.1 illustrates how temperature progresses every 15 seconds, as computed by the sequential stencil method with the Explicit Euler scheme at a grid size of $[256, 256]$. The initial Gaussian bump centered at $(0.5, 0.5)$ with an amplitude of 100°C decays and spreads symmetrically under the zero-flux Neumann boundary condition.



(a) t = 0 s (b) t = 30 s (c) t = 60 s

Figure 5.1: Temperature distribution using the sequential stencil method

5.2 Execution Time Comparison

Table 5.1 through 5.3 report the mean execution times across five runs for each method, grid size, and framework. The sequential baseline reports were calculated based on a single execution due to impractically long execution times. Additionally, any computation exceeding five minutes was terminated and marked with a dash. The speedup is computed as the ratio of sequential to parallel execution time and is omitted where the sequential baseline was not available.

5.2.1 Five-Point Stencil Method

Table 5.1 presents execution times for the five-point stencil method. The sequential baseline becomes impractical after $N = 512$, timing out after $N = 1024$ and above, while both Numba and CuPy complete all grid sizes successfully. Notably, Numba outperforms CuPy consistently, suggesting that Numba's custom kernel provides an advantage for this stencil operation. Additionally, CuPy's results were quite similar for the first three test sizes $N \in \{256, 512, 1024\}$, suggesting that the overhead influenced greatly for smaller grid sizes.

Grid Size	Sequential	Numba	CuPy	Numba Speedup	CuPy Speedup
256	81.663	6.417	13.11	12.726	6.229
512	320.375	6.707	13.141	47.767	24.38
1024	-	10.355	13.259	-	-
1536	-	20.874	22.991	-	-
2048	-	37.676	38.419	-	-

Table 5.1: Average stencil execution times (seconds).

5.2.2 Sparse Laplacian Matrix Method

Table 5.2 presents execution times for the sparse Laplacian matrix method. This method does not include a Numba implementation because this method's sparse matrix-vector computation does not fit

well into Numba's custom-kernel pattern since this computation is heavily reliant on API. The sequential method times out beyond $N = 512$. Interestingly, while the GPU-accelerated results are all under one minute,the results for grid size $N = 2048$ proved to be challenging even on the A100 GPU, exhibiting poor performance, averaging an execution time at almost 1.5 minutes.

Grid Size	Sequential	CuPy	Speedup
256	44.167	15.359	2.876
512	172.228	15.203	11.329
1024	-	22.629	-
1536	-	53	-
2048	-	89.485	-

Table 5.2: Average Laplace execution times (seconds).

5.2.3 Spectral DCT Method

Table 5.3 presents execution times for the spectral DCT method. This method is the most GPU-friendly of the three — CuPy completed all grid sizes, including $N = 2048$, while the sequential baseline timed out only at the largest grid size $N = 2048$. The speedup grows substantially with grid size, reaching $29.341\times$ at $N = 1024$.

Grid Size	Sequential	CuPy	Speedup
256	4.933	1.756	2.809
512	18.302	1.701	10.76
1024	56.804	1.936	29.341
1536	205.68	7.784	26.423
2048	-	14.195	-

Table 5.3: Average spectral execution times (seconds).

5.3 Profiling Results

NVIDIA Nsight Systems profiling was conducted for each GPU-accelerated implementation at a grid size of $[256, 256]$ to analyze behavior of GPU execution for each numerical method. For the CuPy Stencil, the computational loop creates four distinct kernel launches per time step: the convolution kernel accounting for 37% total GPU time, followed by element-wise addition at 23%, scalar multiplication at 21%, and device-to-device copy at 18%. This device-to-device kernel provides a small overhead required for calculating the convolution; due to how CuPy handles copying data after the calculation, there is an additional step of copying the temporary array, causing the kernel launch. Memory operation summary displays that almost all memory transfer is device-to-host (99%) with a total amount of 524MB, suggesting that most operations are done inside GPU memory. The Numba stencil profile is noticeably simple as its summary displays a single kernel launch, accounting for 100% of GPU time with an average call time of 6.052μs per launch. This is the main advantage of Numba where it executes a single kernel without any additional separate kernel launches with it being the main reason why it outperforms CuPy since there are four CuPy kernel launches compared to only one Numba launch. Memory operation summary displays that there is an equal split of host-to-device and device-to-host memory transfer with a total amount of $1,049$MB of host-to-device transfer and 524MB of device-to-host transfer. The CuPy Laplace profile reveals a significant source of overhead: cuSPARSE's internal sparse matrix-vector product accounts for 33% of GPU time while an additional partitioning kernel, required by the CSR format, uses up an additional 26% GPU time. Interestingly, there are $200,272$ CUDA memset operations totaling to approximately 100GB of memory zeroing. These repetitions represent a significant performance cost, explaining why this method underperforms compared to other numerical methods. The CuPy FFT profile is the simplest of all — the kernel is a single element-wise multiplication executed once per time step, averaging 2.615μs per launch across $400,000$ invocations. The DCT and iDCT appear as only 8 kernel instances, confirming that the transformation is applied as expected — once before and once after the computational loop, with there being one discard kernel launch, invoking 2 DCT calls and 2 iDCT calls in 2D space, totaling 8 1D kernel instances. This design explains the spectral method's superior GPU performance.

6 Conclusion

This study evaluated the performance of high-level GPU abstraction frameworks for solving the 2D Heat Equation using three numerical approaches: five-point stencil operation, sparse Laplacian matrix multiplication, and spectral methods. The primary objective was to quantify the trade-offs between abstraction level, development effort, and computational performance when compared to a sequential NumPy/SciPy baseline.

The results demonstrate that high-level frameworks such as CuPy and Numba deliver substantial GPU acceleration relative to sequential execution across all tested grid sizes and numerical methods. In addition to GPU acceleration, these frameworks provide easier development through abstraction while retaining performance levels which are expected to approach those of equivalent CUDA/ROCm implementations due to the frameworks' reliance on optimized backend libraries, although a direct empirical comparison remains outside the scope of this work.

The findings of this work have practical implications for scientists and researchers seeking to leverage GPU acceleration without requiring extensive expertise in low-level CUDA programming. For problems whose computations align with well-supported library operations, high-level frameworks like CuPy offer a compelling combination of high performance and minimal development effort. For problems which require custom kernels and for computations where there are no well-defined libraries, Numba enables higher control over GPU execution with similar performance at the expense of increased complexity relative to CuPy.

6.1 Future Work

Several research directions were outside the scope of this study and, although seemingly a natural fit for this study, they were excluded. First, the study was restricted to NVIDIA's CUDA platform despite the availability of AMD hardware on the EXA cluster. Second, an AMD ROCm implementation was initially planned but proved infeasible due to two major limitations: CuPy's ROCm build was not available in the required version for the EXA cluster in addition to limited functionalities, such as missing APIs, and AMD implementation is still experimental for Numba. It would be prudent to conduct the study once again with AMD implementation once the limitations have been rectified. Additionally, two frameworks — Julia with GPU integration and NVIDIA's Thrust C++ library — were initially considered as testing candidates for the experiment but due to time constraints, their implementations were not included. Further implementations which include said frameworks would provide a better comparison between multiple frameworks, strengthening the claim between abstraction level and performance, thus allowing a more comprehensive study. Lastly, although Backward Euler time-stepping scheme was discussed, it was not implemented due to time constraints, as such, all time integration schemes in this study used Explicit Euler which can be further improved in the future.

References

[1] Wikipedia contributors, *Heat equation*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Heat_equation` (Accessed: 11.11.2025).

[2] Wikipedia contributors, *Euler method*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Euler_method` (Accessed: 07.02.2026).

[3] Wikipedia contributors, *Backwards Euler method*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Backward_Euler_method` (Accessed: 07.02.2026).

[4] L. N. Trefethen, *Spectral Methods in MATLAB,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[5] Wikipedia contributors, *Dirichlet boundary condition*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Dirichlet_boundary_condition` (Accessed: 05.02.2026).

[6] Wikipedia contributors, *Neumann boundary condition*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Neumann_boundary_condition` (Accessed: 05.02.2026).

[7] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, *CuPy: A NumPy-compatible library for NVIDIA GPU calculations,* in Proceedings of Workshop on Machine Learning Systems (LearningSys), Neural Information Processing Systems (NIPS), 2017.

[8] Wikipedia contributors, *Von Neumann stability analysis*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Von_Neumann_stability_analysis` (Accessed: 13.02.2026).

[9] Wikipedia contributors, *Gaussian function*, Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Gaussian_function#Higher-order_Gaussian_or_super-Gaussian_function_or_generalized_Gaussian_function` (Accessed: 07.02.2026).

[10] Wikipedia contributors, *LLVM* Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/LLVM` (Accessed: 13.02.2026).

[11] S. K. Lam, A. Pitrou, and S. Seibert, *Numba: A LLVM-based Python JIT Compiler,* in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015.

[12] Wikipedia contributors, *Thermal diffusivity* Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Thermal_diffusivity` (Accessed: 14.02.2026).

[13] Wikipedia contributors, *FTCS scheme* Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/FTCS_scheme` (Accessed: 14.02.2026).

[14] Wikipedia contributors, *Eigenvalues and eigenvectors of the second derivative* Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors_of_the_second_derivative` (Accessed: 15.02.2026).