CHALMERS
UNIVERSITY OF TECHNOLOGY

GPU-Accelerated FEM Solver: From Heat Equation to Black–Scholes

Project Report

Authors: Brijesh Niranjan, Daniel Fathi, Jan Kula, Jinlun Gao, Simon Rödén

Supervisor: Fredrick Larsson

FEM-1
Chalmers University of Technology

January 30, 2026

Contents

1 Abstract

Solving Finite Element Method problems is sometimes time-consuming depending on the case study. To accelerate computation time, engineers use various CPUs and GPUs to reduce the time taken to solve complex engineering problems. In this project, we use different solvers in Python and CUDA to solve the 2D heat equation. The total computation time is compared across different mesh sizes, with several interesting results observed. In addition, the heat equation formulation is related to the Black–Scholes equation, providing a financial application of the same numerical framework.

2 Introduction

This report investigates performance acceleration achieved by using Graphics Processing Units (GPUs) compared to Central Processing Units (CPUs) when solving systems of linear equations via the Finite Element Method (FEM). Specifically, we utilize the two-dimensional heat equation with an application in financial option pricing.

We evaluate performance for several GPU architectures and solvers, and also compare floating-point precision (float32 vs. float64). Finally, we measure assembly time on CPUs and GPUs.

3 Theoretical Background

The Finite Element Method (FEM) is a numerical technique used to solve complex engineering problems by dividing a system into smaller subproblems called elements. The Finite Element Analysis (FEA) process approximates real-world behavior—forces, stresses, pressure, temperature—by solving these local interactions.

The major steps in FEM are:

- **Discretization:** The geometry is divided into small elements (triangular, quadrilateral, tetrahedral, etc.). The intersection points are called nodes.
- **Applying Equations:** Governing equations (e.g., heat equation) and boundary conditions are applied to each element.
- **Assembly:** Local element matrices are combined into a global stiffness matrix.
- **Solving:** Numerical solvers compute unknown quantities (temperature, displacement, etc.).
- **Post-Processing:** Node results are used to compute stresses, strains, heat flux, etc.

3.1 Heat equation 2D

The 2 dimensional heat equation is given on the form of:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \kappa \Delta u = f & \text{in } \Omega \times (0, T] \\ u = 0 & \text{on } \partial\Omega \times (0, T] \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) & \text{in } \Omega \end{cases} \tag{1}$$

where κ is the heat conductivity and f is a function with initial conditions u_0. In this problem we are using a Dirichlet boundary condition which forces the heat to be 0 on the boundary. t is the time.

In FEM it is said that (1) is the "strong-form" solution of the problem and it can be rewritten into its "weak form" given by (2)

$$\int_\Omega \frac{\partial u}{\partial t} v \, d\mathbf{x} + \int_\Omega \kappa \nabla u \cdot \nabla v \, d\mathbf{x} = \int_\Omega f v \, d\mathbf{x}. \tag{2}$$

This in turn can be represented using matrices and vectors, and doing so one gets the following system

$$M\dot{u} + Ku = Mf, \tag{3}$$

were M is the mass matrix, K is the stiffness matrix and f is the load vector.

We are using the following approximating the derivative of $(\dot{u})$

$$\dot{u} \approx \frac{u^{n+1} - u^n}{\Delta t}. \tag{4}$$

Which gives us the following system

$$M\big(\frac{u^{n+1} - u^n}{\Delta t}\big) + Ku^{n+1} = Mf^n, \tag{5}$$

which can be further simplified and rewritten as so

$$(M + \Delta tK)u^{n+1} = Mu^n + \Delta tMf^n, \tag{6}$$

calling the left-hand side A and the right-hand side for b for simplification as such

$$\underbrace{(M + \Delta tK)}_{A} u^{n+1} = \underbrace{(Mu^n + \Delta tMf^n)}_{b}. \tag{7}$$

$$Au^{n+1} = b \tag{8}$$

Since we are using the explicit Euler scheme (also known as the forward Euler method) we are able to compute the system at time step $n + 1$ by using the information in the previous time-step n. To illustrate the flow when solving this on the computer, we have written how this would look like inside the time-loop inside the function;

$$Au^{n+1} = b \tag{9}$$

$$A^{-1}Au^{n+1} = A^{-1}b \tag{10}$$

$$u^{n+1} = A^{-1}b \tag{11}$$

$$u^n \leftarrow u^{n+1} \tag{12}$$

were the calculation $A^{-1}b$ in (11) is done by a solver. The last step (12) is just the updating of the system where the system at the next time step u^{n+1} becomes the system at the previous time-step u^n and we continue with the next in line time-step. up until $t = T$.

3.2 Matrix properties

- The mass matrix M is always symmetric and Positive Definite.
- The stiffness matrix K is symmetric and positive semi-definite.
- The stiffness matrix K becomes Positive Definite once Dirichlet boundary conditions are applied.
- The sum of a positive Definite and a Semi-Definite matrix (A) is a positive definite matrix.

3.3 Dirichlet Boundary Conditions: The Penalty Method

To enforce Dirichlet boundary conditions ($u_i = u_{BC}$) within the linear system $\mathbf{Au} = \mathbf{b}$ while preserving the symmetry and sparsity pattern of the matrix $\mathbf{A}$ (essential for solvers like Conjugate Gradient), the *Penalty Method* is employed.

This method modifies the diagonal elements of the stiffness matrix and the corresponding entries in the load vector for all boundary nodes $i \in \partial\Omega_D$. A large penalty parameter $\alpha \gg \max(\mathbf{A}_{ij})$ is added to the diagonal entry $\mathbf{A}_{ii}$:

$$(\mathbf{A}_{ii} + \alpha)u_i + \sum_{j \neq i} \mathbf{A}_{ij}u_j = \mathbf{b}_i + \alpha u_{BC} \tag{13}$$

Dividing by α and taking the limit as $\alpha \to \infty$, the terms not involving α vanish:

$$\lim_{\alpha\to\infty} \left(\left(\frac{\mathbf{A}_{ii}}{\alpha} + 1 \right) u_i + \sum_{j \neq i} \frac{\mathbf{A}_{ij}}{\alpha} u_j \right) = \lim_{\alpha\to\infty} \left(\frac{\mathbf{b}_i}{\alpha} + u_{BC} \right) = u_{BC}. \tag{14}$$

$$1 \cdot u_i = u_{BC} \tag{15}$$

This calculation was applied to the matrix A to enforce Dirichlet boundary conditions; we denote this modified system of linear equations as $\tilde{A}$.

This effectively forces the solution at node i to match the boundary value u_{BC} to a precision determined by the magnitude of α. For homogeneous boundary conditions ($u_{BC} = 0$), the modification simplifies to adding α to the diagonal.

While the penalty method effectively enforces $u_i \approx 0$ for boundary nodes, it introduces a significant disparity in the magnitude of the diagonal entries. This leads to a high condition number $\kappa(\tilde{A})$. The performance of iterative solvers such as GMRES and Conjugate Gradient degrades as $\kappa(\tilde{A})$ increases.

3.4 Jacobi Preconditioning and the Penalty Method

Some of the solvers used in this comparison required preconditioning. The Jacobi preconditioner, also known as the diagonal preconditioner, was chosen for this purpose.

The linear system resulting from the Finite Element discretization is given by:

$$\tilde{A}\mathbf{u} = \mathbf{b} \tag{16}$$

where $\tilde{A}$ is the system with Dirichlet boundary conditions enforced using the penalty method.

To mitigate the disparity in the magnitude of the diagonal entries in $\tilde{A}$ introduced by the penalty method, we utilize a Jacobi (diagonal) preconditioner, P. The preconditioner is constructed from the diagonal of the modified system matrix:

$$P = \text{diag}(\tilde{A}) \tag{17}$$

The iterative solver then solves the preconditioned system $P^{-1}\tilde{A}\mathbf{u} = P^{-1}\mathbf{b}$. For a boundary node $i \in \partial\Omega_D$, the diagonal entry of the preconditioned matrix becomes:

$$(P^{-1}\tilde{A})_{ii} = \frac{\tilde{A}_{ii}}{P_{ii}} = \frac{A_{ii} + \alpha}{A_{ii} + \alpha} = 1 \tag{18}$$

For interior nodes, the diagonal entries are similarly normalized to 1. This rescaling effectively "smooths" the spectral distribution of the matrix, ensuring that the large penalty values α do not dominate the Krylov subspace generation, thereby recovering the convergence rate of the solver.

3.5 Black–Scholes 2D

For a multi-asset option, the underlying assets are typically modelled as Brownian motions. In the case of a portfolio with two option depending on two assets, the pricing problem naturally leads to a partial differential equation. We consider two underlying assets $S_1(t)$ and $S_2(t)$ following uncorrelated geometric Brownian motions under the risk-neutral measure

$$\begin{aligned} dS_1 &= rS_1\,dt + \sigma_1 S_1\,dW_1, \\ dS_2 &= rS_2\,dt + \sigma_2 S_2\,dW_2 and \end{aligned} \tag{19}$$

$$\mathbb{E}[dW_1\,dW_2] = \rho\,dt \tag{20}$$

where r is the risk-free interest rate, σ is the volatility of the assets, and $\rho \in [-1, 1]$ is the correlation coefficient. However we will assume the assets to be uncorrelated and ρ to be zero. Let $V(S_1, S_2, t)$ denote the price of the option. The PDE is thereby

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S_1^2 \frac{\partial^2 V}{\partial S_1^2} + \frac{1}{2}\sigma^2 S_2^2 \frac{\partial^2 V}{\partial S_2^2} + rS_1 \frac{\partial V}{\partial S_1} + rS_2 \frac{\partial V}{\partial S_2} - rV = 0. \tag{21}$$

For a portfolio of call option the strike function is

$$V(S_1, S_2, T) = max(S_1 - K_1, 0) + max(S_2 - K_2, 0). \tag{22}$$

Following the approach described in [1], we introduce the following change of variables:

$$x = -\ln(S_1)\,\frac{2}{\sigma_1^2}, \qquad y = -\ln(S_2)\,\frac{2}{\sigma_2^2}, \tag{23}$$

and the scaled time-to-maturity

$$\tau = (T - t), \tag{24}$$

and introducing

$$\kappa_1 = -\frac{r}{\sigma_1^2}, \kappa_2 = -\frac{r}{\sigma_2^2}, \kappa_3 = r + \kappa_1^2 + \kappa_2^2. \tag{25}$$

We can then define $u(x, y, \tau)$ as

$$u(x, y, \tau) = e^{\kappa_1 x + \kappa_2 y + \kappa_3 \tau} v(x, y, \tau). \tag{26}$$

the transformed function $u(x, y, \tau)$ satisfies the correlated two-dimensional heat equation

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}. \tag{27}$$

At $\tau = 0$ (corresponding to $t = T$), the initial condition for u is obtained from the terminal payoff:

$$u(x, y, 0) = (max(S_1 - K_1, 0) + max(S_2 - K_2, 0))e^{(\kappa_1 x + \kappa_2 y)}. \tag{28}$$

3.6 Tolerance threshold

We use a tolerance of `1e-05`. A lower tolerance of `1e-06` would be difficult to reach using `float32`, and `1e-07` would be impossible. This is because the machine epsilon for `float32` is $2^{-23} \approx 1.192 \cdot 10^{-7}$. This corresponds to approximately $-\log_{10}(1.192 \cdot 10^{-7}) \approx 6.92$ digits of accuracy, making any lower bound unfeasible. The tolerance threshold was kept at `1e-05` for the `float64` version to ensure equal comparisons between the two parameters.

4 Problem Description

In this project we are going to do the following things to learn the how to program on the GPU and also to examine its strengths and weaknesses compared to programming on the CPU. More exactly the following things are going to be done:

- Examine different solvers to see how they behave on GPU vs CPU.
- Compare solve time of the different solvers between CPU and GPUs with different size of mesh.
- Compare the assembly time of the system between CPU and GPUs for different size mesh.
- Explore how different floating-point precision (in this case float32 and float64) affect runtime performance.

5 Method

5.1 Code implementation

First, the code was implemented on the CPU. The packages used for the implementation were `Scipy`, `NumPy`, and `Numba`. The implementation followed the theory that was presented in earlier chapters.

The code was divided into two main parts: the assembly of the matrix system and the solver. As the theory previously stated, for this project the assembly is done only once, while the solver is run for as many iterations as there are time steps. Therefore, the majority of the time in this project will be spent on the solver part.

Writing functions for the assembly of the system was done using `NumPy`. Vectorization was prioritized, while Python loops and unsupported dynamic features were avoided. The reason for this is that when we later applied the `Numba` just-in-time `@jit` decorator, these optimizations allowed the code to be compiled effectively for a fast running time.

The solver part of the program was implemented first for the CPU using three different solvers from `Scipy`'s library under `scipy.sparse.linalg`.

For benchmarking the system on the GPU, the code written for the CPU was copied into a new file. The assembly of the system was kept as it was previously (a CPU implementation using `NumPy` compiled with `Numba`), but the solver part was changed.

Here, the `cupyx` implementation was seamless and worked as a drop-in replacement for the CPU implementation. Instead of using `scipy.sparse.linalg`, the `cupy` implementation was used: `cupyx.scipy.sparse.linalg`. This made the switch from doing the solver computations on the CPU to the GPU easy, with little code change, and kept the code looking largely the same.

For the benchmarking of the assembly time on CPU vs GPU, we implemented an assembler that executed code on the GPU. The implementation was done using the `cuda` package from `Numba`. The implementation followed the same structure and logic as the CPU part, but here more attention had to be paid to configuring the thread blocks and grids to parallelize the Black-Scholes logic.

5.2 Solvers and Parameters

We utilized three distinct solvers to evaluate performance:

- Conjugate Gradient - `cg`: An iterative solver optimized for symmetric positive-definite matrices. It is highly memory-efficient and relies on fast matrix-vector products, making it ideal for benchmarking raw GPU throughput.
- Generalized Minimal Residual - `gmres`: A robust iterative solver for general (non-symmetric) systems. It serves as a test case for more complex iterative workloads.
- Linear Solver - `spsolve`: A direct solver that computes an exact solution. It is included to contrast the scalability of iterative GPU methods against a standard CPU-bound direct approach.

The following parameters were used for the solvers:

API-path	Solver	Parameters
`scipy.sparse.linalg`	`cg()`	`atol= 1e-5, rtol=1e-5, M=jacobi`
`scipy.sparse.linalg`	`gmres()`	`atol= 1e-5, rtol=1e-5, M=jacobi`
`scipy.sparse.linalg`	`spsolve()`	-
`cupyx.scipy.sparse.linalg`	`cg()`	`atol= 1e-5, rtol=1e-5, M=jacobi`
`cupyx.scipy.sparse.linalg`	`gmres()`	`atol= 1e-5, rtol=1e-5, M=jacobi`
`cupyx.scipy.sparse.linalg`	`spsolve()`	-

Table 1: Parameters used for the different solvers for both CPU and GPU implementation.

Main function name	Parameters
`solve_heat_equation()`	`T = 10, tol=1e-05, dt=0.05, kappa=1`

Table 2: Parameters used for running the function

5.3 Benchmarking - Time

Runtime performance was evaluated by measuring the assembly and solver phases separately on the CPU and GPU. CPU timings were recorded using `time.perf_counter()`, which provides high-resolution wall-clock timing. The assembly phase was divided into two parts: (i) generation of finite element method (FEM) triplets (row indices, column indices, and values for stiffness and mass contributions), and (ii) construction of global sparse matrices in compressed sparse row (CSR) format using SciPy. Since CSR construction involves sorting indices and summation of duplicate entries, this step was timed independently.

GPU timings were measured using CUDA events (`cp.cuda.Event`) in order to capture device execution time accurately. The GPU benchmark was split into three components: (i) host-to-device memory transfer and conversion of sparse matrices to CuPy CSR format, (ii) GPU setup, which includes system matrix construction, enforcement of boundary conditions via a diagonal penalty operator, Jacobi preconditioner setup, and initialization of the solution vector, and (iii) execution of the time-stepping solver loop. The final CUDA event was explicitly synchronized to ensure that all asynchronous GPU operations had completed before elapsed times were recorded.

In addition, `cProfile` was used to analyze Python-level overhead and identify function-level performance bottlenecks on the CPU. Since Python profilers do not capture asynchronous GPU kernel execution, all reported GPU solver runtimes are based exclusively on CUDA event measurements.

To ensure fair and reproducible measurements, all Numba-accelerated functions were warmed up prior to timing in order to exclude just-in-time compilation overhead. Furthermore, all GPU timings were collected with explicit synchronization through CUDA events.

6 Result

In this part, we have the results of running different solvers on the CPU and GPU and there are analyzed for different mesh sizes and numerical accuracies. The main focus is on the solver time and the assembly time. Solver Performance Here we have the performance of three solvers, Conjugate Gradient (CG), GMRES, and the direct solver (spsolve), this is evaluated on the CPU and on different GPUs (A40, A100, and H100). The tests we have done is for different mesh sizes and for both numerical precisions, float32 and float64.

6.1 Solver time

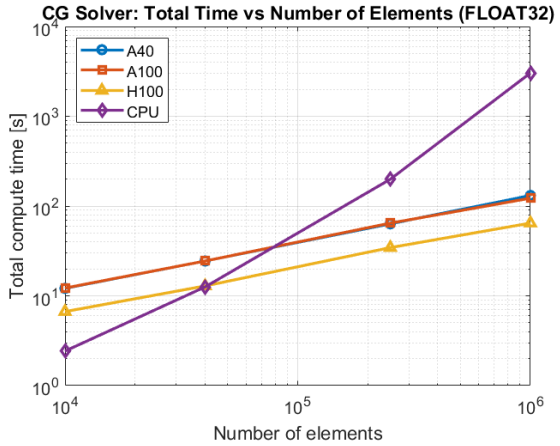
As we can see the results it shows that for small mesh sizes, the CPU solver is faster than the GPU in many cases and the main reason is the high data transfer cost and GPU setup overhead, which are more important when the problem is small. So we can see that as the mesh size increases, the GPU especially the H100 starts to work better than the CPU. We can clearly see that the solving time is shorter.

6.2 Comparison of Solvers

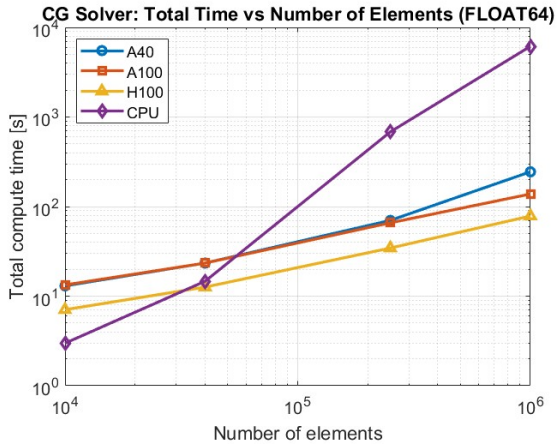
In this part, we will see a comparison between the three implemented solvers, Conjugate Gradient (CG), GMRES, and the direct solver (spsolve). The comparison will be based on the solver time measured on the CPU and on different GPUs (A40, A100, and H100), for different mesh sizes and for numerical precisions, float32 and float64.

As we can see the results in Table 2, we can see that for small mesh sizes (100×100), the CPU is faster than the GPU in all three solvers, this is because the data transfer between CPU and GPU and the GPU initialization overhead are relatively large, and for small problems, these costs are the main part of the runtime. When we compare the iterative solvers, CG are always works better than GMRES on all hardware setups. GMRES takes more time because each iteration is more expensive. We can see the difference when the mesh size grows, which indicates that CG scales better for this problem. When the number of elements increases, we can see the change in performance, in GPU versions, especially on the H100 start to run faster than the CPU. From that we can see that the iterative solvers benefit much from GPU acceleration when the problem becomes larger and parallelism can be used efficiently. The direct solver (spsolve) as show it behaves in a different way, as we can see for all mesh sizes and both numerical precisions, the CPU version is much faster than the GPU versions even for large meshes. So it shows that the direct solver does not work for GPU acceleration in the same way as the iterative solvers have done.

Overall, from the results we can see that the choice of solver is very important for performance. These solvers especially CG, work very well on GPUs and for large problems, while the direct solver is more efficient on the CPU for the problem sizes studied here.

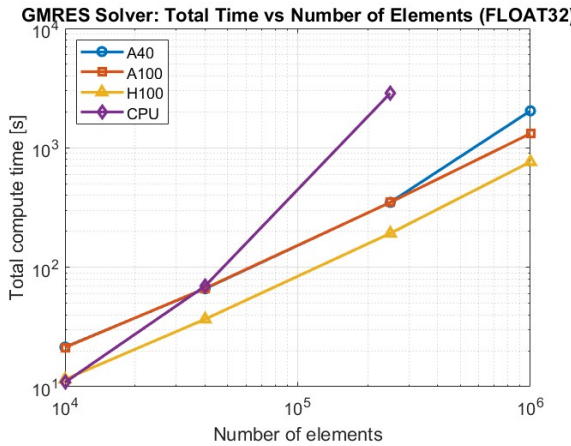


(a) Time vs Number of elements for CG Solver (Float32)

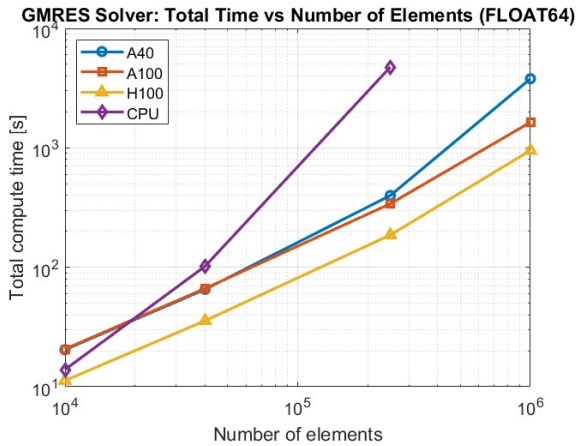


(b) Time vs Number of elements for CG Solver (Float64)

Figure 1



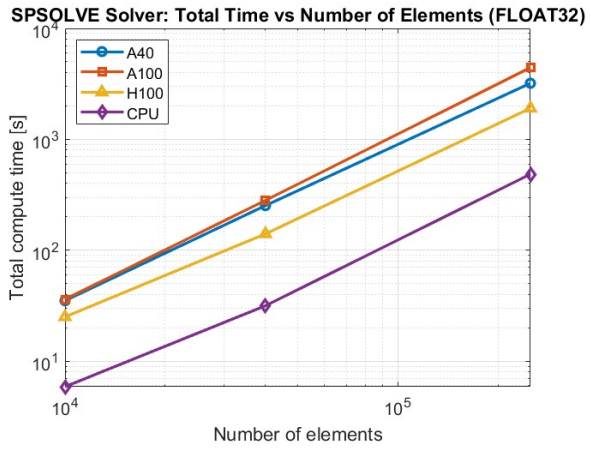
(a) Time vs Number of elements for GMRES Solver (Float32)



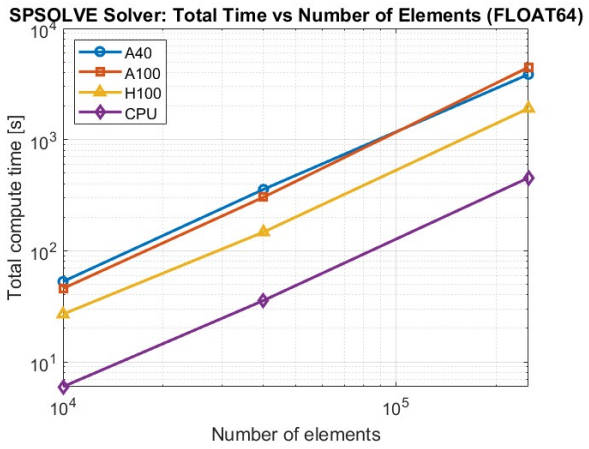
(b) Time vs Number of elements for GMRES Solver (Float64)

Figure 2

The following Figures 1, 2 and 3 are based on the results from the tables in 3, 4, 5, 6, 7, 8, 9 and 10.



(a) Time vs Number of elements for SP Solve Solver (Float32)



(b) Time vs Number of elements for SP Solve Solver (Float64)

Figure 3

Table 3: Regular run timing results (FLOAT32, tol = 10^{-5} 100×100 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	11.835751	0.059179	2606717	12.118
cg	A100	11.923437	0.059617	2606717	12.197
cg	H100	6.501898	0.032509	2606717	6.669
cg	CPU	2.371010	0.011855	2189597	2.423
gmres	A40	21.175593	0.105878	6165725	21.461
gmres	A100	21.027658	0.105138	6165725	21.309
gmres	H100	11.349319	0.056747	6165725	11.520
gmres	CPU	10.808776	0.054044	5536563	10.868
spsolve	A40	29.430895	0.147154	63331	35.037
spsolve	A100	35.774381	0.178872	63331	36.089
spsolve	H100	24.787807	0.123939	63331	25.035
spsolve	CPU	5.811430	0.029057	28087	5.876

Table 4: Regular run timing results (FLOAT64, tol = 10^{-5} 100×100 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	12.752180	0.063761	2791683	13.021
cg	A100	13.061958	0.065310	2791683	13.336
cg	H100	6.897617	0.034488	2791683	7.059
cg	CPU	2.922393	0.014612	2347331	2.978
gmres	A40	20.139586	0.100698	6087272	20.422
gmres	A100	20.403007	0.102015	6087272	20.680
gmres	H100	11.087746	0.055439	6087272	11.255
gmres	CPU	13.846211	0.069231	5947976	13.910
spsolve	A40	52.468535	0.262343	63131	52.777
spsolve	A100	45.418190	0.227091	63131	45.727
spsolve	H100	26.615300	0.133077	63131	26.859
spsolve	CPU	5.913196	0.029566	28087	5.978

F32 200x200	Times speedup
CPU vs A40	≈ 0.5x
CPU vs A100	≈ 0.5x
CPU vs H100	≈ 1.0x

F64 200x200	Times speedup
CPU vs A40	≈ 0.6x
CPU vs A100	≈ 0.6x
CPU vs H100	≈ 1.1x

F32 1000x1000.	Times speedup
CPU vs A40	≈ 24.0x
CPU vs A100	≈ 25.8x
CPU vs H100	≈ 49.2x

F64 1000x1000	Times speedup
CPU vs A40	≈ 25.7x
CPU vs A100	≈ 41.2x
CPU vs H100	≈ 81.3x

(a) Speed-up comparison between GPU and CPU for select dimension using `cg` solver.

F32 200x200 gmres	Times speedup
CPU vs A40	≈ 1.1x
CPU vs A100	≈ 1.1x
CPU vs H100	≈ 1.8x

F64 200x200 gmres	Times speedup
CPU vs A40	≈ 1.6x
CPU vs A100	≈ 1.6x
CPU vs H100	≈ 2.8x

F32 500x500 gmres	Times speedup
CPU vs A40	≈ 8.3x
CPU vs A100	≈ 8.3x
CPU vs H100	≈ 15.1x

F64 500x500 gmres	Times speedup
CPU vs A40	≈ 12.0x
CPU vs A100	≈ 14.1x
CPU vs H100	≈ 25.8x

(b) Speed-up comparison between GPU and CPU for select dimension using `gmres` solver.

F32 200x200 spsolve	Times speedup
CPU vs A40	≈ 0.1x
CPU vs A100	≈ 0.1x
CPU vs H100	≈ 0.2x

F64 200x200 spsolve	Times speedup
CPU vs A40	≈ 0.1x
CPU vs A100	≈ 0.1x
CPU vs H100	≈ 0.2x

F32 500x500 spsolve	Times speedup
CPU vs A40	≈ 0.2x
CPU vs A100	≈ 0.1x
CPU vs H100	≈ 0.3x

F64 500x500 spsolve	Times speedup
CPU vs A40	≈ 0.1x
CPU vs A100	≈ 0.1x
CPU vs H100	≈ 0.2x

(c) Speed-up comparison between GPU and CPU for select dimension using `spsolve` solver.

Figure 4

Table 5: Regular run timing results (FLOAT32, tol = 10^{-5} 200×200 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	24.111796	0.120559	5157164	24.429
cg	A100	24.102410	0.120512	5157164	24.415
cg	H100	12.687378	0.063437	5157164	12.883
cg	CPU	12.392223	0.061961	4362200	12.561
gmres	A40	65.820214	0.329101	19914298	66.147
gmres	A100	66.275232	0.331376	19914298	66.594
gmres	H100	36.574281	0.182871	19914298	36.772
gmres	CPU	69.479671	0.347398	17481690	69.649
spsolve	A40	252.220011	1.261100	63331	252.618
spsolve	A100	280.991474	1.404957	63331	281.377
spsolve	H100	140.076181	0.700381	63331	140.324
spsolve	CPU	31.381975	0.156910	28087	31.569

Table 6: Regular run timing results (FLOAT64, tol = 10^{-5} 200×200 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	23.101424	0.115507	4947822	23.409
cg	A100	23.101365	0.115507	4947822	23.409
cg	H100	12.424588	0.062123	4947822	12.619
cg	CPU	14.479929	0.072400	4184042	14.670
gmres	A40	65.036787	0.325184	19521236	65.352
gmres	A100	66.068466	0.330342	19521236	66.386
gmres	H100	35.492483	0.177462	19521236	35.690
gmres	CPU	101.846542	0.509233	18492915	102.055
spsolve	A40	355.738292	1.778691	63131	356.198
spsolve	A100	302.750722	1.513754	63131	303.132
spsolve	H100	146.812466	0.734062	63131	147.060
spsolve	CPU	35.412215	0.177061	28087	35.651

Table 7: Regular run timing results (FLOAT32, tol = 10^{-5} 500×500 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	59.443992	0.297220	16132509	63.423
cg	A100	60.394504	0.301973	16135255	64.512
cg	H100	31.956310	0.159782	16137189	34.439
cg	CPU	197.998162	0.989991	10678637	198.966
gmres	A40	346.043057	1.730215	106737395	349.949
gmres	A100	345.540968	1.727705	106679898	349.666
gmres	H100	190.002200	0.950011	106681704	192.385
gmres	CPU	2872.500596	14.362503	69898383	2873.931
spsolve	A40	3200.054304	16.000272	3605571	3204.078
spsolve	A100	4431.689367	22.158447	3605538	4435.838
spsolve	H100	1903.935410	9.519677	3607402	1906.401
spsolve	CPU	483.581322	2.417907	28087	484.905

Table 8: Regular run timing results (FLOAT64, tol = 10^{-5} 500×500 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	66.102957	0.330515	16470799	70.092
cg	A100	61.769317	0.308847	16443557	65.885
cg	H100	31.978649	0.159893	16460148	34.430
cg	CPU	679.903844	3.399519	10985551	682.012
gmres	A40	395.095117	1.975476	103119329	398.997
gmres	A100	336.357505	1.681788	103045771	340.522
gmres	H100	183.901349	0.919507	103068364	186.363
gmres	CPU	4743.923726	23.719619	88226080	4745.908
spsolve	A40	3858.465988	19.292330	3601091	3862.462
spsolve	A100	4463.424144	22.317121	3601277	4467.596
spsolve	H100	1903.936967	9.519685	3603057	1906.601
spsolve	CPU	451.782107	2.258911	28266	452.978

Table 9: Regular run timing results (FLOAT32, tol = 10^{-5} 1000×1000 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

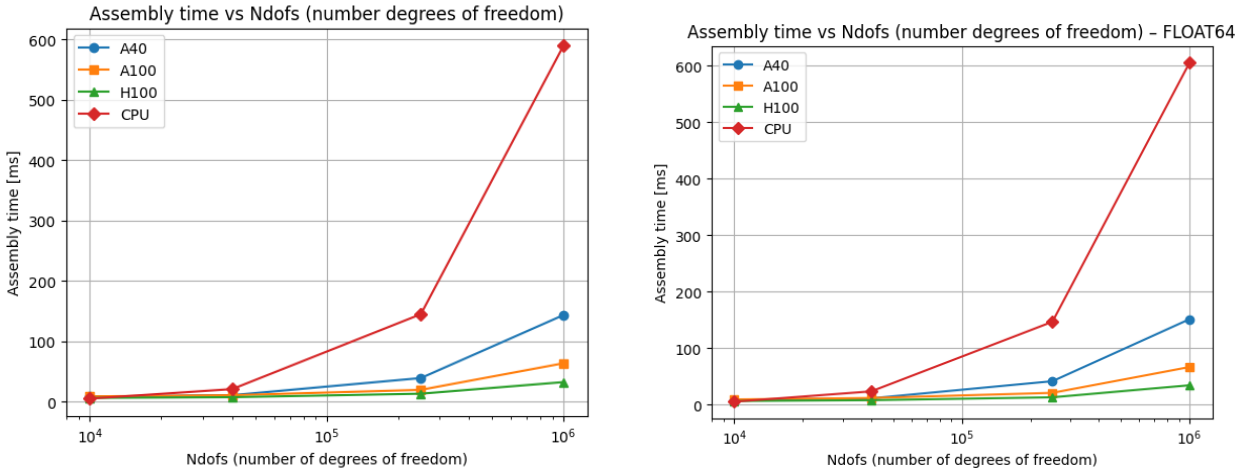
Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	126.359616	0.631798	28116682	131.044
cg	A100	117.488222	0.587441	28138835	122.328
cg	H100	61.605142	0.308026	28078687	64.560
cg	CPU	3028.080706	15.140404	20875970	3033.563
gmres	A40	2036.454614	10.182273	381878611	2041.185
gmres	A100	1312.612297	6.563061	379927279	1317.502
gmres	H100	759.979607	3.799898	379925845	763.087
gmres	CPU	29129.307954	145.646540	227157464	29134.296

Table 10: Regular run timing results (FLOAT64, tol = 10^{-5} 1000×1000 mesh, $dt = 5 \times 10^{-2}$, $T = 10.0$)

Solver	GPU	Cum. Time (s)	per call (s)	nr function calls	Total compute time (s)
cg	A40	239.553241	1.197766	31476500	244.398
cg	A100	133.147511	0.665738	31477480	138.130
cg	H100	75.571924	0.377860	31471163	78.457
cg	CPU	6146.454108	30.732271	23767525	6155.487
gmres	A40	3786.776161	18.933881	353059570	3791.750
gmres	A100	1632.994526	8.164973	353305734	1637.979
gmres	H100	948.177528	4.740888	353045483	951.407

6.3 Assembly time

Because the system assembly utilizes `Numba`, the initial execution incurs a compilation overhead, referred to as a "cold start." To account for this, the code was executed twice to record both the cold and "warm" execution times. The warm run utilizes cached machine code and therefore provides a more accurate representation of the system's performance during iterative usage. The results for the warm runs are presented in Figure 5, while the cold start data is available in the Appendix.



(a) "Warm-start" assembly time float32

(b) "Warm-start" assembly time float64

Figure 5

The Figure 5 is based on the results from tables 11 and 12.

Table 11: Warm-start assembly time of A-matrix, FLOAT32, solver = 'cg'

GPU	100×100	200×200	500×500	1000×1000
A40	8.62800026	11.15868759	39.40291214	143.67657471
A100	9.08528042	11.03123188	19.90163231	63.86463928
H100	6.54303980	7.73772812	13.53382397	32.69270325
CPU	5.40500414	21.18322672	145.13615705	589.61796714

Table 12: Warm-start assembly time of A-matrix, FLOAT64, solver = 'cg'

GPU	100×100	200×200	500×500	1000×1000
A40	8.47526360	11.27555180	41.60835266	151.41993713
A100	9.34467220	11.91388798	20.94934464	66.66809845
H100	6.54307222	8.01103973	13.12515163	34.34406281
CPU	5.37401577	23.56119780	146.82942815	605.34045193

6.4 Black–Scholes Solver Performance

All Black–Scholes performance experiments were conducted on a single workstation equipped with an AMD Ryzen 7 7800X3D CPU (8 cores / 8 threads, base frequency 4.20 GHz) and an NVIDIA GeForce RTX 4070 Ti GPU.

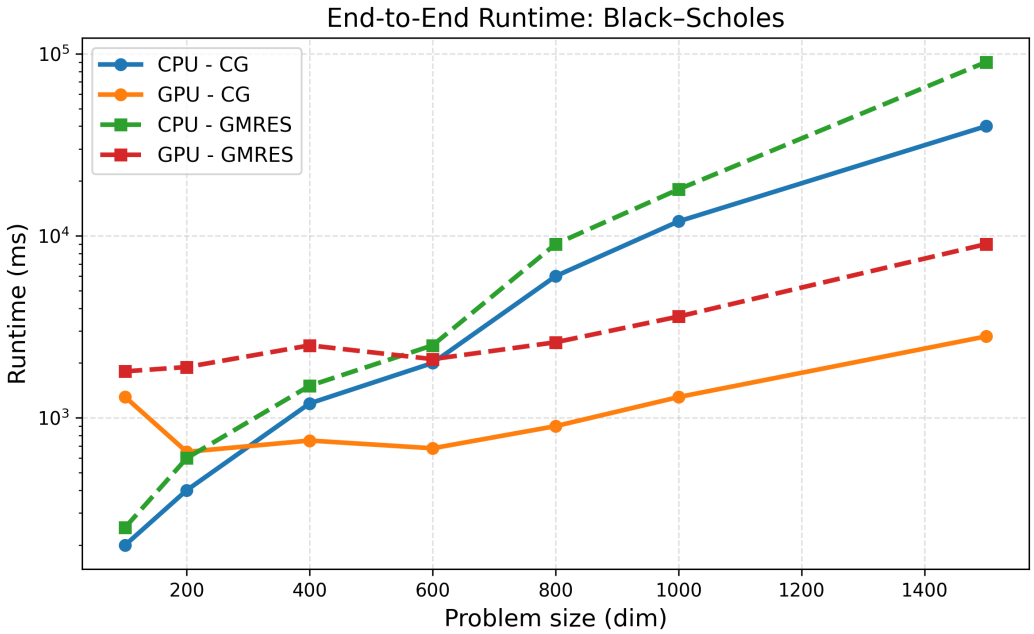


Figure 6: End-to-end runtime for the Black–Scholes problem as a function of problem size. Results are shown for CG and GMRES solvers on both CPU and GPU, and include matrix assembly, data transfer, and solver execution.

Figure 6 shows the end-to-end runtime of the FEM solver applied to the transformed Black–Scholes problem as a function of problem size. Results are reported for both CG and GMRES solvers on CPU and GPU, and include the full cost of matrix assembly, host-to-device data transfer, and time-stepping.

For small problem sizes (dim ≤ 200), the CPU achieves lower runtime due to GPU initialization and data transfer overhead. However, as the problem size increases, the GPU runtime grows at a significantly slower rate than the CPU runtime. Beyond dim ≈ 400, GPU execution consistently outperforms the CPU for both solvers, and the performance gap widens as the problem size increases.

Table 13 reports representative end-to-end runtimes and corresponding CPU-to-GPU speedup factors for the Black–Scholes problem at selected problem sizes. The table provides quantitative support for the scaling trends observed in Figure 6 and Figure 7.

Table 13: End-to-end runtimes and CPU-to-GPU speedup for the Black–Scholes problem. Speedup is reported for both total runtime and linear solve phase only.

Dim	Solver	CPU Total (ms)	GPU Total (ms)	Speedup (Total)	Speedup (Solve)
100	CG	143.88	1297.41	0.11	0.21
100	GMRES	251.10	1740.62	0.14	0.14
200	CG	396.95	573.91	0.69	0.65
200	GMRES	649.53	1780.15	0.36	0.35
400	CG	1188.26	584.38	2.03	2.01
400	GMRES	1639.83	2630.14	0.62	0.58
800	CG	5746.89	826.66	6.95	9.41
800	GMRES	8855.47	2542.92	3.48	3.66
1500	CG	42603.85	2949.59	14.44	21.04
1500	GMRES	90640.88	8925.80	10.15	11.22

The GPU acceleration benefits are further quantified in Figure 7, which reports the CPU-to-GPU speedup

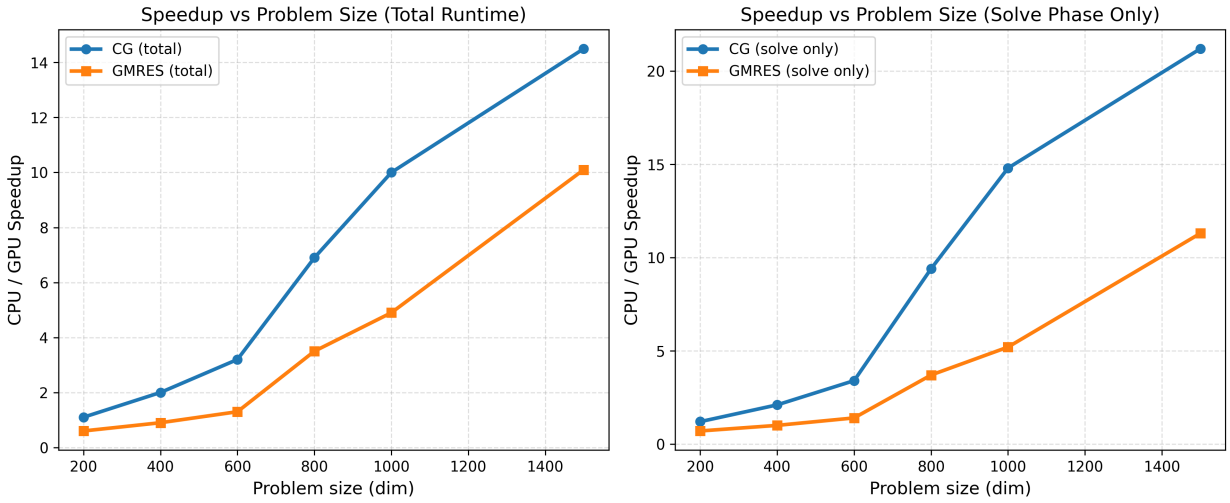


Figure 7: CPU-to-GPU speedup for the Black–Scholes problem as a function of problem size. The left panel shows speedup based on total end-to-end runtime, while the right panel shows speedup considering only the linear solve phase. Results are shown for CG and GMRES solvers.

as a function of problem size for both CG and GMRES. A significantly stronger scaling trend is observed when focusing exclusively on the linear solve phase. In this case, the CG solver exceeds 20× speedup for large problem sizes, whereas GMRES achieves approximately 11× speedup.

The discrepancy between total runtime speedup and solve-only speedup highlights the impact of fixed GPU overheads, such as matrix assembly and host-to-device transfers, which diminish in relative importance as the problem size increases. Overall, these results demonstrate that GPU acceleration is most effective for the iterative Krylov solvers themselves, with the Conjugate Gradient method exhibiting superior scalability. The observed trends closely mirror those obtained for the 2D heat equation, confirming that the transformed Black–Scholes problem preserves the numerical structure that enables efficient GPU acceleration.

6.5 Implementation

Here we discuss our results and analyse the factors influencing the observed performance.

6.5.1 Penalty method

During the implementation phase, we noticed that when running the code using iterative solvers—such as Conjugate Gradient or GMRES—it frequently performed worse than the direct solver. This occurred even for systems with relatively small dimensions, such as $200 \times 200 = 40,000$ degrees of freedom (DOF). This was counter-intuitive, as iterative solvers should theoretically be faster than direct solvers for large sparse systems. The primary reason for this behaviour was likely the failure of the iterative solvers to converge, which stemmed from the way the Dirichlet boundary conditions were enforced. To mitigate this, we added a penalty term to the boundary of the matrix A. This strategy, as explained in the next subsection, was implemented across all three solvers.

6.5.2 Tolerance

The reason we might not have observed a meaningful performance or accuracy difference between `float32` and `float64` is that our test cases did not fully exploit the strengths of double precision. We did not closely monitor the precision of the numerical solutions. Similarly, we chose a tolerance threshold of `1e-05`, which is appropriate for `float32`, whereas a `float64` implementation could push this threshold significantly further. For example, a tolerance of `1e-07` would be near the limit of precision for the `float32` option but would be easily achievable using `float64`.

6.6 Performance of `spsolve()`

The trend observed in the 500×500 case persists in the 1000×1000 case; if anything, the performance gap becomes even more stark. For instance, to solve the 1000×1000 system, the A40 GPU took $23,196.676$ seconds (≈ 6.44 hours), whereas the CPU solver completed the same task in $5,457.54$ seconds (≈ 1.52 hours).

By profiling where each solver spends the majority of its time, the cause of this discrepancy becomes clear. On the A40 GPU, the `spsolve` function spends 115.959185 seconds per iteration, with almost all of that time (115.959126s) occurring within the `cupyx.cusolver.csrlsvqr` function.

Conversely, the CPU-based `spsolve` takes only 27.250337s per iteration, with the majority (27.244150s) spent inside the `scipy.sparse.linalg.dsolve.superlu.gssv` function.

Investigation into this issue (referencing this GitHub issue) reveals that while the `scipy` implementation of `spsolve` utilizes LU decomposition, the `cupy` implementation uses a sparse QR decomposition, which is inherently slower than the LU variant. Furthermore, LU and QR decompositions are notoriously difficult to accelerate on a GPU when the matrix is highly sparse. Since our implementation relies on sparse matrices, these results are expected.

6.7 Performance of different GPUs

We observed that the A40 and A100 performed similarly, likely because the problem sizes were too small to fully utilize the massive parallel resources of the A100, causing the execution to be dominated by overhead. These results show that we did not reach the saturation point of the hardware's compute throughput for these specific configurations.

The H100 was consistently twice as fast as the other graphics cards. This performance boost is likely due to its significantly higher memory bandwidth and the architectural advancements of the Hopper architecture, which provide superior handling of sparse data structures compared to the Ampere-based A40 and A100.

6.8 Single-run Results

Due to time constraints, we were unable to collect data from multiple runs; consequently, these results should be viewed with some skepticism regarding their statistical significance. Our analysis focuses on substantial performance changes. For instance, a $2\times$ speed-up is considered a meaningful result. Conversely,

if one configuration is only a few seconds faster over a multi-minute execution, we treat such a difference as potentially negligible or within the margin of noise.

6.9 Missing data in tables

Notably, results for `spsolve` are not present. That is due to reaching a time-limit of 10 hours and therefore never considered as useful.

7 Conclusion

In this project, we compare GPU-accelerated finite element computations with CPU implementations with using of three solvers, the first one is Conjugate Gradient (CG), the second one GMRES, and the direct solver `spsolve`. The comparison is performed for different mesh sizes, for both float32 and float64 precision, and on different hardware platforms.

From the results, we can see that for small problem sizes, the CPU is generally faster because GPU initialization and data transfer overhead are relatively large. So as the mesh size increases, the iterative solvers benefit more from GPU acceleration, and the H100 GPU gives the best performance. Between the iterative solvers CG consistently work better than GMRES, this is because CG has a lower computational cost per iteration and scales better as the problem size increases.

The implementation of Dirichlet boundary settings using the Penalty Method together with Jacobi preconditioning was important to achieve stable convergence of the iterative solvers. On the other hand, the direct solver `spsolve` worked better on the CPU for all tested cases, since its GPU implementation is based on sparse QR decomposition, which is not so efficient for very sparse systems.

The same solver framework was also applied to a two-dimensional Black–Scholes problem by exploiting its transformation into a heat equation. The Black–Scholes experiments showed the same overall performance trends as the standard heat equation case. In particular, CPU execution was preferable for small problem sizes, while GPU acceleration provided significant speedups for larger systems, especially when using the Conjugate Gradient solver. For the largest problem sizes, the GPU achieved more than an order-of-magnitude speedup in the linear solve phase, confirming that the transformed Black–Scholes problem preserves the numerical structure that enables efficient GPU acceleration.

Overall, This study shows that iterative solver CG, are perfect for GPU acceleration in large scale finite element problems but on other hand the direct solvers remain more efficient on the CPU for the problem sizes considered.

8 Contribution Report

We shared a lot of the work between each other fairly, when it came to the report, presentation and code-writing, but here we will list some things where we diverged.

8.1 Daniel Fathi

- Did the Results and Conclusion sections.
- Analyzed and compared CPU and GPU solver performance.

8.2 Jinlun Gao

- Developed 1-dimensional Black–Scholes solver.
- Extended an existing 2-dimensional heat equation implementation to incorporate the Black–Scholess.

8.3 Jan Kula

- Did more of the programming of the 2-dimensional heat equation code.
- Implemented the Assembly of the system on the GPU.
- Ran the benchmarks and created all tables containing those benchmarks.

8.4 Brijesh Niranjan

- Did the theory behind the FEM method.
- Plotted results from data points
- Code for plotting results

8.5 Simon Rödén

- Wrote the Black-Scholes model in 1 Dimension.
- Found and derived the math behind the Black-Scholes model in multiple dimensions.
- Worked with Jan on the 2d heat equation implementation.

A Tables

Table 14: Cold-start assembly time of A-matrix, FLOAT32, solver = 'cg'

GPU	100×100	200×200	500×500	1000×1000
A40	919.40454102	734.65649414	771.41186523	871.97088623
A100	937.96453857	733.70623779	760.87908936	772.21374512
H100	590.57287598	547.06634521	565.07275391	551.78753662
CPU	1139.59283289	1167.90943500	1288.66877314	1754.55687102

Table 15: Cold-start assembly time of A-matrix, FLOAT64, solver = 'cg'

GPU	100×100	200×200	500×500	1000×1000
A40	744.19506836	740.98162842	748.63726807	850.25384521
A100	975.11425781	710.99597168	795.79650879	797.20758057
H100	516.69915771	529.71240234	534.49792480	549.33648682
CPU	1090.58149578	1211.76704532	1277.99020382	1648.02864334

References

[1] Charles D. Joyner. Black–scholes equation and heat equation. Honors college thesis, Georgia Southern University, 2016. Available at Georgia Southern Commons.