Chalmers University of Technology

GPU-accelerated Computational Methods using Python and CUDA

Accelerating a Poisson solver with CUDA

Author:
Jesper MÖLLBRANT
Raphaël BOUCHEZ

Supervisor:
Prof. Rickard Bensow

January 2025

Contents

1 Introduction

Over the past decade, graphical processing units (GPUs) have widely adopted for high-performance computing. While central processing units (CPUs) excel at sequential processing with their fewer and more powerful cores, GPUs offer a fundamentally different approach. With hundreds or thousands of cores working in parallel, GPUs can distribute workloads across multiple processors simultaneously, enabling significant speedups for suitable computational tasks.

NVIDIA's CUDA platform has played an important role in making GPU computing accessible to developers. This parallel computing platform allows applications to effectively leverage GPU architecture by running sequential portions of code on the CPU while delegating computationally intensive tasks to the GPU's many cores. This hybrid approach takes advantage of each processor's strengths: the CPU's optimization for single-threaded performance and the GPU's massive parallel processing capability.

The Poisson equation, an elliptic partial differential equation, exemplifies the type of problem that can benefit from GPU acceleration. This equation appears frequently in physics and engineering applications, from calculating electrostatic potentials to determining pressure fields in fluid dynamics. Given its fundamental role in many simulations and computational methods, an efficient solution of the Poisson equation is crucial for overall application performance. This report explores the acceleration of computational methods for solving the Poisson equation.

1.1 Definition of the Poisson Equation Problem

The Poisson equation is a partial differential equation of the form:

$$\nabla^2 p(x, y, z) = S(x, y, z),$$

where $p(x, y, z)$ represents the scalar potential field we seek to solve, and $S(x, y, z)$ is the source term defining the distribution of sources and sinks within the domain.

For our study, we defined the problem on a three-dimensional computational domain $[0, 1]^3$, discretized using a non-uniform Cartesian grid with n_i, n_j, and n_k points along the x, y, and z axes, respectively. The grid spacing in each direction is derived from power-law transformations, providing variable resolution across the domain. Specifically:

$$x_i = \left(\frac{i}{n_i}\right)^{1.2}, \quad y_j = \left(\frac{j}{n_j}\right)^{1.5}, \quad z_k = \left(\frac{k}{n_k}\right)^{1.1},$$

where i, j, and k are the grid indices.

The source term $S(x, y, z)$ is initialized with two localized sources of equal magnitude but opposite sign:

$$S(x,y,z) = \begin{cases} +50, & \text{at } \left(\frac{n_i}{4}, \frac{n_j}{4}, \frac{n_k}{4}\right), \\ -50, & \text{at } \left(\frac{3n_i}{4}, \frac{3n_j}{4}, \frac{3n_k}{4}\right), \\ 0, & \text{elsewhere.} \end{cases}$$

Boundary conditions are defined to ensure no flux across the edges of the domain, corresponding to $\frac{\partial p}{\partial x} = 0$, $\frac{\partial p}{\partial y} = 0$, and $\frac{\partial p}{\partial z} = 0$ on all boundaries.

The goal of this problem is to determine the scalar field $p(x, y, z)$ that satisfies the Poisson equation given the non-uniform grid, the specified source term $S(x, y, z)$, and the imposed boundary conditions.

1.2 The iterative algorithm

The most naive way to computationally solve this problem would be to perform continuous relaxations on the matrices of the components of the equation. Implementing an iterative algorithm was the first step of our study process and has helped us better understand the nature of the problem as well as how to solve it. However, it is quite obvious that solving the Poisson Problem with iterations over its components will end up quite inefficient. If we want to perform x iterations over 3-dimensional matrices of size (N_i, N_j, N_k), the cost in time complexity would become $O(N_i \times N_j \times N_k \times x)$, making the scaling quite complex for high sizes of grids.

Nevertheless, we thought that an algorithm of this complexity was a real opportunity to showcase the efficiency of high performance computing optimization methods. This is why we still chose to present multiple versions of this algorithm later in this report. They will be compared over different scenarios to asses their overall performance and the benefits that GPU-computing can leverage.

1.3 The Multigrid method

The multigrid method (MG) is a numerical technique for solving differential equations using multiple levels of grid resolution. At its core, this method uses a hierarchy of discretizations to efficiently solve complex problems that might be computationally intensive on a single grid level.

Building upon the classical multigrid approach, the Algebraic Multigrid (AMG) method offers enhanced flexibility by eliminating the need for explicit geometric information of the problem. Unlike traditional multigrid methods, AMG constructs its hierarchy of levels directly from the system matrix, working with subsets of unknowns without any geometric interpretation. The AMG develops a sequence of coarser grids directly from the input matrix. This abstraction from geometry makes AMG particularly valuable for problems discretized on unstructured meshes and irregular grids.

1.3.1 CPU-Based PyAMG

PyAMG provides a collection of Algebraic Multigrid solvers through a Python interface, operating on CPU cores for traditional sequential processing.

1.3.2 GPU-Accelerated PyAMGX

PyAMGX brings multigrid acceleration to GPUs by interfacing with NVIDIA's AMGX library through Python. This library enables the construction of multigrid solvers that leverage GPU parallel processing capabilities.

2 Problem statement

The growing computational demands of simulations and numerical methods require efficient solvers for partial differential equations such as the Poisson equation. Hence, leveraging GPU acceleration requires a detailed understanding of the trade-offs between computational speed, solution accuracy, and environmental sustainability.

This study addresses three key research questions:

1. Iterative Solver Acceleration: How do GPU acceleration methods affect computational time, residual norm, and CO2e emission when applied to a CPU-based iterative solver?

2. Comparison of Multigrid Methods: How do CPU-based PyAMG and GPU-accelerated PyAMGX compare in terms of computational time, and residual norm?

3. GPU Performance Analysis: How do various NVIDIA GPUs perform relative to each other with respect to computational time, and CO2e emission?

By investigating these questions, this report aims to provide a comprehensive analysis of the computational and environmental efficiency of GPU acceleration methods and hardware in solving the Poisson equation.

3 Methodology

3.1 Acceleration of iterative solver

3.1.1 CPU implementation

For a basic CPU implementation of the iterative solver, we can use Numpy to perform operations on multi-dimensional arrays. The Numpy package is a standard in scientific Python programming. It provides an interface for creating arrays and allows the programmer to efficiently perform linear algebra. The package was utilized to create the an iterative solver using the Jacobi relaxation method.

3.1.2 Parallelizing with numba

The very first step of optimization that we performed was to bring parallelization to the program. By dividing the workload over multiple process, we could reduce the time complexity of the matrix equation solving. To perform this optimization, we used the numba library. numba is a Python accelerator library. More than providing a loop parallelization interface, it allows to compile existing functions to optimized machine code. Using it, we were also able to dodge the very slow Python CAPI and approach the speeds of C or FORTRAN.

```
def slow_python_function():
    ...

@numba.jit(nopython=True)
def omptimally_complied_function():
    ...
```

Figure 1: Compiling function to optimized machine code with numba

Parallelizing the loops did not require much more effort than compiling the function as numba directly provides a replacement of the classic Python loop.

```
from numba import njit
from numba import prange

def slow_loop_iteration():
    for i in range(ni):
        for j in range(nj):
            for k in range(nk):
                ...

@njit(parallel=True)
def quick_parallel_iteration():
    for i in prange(ni):
        for j in prange(nj):
            for k in prange(nk):
                ...
```

Figure 2: Parallelizing loops with numba

3.1.3 GPU-computation with CuPy

After performing parallelization and optimized compilation, there are only so few possibilities left if staying in a CPU solver. This is why, at this stage, we decided to explore shifting the algebra operations to a GPU. GPUs are well known to offer faster computation on highly parallelizable tasks. Our matrix equation is an ideal candidate as we can divide the algebra by addressing 'cells' of the problem's grids.

The easiest way to transfer the matrix calculus to the GPU was to simply replace numba bindings with CuPy bindings. The CuPy library is specially made for this use case. It mimics functions from CuPy and GPU-accelerates them, allowing its user to benefit from fast GPU-computing algebra without high prior knowledge of GPU-programming.

```
import numpy as np
import cupy as cp

# CPU
cpu_array = np.array([1, 2, 3])
cpu_computation = np.linalg.norm(x_cpu)

# GPU
gpu_array = cp.array([1, 2, 3])
gpu_computation = cp.linalg.norm(x_gpu)
```

Figure 3: GPU computation with CuPy

3.1.4 Optimized GPU computing with CUDA kernels

Now at this point, if we want to perform even better than after all the previous optimizations, we have to dive a bit deeper in technicality. The most popular way of programming with GPU is to use Nvidia's CUDA library to send instructions directly to the hardware. Thankfully, things are made a bit easier in Python, for example with numba. numba provides its own cuda bindings, allowing to create custom CUDA kernels for specialized computation. A CUDA kernel is a subroutine that contains instructions for parallelized computation with CUDA. Leveraging numba's bindings, we gain access to a deeper control over the behaviour of our program.

The first step of developing a GPU algorithm is to declare values for the division of our data. GPUs have a hierarchical structure that logically divides the workload. Multiple threads are regrouped in blocks which are themselves regrouped in grids. GPUs contain multiple grids and send structured data to the threads following a certain logic. This is why the first step of the process is to come up with a division strategy by setting setting a number of threads per block and a number of blocks per grid that fits the problem we want to solve.

Then, the next step is to send our data to the GPU so that we can access it and manipulate it inside our kernel subroutine. A reversed transfer will be needed for the result once the computation is done.

```
# CPU to GPU
gpu_data = cuda.to_device(cpu_data)

...

# GPU to CPU
cpu_data = gpu_data.copy_to_host()
```

Figure 4: CPU to GPU data transfers

Once all these steps are realized, we can start to write a kernel subroutine. numba once again provides bindings that make writing the logic of the kernel easier. The subroutine will be ran in parrallel in threads with different positions in the grid structure. Therefore, we can link these positions to our 3 dimensional grids to compute our whole matrices in parallel. This can be done using a numba binding.

```
from numba import cuda

@cuda.jit
def kernel():
    i, j, k = cuda.grid(3)

    # logic based on i, j and k
    ...
```

Figure 5: GPU kernel concept

3.2 Implementing multigrid methods

When studying the multigrid approach for solving the Poisson problem, we first implemented our own solvers. But, in parrallel to this, we also discovered some Python libraries to import for solving the Poisson problem. After trying different approaches, since the aim of using the multigrid method was to come up with an efficient and fast solver, we decided to use premade libraries as their algorithms were performing a lot better than the ones we made.

For a CPU based solver, we implemented the pyAMG library which provides a range of different multigrid solver functions. From this, we only had to set the parameters and the scenario and send it to the library's solver.

For a GPU version, we used the pyAMGX library. This library is a set of bindings for the AMGX toolset developed by Nvidia. AMGX provides multigrid solvers accelerated with CUDA by Nvidia developers and specialists. However, the Python pyAMGX library requires setting up your own instance of AMGX and a set of different tools and library to function corretcly. To get it to work on the VERA cluster, we had to download and compile our own versions of AMGX on our local sessions, adjust some tools versions to ensure cross compatibility of components, and then, manually building a local version of the pyAMGX library to import it manually on our environment. Once this was done, similarily to the CPU version, we just had to set some parameters and send the scenario to the solver.

3.3 Power consumption

If GPU usage are growing over a large variety of use cases, it is also important to point out one of the flaws of using them in large masses: carbon emissions. As big GPU clusters use a lot of power and generate a lot of heat, their threat to the environment is slowly becoming a serious question, especially when the biggest economic powers in the world decide to invest hundreds of billions into large scale AI training.

3.3.1 Calculating CO2e Emission

To assess the environmental impact of different computational methods, we calculated their CO2 equivalent (CO2e) emissions using power consumption data and standardized emission factors.

The power consumption and calculation time data was collected on the VERA cluster at Chalmers University. For each implementation, we used a fixed scenario with the same base values, no precision limits were set, the grid sizes were set to three dimensions of 256 and the solvers were ran on various ranges of time. Then, we extracted the Watts consumption per slice of 15 seconds and averaged it on the total solving time.

Lastly, for converting kWh to CO2e emissions, we used the Nordic electricity mix emission factor of 90 grams CO2e per kWh, as specified by Naturvårdsverket (Swedish Environmental Protection Agency, 2023). Two key metrics were calculated:

1. Average Hourly Emissions:

$$\text{Average Emission (gram/h)} = \frac{\text{Average Power (W)}}{1000} \times 90$$

2. Total Emissions per Calculation:

$$\text{Total Emission (gram)} = \frac{\text{Average Power (W)} \times \text{Calculation Time (s)}}{3600 \times 1000} \times 90$$

3.3.2 Comparing of GPUs

After comparing different solvers and algorithms, we also thought it could be interesting to see how different GPUs perform and the same problems and try to get an idea of each GPU's perofrmance to cost ratio.

For this, we ran a set scenario on the pyAMGX solver on different GPUs, extracted the Watts usage per slice of 15 seconds and averaged it over the total time spent.

4 Result

4.1 Iterative Solver Acceleration

4.1.1 CPU solver

To asses the performance of this algorithm, we ran it for 1000 iterations on different grid sizes and measured the computing time of the solving process.

As we could expect, running high dimensional linear algebra on a single-threaded program on the CPU was not very promising. The values rise very quickly to the hundreds of seconds and even come close to a thousand for a 3 dimensional 256 sized grid.

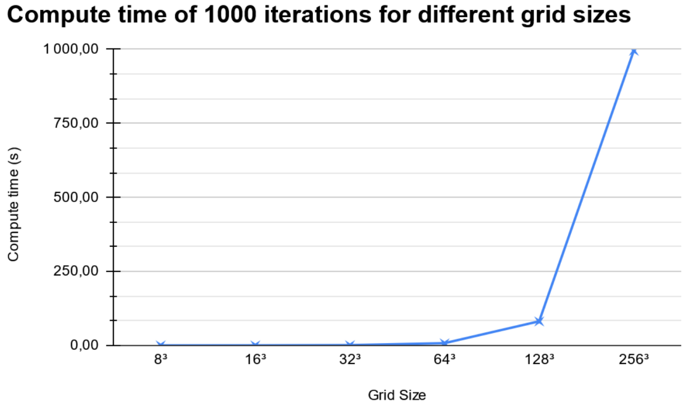


Figure 6: Compute time of 1000 iterations of the CPU iterative solver on different grid sizes

4.1.2 Parralel solver

Using numba to parallelize our algorithm and compile our functions to optimized machine code, we were able to reduce the compute time from 1000 seconds to approximately 70 seconds. While this number still represent a long waiting time for our problem solver, the improvement seemed very promising. It was even more impressive for the fact that our second solver remained completely CPU-based at this stage.

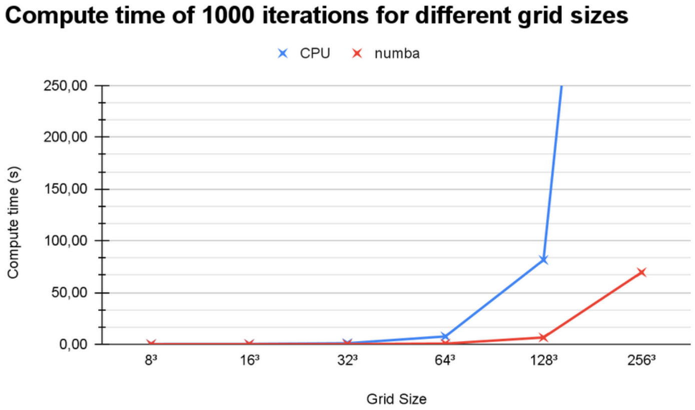


Figure 7: Compute time of 1000 iterations of the numba parallelized solver on different grid sizes

4.1.3 CuPy solver

With CuPy functions, the total compute time of the same problem as for previous solvers was reduced to approximately 10 seconds. By adopting GPU-computing, we were finally able to reach reasonable solving time and a drastically better scaling to large grid dimensions.

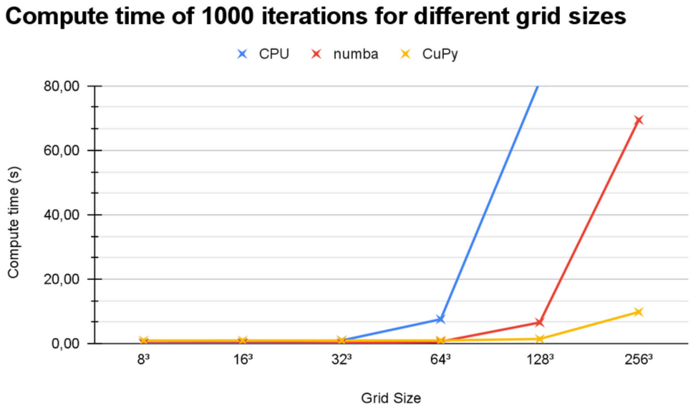


Figure 8: Compute time of 1000 iterations of the CuPy solver on different grid sizes

However, at this point, we realized one of the main critical points of programming with GPUs: the data transfer overhead. While GPUs benefit from very fast parallel computations, transferring data from the CPU to the GPU and back is very costly and can often end up longer than actually computing with the data. This problem especially occurs on smaller sizes of context that propose lesser levels of problem division. This is why, on small grid sizes, CuPy ended up spending more time than both CPU solvers, as it had to take time for transferring data before and after its solving.

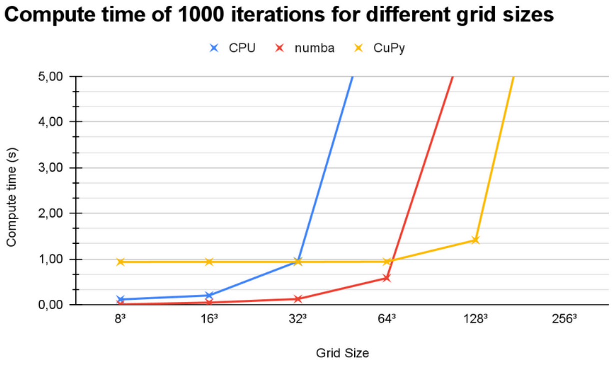


Figure 9: Compute time of 1000 iterations of the CuPy solver on smaller grid sizes

4.1.4 CUDA kernels solver

Implementing custom CUDA kernels allowed us to optimize the data transfers, addressing the main problematic of our last GPU-based solver. By applying this new method, we were able to fix our bad performance on low grid levels.

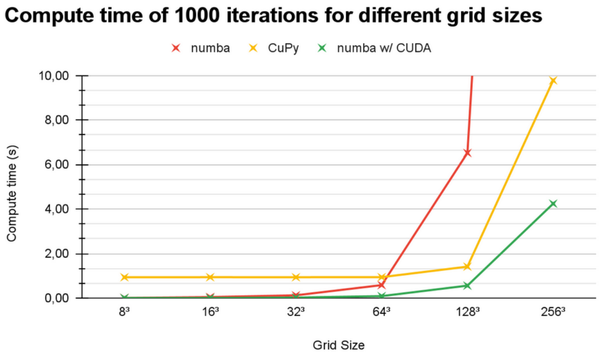


Figure 10: Compute time of 1000 iterations of the CUDA solver on smaller grid sizes

Also, centralizing our matrix operations and fine-tuning the program to our use case, we were able to reduce even more the total computation time to as little as 4 seconds.

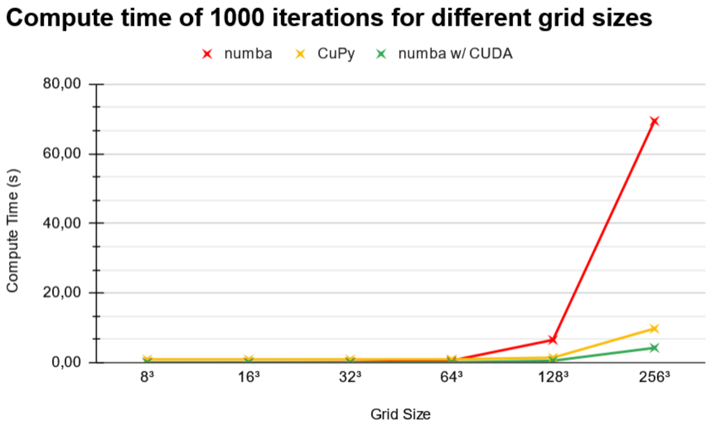


Figure 11: Compute time of 1000 iterations of the CUDA solver on different grid sizes

4.1.5 Co2e Emission

The environmental impact of different acceleration methods is presented in Table 1. The result presents two important aspects of emissions: the average hourly rate and the total emissions for completing the calculation.

GPU-based methods generally showed higher hourly emission rates compared to CPU-based approaches, due to their higher power consumption. The CuPy implementation had the highest hourly emission rate at 22.97 grams/hour, followed by Numba with CUDA at 13.06 grams/hour. In contrast, the basic CPU implementation had the lowest hourly rate at just 0.47 grams/hour.

However, when considering total emissions for the complete calculation, a different pattern emerges due to the varying computation times:

- GPU-accelerated Numba with CUDA and PyAMGX achieved the lowest total emissions (0.0154 and 0.0104 grams respectively) despite their higher hourly rates, thanks to their superior computational efficiency.
- The basic CPU implementation, despite its low hourly rate, resulted in the highest total emissions (0.1291 grams) due to its significantly longer computation time.
- CuPy presented an interesting case where its combination of high hourly emissions and relatively slower runtime resulted in higher total emissions (0.0624 grams) than the CPU-based Numba njit (0.0456 grams).

These results demonstrate that faster computation times can lead to a lower overall environmental impact, even when using hardware with significantly higher power consumption. This highlights the importance of considering both power consumption and computational efficiency when evaluating the environmental impact of different solver implementations.

Solver	**Average Emission (Gram/h)**	**Emission of Calculation Time (Gram)**
Numba w/ CUDA	13.06	0.0154
CuPy	22.97	0.0624
PyAMGX	9.83	0.0104
CPU	0.47	0.1291
CPU Numba njit	2.37	0.0456

Table 1: Comparison of emissions from different solvers.

4.2 Comparison of Multigrid solvers

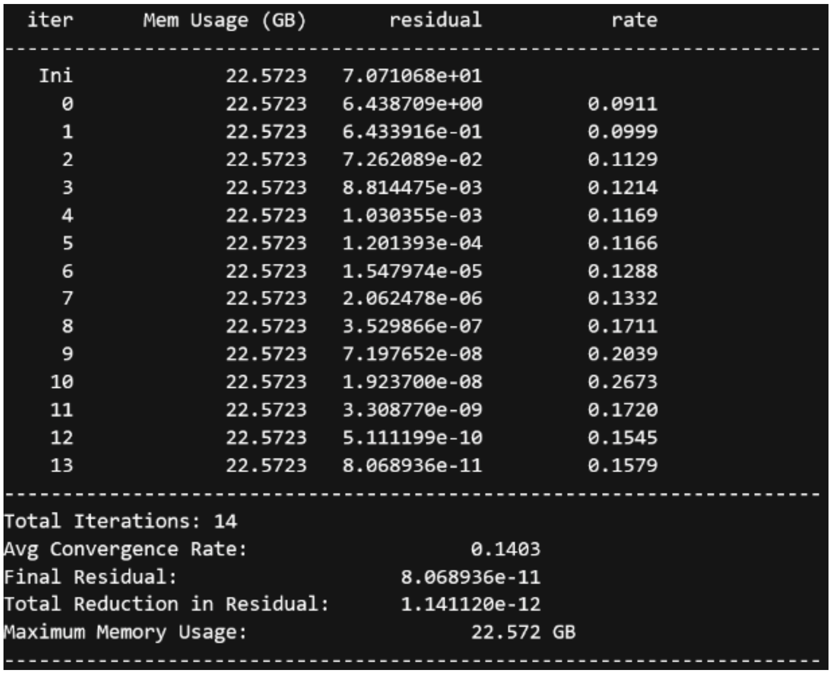
4.2.1 Accuracy

Comparing the multigrid solvers to its iterative counterparts is quite tricky. As the structure of their algorithm is very different, we have to set a fair baseline unit for them to fight in common ground. Setting a number of iteration would not be the ideal as the mutligrid solvers rely on completely different logical operations. This is why we simply chose to run both of them on the same problem preset, for a set amount of time, and compare their results. This way, we would be able to get an idea of how efficiently both of these solver types spend their computation times.

First, we ran the iterative CPU algorithm against the CPU-based pyAMG solver. After 1000 seconds, the results were shocking: the residual norm of the iterative solver was 0,20 against 6,75e-09 for pyAMG. This enormous gap not only showcased how inefficient iteratively solving a Poisson problem is, but also how good the pyAMG library performed.

Now, if we look back at the benefits of GPU-acceleration on our past solvers, by combining the efficiency of the multigrid method and the speed of GPU-computing, we could be able to reach precise results in only a few seconds.

The next step was to run our pyAMGX solver on the same problem as before. Once again, the results were very positive. After only 3,81 seconds, the solver reached a residual norm as little as the precision of the GPU it was running on and automatically stopped. The residual norm reached a value of approximately 8.07e-11.



```
 iter      Mem Usage (GB)       residual          rate
-----------------------------------------------------------------------
   Ini            22.5723   7.071068e+01
     0            22.5723   6.438709e+00         0.0911
     1            22.5723   6.433916e-01         0.0999
     2            22.5723   7.262089e-02         0.1129
     3            22.5723   8.814475e-03         0.1214
     4            22.5723   1.030355e-03         0.1169
     5            22.5723   1.201393e-04         0.1166
     6            22.5723   1.547974e-05         0.1288
     7            22.5723   2.062478e-06         0.1332
     8            22.5723   3.529866e-07         0.1711
     9            22.5723   7.197652e-08         0.2039
    10            22.5723   1.923700e-08         0.2673
    11            22.5723   3.308770e-09         0.1720
    12            22.5723   5.111199e-10         0.1545
    13            22.5723   8.068936e-11         0.1579
-----------------------------------------------------------------------
Total Iterations: 14
Avg Convergence Rate:                     0.1403
Final Residual:                     8.068936e-11
Total Reduction in Residual:        1.141120e-12
Maximum Memory Usage:                     22.572 GB
-----------------------------------------------------------------------
```

Figure 12: Evolution of the residual norm with the pyAMGX solver

4.3 GPU Performance Analysis

The GPU we compared in this part were 3 GPUs available on the VERA cluster at Chalmers. They all had different specifications that could impact the performance of the solving, but the one we expected to matter the most were the memory and the bandwidth.

GPU	Memory	Memory Bandwidth
A100	80GB	1,935GB/s
A40	48GB	696GB/s
T4	16GB	300GB/s

In order to collect some data for those GPUs, we measured the computation time of the pyAMGX solver on all of them for a set scenario. Also, to have an idea of their power consumption for this specific use case, we ran multiple problems in a row for a few minutes, extracted the power consumption of this period of time, and averaged it. Since the solving of a single problem was too quick to extract power consumption data, we projected the average consumption on the short compute time of the solver to get an idea of how much solving one instance of equation costs.

4.3.1 Memory

As expected, GPU memory had a strong influence on our solvers. At first, we wanted to run these tests on the same presets as all the other tests we had done: on 256-sized 3-dimensional grids. However, due to the lower memory capacity of the T4 GPU, the solver would crash during the initialization of the problem, indicating an OutOfMemory exception. This is why we had to downgrade the scenario to a 128-sized 3-dimensional grid problem in order for all GPUs to be able to run on the same conditions.

4.3.2 Computation Time

Now even if it is quite obvious that faster GPUs will perform better, it is interesting to see if the increase in bandwidth has the same scaling as our compute time. Here is what we observed:

GPU	**Bandwidth Increase**	**Solving Speedup**
T4 (baseline)	-	-
A40	232%	282%
A100	645%	477%

Even if the hierarchy is respected, we can see a disparity between the growth of bandwidth and speedup. The speedup of going from a T4 GPU to an A40 exceeds the gain of bandwidth of the exchange. However, even if switching from an A40 to an A100 makes us gain a lot of time, the speedup of this upgrade is significantly lower than the bandwidth increase. These observations can make us question the possibility of GPU computing to reach a limit in its improvements over technological advancements.

4.3.3 Co2e Emission

The environmental impact of different GPU hardware is shown in Table 2. The result presents three NVIDIA GPUs: the A100, A40, and T4, displaying both their hourly emission rates and total emissions for the complete calculation. The emission patterns reveal that the newer, high-performance GPUs (A100 and A40) showed substantially higher hourly emission rates (11.40 and 12.97 grams/hour respectively) compared to the T4 (4.50 grams/hour). However, the A100's superior computational efficiency led to the lowest total emissions (0.0010 grams) for the complete calculation, approximately half that of both the A40 and T4 (0.0020 and 0.0019 grams respectively).

This inverse relationship between hourly emissions and total environmental impact highlights that while more powerful GPUs may have a higher power consumption, their ability to complete calculations more quickly can result in lower overall emissions. This advantage could become even more pronounced in more computationally intensive problems, where the performance gap between the GPUs could be larger.

GPU	Emission of Calculation Time (Gram)	Average Emission (Gram/h)
A100	0.0010	11.4003
A40	0.0020	12.9744
T4	0.0019	4.5018

Table 2: Comparison of emissions for different GPUs.

5 Discussion

Our research into GPU acceleration of Poisson equation solvers revealed several significant insights about performance, efficiency, and environmental impact.

5.1 Implementation Method and Performance

The choice and design of implementation methods proved crucial for both computational efficiency and environmental impact. Our findings demonstrate that simply utilizing GPU hardware is not sufficient; the algorithm must be well-suited to the problem at hand.

The contrasting performance of PyAMGX and CuPy implementations particularly illustrates this point: - PyAMGX, despite its high power consumption, achieved low overall CO2e emissions due to its algorithm's excellent suitability for Poisson equations, resulting in significantly reduced computation times. - CuPy, while also GPU-accelerated, showed less favorable results for our specific problem size. Its higher power consumption was not sufficiently offset by computational speed gains, leading to higher overall emissions. However, this implementation might prove more efficient for larger, more complex problems.

5.2 Benefits of GPU Acceleration

Our study confirmed several key advantages of GPU acceleration when properly implemented:

1. Computational Efficiency - Significantly reduced calculation times - Improved accuracy in residual calculations

2. Environmental Impact - Lower carbon footprint for large-scale computations - Reduced total energy consumption despite higher power usage

6 Conclusion

- Impact of GPU Acceleration: GPU acceleration significantly improves computational time, residual norm, and CO2e emissions when applied to a CPU-based iterative solver.

- Comparison of PyAMG and PyAMGX: The GPU-based PyAMGX outperforms the CPU-based PyAMG, achieving significantly lower computation times and residual norms.

- GPU Performance Differences: The choice of GPU has a notable effect on computation time. The A100 performs best, followed by the A40, with the T4 showing the slowest performance. While CO2e emission differences between GPUs were less pronounced, the A100 still demonstrated the best efficiency in terms of emissions.

This study highlights the benefits of GPU acceleration, showing that carefully selected hardware and methods can lead to substantial improvements in both computational efficiency and environmental impact.