

CHALMERS UNIVERSITY OF TECHNOLOGY

# GPU-ACCELERATED COMPUTATIONAL METHODS USING PYTHON AND CUDA

COMPUTATIONAL FLUID DYNAMICS

*Author:*

Erik HASSELWANDER

Yuhua CHENG

Kyriakos GAVRAS

*Supervisor:*

Prof. Lars DAVIDSON

February 6, 2024



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

### **Abstract**

This report shows that it's possible to achieve a speedup of up to 39 times by accelerating CFD codes written in `Python` using CUDA methods. The report also details the performance of different solvers and the limitations of such acceleration. The report further shows how Nvidias multigrid solver AMGX, using the `Python` interface `pyamgx`, is not able to achieve a speedup when compared to the native GMRES solver in the `CuPy` package.

# 1 Introduction

General purpose graphics processing units have seen a sharp rise in popularity in recent years. Compared to regular CPUs they have hugely superior performance in highly parallelizable workloads. Writing code for GPUs usually requires writing custom kernels, but there exists many packages that allow direct access to functions that can run on GPUs such as the `Python` package `CuPy`.

## 1.1 Computational fluid dynamics

Computational fluid dynamics is the act of solving Navier-Stokes equations numerically. This can be done in many different ways, but this report will target a finite difference model as well as a finite volume model. These models are different ways to discretize the equations to get approximate numerical solutions. To get results that match experimental results large computational meshes are usually required which are slow to resolve. The computations are however highly parallelizable and by using GPGPU to solve the problem faster a higher resolution mesh can be used.

## 1.2 Why GPU Computing

In the past, computers were dominated by single core central processing unit(CPU) appears to execute instruction in a sequential manner but with pipelining hidden at micro-structure level, several instructions can be executed at the same time depends on the available hardware units. The software industry was driven by the faster clock frequency and more hardware within a unit in the past which was predicted by the Moore's law. But this trend has slowed down since around 2004 due to the energy usage and heat dissipation problems. Since we cannot increase the performance of a single core CPU, the trend has turns to encapsulated several cores within one chip. This trend has changed the software and hardware industry tremendously, because previous code works pretty well does not apply at multi-core era anymore [4].

Still, the multi-core design method tries to maintain the speed of each core and execute single instruction as fast as possible. Another direction is trying to execute as many instructions as possible per unit time which is often called the throughput. On type of hardware aims to increase the throughput of instruction execution is called GPGPU. Intel and NVIDIA are two companies represent these two different directions. For comparison, a modern CPU like intel 14th generation CPU has up to 24 cores [1], while the NVIDIA 40 series graphic card can have tens of thousands cores [7]. The huge number of thread cores within GPGPU makes it extremely suitable for scientific programs where computation is often massively huge for example matrix multiplication and numeric simulation.

In all, CPU is good at execute sequential instruction fast, while GPU can execute instructions in parallel with high throughput. For CFD problems we are trying to study in the report, it's inherently suitable for parallel execution.

## 1.3 CUDA Programming model

CUDA is a programming model designed and maintained by NVIDIA for general purpous parallel programming. It allows developers to utilize the parallel hardware of NVIDIA's GPU devices. NVIDIA also develops lots of libraries using CUDA for developers to use as well such as `cuSolver` and `cuSparse`. Each CUDA program is divided into two parts, device code executes on GPU which is often called kernel code, and host code executes on CPU which is responsible for launching device

code and transfer data between CPU and GPU. CUDA's kernel code can be executed in parallel on GPU. GPU also has its own memory, CUDA kernel code can only access data on device memory. So in order for the GPU core to access the data, the CPU needs to copy data from host memory to GPU memory. The data traffic between GPU and CPU is often a huge overhead of CUDA programs. GPU memory is divided into global memory and shared memory. Shared memory is much faster than global memory and can be shared but is much smaller in size than global memory [6].

## 1.4 CuPy

CuPy [2] is a NumPy equivalent designed for GPU. It wraps upon lots of CUDA library functions and provides a Python API. It allows users to execute high performance operations on NVIDIA GPUs easily. In most cases, the operations in NumPy are also provided in CuPy as well. So developers can modify the NumPy program operated on CPU to CuPy program by changing few lines of code. In the `cupyx.scipy.sparse` module, several sparse solvers are provided as well which makes it suitable for CFD problems.

## 1.5 pyamgx

`pyamgx` [8] is a Python wrapper for NVIDIA's AMGX library. AMGX is a distributed multigrid linear solver library on GPU. It includes a flexible solver composition system that allows a user to easily construct complex nested solvers and preconditioners. The library is well suited for implicit unstructured methods. The AMGX library offers optimized methods for massive parallelism, the flexibility to choose how the solvers are constructed, and is accessible through a simple C API that abstracts the parallelism and scale across a single or multiple GPUs using user provided MPI.

## 2 Method

### 2.1 The code we're working with

This report will be limited to accelerating a lid driven cavity flow from Lorena Barbas 12-steps to Navier-Stokes code as well as a boundary layer flow in `pyCALC-RANS` [3]. The code will mainly be accelerated using CuPy and its sparse functionality from `cupyx.scipy.sparse`. Focus will lie on benchmarking different solvers on different hardware.

### 2.2 How we decided what to speed up

To figure out which parts of our code were slow and needed speed up, we used a tool called *cProfile*. This tool showed us details about how much time, different parts of our code were taking.

Here is a simple snippet of the code:

```
1 import cProfile
2 import stats
3
4 profiler = cProfile.Profile()
5 profiler.enable()
6
7 # run the lid-drive cavity flow
8 u, v, p = cavity_flow(nt, nit, u, v, dt, dx, dy, b, p, rho, nu)
9 profiler.disable()
10
11 stats = pstats.Stats(profiler).sort_stats('tottime')
12 stats.print_stats(10)
```

Listing 1: cProfiler

The results helped us see which parts were taking up the most time. We then used insights to decide where to focus our efforts for making the code faster. The graphs we created from this data made it easy to see which functions were slowing things down the most.

### 2.3 How we measured the speed up

We measured the speedup through benchmarking, which involved testing our code with different mesh sizes. We analyzed the performance using the statistics from *pstats*. By examining this data, we identified which functions scale the worst by increasing the mesh sizes. This allowed us to pinpoint areas that needed improvement. Benchmarks are between the M1 chip and the NVIDIA RTX 3090.

### 2.4 GPU implementations

Two GPU implementations have been proposed. The first implementation will only accelerate the solver using `cupyx`, while the second implementation looks to convert the entire codebase to `cupy` instead of `numpy` or `scipy`. For the first implementation, since only the solver is accelerated using CuPy, we need to copy solver input data from CPU to GPU and output data back from GPU to CPU between iterations. This method will introduce lots of memory traffic into the program. For the second implementation, the whole program is converted to using CuPy which means the whole

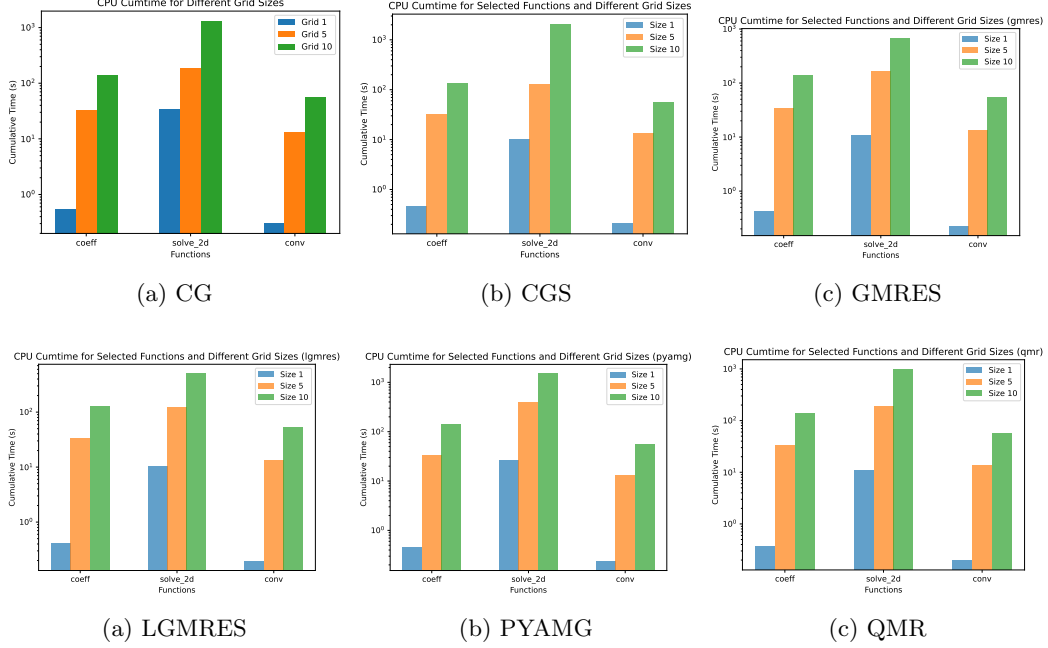


Figure 2

CFD program is run on GPU. The only thing we need to do it copy data from CPU to GPU at the beginning of the program and copy data from GPU to CPU at the end of the program. No memory overhead is introduced in between. For the **pyCALC-RANS** code, we cannot replace Numpy with CuPy directly. Since there are some operations that are used in the code but does not exist in the CuPy library. In order to work around this, we try to change the origin code compatitable with the CuPy operations. For example we use the `cupy.append` to replace the logic of `np.insert`, replace `np.max()` function call with `cupy array max method` call, and replace `np.matrix.flatten` with `cupy array flatten method`. After these fixes, the **pyCALC-RANS** is guaranteed to run GPU entirely.

After we made sure our code can run on NVIDIA GPU entirely, we replace the solver in the program with the solver in CuPy. However, the only solver in CuPy works for CFD program is `gmres`.

## 2.5 pyamgx implementation

The package **pyamgx** was used to allow the AMG solver API to be accessed using **Python**. As the whole code was written using **CuPy** the sparse matrix, the solution vector and the load vector (i.e.  $Ax = b$ ) were all loaded in the GPU memory. By using the **pyamgx** functions `pyamgx.Matrix.upload_CSR` and `pyamgx.vector.upload_raw` it was possible to directly pass the memory pointers and sizes of the arrays to **amgx** and therefore not having any transfer overhead to convert the arrays from the **CuPy** format to the AMG format. Further AMG is a heavily configurable package and a few configurations from the NVIDIA github repository for AMG were used[5].

## 2.6 Tests

Benchmarking the code shows that the majority of time is spent inside the `solve2d`-function. This function solves a finite volume problem on the form  $A\phi = b$ .

The current implementation of `pyCALC-RANS` allows for a variety of solvers that come with the `SciPy` package which run on the CPU.

For the chosen case, lid driven cavity flow, the GPU implementations and the existing CPU implementations will be compared. The planned tests are to run each solver and implementation for meshes of size 120x120, 240x240, 480x480 and 960x960.

## 3 Results

### 3.1 Lorena Barba 12 step code

Fixed overflow issue with smaller time-step for larger mesh sizes in Lorena Barba’s code. Achieved significant speedup using an NVIDIA RTX 3090 GPU compared to the M1 chip CPU. Log scale used for cleaner visualization of exponential increase in execution times.

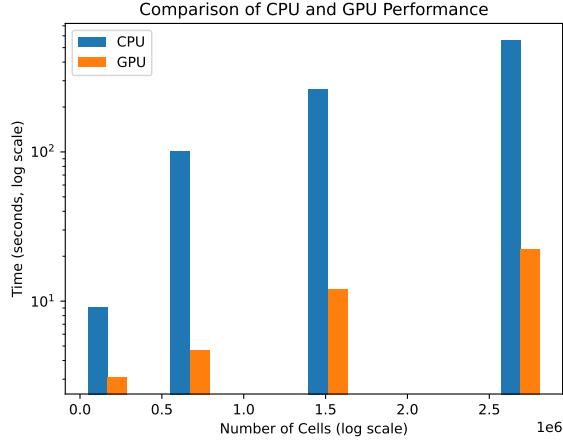


Figure 3: CPU and GPU Performance Comparison

### 3.2 pyCALC-RANS with CuPy

Accelerating the `pyCALC-RANS` code using the python package `CuPy` was possible. The provided default configuration for the `boundary-layer-RANS-kom` was tested for different CPU solvers as well as the GPU based `CuPy` solver `GMRES`, while `CuPy` was also used as a drop-in replacement for `NumPy` for the GPU based runs. The code was ran for 3 iterations and figure 4 shows the total runtime. The runtime of just the `SIMPLEC` iterations was also measured and figure 5 shows the runtime per iteration. A maximum speedup by a factor 39.8 was obtained using the `CuPy` acceleration for a meshsize of 12.5 million cells. The missing datapoints for the `qmr` and `lgmres` solvers are due to the benchmarking machine running out of memory.



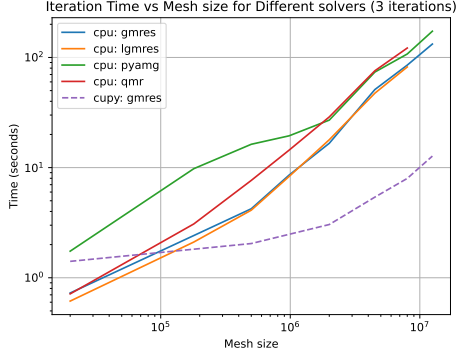


Figure 4: CPU and GPU Performance Comparison

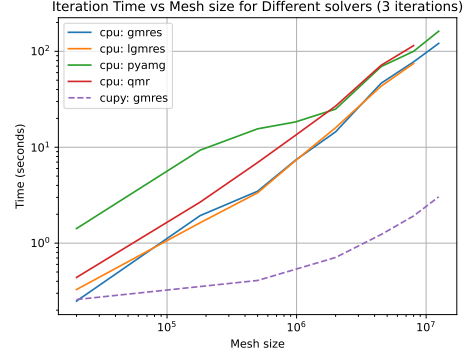


Figure 5: CPU and GPU Performance Comparison

### 3.3 Comparison between cupy and pyamgx solvers

The code was also tested using solvers from the `pyamgx` package for a few select configurations. The tested `pyamgx` configurations were a classical GMRES solver configuration (`GMRES.json[5]`), a multigrid solver using FGMRES as the outer solver (`FGMRES_AGGREGATION.json[5]`) and a multigrid solver using only sweeps of an aggregated multicolor DILU solver (`AGGREGATION_DILU.json[5]`)<sup>1</sup>. The total runtime of these solvers were compared to the GMRES solver in CuPy which is displayed in figure 6. Figure 7 shows the runtime per iteration as in the section above for all tested solvers.

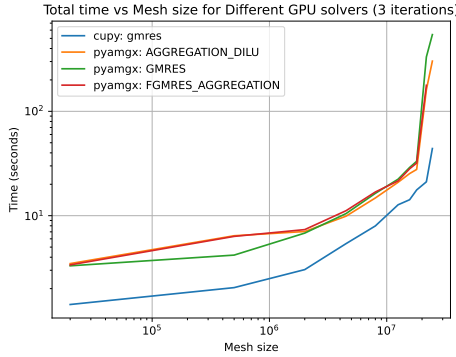


Figure 6: CPU and GPU Performance Comparison

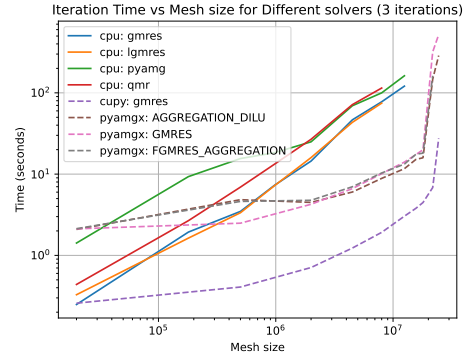


Figure 7: CPU and GPU Performance Comparison

In the figures a sharp rise in computational time can be seen for meshes larger than about 20 million cells. This occurs when the codes memory exceeds the available memory on the GPU (24 gb), which caused the operating system to swap data onto the regular CPU memory giving rise to a large decrease in performance.

<sup>1</sup>The solver configurations can be found at <https://github.com/NVIDIA/AMGX/tree/main/src/configs>.

## 4 Discussion

The results show a significant improvement in the performance of the programs can be achieved by utilizing GPU acceleration, in particular using the `Python` package `CuPy`. This does however not mean that a CFD code will always run faster on a GPU.

The results shows that while the per-iteration time is always faster for all meshsizes, the total runtime is not always larger. GPU based operation can introduce large overhead as the program needs to generate the kernels that run on the GPU from the original python. The runtime of things outside the solvers of the linear systems can also scale badly on the GPU, or not be parallelizable, and it can therefore make sense to do some types of computation on the CPU in a hybrid configuration.

There's also a large overhead associated with transferring data between the GPU and CPU which can cause a code that needs to do some treatment of the data on the CPU between iterations to be significantly slower compared to running the entire code. This also applies to problems that are too large to fit solely inside the GPU memory, such problems could see huge reductions in speed due to having to move data between the two different types of memory, and this is also seen in the results where a too large mesh sees a sharp increase in time due to this phenomena. This also shows in the results where a quick interpolation of the CPU results would beat the GPU results for very large meshes if the machine had enough memory to run such cases on the CPU.

The results also show that the `CuPy` native `GMRES` solver is the fastest, even beatin the `AMGX` solver speed. This could be due to either that the `CuPy` solver is quicker due to being a newer and faster-running solver. It could also be that the `AMGX` solver has to do extra work to convert the `CuPy` arrays to something else on the GPU, even when passing the objects as direct memory pointers. There could also be that the configurations of the `AMGX` solver as well as the convergence criterion in the `SIMPLEC` loop were not ideal for the `AMGX` solver. It is however noteworthy that both GPU solvers scaled similarly, as seen by the similar slopes of the lines in the log-log plots as well as the fact that the `pyamgx` solver still outscaled the CPU based solvers by a large factor.

Another consideration is stability and convergence. Stability was not investigated in this study and only a few iterations of the different solvers were ran. Future studies could look at stability and convergence to further improve the runtime of the programs by tweaking parameters like relaxation factors and convergence criterion in the `SIMPLEC` loop. It would also be possible to get a larger selection of converging solvers by tweaking such parameters. It is however important to point out that the runtime results are still valid due to the same convergence criterion being used for all the solvers and equations in the `SIMPLEC` loop.

## 5 Conclusion

The main conclusion is that it is possible to accelerate both simple and complex CFD codes written in `Python` using `CUDA`. This report shows that large decreases in computational time can be obtained and this report measured an increase of up to 39.8 times decrease in computational time per iteration by moving to GPU based solvers and data structures.

This report also highlights the importance of considering overhead and memory limitations as the main limitants to the usability of such acceleration while also highlighting the need for further studies where stability and convergence is studied in this context.

## References

- [1] Intel Corporation. *Intel Core 14th Gen Desktop Product Brief*. Accessed on: February 6, 2024. 2023. URL: <https://download.intel.com/newsroom/2023/client-computing/Intel-Core-14th-Gen-Desktop-Product-Brief.pdf>.
- [2] CuPy. *CuPy*. Accessed on: February 6, 2024. 2023. URL: <https://cupy.dev/>.
- [3] Lars Davidson. *pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow*. 2023. URL: [https://www.tfd.chalmers.se/~lada/postscript\\_files/py-calc-rans.pdf](https://www.tfd.chalmers.se/~lada/postscript_files/py-calc-rans.pdf).
- [4] Charles E. Leiserson et al. “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” In: *Science* 368.6495 (2020), eaam9744. DOI: 10.1126/science.aam9744. URL: <https://www.science.org/doi/abs/10.1126/science.aam9744>.
- [5] NVIDIA. *Algebraic Multigrid Solver (AmgX) Library*. Accessed on: February 6, 2024. 2017. URL: <https://github.com/NVIDIA/AMGX>.
- [6] NVIDIA. *CUDA C++ Programming Guide*. Accessed on: February 6, 2024. 2023. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [7] NVIDIA. *NVIDIA RTX 4070 family*. Accessed on: February 6, 2024. 2023. URL: <https://www.nvidia.com/sv-se/geforce/graphics-cards/40-series/rtx-4070-family/>.
- [8] pyamgx. *pyamgx*. Accessed on: February 6, 2024. 2023. URL: <https://pyamgx.readthedocs.io/en/latest/>.