CHALMERS UNIVERSITY OF TECHNOLOGY

GPU-ACCELERATED COMPUTATIONAL METHODS USING PYTHON AND CUDA

COMPUTATIONAL FLUID DYNAMICS

Author:

Weilong CHEN

Panagiotis MORAITIS

Johannes HANSSON

Supervisor:

Prof. Lars DAVIDSON

February 12, 2023

Abstract

In this study, we performed a detailed evaluation of the use of GPUs (Graphical Processing Unit) for CFD (Computational Fluid Dynamics) simulations. CFD is a widely used tool for analyzing the flow of fluids and predicting their behavior in a variety of applications, including aerospace, automotive, and chemical engineering.

In recent years, the use of GPUs has become increasingly popular in scientific computing, particularly in the field of CFD. The high parallelization capabilities of GPUs make them well-suited for handling the complex calculations required for CFD simulations.

To evaluate the performance of GPUs in CFD, we implemented CFD algorithms[1] using the Cupy library in Python. CuPy is a library that allows users to perform array operations using a GPU, and is designed to be used with the popular numerical computing library NumPy.

Our results showed that using a GPU with Cupy was able to significantly improve the computation time for the CFD algorithms tested. In some cases, the use of a GPU resulted in a speedup of over 110x compared to using a CPU alone. Investigations also dive into different GPUs, matrix sizes and block sizes.

Overall, our study demonstrates the potential of using GPUs, specifically with the CuPy library in Python, for speeding up CFD simulations. This approach should be considered as a viable alternative to traditional CPU-based methods for those looking to improve the efficiency of their CFD simulations.

Keywords— GPU, CUDA, CFD, CuPy, NumPy, Python, Poisson Solver

Abbreviations

CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
float32	32bit Floating Point Number (Single Precision Data type)
float64	64bit Floating Point Number (Double Precision Data type)
GPU	Graphical Processing Unit
VRAM	Video Random Access Memory

Contents

1 Background knowledge

1.1 CFD info

1.1.1 General Transport Equation

The Navier-Stokes equations are a set of equations that describe the motion of fluids and can be derived from the conservation of mass, momentum and energy principles. There are significant commonalities between the various equations. If we introduce a general variable ϕ, the equations can be written in the following form, which is so-called transport equation for propertyϕ[2].

$$\frac{\partial(\rho\phi)}{\partial t} + div(\rho\phi\mathbf{u}) = div(\tau grad\phi) + S_\phi$$

1.1.2 Explicit and Implicit Scheme

This distinction is commonly used in numerical analysis, where explicit and implicit schemes refer to different ways of solving equations numerically.

For spercific CFD problem, an explicit scheme uses known information from the previous time step to determine the next step, while an implicit scheme uses the unknown information to determine the next step.

In an explicit scheme, the new values of the variables being solved for are calculated using only the known, previously calculated values. This makes explicit schemes relatively simple to implement, but they can be computationally expensive and may not be stable for certain types of equations.

On the other hand, an implicit scheme uses the unknown values to calculate the next step, which can make the scheme more stable and accurate. However, this also means that the equations must be solved iteratively, which can be more computationally complex.

Overall, the choice of whether to use an explicit or implicit scheme depends on the specific problem being solved and the desired trade-offs between accuracy, stability, and computational complexity.

For our problem, we have two explicit equations and one implicit Poisson equation, which needs iterations to solve and consumes most of the time. The Gauss-Seidel method is an iterative method for solving systems of linear equations. It is a variant of the Jacobi method and is considered to be an "implicit" method. This is because it uses the current estimates of the solution at each iteration to update the solution for the next iteration.

1.1.3   Pressure Velocity Coupling

From the previous general transport equation, we often assume ϕ is the unknown parameter, while the velocity field is given before solving the equations. But the first step is always to get the correct velocity and pressure field.

For the pure momentum equations where ϕ are u and v, the source term will be the gradient of pressure., which will be derived through continuity equation. Therefore, we need to solve momentum and continuity equation at the same time. During this process, we achieve the so-called pressure-velocity coupling.

In our code, it's an unsteady problem, the scheme is showed below. As you will see, we need to use a linear solver or iterations to get the pressure field first and then iterate to get the next time-step velocity field across the whole area.

1.1.4   Poisson Equation

For the discretized form poisson equation, each node depends on the four nodes around it, which will form the so-called sparse matrix. It's a penta diagonal matrix. As we will see later, the Poisson solver takes more than 90% of total running time.

The variety of methods for solving Poisson equation[3] include:

- Direct methods

    - Gaussian Elimination
    - LU decomposition method

- Iterative methods

    - Mesh relaxation methods
        * Jacobi
        * Gauss-Seidel
        * Successive over-relaxation method(SOR)
        * Alternating directions implicit(ADI) method
    - Matrix methods
        * Thomas tridiagonal form
        * Sparse matrix methods
        * Conjugate Gradient method
        * Multi-Grid method

In our report, we mainly use the Gauss-Seidel method which is provided with the code. It's reasonable to try different solvers first before parallelization. However, since this is a GPU course, we will focus more on GPU acceleration.

1.2 Why GPU

GPUs are often used in scientific applications because they are able to perform many calculations in parallel, which makes them much faster than CPUs for certain types of computational problem. This is because GPUs are designed to handle many threads (small programs executing simultaneously) at the same time, which is useful for tasks that involve a large amount of data and can be broken down into many smaller, independent calculations.

For this reason, GPUs are often used to accelerate the training of machine learning models and the execution of other data-intensive tasks. They can significantly reduce the time it takes to perform these tasks, making them more practical to run on large datasets.

However, CPUs are still important for many other types of tasks, especially those that require more complex control flow or involve interacting with other systems. In addition, CPUs are typically more flexible than GPUs and can perform a wider range of tasks, so they are still an important component of most computing systems.

1.3 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs. This platform allows developers to leverage the power of the GPU special architecture to perform complex calculations and operations in parallel, leading to a significant improvements in performance and speed. CUDA is widely used in a variety of fields, including scientific computing, machine learning, image and video processing, and gaming.

1.3.1 CUDA Architecture

In CUDA, the host is the CPU and the device is the GPU. The host is responsible for managing the overall operation of the system, while the device is responsible for executing parallel tasks.

CUDA uses the concept of threads to divide a computation into smaller, independent parts that can be executed concurrently. A thread is a sequence of instructions executed by the device. In CUDA, threads are organized into thread blocks, which are groups of threads that can be executed on the same GPU processing core. Thead blocks are organized into grids, which are groups of thread blocks that can be executed on the GPU.

1.3.2 CuPy Library

CuPy is a library for GPU-accelerated computing in Python that provides a NumPy-like interface. It was developed as a way to provide users with the ability to perform fast numerical computations on their GPUs without having to learn a new API or write CUDA code directly.

CuPy is implemented in C++ and CUDA, and it provides functions and classes that mirror those in NumPy. This allows users who are familiar with NumPy to easily transition to using CuPy, as the syntax and functionality are largely the same.

CuPy is designed to be highly performant, making it suitable for use in a wide range of applications that require fast numerical computations. It can be used to accelerate machine learning algorithms, scientific simulations, and other data-intensive tasks.

One of the key advantages of CuPy is that it can take advantage of the parallel processing capabilities of the GPU to perform calculations much faster than would be possible on the CPU alone. This is especially useful for tasks that involve processing large amounts of data, such as training machine learning models on large datasets.

CuPy also provides many of the same functions and features as NumPy, such as support for array creation, indexing, slicing, and mathematical operations. It also includes additional functionality for working with GPU arrays, such as the ability to copy data between the CPU and GPU and the ability to perform reductions and broadcasts on GPU arrays.

Overall, CuPy is a powerful tool for GPU-accelerated computing in Python that allows users to perform fast numerical computations and leverage the power of their GPUs without having to write CUDA code directly.

2 Problem description - CFD code

2.1 General Problem Description

We use the example Navier Stokes code[4]. It is a cavity flow problem. The domain is rectangle. We have three varibles to solve, which are u-velocity, v-velocity and pressure. We have three equations to discretize, which are two momentum equation and one poisson equation derived from continuity equation.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = -\rho \left(\frac{\partial u}{\partial x} \frac{\partial u}{\partial x} + 2\frac{\partial u}{\partial y} \frac{\partial v}{\partial x} + \frac{\partial v}{\partial y} \frac{\partial v}{\partial y} \right)$$

The initial condition is $u, v, p = 0$ everywhere, and the boundary conditions are:

- $u = 1$ at $y = 2$ (the "lid");

- $u, v = 0$ on the other boundaries;

- $\frac{\partial p}{\partial y} = 0$ at $y = 0$;

- $p = 0$ at $y = 2$

- $\frac{\partial p}{\partial x} = 0$ at $x = 0, 2$

2.2 Discretization Scheme

The momentum equation in the u direction:

$$u_{i,j}^{n+1} = u_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x} \left(u_{i,j}^n - u_{i-1,j}^n \right) - v_{i,j}^n \frac{\Delta t}{\Delta y} \left(u_{i,j}^n - u_{i,j-1}^n \right) - \frac{\Delta t}{\rho 2 \Delta x} \left(p_{i+1,j}^n - p_{i-1,j}^n \right) +$$
$$\nu \left(\frac{\Delta t}{\Delta x^2} \left(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n \right) + \frac{\Delta t}{\Delta y^2} \left(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n \right) \right)$$

The momentum equation in the v direction:

$$v_{i,j}^{n+1} = v_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x} \left(v_{i,j}^n - v_{i-1,j}^n \right) - v_{i,j}^n \frac{\Delta t}{\Delta y} \left(v_{i,j}^n - v_{i,j-1}^n \right)) - \frac{\Delta t}{\rho 2 \Delta y} \left(p_{i,j+1}^n - p_{i,j-1}^n \right)$$
$$+ \nu \left(\frac{\Delta t}{\Delta x^2} \left(v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n \right) + \frac{\Delta t}{\Delta y^2} \left(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n \right) \right)$$

For above two equations, as we talk before, they are both explicit scheme, which means the next time step exactly depends on the previous time step parameters. But for the Poisson equation below, dut to the implicit nature of the Gauss-Seidel method also has some drawbacks. For example, it requires the solution of a system of linear equations at each iteration, which can be computationally expensive.

$$p_{i,j}^n = \frac{\left(p_{i+1,j}^n + p_{i-1,j}^n \right) \Delta y^2 + \left(p_{i,j+1}^n + p_{i,j-1}^n \right) \Delta x^2}{2 \left(\Delta x^2 + \Delta y^2 \right)} - \frac{\rho \Delta x^2 \Delta y^2}{2 \left(\Delta x^2 + \Delta y^2 \right)} \times$$
$$[\frac{1}{\Delta t} (\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}) - \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}$$
$$- 2\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} - \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}]$$

3 Performance improving methods

One of the key goals in this project is to make our CFD solver as fast as possible. This will be achieved mainly through GPU acceleration. Some degree of code optimization will also be considered, but is not part of the main focus of this report.

This section begins with an evaluation of the current state of the code via performance profiling. Then, we look at the main performance bottleneck identified via the profiling, the Poisson solver, and how this calculation is accelerated using CuPy GPU acceleration.

3.1 Performance Profiling

In many types of simulation software it is common to have one or a few operations that take a majority of the simulation run time. These operations then become a major limiting factor when complexity of the simulated system increases. It is generally desirable to increase the computational power of one's computational tools, because this allows the study of even more complex systems.

When we want to increase the computational power of any software tool, an important first step is to identify performance bottlenecks. These bottlenecks are often in the form of functions that take a long time to complete. There are specialized development tools that help identify such bottlenecks, and in this project we have, among others, used the Python tool line-profiler[5] for line-by-line profiling. This tool gives us fine-grained information about the code that helps us decide where to focus our optimization efforts. The best performance return on investment of development time is often given when effort is dedicated to improving the slowest part of a program.

For the case studied in our report, the line-by-line profiling indicated that about 90% of the simulation run time was spent on solving the Poisson equation for the pressure-velocity coupling. From this result it is clear that the Poisson solver is going to be a major factor in the performance of our entire CFD solver. For example, if we manage to improve performance in this function by a factor two, then this will translate into an almost factor two in increased performance for the CFD solver as a whole.

3.2 Performance of the Poisson equation solver

Since we know that the Poisson equation solver represents about 90% of the simulation run time, we spend extra effort studying this function. We see that the finite difference discretization scheme that is implemented can be computed in parallel, so called massive parallelism. This means that this code is very well-suited for running on GPUs. The original code is written in Python, making heavy use of the NumPy library. In order to run the code on a GPU, we rewrite parts of the implementation so that it

uses the CuPy library instead of NumPy. The re-write is fairly simple and provides an almost drop-in replacement to NumPy which is able to run on the GPU. This version of the code runs all calculations on the GPU, the CPU is used mostly for organizing the computations and synchronizing results back and forth from main computer memory.

In order to increase computational performance beyond the initial CuPy implementation, we sought to optimize the solver by looking at the operations performed behind the scenes in the CuPy code. GPU performance profiling using the Nvidia tool nvprof indicated that the CuPy code launches a large number of CUDA kernels. We assume that CuPy launches a CUDA kernel for each individual mathematical operation. This hints to a possible inefficiency in the code since it takes some time to launch each kernel. Also, if the kernels are doing a single mathematical operation each, such as addition or multiplication, then we lose the ability to optimize memory access patterns and memory locality for even greater performance.

Finally, some additional performance was extracted by removing unnecessary operations such as redundant memory allocations and data copying.

3.3   Speedup affected by various factors

The overall speedup of a computation involving matrices can be affected by various factors such as the size of the matrices, the block size used for parallelization, and the specific GPU processor model being used. We run different tasks considering one changeable variable while others are fixed.

3.3.1   Speedup as a function of matrix size

An important performance metric is how the simulation run time scales with the size of the computational grid. In tables 1 and 2 we present run time measurements as a function of grid size for a simulation of 10 time steps, each with 10 internal grid update steps for the Poisson solver. In this simulation, the computational grid is a structured two-dimensional rectangular grid with "matrix dimension" grid points in each dimension. In other words, a matrix dimension of 128 means we have a 128×128 matrix as our grid. Simulation run times are reported for both the CPU solver and the corresponding GPU solver. A factor speedup of the GPU solver relative to the CPU solver is also presented. Note that in table 1 the two solvers use single-precision floating point numbers (also referred to as float32) and in table 2 the solvers use double-precision floating point numbers (float64). Profiling runs were performed on a Chalmers-provided StuDAT computer with an Intel Core i5-9500 CPU and an NVIDIA GeForce RTX 2060 GPU. Also note that the grid sizes are powers of two, with the upper limit of 8192 chosen due to restrictions on VRAM size on this particular GPU model.

3.3.2   Speedup as a function of block size

In table 3, we conducted a simulation study to investigate the effect of different block sizes on the run time of a Poisson equation solver. We tested block sizes ranging from 2 to 1024, and measured the speedup performance as a function of block size. The simulation consisted of 10 time steps, each with 10 internal grid update steps for the Poisson solver. The experiments were performed on a StuDAT Linux computer with an Intel(R) Core(TM) i5-9100 CPU and an NVIDIA GeForce GTX 1060 GPU. In this simulation, the block size represents the number of threads in each block. For our problem, if the block size is n, it means we have n^2 threads in each block. The results of our simulation showed that as the block sizes increased from 2 to 32, the speedup performance also increased, from 4 to more than 40. However, on larger block sizes, the speedup tended to be stable, indicating that there is a trade-off between the block size and the speedup performance.

It is worth noting that, in general, the larger the block size, the more memory is required. Therefore, it is important to carefully consider the trade-offs between block size, speedup performance, and computational resources when choosing the block size for a given problem.

3.3.3   Speedup as a function of GPU processor model

In table 4 we present a comparison of computational performance, measured by execution times given two GPU processors of different generations. Also, in section 3.4, we compare their execution times to the CPU performance. Here, again we are using the StuDAT computers including the same CPU Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz. As for the GPU we are using a GTX 1060 with 6GB VRAM and next generation RTX2060 with the same amount of VRAM memory. In addition, we use the Alvis cluster to get access to the Tesla T4. From the result, the general trend shows that the two consumer Nvidia models GTX1060 and RTX2060 have quite powerful performance. However, when we increase the grid size to 2048 we have noticeable performance gains of more than 3 seconds. While with the Tesla T4 card we can observe noticeable differences in performance earlier from the grid size of 1024 and then on bigger grid sizes the performance gap is not big comparing with the consumer based GPUs. Especially, the RXT2060 scores the best performance as it is optimized in computations in single precision float. On the other side, when we compare with the CPU, as we noticed earlier the performance is equivalent for grid sizes up to 1024 and then the GPUs have great timing benefits.

3.4   Kernel function code

There are several ways of approaching GPU acceleration in Python programming. The two main alternatives are Numba and CuPy. Below we describe two implementations of our CFD solver, one using Numba and the other using CuPy.

For numba[6] code, the kernel first uses the cuda.grid(2) function to identify the indices of the block and thread that is currently executing the kernel. It then uses an if statement to check if the current thread's indices are within the bounds of the input array p. If the indices are within bounds, the kernel enters a for loop that iterates a specified number of times (indicated by the variable nit). Within the for loop, the kernel uses a series of if statements to check if the current thread's indices are located on the edge of the input array. If the thread is on the edge, the kernel assigns specific values to the corresponding element of the input array p. In the next step kernel applies the mathematical equation on the array p, the equation is a Poisson equation which is used to calculate the pressure. It is a partial differential equation that describes the distribution of a physical quantity such as pressure, temperature, or fluid flow in a given region. Finally, the kernel uses the cuda.syncthreads() function to synchronize all threads after one iteration of the for loop. This ensures that all threads have completed the current iteration before moving on to the next iteration.

In cupy code, we give kernel function in another form which is much more straight forward than Numba. It's a C-language type script. The kernels, $b\_equation, p\_equation,$ $u\_equation, v\_equation$, are defined using the $cupy.ElementwiseKernel$ method which takes in several arguments. The first argument is the input parameter signature in the form of $Tparam1, Tparam2 ..., TparamN$. It defines the types and names of the input parameters. The second argument is the output parameter signature in the form of $Toutput$. It defines the type and name of the output parameter. The third argument is the kernel function in the form of a string $expression$. It defines the operation to perform on the input parameters and assigns the result to the output parameter. The fourth argument is the kernel name in the form of a string $kernel\_name$. The fifth argument is $options = cuda\_compile\_flags$ which is optional and tells the compiler to use certain flags when compiling the kernel.

For our case with mainly matrix computation, Cupy would be the better choice as it provides a NumPy-like interface for array computations on CUDA-enabled GPUs. Cupy can be used as a drop-in replacement for NumPy and allows to perform matrix computations on a GPU. This can greatly accelerate performance for large matrices and complex computations.

Numba, on the other hand, is a just-in-time (JIT) compiler for Python code and does not have a specific matrix operation, it can specify certain parts of the code should be executed on the GPU, but it doesn't provide a NumPy-like interface.

Compared to Cupy code, the numba is hard to implement because the difficuly of

defining the kernel for all functions. It's also slower if we only use numba for poisson kernel function, so we choose Cupy kernel for our problem.

```
@cuda.jit
def pressure_poisson(p, dx, dy, b):
    i,j = cuda.grid(2)
    #indentify the index of block and thread
    if i < p.shape[0] and j < p.shape[1]:
        for q in range(nit):
            if i>0 and i<p.shape[0]-1 and j>0 and j<p.shape[0]-1:
                #for each thread, we assign same operations.
                p[i,j] = ((p[i+1,j]+p[i-1,j]) * dy**2 +(p[i,j+1]+p
    [i,j-1]) * dx**2)/(2 * (dx**2 + dy**2)) - dx**2 *dy**2/ (2*(dx
    **2+dy**2))*b[i,j]
            if i == p.shape[0]-1:
                for x in range(p.shape[1]):
                    p[x,i] = p[x,i-1]
            if j == 0:
                for y in range(p.shape[1]):
                    p[j,y] = p[j+1,y]
            if i == 0:
                for z in range(p.shape[1]):
                    p[z,i] = p[z,i+1]
            if j == p.shape[1]-1:
                for m in range(p.shape[1]):
                    p[j,m] = 0
            cuda.syncthreads()
            #For next iteration, sync all threads after one
    iteration.
```

Listing 1: Pressure Poisson using numba

```
b_equation =cupy.ElementwiseKernel ('T ul, T ur, T uu, T ud, T vl,
     T vr, T vu, T vd, T rho, T dt, T dx, T dy',#input parameters
    'T b',#output parameters
    'b = (rho * (1 / dt * ((ur - ul) /(2 * dx) + (vd - vu) / (2 *
    dy)) - ((ur - ul) / (2 * dx))*((ur - ul) / (2 * dx)) - 2 * ((ud
     - uu) / (2 * dy) * (vr - vl) / (2 * dx))- ((vd - vu) / (2 * dy
    ))*((vd - vu) / (2 * dy))))',#execution function
    'b_equation',#function name
    options=cuda_compile_flags)


p_equation = cupy.ElementwiseKernel('T pnu, T pnd, T pnl, T pnr, T
     dx, T dy, T b', 'T p',
```

```
9       'p = (((pnr + pnl) * dy*dy +  (pnd + pnu) * dx*dx) / (2 * (dx*
    dx + dy*dy)) - dx*dx * dy*dy / (2 * (dx*dx + dy*dy)) * b)',
10      'p_equation', options=cuda_compile_flags)
11
12 u_equation =cupy.ElementwiseKernel ('T un, T vn, T unl, T unr, T
    unu, T und, T pl, T pr, T nu, T rho, T dt, T dx, T dy', 'T u',
13      'u = (un-un * dt / dx * (un - unl) - vn * dt / dy * (un - unu)
     - dt / (2 * rho * dx) * (pr - pl) + nu * (dt / dx*dx * (unr -
    2 * un + unl) + dt / dy*dy * (und - 2 * un + unu)))',
14      'u_equation', options=cuda_compile_flags)
15
16 v_equation =cupy.ElementwiseKernel ('T vn, T un, T vnl, T vnr, T
    vnu, T vnd, T pd, T pu, T nu, T rho, T dt, T dx, T dy', 'T v',
17      'v = (vn - un * dt / dx * (vn - vnl) - vn * dt / dy * (vn -
    vnu) - dt / (2 * rho * dy) * (pd - pu) + nu * (dt / dx*dx * (
    vnr - 2 * vn + vnl) + dt / dy*dy * (vnd - 2 * vn + vnu)))',
18      'v_equation', options=cuda_compile_flags)
```

Listing 2: Cupy Kernel Function

Table 1: Simulation run time as a function of computational grid size for both the CPU and GPU solvers. The simulation contained 10 time steps and 10 internal steps for the Poisson solver for each time step. In this measurement, the two solvers used single-precision floating point numbers (float32). Results were measured on a Studat computer with Intel Core i5-9500 CPU and NVIDIA GeForce RTX 2060 GPU. The block size used was 128.

Matrix dimension	CPU time [s]	GPU time [s]	Factor speedup
128	0.0093	0.0155	0.5983
256	0.0268	0.0154	1.7385
512	0.1021	0.0158	6.4609
1024	0.5359	0.0160	33.5012
2048	2.8775	0.0418	68.7851
4096	12.1503	0.1583	76.7547
8192	47.4050	0.6207	76.3677

4   Discussion

Due to the nature of how GPUs operate, we expect CPUs to outperform GPUs for very small problem sizes. This comes from the fact that GPUs need a few extra setup

Table 2: Equivalent of table 1, but using double-precision floating point numbers (float64) instead.

Matrix dimension	CPU time [s]	GPU time [s]	Factor speedup
128	0.0161	0.0157	1.0246
256	0.0515	0.0158	3.2495
512	0.2226	0.0254	8.7664
1024	1.2647	0.0947	13.3479
2048	5.7884	0.3134	18.4685
4096	24.2251	1.1702	20.7015
8192	95.5761	4.5081	21.2010

Table 3: i5-9100 and NVIDIA GeForce GTX 1060 GPU, use different block sizes ranging from 2 to 1024. Matrix size is 4096.

Block size	CPU time [s]	GPU time [s]	Factor speedup
2	13.3661	2.1935	6.0935
4	13.2612	1.2404	10.6911
8	13.2649	0.6845	19.3790
16	13.4899	0.4329	31.1617
32	13.2649	0.3199	41.4658
64	13.2634	0.3095	42.8543
128	13.4924	0.3010	44.2068
256	13.2974	0.3008	44.2068
512	13.2589	0.3052	43.4433
1024	13.5870	0.3135	43.3397

Table 4: Different GPU processors performance comparison with matrix size is 4096. Speedup related to the fastest gpu architecture at the given grid - using single precision float numbers.

Grid	RTX2060	GTX1060	CPU	Tesla T4	Factor Speedup
128	0.0154	0.0171	0.0092	0.0093	0.98
256	0.0154	0.0172	0.0268	0.0173	1.74
512	0.0163	0.0174	0.1021	0.0196	6,26
1024	0.0162	0.0224	0.5358	0.0180	39.48
2048	0.0419	0.0804	2.8775	0.0569	68,67
4096	0.1569	0.3042	12.1503	0.2477	77.43
8192	0.6116	1.1957	47.4050	1.126	77.50

Speedup versus different Matrix sizes
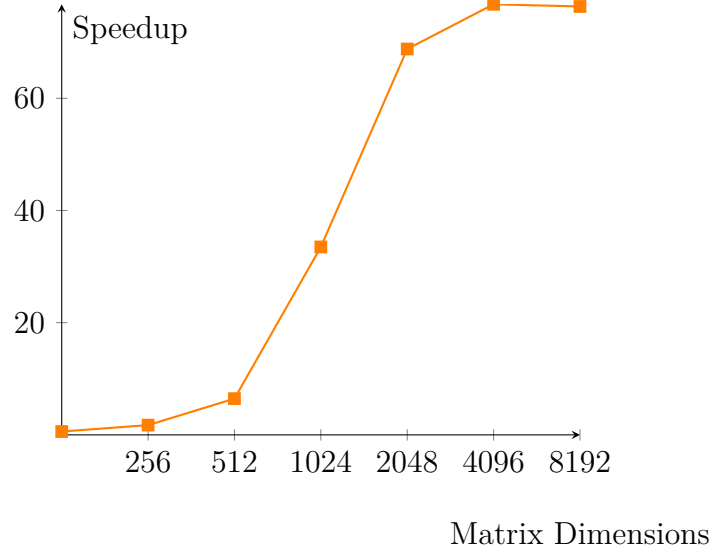
Figure 1: Speedup vs different Matrix sizes

tasks before they start a computation that CPUs do not need, for example copying data and launching computational kernels. These extra tasks take time and delay the start of the computation. If the computation itself is small, then the setup could take significantly more time than the computation, leading to slow response times. This is exactly what we see in, for example, the 128×128 case in table 1. The CPU finishes more quickly than the GPU, whereas the GPU is faster in all other cases. It is also interesting to note that the GPU run time in this measurement is practically constant for matrix dimensions between 128 and 1024, a factor 64 increase in computational
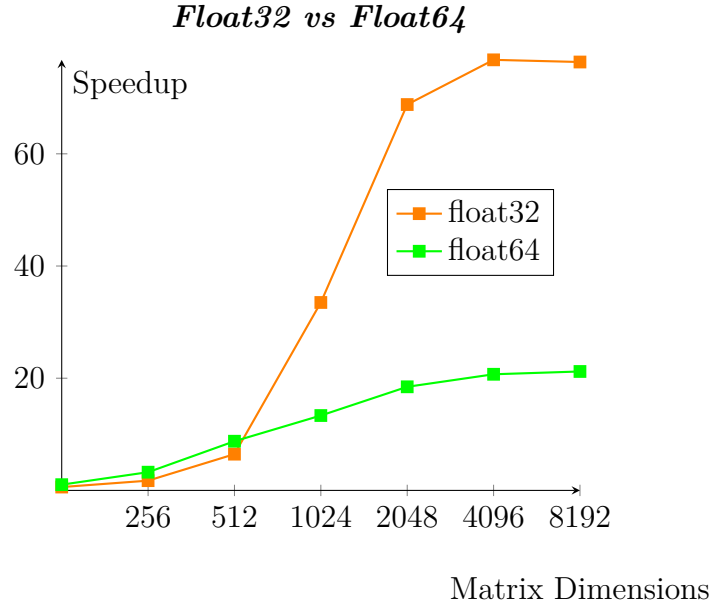
Float32 vs Float64

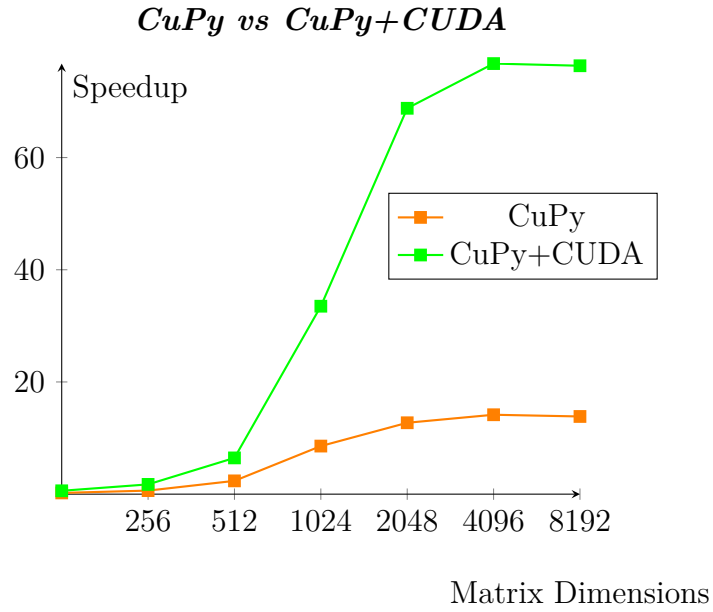*Figure 2: Comparison between float32 and float64*

CuPy vs CuPy+CUDA

Figure 3: Comparison of speedup for the pure CuPy implementation and the hybrid CuPy+CUDA version. The Hybrid version has purpose-written CUDA kernels in C++ for performance-critical code segments. The results clearly show that the hybrid version is significantly faster than the pure CuPy implementation.
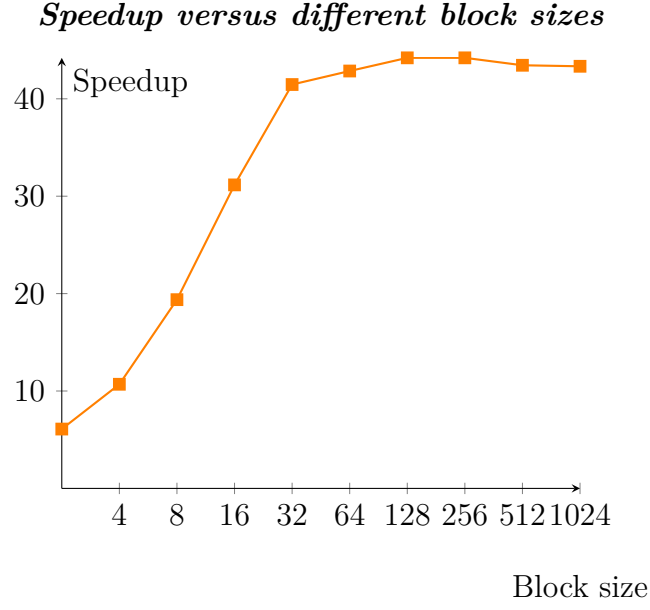
17

Figure 4: Comparison between different block sizes
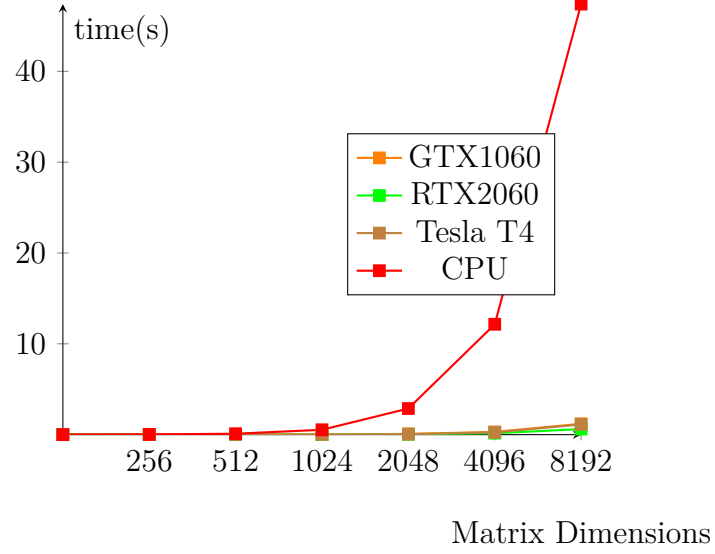

Figure 5: Executing time(s) versus different GPU processors

requirements. This suggests that the constant setup time makes up a large fraction of the computational time. This, in turn, is indicative of underutilization of the GPU due to too small problem sizes.
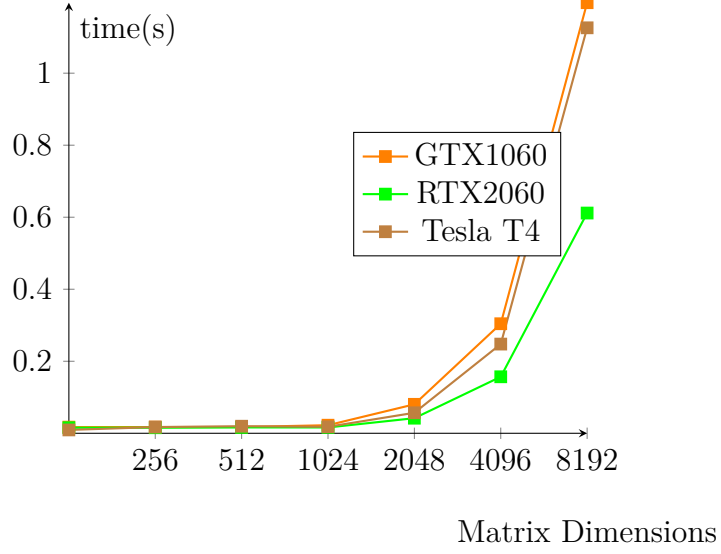
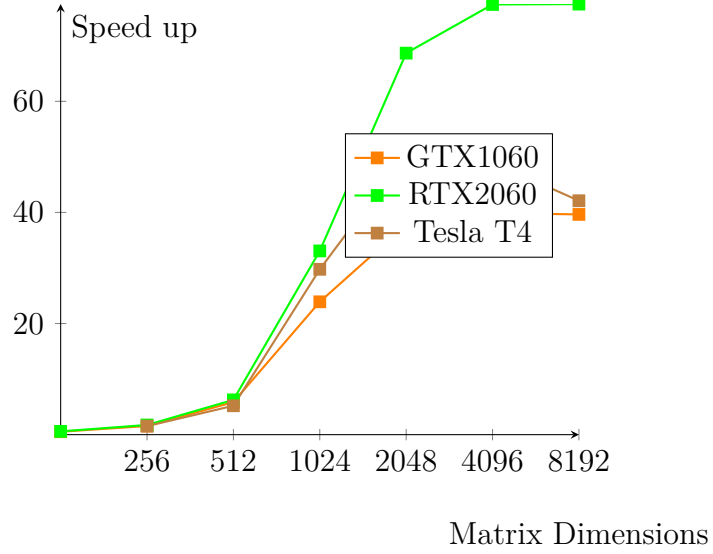*Figure 6: Executing time(s) versus different GPU*

Figure 7: Speed up versus different GPU

That GPUs outperform CPUs on larger problem sizes is also to be expected, especially for embarrassingly parallel workloads such as the CFD solver studied in this project. GPUs generally have much more computational power than CPUs, and will

19

therefore outperform CPUs as long as the setup time is small in comparison to the time for actual computations. Setup time is generally constant in time, so it becomes insignificant as the size of the computational problem increases. In other words, the bigger the grid, the better the speedup when using GPUs, at least up to a point. Performance again decreases for very large problem sizes, where the size of the GPU memory poses a limiting factor. For these very large cases, performance is very dependent on how the memory is used. It could be that GPUs still offer massive performance gains, or it could be that CPUs once again become faster than GPUs. It is difficult to be more precise without looking at a particular application. In this report we limit ourselves to problem sizes that fit in device memory, so we have not studied the performance scaling of very large grid sizes. We propose that future studies should look at how these memory issues can be dealt with to minimize performance penalties as the grid size increases.

If we compare tables 1 and 2 we see that the CPU takes about twice as much time to finish the simulation when using float64 as compared to when it uses float32. This factor two difference in performance could be due to the fact that the float64 requires twice as much data bandwidth from main memory. It could also be due to the fact that CPUs often have a factor two better arithmetic performance when using float32, in part due to vectorized processing of several data elements in a single clock cycle. The GPU, on the other hand, performs much better with float32 than float64. This is likely due to the fact that consumer-grade Nvidia GPUs are specialized on float32 computations, at the expense of slower float64 ones.

In table 3 we see that block size is a relatively important consideration when running computations on the GPU. In particular, it is important to have a sufficiently large block size to achieve good performance on the GPU. From the table we see that block sizes of 16 and below significantly reduce performance. Block sizes of 64 and above have approximately the same performance, indicating that we need at least this block size to achieve good performance. This knowledge has implications when choosing an appropriate grid size. Ideally, we would like the number of grid points to be an integer factor of at least 32 or maybe 64 for optimal performance. Larger block sizes can also be used, but might result in sub-optimal performance when the grid size does not perfectly match the block size up to an integer factor. Also, notice that CPU run time is essentially constant in all cases. Block size does not affect CPU time because this size parameter is GPU-specific.

In figure 3 we present the speedup as a function of grid dimension for two different versions of the solver, the initial pure CuPy version and the CuPy+CUDA hybrid. The large speedup of the hybrid version is attributed to more efficient use of memory locality as compared to the pure CuPy version. We believe that the lower performance of the pure CuPy version is due to each mathematical operation being performed by separate kernels. This hinders the use of memory locality, leading to lower computational performance.

The purpose-written CUDA kernels are able to achieve much better use of memory locality, which leads to higher performance. The use of a CuPy+CUDA hybrid leads to some increase in code complexity, but we believe that it is worth it due to the significant performance increases.

Finally, table 4 shows the run time required to simulate our example CFD case, both for the CPU solver and the GPU solver running on different GPUs. It is clear that the GPUs are significantly faster than the CPU, and it is also clear that there is clear difference between different models of GPUs. The fastest model is the Nvidia RTX2060, then the Tesla T4 by small margin comparing to the GTX1060. This difference in performance is attributed to the different models having varying computational capabilities.

5   Limitations and Future Work

In the limitations and future work section of this report, several areas for potential improvement were identified. One potential area for improvement is the comparison of the performance of utilizing a single GPU versus multiple GPUs in parallel. This could potentially enhance the overall performance of the simulation by distributing the computation across multiple GPUs. Furthermore, expanding the scope of the simulation from the current cavity flow problem to more complex flow problems such as combustion reaction flow could increase the applicability of the simulation to a wider range of problems and provide new insights into various areas of research. Additionally, compatibility between the CPU and GPU should be taken into consideration when designing the simulation in order to optimize performance. Implementing the simulation using CUDA in C++, as opposed to Python, could also potentially enhance performance by utilizing the lower-level programming capabilities and optimized libraries provided by CUDA. Furthermore, experimenting with different sparse matrix solvers may also improve the performance of the simulation. Additionally, optimization with different data structures and algorithms, utilization of newer GPU architectures and technologies, scalability to larger and more complex problems and use of machine learning techniques can also be considered as potential areas for future work.

6   Conclusion

The Navier Stokes code used in this work is a cavity flow problem that involves solving three variables (u-velocity, v-velocity, and pressure) using three equations (two momentum equations and one Poisson equation). The line-by-line profiling showed that about 90% of the simulation run time was spent on solving the Poisson equation for

pressure-velocity coupling, making it a major factor in the performance of the entire CFD solver. To improve performance, the code was rewritten using the CuPy library to allow for massive parallelism on GPUs. Additional performance was also gained by optimizing the solver through the use of purpose-written CUDA kernels and the removal of unnecessary copy operations in the original NumPy code.

The GPU acceleration and code optimizations lead to an increase in computational performance of more than a factor 70 for the single-precision case when compared to the CPU solver. This facilitates the study of even larger physical systems, or better statistical analyses on smaller such systems. Neither of these studies would be practical using the original CPU code. Our results also indicate that it is important to choose the block size parameter correctly since performance is dependent on this parameter being at least 32, but preferably 128 or greater. Additionally, it is important to keep in mind that the speedups are, to a large degree, hardware-dependent. For example, we found that the best computational performance was achieved when using the RTX2060, out of the GPU models we benchmarked. However, newer GPUs and CPUs will change the relative effectiveness of GPU acceleration.

References

[1] Lars Davidson. *pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow.* Tech. rep.

[2] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method.* Pearson education, 2007.

[3] Lawrence C Evans. *Partial differential equations.* Vol. 19. American Mathematical Soc., 2010.

[4] LA Barba. "CFD python: 12 steps to navier-stokes". In: *Blog post http://lorenabarba. com/blog/cfd-python-12-steps-to-navier-stokes* (2013).

[5] Robert Kern et al. *line_profiler.* Github repository. URL: `https://github. com/pyutils/line_profiler` (visited on 01/05/2023).

[6] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC.* 2015, pp. 1–6.

A Statement of contributions

A.1 Johannes

a. Profiled the original Navier-Stokes solver written in NumPy

b. Studied different ways of solving the Poisson equation efficiently

c. Experimented using different solvers for the Poisson equation. Tried several sparse, iterative and factorization solution methods.

d. Implemented CFD solver that is able to use both the CuPy and NumPy frameworks, depending on user preference

e. Implemented performance-critical parts of the solver in raw CUDA C++ for significantly better computational performance.

f. Generated results for speedup when using single and double precision for the RTX2060 GPU

g. Generated tables of the above results for use in the report

h. Wrote sections in the report about performance profiling, performance of the original NumPy code and the move to CuPy, results for speedup as a function of grid size when using single and double precision, as well as the discussion. Wrote part of the conclusions.

i. General editing of the report.

j. Prepared and delivered presentation. Here I was responsible for presenting the performance profiling, implementation using CuPy, performance improvement using pure CuPy and the CuPy+CUDA hybrid, programming details about said hybrid, and a quick overview of the achieved speedup.

A.2 Panos

a. Experimented using cupy and numba as also multi cpu architecture

b. Experimented using different gpu processors from studat computers to benchmark time scores

c. Experimented and benchmarked the server based gpu capability of Alvis and Vera clusters

d. Wrote the background knowledge for GPU/CUDA/CUPY.

e. Spend time with different python profilers to profile the given problem and find the most critical part.

f. Wrote some parts of the experiment sections, including the performance using different gpu processors and the relevant graphs.

g. Contributed in the presentation slides. Worked on the slides with the experiments.

h. Started experimenting with the multi-gpu scheme on the Alvis cluster without having

time to get some initial results (future work).

A.3 Weilong

a. Theory related to CFD, Poisson and discretization scheme.

b. Try different Poisson solvers and benchmark before mid-term presentation.

c. Implement CFD code using Numba. Define CUDA kernel in Numba and benchmark. However, slower than Cupy version. Analyse the reason. Compare the difference between the numba code and Cupy code.

d. Experimented the Cupy code with different block sizes and draw the corresponding table, graph.

e. Experimented Cupy code on Vera2 and fix Linux problems.

f. All content in report from title page to page 5, all content in Part two.

g. All content in 3.3.2, 3.4 and all content in Part 5, Introduction part of 3.3.

h. Python code listing, most line graphs, caption of table and graphs. References.

i. General correction of the report.

j. PPT template detail, head of each page; Create Page 0 to 10 and page 31 to the end. In presentation, responsible for the introduction, CFD and why GPU part.