

Opportunities for GPU acceleration in CFD

Final report for project course TRA220

January 26, 2024

Benedick Allan Strugnell-Lees **benedick**

Program: 1st year PhD in controll engineering

Joar Forsberg **joarf**

Program: 2nd year, High Performance Computing.

Viktor Sundström **viktorsu**

Program: 1st year, Applied Mechanics.

Examiner: Rickard Bensow.

Abstract

This report investigates ways to accelerate a Computational Fluid Dynamics (CFD) solver developed by Lars Davidson. The report initially investigates how implementing the CuPy Library can accelerate the code and how this scales for different GPUs. It was found that the time to run a simulation is decreased by approximately two orders of magnitude when switching to the GPU implementation. During this investigation it was noted that the available VRAM is the limiting factors on the GPU. In order to reduce VRAM usage and take advantage of recent developments in tensor cores, the feasibility of using half precision floats was investigated. It was concluded that the performance increase would be significant, but it is hard to implement for the sparse CFD solvers and current support in Python libraries is limited.

Code

The code for this project is uploaded on github

<https://github.com/vrogly/Acceleration-of-CFD-code.git>

Original pyCALC-RANS solver by Lars Davidson [1]

<https://www.tfd.chalmers.se/~lada/pyCALC-RANS.html>

Keywords: CUDA, CFD

1 Introduction

This project will investigate ways to accelerate a simple CFD solver by using the CUDA interface for GPUs and tools made for Python. This is done by modifying and accelerating a simple CFD solver written in Python by Lars Davidsson [1]. Changes to the solver will be detailed later in the report.

The subsections in this introduction will give an overview of how CFD can be reduced to a problem of the form $Ax = b$, something that computers are very good at solving, but why the A -matrix specific to CFD is less suitable for GPUs in particular.

1.1 From Governing Equations to $Ax = b$

A major challenge in CFD is to go from the Navier Stokes equations, that governs all fluid flow, to linear system of equations that can be solved numerically. These derivations are extensive, and the reader is referred to Versteeg and Malalasekera [2] or the documentation of the pyCALC-RANS project [1] for a thorough explanation. The general idea is however:

1. Look at a small volume and integrate the Navier Stokes equations over it
2. Using calculus relations, it can be rewritten in terms of the values on the volumes boundaries that are assumed to be constant at each face
3. The velocity and its derivative at the boundaries must be interpolated in various ways
4. Now, for each cell there is a linear equation relating its values to the surrounding cells. The factor, e.g. a_E for the cell to the east (right), in front of each value from the surrounding cells, e.g. u_E for the velocity in the x -direction of the cell to the east, can be collected into a matrix.
5. the final system of equation looks like $Ax = b$, where A is a matrix containing all coefficients, b usually contain boundary and source terms (although not always for stability reasons), and x are the values that are being solved for, e.g. a list of the values of u for every single cell.

The equation for each cell will also include boundary conditions, if applicable. However, just solving this equations is not enough, as the pressure is not solved for.

The transport equations that governs k and ω from the chosen turbulence model are discretized in a similar way.

1.2 SIMPLEC Loop

In order to get a dependence on both the pressure and continuity equation, algorithms based on a pressure correction are typically used. This yields an equation for the pressure, similar to the ones for velocities described above. That solution is then used to update the values of u and v .

Using this, an iterative way of converging to the solution called the SIMPLEC algorithm, as implemented by [1], consists of the general steps

1. Calculate coefficients for u, v
2. Solve for u, v
3. Calculate coefficients for p
4. Solve for p
5. Pressure corrections

6. Calculate new faces
7. Calculate coefficients for k
8. Solve for k
9. Calculate coefficients for ω
10. Solve for ω
11. Goto step 1.

1.3 Sparse Matrices

The matrix for the systems of equations that are generated in the code by [1] are for reasonable resolutions more than 99,9% zeros. This means that methods normally used for solving linear systems of equations, that have a time complexity of $\mathcal{O}(n^3)$ or slightly less for large enough matrices [3], are infeasible. Solvers specifically design for sparse matrices are typically used. One example is the Jacobi algorithm [4], which will be shown to have $\mathcal{O}(n)$ time complexity in section 3.

1.4 Accelerating Sparse Matrix Operations

Deep learning applications are currently a major focus of the GPU industry, illustrated by that at the time of writing, 2024-01-26, all 15 main news cards on developer.nvidia.com are related to opportunities with AI and Nvidia GPUs. The accelerators made by Nvidia are primarily made for deep learning and as such their architecture is optimized for this purpose, especially with the advent of tensor cores [5]. As the matrices in AI are inherently (at least primarily) dense these accelerator lends themselves to dense matrix operations. Whereas CFD applications are inherently sparse it is hard to leverage these accelerators to their fullest extent when accelerating CFD applications but even so there are gains to be made.

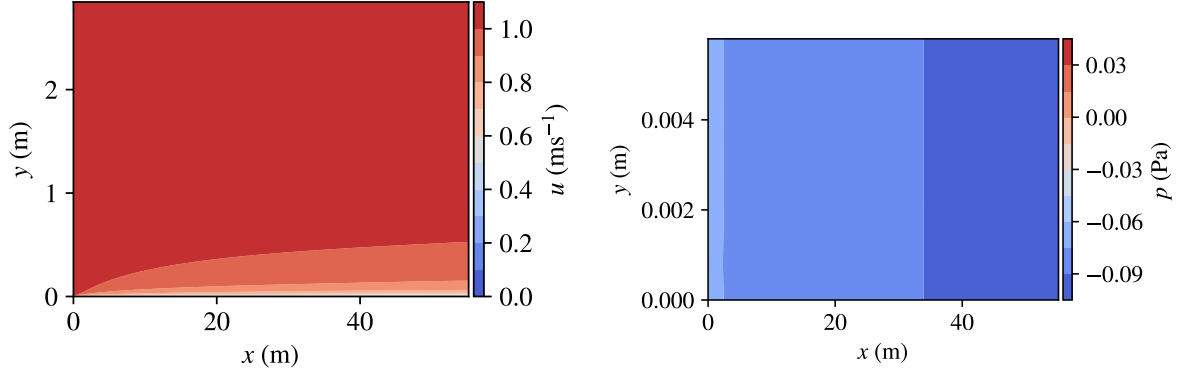
2 Quantitative investigations

The code available on GitHub are designed to test large batches of simulations whilst varying a few parameters. A list of all simulation that forms the bases for this section is found in table 1 at the last page of this report. These were deployed to the Vera cluster, where they, including failed runs, took 19 000 core hours. All CPU simulations were allocated 16 cores although since the program is single threaded this will not make a difference.

2.1 Simulated case

The simulated case was `komega-case-1e-8` from [1], with all convergence limits removed, see figure 1. The general description is that of a boundary layer flow, with a liquid with kinematic viscosity $\nu = 3,57 \cdot 10^{-5} \text{ m s}^{-2}$ ¹ and free stream velocity $u_\infty = 1 \text{ m s}^{-1}$ passing a no slip wall.

¹Presumably Gear oil SAE 140 at 98 °C by matching values from https://www.engineersedge.com/fluid_flow/kinematic-viscosity-table.htm



(a) Velocity component in flow direction for the entire domain.

(b) Pressure close to the boundary.

Figure 1: Plots of fields from simulation with original settings.

2.2 CFD Numerics

The original code by [1] was not under relaxed. To increase stability for larger meshes, the pressure was under relaxed to 0,5 [6]. This heavily affects the convergence speed, and since different solvers might converge at different rates, the number of iterations were kept fixed at 500.

2.3 Profiling

The main profiling was performed on the main SIMPLEC loop as described in section 1.2, as this is where most of the computational time is being spent. The functions corresponding to distinct moments in the SIMPLEC loop were timed cumulatively over the entire simulation. For each of u, v, p, k, ω there is a corresponding "Solve" and "IO" part. Solve reformulates the current coefficient matrices and values to the sparse equation $Ax = b$ and solves it. IO corresponds to building the coefficient matrices, adjusting them with boundary conditions. The matrix operations being performed is mainly addition, multiplication and cropping. This is in various amounts done using approximately 16 different matrices meaning that a significant time is spend moving around these in memory.

The RAM and VRAM (when applicable) were also monitored (although they showed very little variation between iterations). Since the simulations were run for a fixed number of iterations, even for very fine meshes, most did not converge to any practical tolerances. The residuals were also monitored.

2.4 Mesh

The original mesh supplied in [1] had a resolution of 300×90 cells. To make resolutions comparable with each other, the core properties were kept the same when reworking the mesh. The original mesh had a $y^+ = 1$ and resolved the boundary with its high resolution and not any wall functions. In principle the cells could be arbitrary close to the wall, but since all simulations will be run the same number of iterations, and not necessarily until convergence, it was decided to keep the same y^+ for all simulations. The original mesh had a maximum cell aspect ratio of $\alpha \approx 700$, when increasing the number of cells, this value was kept roughly the same.

The expansion rate between cells was originally 1,01 for the x -axis and 1,1 for y . This had to be decreased to accommodate the increased number of cells and end up with the same size of the entire flow domain as the original mesh, i.e. $55,6\text{m} \times 2,9\text{m}$. As an example, for the mesh with a million cells, this amounted to the expansion rate 1,0016 and 1,010 respectively.

A comparison between the original mesh and the one million cell mesh is shown in figure 2

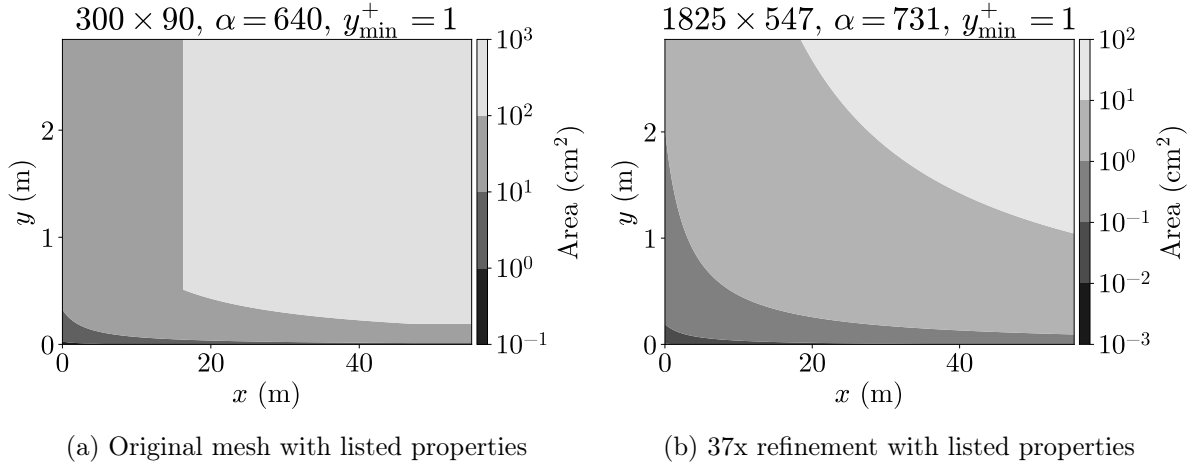


Figure 2: Comparison between size of cells in original mesh and one with increased number of cells, α is maximum aspect ratio of any cells.

2.5 Resolution

The time for each simulation at different resolutions are showcased in figure 3. Noteworthy is that for the simulation speed, changing GPU typically makes a larger difference than going from doubles (FP64) to floats (FP32). The A100 has an advertised 9,7 TFLOPS for FP64 and 19,5 TFLOPS for FP32 [7], indeed the speed improvement is roughly 1,6 which matches that relation.

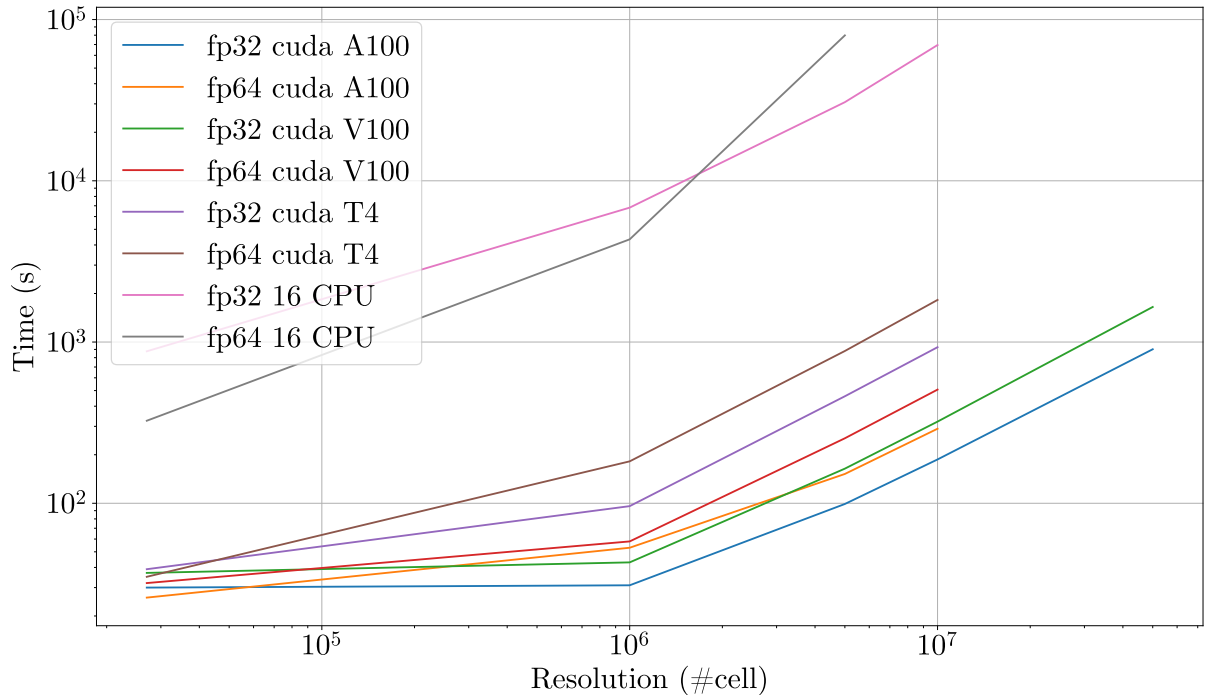


Figure 3: Time to run simulation plotted against the resolutions for different hardware. No GPU was capable of running 50 M cells with FP64 due to VRAM limitations, even though A100 is one of the most capable GPUs available with 80 GB VRAM [7].

2.6 Memory usage

A graph displaying the relationship between mesh resolution (number of cells) and memory (VRAM) usage is displayed in 4 As can be extrapolated from the figure the relationship between resolution and memory usage is not a linear one. Going from 1 M cells to 50 M cells results in a drastic increase in memory usage.

Running out of memory quickly becomes a factor when scaling up the simulation. One workaround to this problem would be to use FP16 instead of FP32 and thus, effectively halving the total memory usage, this will be investigated more in section 4.

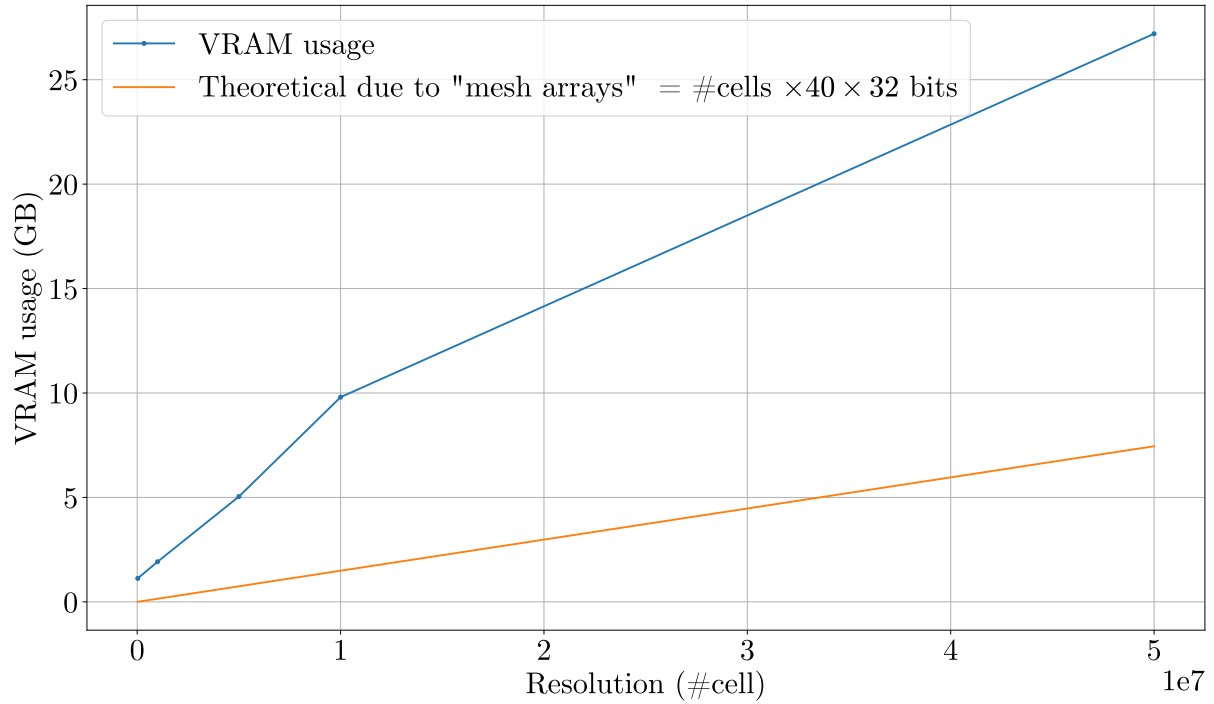


Figure 4: Graph depicting the relationship between memory and resolution for simulations with 32 bit floats. "Mesh arrays" represent the approximately 40 arrays occupying memory by storing values for individual cells.

When running the simulation and including global variables, there is approximately 40 values defined for any given node (these range from the value of interest, u, v, p, k, ω , but also their respective interpolation to boundaries (4 per value), respective coefficients and some other variables. Most of these are globally defined and will occupy memory throughout the entire simulation. At any given moment 40 is a fair approximation for the total amount of values that are loaded for each individual cell in the mesh during the SIMPLEC loop). The memory usage due to this is also displayed in figure 4, there it is significant but not dominant contribution to the memory usage. Hence reformulating these global variables to locals could result in significant memory savings.

2.7 Different solvers

To test different solvers, everything were kept fixed between simulations except for the pressure solvers, which typically takes the longest time. The results are shown in figure 5.

It can be seen from 5 that the GMRES is both faster and more memory efficient than the Direct solver. This is expected as with a larger memory footprint more cache misses are expected which would affect the total execution time. It should be noted, however, that these values were

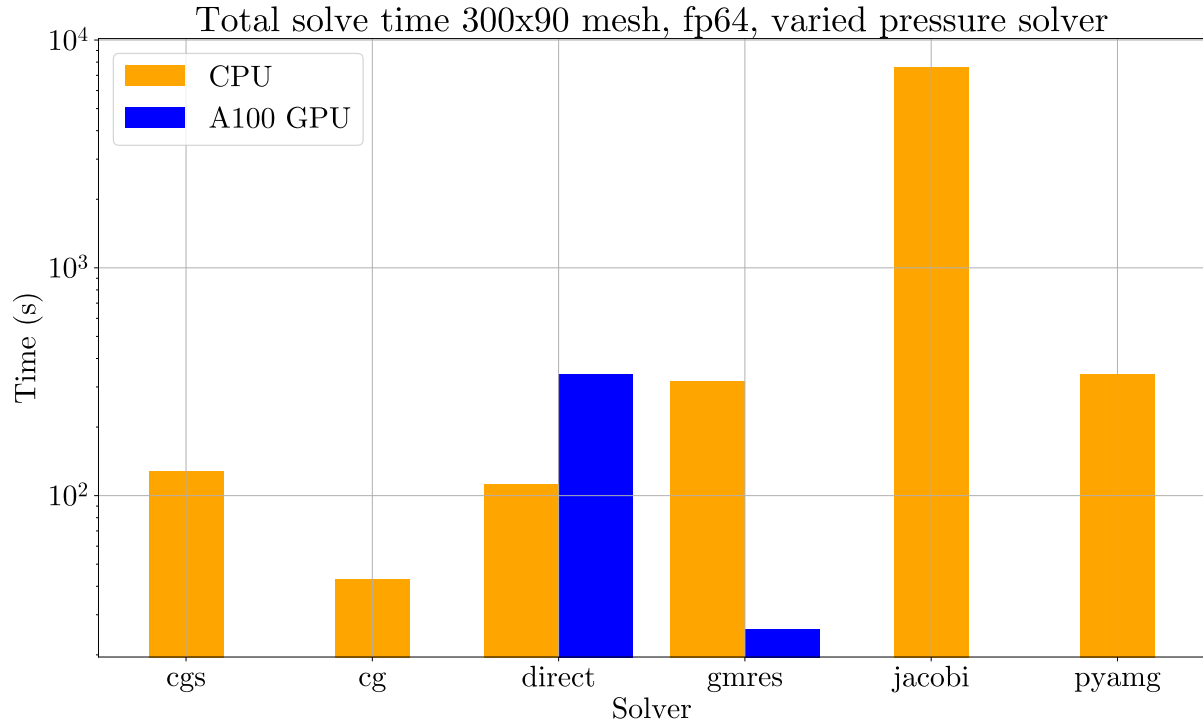


Figure 5: Graph comparing the speed of different solvers applied to the pressure equation. VRAM usage for the Direct solver (Cupy): 1,75 GB, GMRES solver (Cupy) 1,26 GB.

generated from running the simulation with a small mesh with 27k cells. This means that the total execution time will be disproportionately affected by the transfer time to and from device.

2.8 Time Distribution in SIMPLEC Loop

In general, it was observed that for the CPU, most time were spent solving for pressure for smaller meshes and gradually stood for a smaller percentage as the resolution increase, as illustrated by the large difference seen in figure 5.

For both CPU and GPU the relative time spent on IO decreases, likely due to the solvers increasing more rapidly in computational cost than the small operations performed in the IO.

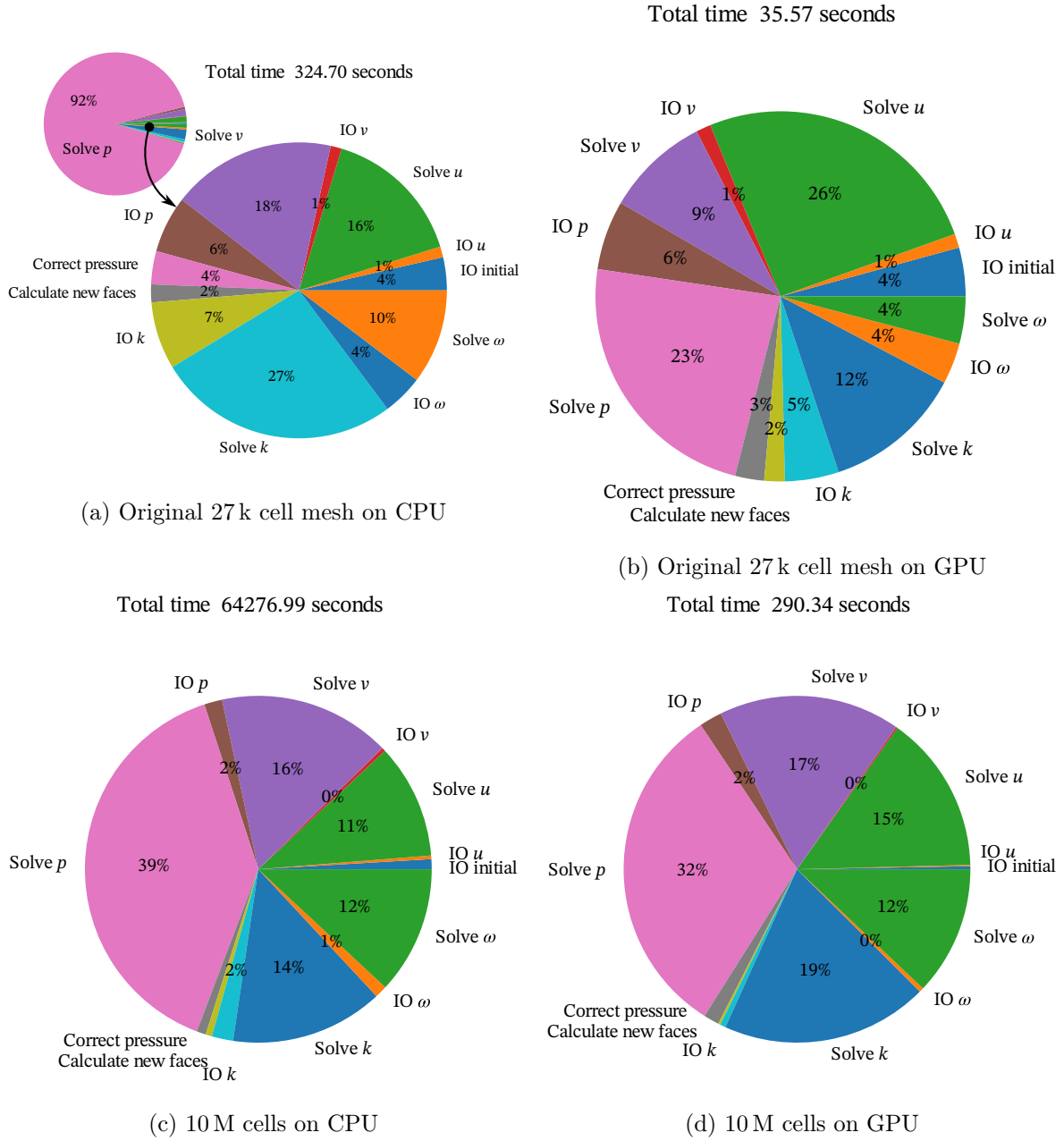
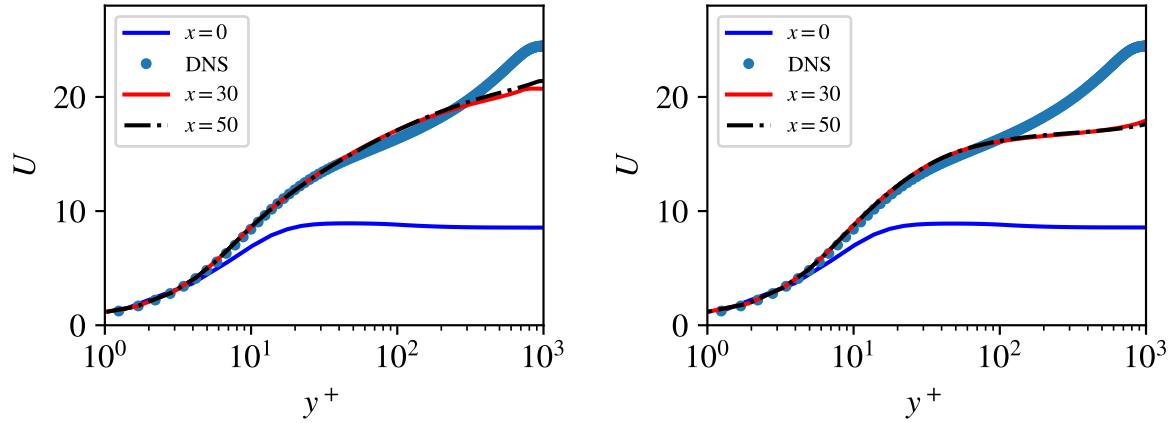


Figure 6: Distribution of time spent on different stages in the All 500 iterations, fp64

2.9 Convergence

Most simulations did not converge to the equivalent DNS simulation. However, for the original mesh, with 500 fixed iterations, the CPU version converged more than the GPU, as seen in figure 7.



(a) Original CPU implementation

(b) GPU implementation

Figure 7: Original mesh and fixed 500 iterations compared to DNS data.

3 Implementing a Jacobi Solver With Sparse Data Types

Iterative/implicit methods for solving the system of equations formed in the introduction have good convergence for sparse (diagonally dominant) matrices. However, naïve implementations of them such as the one listed on Wikipedia [4] do not fully take into account the sparsity. Implementing that Jacobi-algorithm directly in python would look like the following

```
# This loops over an extreme amount of zeros and will never finish
if solver_local == "jacobi_dense":
    A # (n x m) x (n x m)
    phi_updated = phi
    it = 0
    while it < nmax:
        for i in range(A.shape[0]):
            sigma = 0
            for j in range(A.shape[0]):
                if j != i:
                    sigma += A[i,j] * phi[j]
            phi_updated[i] = (su[i] - sigma) / A[i, i]
    phi = phi_updated
    if np.linalg.norm(A.dot(phi)-su) <= tol_conv:
        it = nmax
    it += 1
# E.g. n x n mesh, 5*(n x n) non zero elements --> ~ n^4 operations
```

However, running this on the original mesh, even on a cluster takes several hundreds of hours. This is due to the for-loop looping over all elements and explicitly adding zeros. Even if the Python interpreter is "smart" enough to not perform the addition (since the value won't change anyway when the array value is zero), it still accounts to lookups in the A-array which takes time. For a $N \times N$ mesh, this would correspond to doing N^4 lookups.

In scipy, and equivalent cupy-package, there is a special data structure for sparse matrices. This uses a structure for describing all non zero elements. Implementing this yields the following code,

```
if solver_local == "jacobi_sparse": # Modified to use sparse format
    if iter == 0:
        print("solver in solve_2d: jacobi (sparse) method")
    iMask = A.indptr # 1 x # (n x m)
    jMask = A.indices # 1 x # non zero elements
    vals = A.data # 1 x # non zero elements
```

```

phi_updated = phi
it = 0
while it < nmax:
    i = 0
    while i < iMask.shape[0]-1:
        sigma = 0
        Aii = 0
        for onRow in range(int(iMask[i]), int(iMask[i+1])):
            j = jMask[onRow]
            if j != i:
                sigma += vals[onRow]*phi[j]
            else:
                Aii = vals[onRow]
        phi_updated[i] = (su[i] - sigma) / Aii
        i += 1
    phi = phi_updated
    if np.linalg.norm(A.dot(phi)-su) <= tol_conv:
        it = nmax
    it += 1
# E.g. n x n mesh, 5*(n x n) non zero elements --> ~ 5n^2 operations

```

Using this in the original code and mesh converges in three hours. For a $N \times N$ mesh, this would correspond to doing $5N^2$ lookups (all of which would update the value and does indeed correspond to the $\mathcal{O}(n)$ complexity for a fixed number of iterations of the algorithm, as discussed earlier. Here “ n ” would correspond to N^2 which is the length of the vector in $Ax = b$), which is a significant difference compared to the original implementation.

4 Reduction to Half Precision Floating Point Numbers

It was speculated that changing the resolution from 32-bit floats (FP32) to 16-bit floats (FP16) would lead to an improvement in execution speed of the simulation. This is because, for one, it could speed up memory operations as well as transferring to and from device. It could also increase the speed of the floating point operations themselves. This, would, of course, come at the cost of the precision of the simulation. When using lower precision floats, this will lead to the simulation having a lower degree of accuracy.

The first reason to believe that reducing the precision might decrease the execution time is because this would increase the memory utilization. Besides the obvious benefit of having faster data transfers to and from the device using a lower precision can also reduce the number of instructions that has to be executed. Nvidia’s GPUs are able to perform vectorized memory operations, meaning that they can, with just one operation, perform multiple reads/writes to memory [8]. These vectorized memory operations are capable of reading or writing 128 bytes at a time. This means that eight FP16 variables could be written/read as opposed to only four FP32 variables. This means that it would be possible to perform as many reads/writes with only half the number of instructions, which, of course would decrease the execution time. Whether these vectorized memory operations actually are utilized depends on how the CUDA kernels themselves are programmed however. It is unknown if the kernels launched by CuPy actually utilize these instructions.

If the accelerator in question has specialized hardware for the execution of FP16 operations then it would be probable that a decrease in the time it takes to perform logical and arithmetical operations would result from this. This would have great benefits on the execution time. The GPUs assigned for use in this study do in fact have such hardware.

These GPUs have two types of cores: CUDA cores and Tensor cores. The CUDA cores are regular cores that are capable of executing any instruction on any data whereas the Tensor cores only capable of executing some particular instructions and have strict requirements regard the layout of the data processed. The Tensor cores also see a benefit from reducing the precision of

the data. They can execute FP16 instructions faster than higher resolution instructions. CUDA cores do not see the same benefit. They execute FP16 and FP32 instructions equally fast and so they do not benefit from reducing the precision [9].

Due to the nature of the simulation it is unable to utilize the faster, specialized Tensor cores. Tensor cores require dense matrices and normal CFD-matrices are too sparse. Sadly, this means that there was no decrease in execution time after changing the precision from FP32 to FP16. In order to make use of these cores, the simulation functions would have had to be reworked fundamentally, which would have been very time consuming. Another possibility would have been to treat the sparse matrix as a dense one but doing this would also not have been viable since the performance penalty from treating it as dense would have far outweighed the performance gains.

The `cupyx.scipy.sparse` methods does not support FP16 either, which means that these methods would have had to be re implemented. The sparse Jacobi solver written for this project would work as a substitute. This unveils further problems with the decreased precision, convergence criterion for residuals are specified close to the limit of the smallest values that FP16, $6 \cdot 10^{-8}$, can handle and it is hard to achieve convergence (indeed most sought after convergence limits in CFD would fall into the subnormal range of FP16 which further complicates any residual measures for convergence).

Even though there was no reduction in execution time that as was hoped, this endeavour was not completely fruitless. When dealing with very high resolution meshes, memory becomes a bottleneck. When the resolution becomes too big and the system will run out of memory, the simulation breaks. With the usage of FP16 the resolution of the mesh can be doubled and still occupy the same amount of memory as compared to FP32.

5 Standalone investigation of TDMA

This investigation is implemented into the main CFD code, but showcases some ways to accelerate the TDMA-solver.

5.1 Use of the TDMA as an example of GPU specific python

The Tri-Diagonal Matrix Algorithm (TDMA) is a specialized numerical technique used for solving linear systems of equations that arise in the context of implicit methods for solving partial differential equations (PDEs)[10].

The following steps give an idea of how this method is implemented in our approach:

1. Tri-Diagonal Matrix Structure: Non-zero coefficients are only present along the main diagonal and the two adjacent diagonals[11].
2. The TDMA algorithm is an iterative method designed specifically for solving tri-diagonal systems. It proceeds through three main steps for each row of the matrix:
 - (a) Forward elimination: Transform the matrix to an upper triangular form.
 - (b) Backward substitution: Solve for the unknowns in reverse order.
 - (c) Update the solution vector.
3. Efficiency and Complexity: The TDMA algorithm is computationally efficient with a complexity of $O(n)$, making it suitable for solving large linear systems quickly. Its efficiency comes from the fact that each step involves only basic arithmetic operations, and the structure of the matrix allows for a streamlined process.

5.2 Matrix Configuration

Tri-diagonal systems for n unknowns [10]:

$$\begin{aligned}
 & a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i \\
 & \text{Boundaries: } u_0 = B_0; u_{n-1} = B_{n-1} \\
 & \text{Vector } u \text{ comprises unknowns:} \\
 & \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & 0 & \dots & 0 \\ 0 & a_2 & b_2 & c_2 & \dots & 0 \\ & \cdot & \cdot & \cdot & \cdot & \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ \cdot \\ \cdot \\ \cdot \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} B_0 \\ d_1 \\ \dots \\ d_{n-2} \\ B_{n-1} \end{pmatrix} \\
 & \text{Re-arrangement of the matrix} \\
 & \begin{pmatrix} b_1 & c_1 & 0 & & 0 \\ a_2 & b_2 & c_2 & & \cdot \\ 0 & a_2 & b_2 & c_2 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & & a_{n-2} & b_{n-2} \end{pmatrix} \begin{pmatrix} u_1 \\ \cdot \\ \cdot \\ \cdot \\ u_{n-2} \end{pmatrix} = \begin{pmatrix} d_1 - a_1 B_0 \\ d_2 \\ \dots \\ d_{n-3} \\ d_{n-2} - c_{n-2} B_{n-1} \end{pmatrix}
 \end{aligned}$$

5.3 TDMA for CPU

To give an instance of the implemented method in code:

```

def TDMA solver(a, b, c, d):
    neq = len(a) # number of equations
    ac, bc, cc, dc = map(np.array, (a, b, c, d))
    for it in range(1, neq):
        mc = ac[it]/bc[it-1]
        bc[it] = bc[it] - mc*cc[it-1]
        dc[it] = dc[it] - mc*dc[it-1]
    xc = ac
    xc[-1] = dc[-1]/bc[-1]
    for il in range(neq-2, -1, -1):
        xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]
    return xc

```

5.4 CuPy

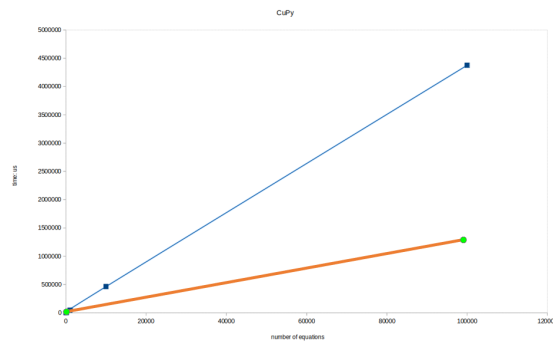
CuPy is a library for numerical computations in Python that provides an interface similar to NumPy, but it is specifically designed to accelerate code on NVIDIA GPUs. The primary advantage of CuPy is its integration with NVIDIA GPUs allows you to perform computations on the GPU, which can significantly accelerate certain types of numerical tasks, especially those involving large-scale parallelism[12].

CuPy introduces its array type, which is similar to NumPy arrays. It manages memory on the GPU efficiently and allows for seamless data transfer between the CPU and GPU. This helps in minimizing data transfer overhead and maximizing computational efficiency. Because CuPy is built on top of CUDA, the parallel computing platform and programming model developed by NVIDIA, this close integration with CUDA allows CuPy to harness the full power and features of NVIDIA GPUs.

In the example of TDMA the following features of CuPy were used:

1. Easily accelerate codes that only rely on NumPy

2. Custom kernels (element wise)
3. GPU's memory management



Comparison of CPU and GPU runtime for equations.

Demonstrating this in implementable code:

```
import cupy as cp

mul = cp.ElementwiseKernel('float64 x, float64 y', 'float64 z',
                           'z = y*x', 'mul')
div = cp.ElementwiseKernel('float64 x, float64 y', 'float64 z',
                           'z = x/y', 'div')

def TDMA solver(a, b, c, d, l):
    res = cp.zeros(l)
    mc = div_gpu(a[1:], b[:l-1])
    b[1:] -= mul(mc[:l-1], c[:l-1])
    d[1:] -= mul(mc[:l-1], d[:l-1])
    res = div(d, b)
    for i in range(l-2, -1, -1):
        res[i] -= div((mul(c[i], res[i+1])), b[i])
    return res
```

5.5 Numba

NUMBA is a Just-In-Time (JIT) compiler for Python that translates a portion of Python code into machine code at runtime, through the use of the `@cuda.jit` decorator, enabling significant speedup for numerical computations [13].

Advantages:

1. Ease of Use: NUMBA is easy to use since it integrates well with Python.
2. Dynamic Compilation: NUMBA performs dynamic compilation at runtime, optimizing the code for the specific hardware it is running on. Meaning that the same code can achieve good performance on different architectures without manual optimization.
3. Array-Oriented Operations: NUMBA is particularly well-suited for array-oriented operations; it can automatically parallelize and optimize array operations.

Disadvantages:

1. Limited Support for Certain Features: NUMBA does not support all Python features, especially those related to complex data structures or certain libraries[14].
2. Learning Curve: NUMBA can require some time in order to understand how to use it effectively, especially when dealing with complex CFD codes when compared with CuPy.
3. Dependency on Python: Since NUMBA relies on Python, it inherits some of Python's limitations, meaning that it is not suitable for all types of parallelism.

For this project specifically, it was initially attempted to implement a NUMBA method, but experienced extensive problems with the code (many of which may have been owing to Python's limitations). Additionally, because it was attempted to make use of additional libraries in order to speed up sub-processes, it was recognized that it was likely that these issues would only be exacerbated. Ultimately, it was decided not to implement NUMBA as it was hoped that several different methods could be attempted and there wasn't the time to spend required to properly implement it for the given CFD problem.

To give a tested example of NUMBA implementation in code:

```
@cuda.jit
def TDMAsolver_cuda(a, b, c, d, res):
    it = cuda.grid(1)
    il = cuda.grid(1)
    for it in range(1, len(a)):
        mc = a[it]/b[it-1]
        b[it] = b[it] - mc*c[it-1]
        d[it] = d[it] - mc*d[it-1]
    res[-1] = d[-1]/b[-1]
    for il in range(len(a)-2, -1, -1):
        res[il] = (d[il]-c[il]*res[il+1])/b[il]
    return

threadsperblock = 1024
blockspergrid = (a.size + (threadsperblock - 1)) // threadsperblock
a_gpu=cuda.to_device(a)
b_gpu=cuda.to_device(b)
c_gpu=cuda.to_device(c)
d_gpu=cuda.to_device(d)
res = cuda.device_array_like(a)
```

```
TDMAsolver_cuda[blockspergrid, threadsperblock](a_gpu, b_gpu, c_gpu, d_gpu, res)
res_cpu = res.copy_to_host()
```

The results of our simulation's time division were as follows:

3.915 sec safe cuda api call, Total time for len(a) = 100000 is 4.488 seconds

5.6 PyTorch

PyTorch tensors are fundamental data structures in PyTorch, a popular open-source machine learning library. PyTorch tensors are designed to provide efficient computation on parallel architectures, especially GPUs[15]. PyTorch tensors share similarities with NumPy arrays, making it easy to transition between the two. Tensors can be created from Python lists, NumPy arrays, or other existing tensors. PyTorch also seamlessly supports GPU acceleration, allowing users to perform computations on GPUs. Tensors can be easily moved between CPU and GPU using the `.to()` method[16].

Furthermore, PyTorch facilitates more straightforward vectorization of code. The implemented TDMA using PyTorch demonstrates superior speed compared to alternative GPU-accelerated libraries. While its unconventional approach may stand out, it unquestionably merits attention[17].

To give an implemented section of code using PyTorch:

```
def th_delete(tensor, indices):
    mask = torch.ones(tensor.numel(), dtype=torch.bool)
    mask[indices] = False
    return tensor[mask]
def TDMAsolver(ac, bc, cc, dc):
    nf = int(ac.shape[0]) - 1
    acd = th_delete(ac, 0)
    bcd = th_delete(bc, nf)
    ccd = th_delete(cc, nf)
    dcd = th_delete(dc, nf)
    dbc = th_delete(bc, 0)
    ddc = th_delete(dc, 0)
    mc = torch.add(acd, torch.pow(bcd, -1), alpha=-1)
    dbc = dbc - torch.mul(mc, ccd)
    ddc = ddc - torch.mul(mc, dcd)
    ac[-1] = ddc[-1]/dbc[-1]
    for il in range(nf-1, -1, -1):
        ac[il] = (dc[il]-cc[il]*ac[il+1])/bc[il]
    return ac
```

Performance was better than the one of CuPy and Numba

24 function calls in 3.738 seconds

<i>ncalls</i>	<i>totttime</i>	<i>percall</i>	<i>cumtime</i>	<i>percall</i>	<i>function</i>
1	3.698	3.698	3.738	3.738	<i>TDMAsolver</i>
6	0.016	0.003	0.017	0.003	<i>th - delete</i>
1	0.010	0.010	0.010	0.010	<i>torch.add</i>
1	0.007	0.007	0.007	0.007	<i>torch.pow</i>

5.7 PyAMG

PyAMG, or Python Algebraic Multigrid, is an open-source Python library that provides tools for solving large sparse linear systems of equations. It focuses on algebraic multigrid methods,

a class of iterative solvers that are particularly effective for solving linear systems arising from discretized partial differential equations (PDEs)[18]. PyAMG is designed to be scalable and efficient, making it suitable for solving problems with large, sparse matrices often encountered in scientific and engineering simulations. AMG is a multilevel iterative method that leverages a hierarchy of coarser grids to accelerate the convergence of iterative solvers. It is effective for solving linear systems arising from the discretization of PDEs.

PyAMG is designed to handle large, sparse linear systems efficiently. It can scale to problems with millions or even billions of unknowns, making it suitable for high-performance computing environments. It complements SciPy’s sparse linear algebra capabilities by providing additional advanced solvers. PyAMG supports parallel computing, allowing users to take advantage of multiple processors or cores.

Here’s a simple example that demonstrates how to use PyAMG to set up a solver for a sparse matrix and solve a corresponding linear system.:

```
import pyamg
import scipy.sparse
A = scipy.sparse.random(1000, 1000, density=0.1, format='csr ')
b = scipy.sparse.random(1000, 1, density=0.1)
ml = pyamg.smoothed_aggregation_solver(A)
x = ml.solve(b)
```

6 Future work

The accelerators that were used in this study are mainly designed to accelerate ML-applications. As this is the case there is a lot of performance being left on the table. If using Tensor cores was an option it would be possible to realize much greater performance gains. But as CFD is not suited for these cores it is hard to utilize them. Tensor cores are only able to work on dense data and CFD is fundamentally sparse. If there were accelerators that are suited specifically for the needs of CFD a much larger performance increase could have been seen. Designing such an accelerator is no easy feat, however. Even so, it is a venture that it worthy of looking into.

Whereas Tensor cores are made for dense workloads there have been attempts to create software solutions for utilizing these cores for sparse matrix operations. [19] explores the possibility of performing sparse matrix-matrix multiplications on Tensor cores. While these simulation does not make use of matrix-matrix multiplication their algorithms do make use of the multiply-and-accumulate (MAC) functionality of the Tensor cores. Many of the algebraic solvers used in this study do make frequent use of MAC operations. As such it would be interesting to explore if the algorithms proposed in their paper could be adapted for the purposes of CFD.

7 Summary

This project has shown significant speed gains by using GPU and the CUDA/Python framework. The main limiting factor remains VRAM as it cannot be scaled easily. The A100 with 80 GB VRAM, which was the maximum amount available, could not handle 50 million cells with double precision, which is not an unrealistic requirement for real CFD cases.

References

- [1] L. Davidson, "pycalc-rans: A python code for two-dimensional turbulent steady flow," tech. rep., Tech. rep. Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, 2021.
- [2] H. K. Versteeg och W. Malalasekera, *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.
- [3] Wikipedia contributors, "Computational complexity of mathematical operations — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Computational_complexity_of_mathematical_operations&oldid=1189645589, 2023. [Online; accessed 26-January-2024].
- [4] Wikipedia contributors, "Jacobi method — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Jacobi_method&oldid=1146436163, 2023. [Online; accessed 21-January-2024].
- [5] S. Li, K. Osawa, och T. Hoefer, "Efficient quantized sparse matrix operations on tensor cores," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2022.
- [6] Simscale, "Relaxation factors." <https://www.simscale.com/docs/simulation-setup/numerics/relaxation-factors/>, 2022. [Online; accessed 26-January-2024].
- [7] NVIDIA Corporation, "NVIDIA A100 Tensor Core GPU," tech. rep., NVIDIA Corporation, 2024. Accessed: 2024-01-26.
- [8] NVIDIA Developer, "Cuda pro tip: Increase performance with vectorized memory access." <https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/>, 2014. Accessed: 2024-01-24.
- [9] NVIDIA Corporation, "Tesla V100 GPU Architecture," tech. rep., NVIDIA Corporation, 2017. Accessed: 2024-01-24.
- [10] Wikipedia, "Tridiagonal Matrix Algorithm." https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm, 2023. Accessed: 2024-01-22.
- [11] Pankaj Dumka, Dr. Dhananjay R. Mishra, "Understanding the TDMA/Thomas algorithm and its Implementation in Python." https://www.researchgate.net/publication/364389215_Understanding_the_TDMAThomas_algorithm_and_its_Implementation_in_Python, 2023. Accessed: 2024-01-23.
- [12] CuPy, "User Guide." https://docs.cupy.dev/en/stable/user_guide/index.html, 2023. Accessed: 2024-01-23.
- [13] NUMBA Developer, "Numba documentation." <https://numba.pydata.org/numba-doc/latest/index.html>, 2023. Accessed: 2024-01-22.
- [14] NUMBA Developer, "Environmental Variables." <https://numba.pydata.org/numba-doc/latest/reference/envvars.html>, 2023. Accessed: 2024-01-22.
- [15] NVIDIA Developer, "PyTorch - NVIDIA NGC Catalog." <https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch>, 2023. Accessed: 2024-01-22.
- [16] PyTorch Developer, "Introducing Accelerated PyTorch Training on Mac." <https://pytorch.org/blog/introducing-accelerated-pytorch-training-on-mac/>, 2022. Accessed: 2024-01-24.

- [17] PyTorch Developer, "PYTORCH DOCUMENTATION." <https://pytorch.org/docs/stable/index.html#pytorch-documentation>, 2022. Accessed: 2024-01-24.
- [18] ResearchGate, "PyAMG: Algebraic Multigrid Solvers in Python." https://www.researchgate.net/publication/360028095_PyAMG_Algebraic_Multigrid_Solvers_in_Python, 2022. Accessed: 2024-01-24.
- [19] O. Zachariadis, N. Satpute, J. Gómez-Luna, och J. Olivares, "Accelerating sparse matrix-matrix multiplication with gpu tensor cores," *Computers & Electrical Engineering*, vol. 88, s. 106848, 2020.

Division of work

A Benedick

Not including troubleshooting and technical support, I worked with implementing the TDMA and NUMBA solvers, and ultimately tested them on independent problems on a GPU associated with my PhD (owing to Vera not actually working for a long enough for me to implement them there). I also implemented PyTorch based solutions, attempting to use the built in functionality of PyTorch to do more of the computational distribution with built in library functions.

B Joar

A lot of technical support in order to install various libraries such as pyamgx. Made a lot of attempts to perform profiling on the CUDA kernels launched by CuPy in order to gain deeper insights about memory utilization, core utilization, etc. on Nvidia GPUs. Implemented a solver with pyamgx. Ran many jobs on Vera.

C Viktor

In terms of actual work:

Reworked the CFD-code to make it suitable to deploy to Vera and extracted all relevant data and plots from individual simulations. Also ran all final state simulations. Monitored convergence and corresponding measures to ensure that simulations between GPU and CPU as well as between different resolutions were comparable to each other (such as rescaling of mesh, under relaxing pressure and alike). Jacobi implementation for sparse matrices. Also some project management to coordinate us as a group and our areas of focus.

In terms of report:

Formatting, organization, structure, everything CFD related, most of the results, Jacobi, discussion about subnormal numbers for FP16. Also spent quite some discussing and proof reading the the other group members parts.

Table 1: All simulations that were successfully run with the final setup. Multiple occurrences of GPU simulations are due to running all of them on on T4, V100 and A100 GPUs. Variations using fp16 and pyAMGx was not deployed to cluster but tested locally.

Iterations	mesh	precision	solver u,v	solver pressure p	solver k,ω	useCupy
500	x300y90	fp64	gmres	direct	gmres	false
500	x300y90	fp32	gmres	gmres	gmres	true
500	x1825y547	fp32	gmres	gmres	gmres	true
500	x12909y3872	fp32	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	pyamg	gmres	false
500	x1825y547	fp64	gmres	gmres	gmres	false
500	x300y90	fp64	gmres	gmres	gmres	true
500	x4082y1224	fp64	gmres	gmres	gmres	false
500	x300y90	fp64	gmres	cgs	gmres	false
500	x4082y1224	fp32	gmres	gmres	gmres	true
500	x5773y1732	fp64	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	gmres	gmres	false
500	x4082y1224	fp32	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	gmres	gmres	true
500	x182y54	fp32	gmres	gmres	gmres	false
500	x300y90	fp32	gmres	gmres	gmres	true
500	x4082y1224	fp32	gmres	gmres	gmres	true
500	x4082y1224	fp64	gmres	gmres	gmres	true
500	x57y17	fp32	gmres	gmres	gmres	false
500	x1825y547	fp32	gmres	gmres	gmres	true
500	x300y90	fp32	gmres	gmres	gmres	false
500	x5773y1732	fp32	gmres	gmres	gmres	false
500	x1825y547	fp64	gmres	gmres	gmres	true
500	x4082y1224	fp64	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	cg	gmres	false
500	x4082y1224	fp32	gmres	gmres	gmres	false
500	x5773y1732	fp32	gmres	gmres	gmres	true
500	x5773y1732	fp64	gmres	gmres	gmres	false
500	x4082y1224	fp64	gmres	gmres	gmres	true
500	x182y54	fp64	gmres	gmres	gmres	false
500	x300y90	fp64	gmres	gmres	gmres	true
500	x1825y547	fp64	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	direct	gmres	true
500	x5773y1732	fp32	gmres	gmres	gmres	true
500	x1825y547	fp32	gmres	gmres	gmres	true
500	x1825y547	fp64	gmres	gmres	gmres	true
500	x5773y1732	fp64	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	gmres	gmres	false
500	x1825y547	fp32	gmres	gmres	gmres	false
500	x300y90	fp64	gmres	gmres	gmres	true
500	x57y17	fp64	gmres	gmres	gmres	false
500	x300y90	fp64	gmres	gmres	gmres	true
500	x12909y3872	fp32	gmres	gmres	gmres	true
500	x300y90	fp32	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	direct	gmres	true
500	x5773y1732	fp64	gmres	gmres	gmres	true
500	x5773y1732	fp32	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	gmres	gmres	true
500	x300y90	fp64	gmres	direct	gmres	true
500	x300y90	fp64	gmres	jacobi	gmres	false