Acceleration of CFD Python code using CUDA

GPU-accelerated Computational Methods using Python and CUDA HT22

TRA100/TRA105

Marios Aspris, Xingyuan Li, Andhika Pratama, Patricia Vanky

Chalmers University of Technology
Gothenburg, Sweden 2023

1 Introduction

Graphics Processing Units (GPUs) are specialized hardware designed to accelerate the processing of graphics and visualizations. In recent years, GPUs have become increasingly popular for a variety of non-graphics related tasks, including scientific computing, machine learning, and data analysis. One of the main reasons for this shift is the ability to utilize the parallel processing capabilities of GPUs to significantly increase the speed of specific types of computations.

Writing code for the GPU is typically done in CUDA C/C++, or a higher-level languages such as Python, Fortran, or C# with bindings to the CUDA API and can be used. In this report, we will discuss the use of GPU-accelerated Python code for computational fluid dynamics, and provide a brief overview of the tools and libraries available for GPU programming in Python. We will also discuss the issues that we have run into when applying CUDA code to accelerate already existing Python CFD code.

2 Methodology

This section will firstly introduce the benefit of parallelizing a computation on the GPU. It will also briefly presents the libraries available for GPU programming in Python that were introduced in the course. This list is not all conclusive but shows the what the project has been based on. Furthermore, the approach taken in this project will be explained.

2.1 Parallel discreatization

Parallel discretization is a method for dividing a continuous problem into discrete pieces, which is utilized to be able to solve the problem on a GPU for example. There are several different approaches to parallel discretization, but a common approach is to divide the continuous problem into a number of smaller discrete elements that can be solved independently. These elements can then be solved concurrently, using multiple processors to work on different parts of the problem simultaneously. This allows the solution to be computed more quickly, as the workload is distributed across multiple resources.

One key difference between CPUs and GPUs is the way they are designed to handle parallelization. CPUs have larger memory and are designed to handle a wide range of tasks, but they are not as efficient at handling tasks that can be easily divided into smaller pieces and solved concurrently. In contrast, GPUs does not have as much memory but are specifically designed to handle many small parallel processes concurrently very efficiently.

To be able to solve the computation concurrently the the domain has to be divided into blocks, and these blocks in turn contain a number of threads. The calculations are then performed on the threads inside of the blocks, similarly to how serial computations are performed but each block can run concurrently.

2.1.1 Shared memory

Shared memory on Nvidia GPUs is a memory on the hardware that is available per thread block. Shared memory is similar to an L1 Cache that is available on CPU architectures. The difference is that shared memory can be managed by the programmer explicitly as oppose to L1 which is automatically managed. Shared memory is available for the all the threads in a block and can be managed to exchange data between threads, with predictable timing. Shared memory is faster and more efficient than global memory and can be accessed with much lower latency, making it a useful tool for optimizing the performance of GPU kernels. The drawback in managing shared memory is synchronizing threads to avoid race conditions, and having a correct implementation for the problem we are trying to solve.

2.2 Python libraries

2.2.1 Numba

Numba is a just-in-time (JIT) compiler for Python that allows developers to write high-performance code using Python and execute it on the either CPU or GPU. Numba CUDA is a subset of Numba that provides support for programming NVIDIA GPUs using the CUDA parallel computing platform and programming model. It allows developers to write GPU-accelerated code using Python and execute it on NVIDIA GPUs.

2.2.2 CuPy

CuPy is an open-source library in Python that uses the GPU parallelization. The library is compatible with the commonly used library for scientific computing, NumPy, meaning that it provides an interface and functionality similar to NumPy, but is designed to make use of the parallel processing on GPUs to accelerate the computations.

2.2.3 Advantages and disadvantaged

There are several advantage of using CuPy over Numba and vice versa which will be briefly discussed in this section.

The first advantage of using CuPy being the NumPy compatibility, meaning that there is most likely limited rewriting necessary if the original program is based on NumPy. More over, CuPy in general allows for writing code for on the GPU parallelization without having to write low-level CUDA C/C++ code, by the use of RawKernels that can be compiled for CUDA inside a Python script. CuPy is also widely used and a well-supported library that is actively developed and maintained, making it a reliable choice for GPU acceleration in Python.

On the other hand, while CuPy is designed to be compatible with NumPy, it may not support all NumPy functions and features. CuPy will provide significant performance improvements over NumPy for certain types of array operations, however, it may not always be faster than Numba. As Numba is a just-in-time compiler that it is designed to provide high performance for a wide range of applications it can be more versatile.

2.2.4 Our approach

Although it is a great advantage to be able to use the NumPy framework that we are already familiar with, we have decided to use Numba as a base for the acceleration of our code. This choice was made as it would generate more learning and understanding as it requires to write code that is more similar to CUDA C/C++ although wrapped in a python package.

We would also like to note that we are aware of the fact that the algorithms used in these cases below are not computationally optimal in general. As the course is about GPU-acceleration we decided to focus on accelerating simple code, in this case the Gauss-Seidel solver (see section **??**, rather than spending time on finding the best algorithm and accelerating that. It would have been interesting to consider sparse matrices though, as it is such a central concept for these problems. However, we found that in order to do that we would have to rely on CuPy or SciPy or it would get much to complicated for this course, and we did not want to do that according to the above discussion.

2.3 Linear Solvers

2.3.1 Gauss-Seidel solver

The Gauss-Seidel algorithm is an iterative method for solving systems of linear equations. It solves problems by approximating the solutions of the governing equations on a grid and works by iteratively updating the

values of the unknown variables at each grid point, based on the values of the unknowns at the neighboring points. This is done in a sequential manner, starting from one corner of the grid and working towards the opposite corner. The updated values are then used to update the values of the unknowns at the next grid point, and the process is repeated until the solution has converged to a satisfactory level of accuracy.

2.3.2 Tridiagonal Matrix Algorithm

The Tridiagonal Matrix Algorithm (TDMA), is a method for solving systems of linear equations that are in the form of a tridiagonal matrix. This matrix is a special type of matrix that has non-zero entries only on the main diagonal and the two diagonals immediately above and below it, which is the case for this diffusion problem. TDMA works by using the values of the unknowns at the neighboring points to eliminate the unknowns at the intermediate points, and then using the resulting equations to solve for the unknowns at the remaining points, similarly to the Gauss-Seidel algorithm. Although TDMA is generally faster and more accurate than Gauss-Seidel as it only considers the non-zero entries of the matrix. However, it is only applicable to systems of equations with a tridiagonal matrix structure and cannot be used for more general systems.

3 Acceleration of Provided Simple Poisson Solver

As a first task the provided simple Poisson solver has been accelerated using Numba. In this section only the Gauss-Seidel function (see explaination of algorithm in section 2.3.1) and the modification made to it will be discussed, the full code including the set up for launching the GPU-kernel can be seen in Appendix A. The code for the function to be computed on the CPU can be viewed below.

```
def solve_gs(phi3d,aw3d,ae3d,as3d,an3d,al3d,ah3d,su3d,ap3d,tol_conv,nmax):
   print('solve_3d gs called,nmax=',nmax)
   acrank_conv = 1
   for n in range(0,nmax):
      phi3d=((ae3d*np.roll(phi3d,-1,axis=0)+aw3d*np.roll(phi3d,1,axis=0) \
      +an3d*np.roll(phi3d,-1,axis=1)+as3d*np.roll(phi3d,1,axis=1) \
      +ah3d*np.roll(phi3d,-1,axis=2)+al3d*np.roll(phi3d,1,axis=2))*acrank_conv+su3d)/ap3d

      res= ap3d*phi3d-\
      ((ae3d*np.roll(phi3d,-1,axis=0)+aw3d*np.roll(phi3d,1,axis=0) \
      +an3d*np.roll(phi3d,-1,axis=1)+as3d*np.roll(phi3d,1,axis=1) \
      +ah3d*np.roll(phi3d,-1,axis=2)+al3d*np.roll(phi3d,1,axis=2))*acrank_conv+su3d)

   resid=np.sum(np.abs(res.flatten()))
   return phi3d, resid
```

The function uses the NumPy feature "roll" which shifts the elements of an array circularly along a given axis by a specified number of positions. To be able to split the computation into smaller pieces for parallelization on the GPU using Numba, the numpy.roll function must be replaced with a proper matrix indexing. This can be seen in the code below. Other difference made is the introduction of the @cuda.jit decorator which modifies the function to run on the GPU. Furthermore, the indexing has been defined by the cuda.grid(3) function, which defines the shape of the grid of threads that will be launched on the GPU to execute the CUDA kernel. It is also customary practice to return empty for a GPU kernel as all arrays must have been predefined on the device before hand as to allocate space. Finally, the for-loop used for the iteration has been moved to outside of the kernel, meaning that the kernel is launched once per iteration. This was done because keeping it inside generated incorrect results which will be shown and discussed in more detail in a following section.

```
@cuda.jit
def solve_gs(phi3d,ap3d,aw3d,ae3d,as3d,an3d,al3d,ah3d,su3d,tol_conv,nmax,res):
    print('solve_3d gs called,nmax=',nmax)
    acrank_conv = 1
```

```
    i,j,k = cuda.grid(3)

    if 0 < i < phi3d.shape[0] and 0 < j < phi3d.shape[1] and 0 < k < phi3d.shape[2]:

        phi3d[i,j,k] = ((ae3d[i,j,k]*phi3d[i-1,j,k]+aw3d[i,j,k]*phi3d[i+1,j,k]\
                  +an3d[i,j,k]*phi3d[i,j-1,k]+as3d[i,j,k]*phi3d[i,j+1,k]\
                  +ah3d[i,j,k]*phi3d[i,j,k-1]+al3d[i,j,k]*phi3d[i,j,k+1])\
              *acrank_conv+su3d[i,j,k])/ap3d[i,j,k]

         res[i,j,k] = ap3d[i,j,k]*phi3d[i,j,k]-\
              ((ae3d[i,j,k]*phi3d[i-1,j,k]+aw3d[i,j,k]*phi3d[i+1,j,k]\
                +an3d[i,j,k]*phi3d[i,j-1,k]+as3d[i,j,k]*phi3d[i,j+1,k]\
                +ah3d[i,j,k]*phi3d[i,j,k-1]+al3d[i,j,k]*phi3d[i,j,k+1])\
                    *acrank_conv+su3d[i,j,k])/ap3d[i,j,k]

    return
```

Less related to the GPU set up, but still a difference in the code is the boundary condition. It was not clear how the boundary condition was implemented in the provided CPU-code, so we decided to switch the boundary condition to a Dirichlet boundary (constant as the initial condition) for all sides. This of course changes the results slightly but not enough so that it is not clear that the solutions are the same on the CPU and GPU. The grid also had to be greatly increased to utilize the GPU properly, the difference is then very minimal as the source then becomes very small in comparison to the domain with the current set up.

Lastly, the final row of the residual computation in the function is missing in the GPU-function. It was proven hard to reduce a matrix by summing all the all elements using Numba as the entire sum-functionality would have to be written from scratch with indexing to allow for parallelization. Therefore, this last row is now computed outside of the function using CuPy, where the sum-feature is already implemented for the GPU, this can be seen in the code below. There are some problems with this approach, for one the arrays has to be modified from Numba arrays to CuPy arrays which take unnecessary time. Also, as was mentioned earlier, we do not know what exactly is done within the CuPy sum-function, so we are less in control of the parallelization and we still don't know how it is done which is the point of this course.

```
for n in range(0,niter):
  solve_gs[blockspergrid,threadsperblock](p3d_gpu,ap3d_gpu,aw3d_gpu,ae3d_gpu,as3d_gpu,
    an3d_gpu,al3d_gpu,ah3d_gpu,su3d_gpu,convergence_limit,niter,res_gpu)
p3d = p3d_gpu.copy_to_host()
res = cp.asarray(res_gpu) #probably slow to switch, but I can't make it work in numba
resid = cp.sum(cp.abs(cp.ravel(res))).get()
```

Note that the second line simply copies the results generated in the GPU kernel back from the device to the host and that the .get() extension on the last line does the same.

In the GPU code presented above the computation takes a naive approach of always accessing the global memory. A better approach would be to introduce shared memory so that all threads in a block can be solved directly without accessing the global memory. Just like with the reduction of an array, this code has to be written from scratch. As to limit the complexity of the problem, this will not be introduced in this three dimensional code. Instead we have moved on to a diffusion case, similar to this, but in two dimensions. Although first the time savings for the current implementation will be presented in the following subsection.

3.1 Benchmarking

The benchmarking has been performed over 20 runs of 1 loop each for the Gauss-Seidel function using the python function %timeit. The mean and standard deviation of the run times are presented in Table 1. Before the benchmarking 20 runs we completed as a CPU and GPU warm-up as to make sure that the results are as fair as possible.

As can be seen in Table 1, the GPU-accelerated code performs more efficiently even for smaller simulations with a cell count of 1000 cells, although very slightly. The larger simulation, with one million cells, saw an even greater improvement, with a simulation time reduction of a factor of 12. This demonstrates that the greatest advantage of GPU acceleration is its ability to parallelize larger problems.

Table 1: Run time, including standard deviation, for the simple Poisson solver on Chalmers' computer

Number of cells	Run time CPU	Run time GPU
10^3	15.8 ms $\pm$ 971 μs	13.8 ms $\pm$ 86.3 μs
10^6	3.28 s $\pm$ 12.6 ms	263 ms $\pm$ 2.37 ms

4 Acceleration of 2D Diffusion Case

The CPU-code for the 2D Diffusion Case is a manipulation of the first task in the CFD-course, MTF072. As previously mentioned, we are aware of the fact that the code is not optimal, but as it is developed as a means to learn how to implement CFD-code it is also simple enough and a good baseline to modify for implementation on the GPU. The full modified CPU-code for the diffusion case can be found in Appendix B, where as this section will handle the accelerated parts. A brief overview of the code will be given in this section though.

The code has been divided into three functions, Poisson, Mesh generation, and Gauss-Seidel Solver where Poisson is the main function which includes both the other functions as well as the initialization of values and set-up of the GPU-kernel. Plotting of the results is done outside of the these three functions. A fourth function used to calculate the residual has been added after this profiling was done. This function has therefore not been profiled and the calculation has not been used to measure convergence. The Gauss-Seidel solver is run for a fix number of 100 iterations, where the function covers one iteration each meaning that it is called multiple times, once per iteration. Running until convergence instead of a set number of iterations would require much longer simulation times which would not be feasible to consider for benchmarking in this project.

4.1 Profiling

As a means to decide what parts of the code to accelerate the code was profiled based on the cumulative time of all processes. This was done using the python library cProfiling. The profiling showed that the Gauss-Seidel solver function took the longest time to run in general, approximately 1.3 s per iteration, see Table 2. As it was run also a total of 100 times this means a cumulative time of 130 s. The cumulative time of the Poisson function, which includes the entire program except for some importing and final plotting, was around 132 s. As expected this means that almost all time was spent on the calculations and that the set-up is negligible. However, the meshing function is also rather slow, with a cumulative time of around 0.9 s, and could be sped up as well. The main focus will be put into accelerating the Gauss-Seidel solver.

Table 2: Profiling of 2D diffusion case on the CPU

Function	Number of calls	Total time	Per Call	Cumulative time
Poisson	1	0.013	0.013	132.593
Gauss-Seidel	100	131.579	1.316	131.579
Generate mesh	1	0.998	0.998	0.998

4.2 Code acceleration

The CPU-code of the Gauss-Seidel function for the 2D diffusion case can be seen below. The input variables are first the grid parameters in x and y direction, nI and nJ, followed by the coefficients aE, aW, aS, and

aN. Finally there is the temperature, T, which is the variable that we are solving for, and lastly the source, Su. Note that the loop for Gauss-Seidel iterations is outside of the function, meaning that the function is called once per iteration.

```
# GAUSS-SEIDEL FUNCTION
def solve_gs (nI, nJ, aE, aW, aN, aS, aP, T, Su):
    # Solve for T using Gauss-Seidel
    for i in range (1,nI-1):
        for j in range (1,nJ-1):
            T[i,j] = (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + aN[i,j] * T[i,j+1]
            + aS[i,j] * T[i,j-1] + Su[i,j]) / aP[i,j]
```

Before introducing the GPU-code with shared memory we will show the naive approach. The code is very similar to the simple Poisson case, the difference is that the simple Poisson includes the calculation of residual which is not considered in this case, otherwise the code and manipulations follows the same structure as described in section 3.

```
# GAUSS-SEIDEL FUNCTION
@cuda.jit
def solve_gs (aE, aW, aN, aS, aP, T, Su):

    # Global indices
    i,j = cuda.grid(2)

    # Thread check
    if 0 < i < T.shape[0]-1 and 0 < j < T.shape[1]-1:
        T[i,j] = (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + aN[i,j] * T[i,j+1]
        + aS[i,j] * T[i,j-1] + Su[i,j]) / aP[i,j]
```

4.2.1 Shared memory implementation

Two different, but very similar, approaches has been taken when implementing the shared memory, none with a desirable outcome. As mentioned previously for the 3D simple Poisson case, incorrect results where achieved when including the iteration-loop inside of the kernel. As the idea of shared memory is that the global memory is copied into a local memory that can be accessed by all threads within a block the looping has to be performed inside of the kernel for the full potential of the shared memory to be utilized. If the looping is done outside of the kernel, the global memory will have to be copied into the local memory once for every iteration which means that the advantages of shared memory will only be applied for the one calculation instead of all calculations which means it will loose the main part of its benefit. Additionally, the global memory will have to be copied to the local one multiple times, which will most likely make the code slower than without using shared memory at all.

The other approach is to put the iteration loop inside of the kernel which should make the code faster, but will generate an incorrect result. This approach is of course not desirable in any way as a code that produces incorrect results can not be used in practice, but we have decided to show it anyway as it is an interesting issue when it comes to GPU-programming. The code with shared memory can be seen below and the correct/expected and incorrect results for 100 Gauss-Seidel iterations (i.e not converged) can be seen in Figure 1.

```
# GAUSS-SEIDEL FUNCTION
@cuda.jit
def solve_gs (aE, aW, aN, aS, aP, T, Su):

    # Define shared memory size
    SM_size = 32
    SM_size_T = 34

    # Global indices
    i,j = cuda.grid(2)

    # Thread check
```

```
    if 0 < i < T.shape[0]-1 and 0 < j < T.shape[1]-1:

        # Allocate shared memories for the coefficients
        aE_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)
        aW_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)
        aN_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)
        aS_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)
        aP_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)
        Su_sm = cuda.shared.array(shape = (SM_size, SM_size), dtype = np.float32)


        # Local indices
        li = cuda.threadIdx.x
        lj = cuda.threadIdx.y

        # Load data into the shared memory arrays
        aE_sm[li,lj] = aE[i,j]
        aW_sm[li,lj] = aW[i,j]
        aN_sm[li,lj] = aN[i,j]
        aS_sm[li,lj] = aS[i,j]
        aP_sm[li,lj] = aP[i,j]
        Su_sm[li,lj] = Su[i,j]

        #Temperature
        T_sm = cuda.shared.array(shape = (SM_size_T, SM_size_T), dtype = np.float32)
        # Local indices
        ti = cuda.threadIdx.x+1
        tj = cuda.threadIdx.y+1

        #Inner region
        T_sm[ti,tj] = T[i,j]
        #Outer region
        if ti == 1:
            T_sm[ti-1,tj] = T[i-1,j]
        if tj == 1:
            T_sm[ti,tj-1] = T[i,j-1]
        if ti == SM_size-1:
            T_sm[ti+1,tj] = T[i+1,j]
        if tj == SM_size-1:
            T_sm[ti,tj+1] = T[i,j+1]

        cuda.syncthreads()

        for n in range(nIterations):

            T_sm[ti,tj] = (aE_sm[li,lj] * T_sm[ti+1,tj] + aW_sm[li,lj] * T_sm[ti-1,tj]
                 + aN_sm[li,lj] * T_sm[ti,tj+1] + aS_sm[li,lj] * T_sm[ti,tj-1]
                        + Su_sm[li,lj]) / aP_sm[li,lj]

            #cuda.syncthreads()
            T[i,j] = T_sm[ti,tj]
```

The incorrect results in Figure 1b, show a temperature distribution that appear to be bounded by the blocks instead of the smooth square source that has been defined in the set-up phase. It appears as though a boundary condition has been implemented for every block making the temperature distribution incorrect. The issue remains when the threads are synced after every iteration and we have not been able to find what causes this behavior.

One possible idea of how it could be solved that has been found from looking at other examples of how shared memory has been implemented, is to loop over each block separately to load the data into shared memory arrays and then again when performing the calculations. We have tried to implement this approach without success and can therefore not report on whether it is the solution or not. This proved to be a rather complex implementation in regards with the indices on the matrices. In the implementation of shared memory we

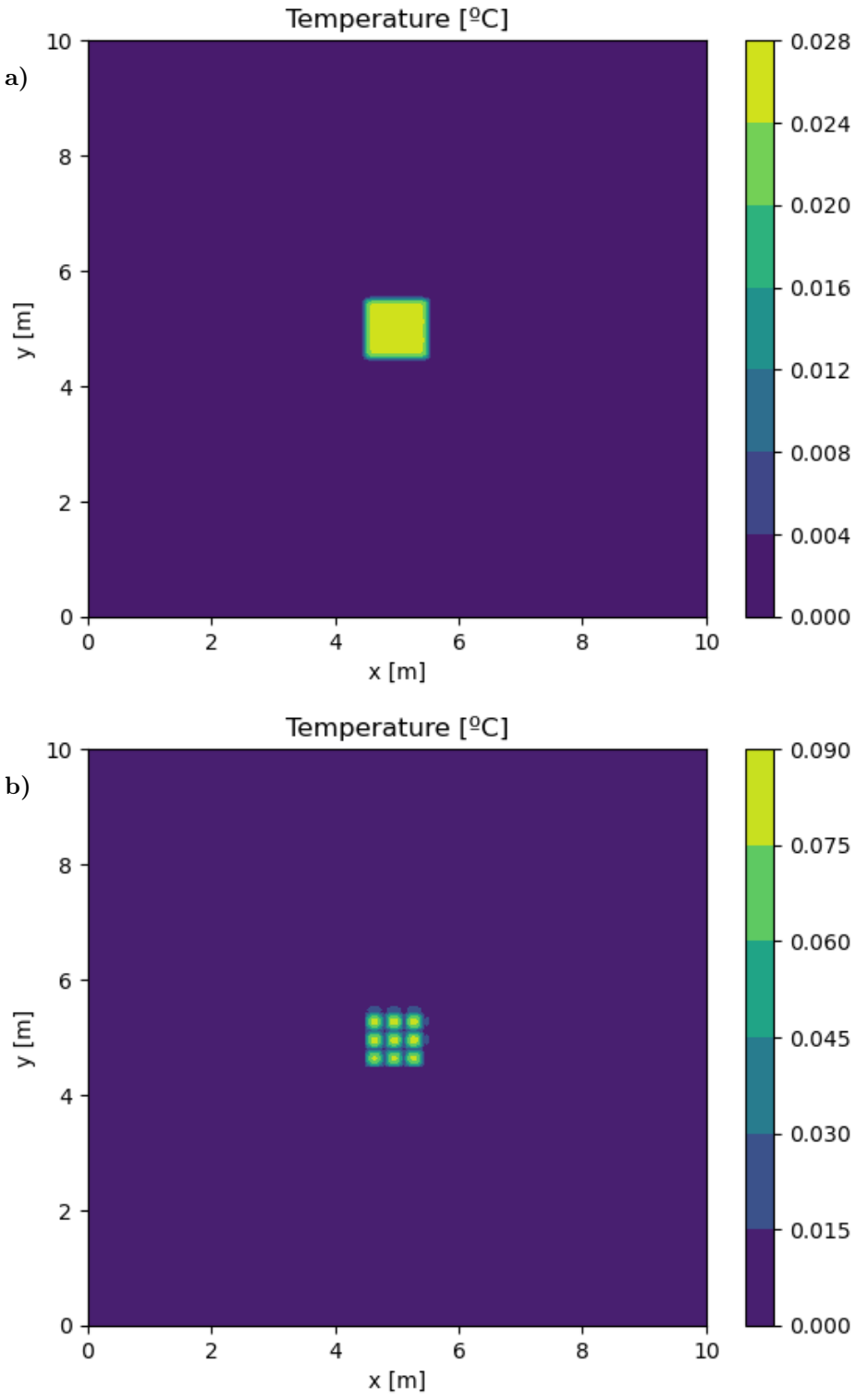

Figure 1: a) Expected results (100 iterations, not converged), b) Results produced by GPU-code with iteration-loop inside of the kernel

need to handle matrix multiplication differently from the simple case where we iterate over the block of threads and allocate shared memory from part of the matrix and atomically add the sum. The indices in the matrix T need to be shifted +1/-1 in position in the Gauss-Seidel equation. We could make use of halo cells in the original result matrix, but it was too complex of a solution to find in time.

4.2.2 Benchmarking

In Table 3 the benchmarking for the 2D Diffusion Case run on two different meshes for three different graphic cards can be seen. Note that the run times presented for this case is not the run time for a converged solution but with a number of Gauss-Seidel iterations fixed at 100 and without running the residual calculation. As in the previous case, the benchmarking has been performed over 20 runs of 1 loop each python function %timeit with 20 runs of warm-up before hand. In this case it is the entire Poisson function that has been time, i.e the set-up and meshing as well as the solver, although as the profiling showed the Gauss-Seidel is the most time-consuming part.

Table 3: Run time, including standard deviation, for the 2D diffusion case using Gauss-Seidel

Number of cells	Run time CPU double-precision	Run time CPU single-precision	Run time GPU double-precision	Run time GPU single-precision	Run time GPU shared memory
NVIDIA GeForce GTX-1060 6GB					
10^4	1.35 s ± 3.02 ms	1.37 s ± 2.9 ms	19.8 ms ± 90 μs	43.8 ms ± 613 μs	21.3 ms ± 180 μs
10^6	133 s ± 2.3 s	134 s ± 47.9 ms	1.13 s ± 5.34 ms	3.48 s ± 49 ms	1.93 s ± 10.6 ms
NVIDIA GeForce GTX-1650 4GB					
10^4	2.09 s ± 14.2 ms	1.92 s ± 19.3 ms	61.4 ms ± 714 μs	60.5 ms ± 1.84 ms	49 ms ± 910 μs
10^6	210 s ± 519 ms	186 s ± 3.55 s	3.34 s ± 101 ms	3.21 s ± 61.1 ms	3.27 s ± 68.2 ms
NVIDIA GeForce RTX-3060 8GB					
10^4	1.157 s ± 16.1 ms	0.98 s ± 6.5 ms	43.7 ms ± 414.5 μs	46.2 ms ± 467.5 μs	54.7 ms ± 232 μs
10^6	108.5 s ± 12.86 ms	94.7 s ± 876 ms	1.943 s ± 24.3 ms	1.726 s ± 18.3 ms	2.03 s ± 19.5 ms

The run time presented for the GPU-code with shared memory is the code with the iteration done inside of the kernel even though this produces incorrect results. The reason for this is that if the looping is done outside of the kernel the shared memory is regenerated for every iteration which defeats the purpose. When the iteration is done inside of the kernel we can see that the run time is shorter than for the GPU-code where shared memory is not implemented for two out of the three graphic cards as expected. The difference is not as significant as we would have expected though. The minimal difference is probably cause by the fact that there is something wrong with the results in the shared memory case making the two cases incomparable.

The important part to note is that the naive GPU code, without shared memory, is significantly faster than the code run on the CPU. Already at a cell count of 10 000 the speed-up is around a factor of 20 depending on the graphics card. When we have one million cells the speed-up is at a factor of 70. It should also be noted that even though we have seen large impacts on the simulation time of the GPU-code, the original CPU-code was not optimized to begin with. This means that if we had started with faster CPU-code, the GPU speed-up might not have been as impressive. The idea was to also accelerate some faster method than the Gauss-Seidel, for example TDMA, and compare the speed-up. Unfortunately, we ran out of time for this due to spending the time on trying to solve the issues with shared memory. We believe that the run times using TDMA would be much lower in general, due to it being a sparse matrix solver, however it would be likely that this means that the speed-up would not be as significant as the code was more optimal to beging with.

For the benchmarking showed above, both double and single precision has been used i.e floating-point numbers that uses 64 or 32 bits respectively to represent a number. Because double precision uses twice as many bits as single precision, it requires more memory and processing power to perform operations on

double precision numbers but single precision provides less precision. For two out of the three GPUs, the computational time is reduced by introducing single precision, although not by a large capacity. However, as can be seen in this case the double-precision actually performs better than single-precision for the NVIDIA GeForce GTX-1060 6GB. There is not a significant difference in most cases, except for the fine-mesh GPU-case, where the time more than doubles for the single-precision. We cannot explain this behavior, and the benchmarking has been rerun to double check the results without change. Overall, it can be said that there should be a slight improvement in limiting the variable to single precision, that could be beneficial for larger problems than this, but in this smaller case is barely noticeable.

5 Conclusion

The advantage of using the parallelization power of a GPU for CFD-simulations is clear. Even with quite small efforts, large improvements of simulation time was achieved. However, it got very complicated fast after that initial step. Although this group consisted of different backgrounds that could help in understanding the complexity of the problems, we did not have enough knowledge when trying our hands at more complicated coding.

It is still unclear how much of a speed-up one would get when the baseline CPU- code is more optimized than what has been used in this project. However, utilizing the GPU overall can provide a speedup given that the underline computation is matrix multiplication, which is essentially faster on the external hardware. In a production environment, even if the speedup is 10% that could translate into a significant amount of hours for simulations that CFD engineers can save for example. Here is also important to mention that when testing the performance of the code on a larger system such as a server, there might be also an additional overhead of data transferring between the host CPU and the GPU since they might not be on the same machine.

Writing the code in Python using the libraries provided, can be beneficial and fast to experiment and get early results. We could easily avoid to have to deal with nested for loops for matrix multiplication, or writing CUDA specific code, things we would have encountered doing it in C. But, in our opinion it is better for debugging and understanding the system to use the native api to the heterogeneous hardware, rather than depending on an external Python library. A better solution to do this would have been to use the Thrust Cuda library that is available by Nvidia, and is made to mimic the standard template library in C++11 and above version. It makes things easier, and is much more readable and easy to use rather than C. It is also included in the latest version of CUDA 12, under the cuda namespace. However, there is no support yet for shared memory, something that we tried to implement in this project. It would have been worthwhile to try this solution, but it has been out of our scope given the timeline.

Appendix

A Simple Poisson for GPU

```
import sys
import math
import time
from timeit import default_timer as timer
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from numba import cuda, float32
import cupy as cp

@cuda.jit
def solve_gs(phi3d,ap3d,aw3d,ae3d,as3d,an3d,al3d,ah3d,su3d,tol_conv,nmax,res):
    print('solve_3d gs called,nmax=',nmax)
    acrank_conv = 1
    i,j,k = cuda.grid(3)

    #Including shared memory should be faster

    if i < phi3d.shape[0] and j < phi3d.shape[1] and k < phi3d.shape[2]:

       phi3d[i,j,k] = ((ae3d[i,j,k]*phi3d[i-1,j,k]+aw3d[i,j,k]*phi3d[i+1,j,k]\
                +an3d[i,j,k]*phi3d[i,j-1,k]+as3d[i,j,k]*phi3d[i,j+1,k]\
                +ah3d[i,j,k]*phi3d[i,j,k-1]+al3d[i,j,k]*phi3d[i,j,k+1])\
            *acrank_conv+su3d[i,j,k])/ap3d[i,j,k]

        res[i,j,k] = ap3d[i,j,k]*phi3d[i,j,k]-\
            ((ae3d[i,j,k]*phi3d[i-1,j,k]+aw3d[i,j,k]*phi3d[i+1,j,k]\
              +an3d[i,j,k]*phi3d[i,j-1,k]+as3d[i,j,k]*phi3d[i,j+1,k]\
              +ah3d[i,j,k]*phi3d[i,j,k-1]+al3d[i,j,k]*phi3d[i,j,k+1])\
            *acrank_conv+su3d[i,j,k])/ap3d[i,j,k]

    #resid=np.sum(np.abs(res.flatten())) #Now done outside function
    return

def poisson(solver,niter,convergence_limit):

    #global ni,nj,nk,x,y,z,p3d
    #print('\nhostname: ',socket.gethostname())
    print('\nsolver,convergence_limit,niter',solver,convergence_limit,niter)

    # set grid x
    xmax=1
    ni=10
    dx=np.float32(xmax/ni)
    x = np.linspace(0, xmax, ni, dtype=np.float32)

    # set grid y
    ymax=1
    nj=10
    dy=np.float32(ymax/nj)
    y = np.linspace(0, ymax, nj, dtype=np.float32)

    # set grid z
    zmax=1
    nk=10
    dz=np.float32(zmax/nk)
    z = np.linspace(0, zmax, nk, dtype=np.float32)


    # initial coefficients
    aw3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
```

```
    ae3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    as3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    an3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    al3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    ah3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    ap3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20
    su3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20

    # initial solution
    p3d=np.ones((ni,nj,nk), dtype=np.float32)*1e-20

    # compute coefficients, see Chapter 4, Eq. 17 where a 2D version is derived

    # http://www.tfd.chalmers.se/~lada/comp_fluid_dynamics/lecture_notes.html

    aw3d=np.ones((ni,nj,nk), dtype=np.float32)*dy*dz/dx
    ae3d=np.ones((ni,nj,nk), dtype=np.float32)*dy*dz/dx

    as3d=np.ones((ni,nj,nk), dtype=np.float32)*dx*dz/dy
    an3d=np.ones((ni,nj,nk), dtype=np.float32)*dx*dz/dy

    al3d=np.ones((ni,nj,nk), dtype=np.float32)*dx*dy/dz
    ah3d=np.ones((ni,nj,nk), dtype=np.float32)*dx*dy/dz

    ap3d=aw3d+ae3d+as3d+an3d+al3d+ah3d

    su3d[int(0.4*ni):int(0.6*ni),int(0.4*nj):int(0.6*nj),int(0.4*nk):int(0.6*nk)]=100*dx*dy*
    dz

    ap3d_gpu = cuda.to_device(ap3d)
    aw3d_gpu = cuda.to_device(aw3d)
    ae3d_gpu = cuda.to_device(ae3d)
    an3d_gpu = cuda.to_device(an3d)
    as3d_gpu = cuda.to_device(as3d)
    al3d_gpu = cuda.to_device(al3d)
    ah3d_gpu = cuda.to_device(ah3d)
    su3d_gpu = cuda.to_device(su3d)
    p3d_gpu  = cuda.to_device(p3d)


    threadsperblock = (8, 8, 8) #What should this be to be efficient?
    blockspergrid_x = math.ceil(p3d.shape[0]/threadsperblock[0])
    blockspergrid_y = math.ceil(p3d.shape[1]/threadsperblock[1])
    blockspergrid_z = math.ceil(p3d.shape[2]/threadsperblock[2])
    blockspergrid = (blockspergrid_x, blockspergrid_y, blockspergrid_z)

    res_gpu = cuda.device_array((ni,nj,nk))

    # call solver
    for n in range(0,niter):
        solve_gs[blockspergrid,threadsperblock](p3d_gpu,ap3d_gpu,aw3d_gpu,ae3d_gpu,as3d_gpu,
    an3d_gpu,al3d_gpu,ah3d_gpu,su3d_gpu,convergence_limit,niter,res_gpu)
     p3d = p3d_gpu.copy_to_host()
     res = cp.asarray(res_gpu) #probably slow to switch
     resid = cp.sum(cp.abs(cp.ravel(res))).get()

    return nk,x,y,p3d

#Choose Solver
solver='gs'

# number of iterations in GS solver
niter=100

# convergence limit
convergence_limit=1e-7
```

```
126
127 start= time.time()
128
129 nk,x,y,p3d = poisson(solver,niter,convergence_limit)
130
131 print("time", time.time()-start)
132
133 plt.close('all')
134 plt.interactive(True)
135 plt.rcParams.update({'font.size': 22})
136
137
138 ############## plot results
139 fig1,ax1 = plt.subplots()
140 plt.subplots_adjust(left=0.20,bottom=0.20)
141 # plot results in mid-plane in z direction
142 nk2=int(nk/2)
143 plt.contourf(x, y, p3d[:,:,nk2], 20, cmap='RdGy')
144 plt.ylabel('$y$')
145 plt.xlabel('$x$')
146 plt.title('$\phi$ in plane $z=z_{max}/2$')
147 plt.colorbar();
148 plt.savefig('poisson-p3d.png',bbox_inches='tight')
```

B 2D Diffusion for CPU

```
1  # Packages needed
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import time
5  from numba import jit
6
7  # SOLVER INPUTS
8  nIterations  = 100 # maximum number of iterations
9  resTolerance = 0.001 # convergence criteria for residuals each variable
10
11
12 # MESH GENERATION FUNCTION
13 def mesh_gen(mI, mJ, xL, yL, dx, dy, x_M, y_M, x_N, y_N):
14     # Fill the coordinates
15     for i in range(mI):
16         for j in range(mJ):
17             # For the mesh points
18             x_M[i,j] = i*dx
19             y_M[i,j] = j*dy
20
21             # For the nodes
22             if i > 0:
23                 x_N[i,j] = 0.5*(x_M[i,j] + x_M[i-1,j])
24             if i == (mI-1) and j>0:
25                 y_N[i+1,j] = 0.5*(y_M[i,j] + y_M[i,j-1])
26             if j >0:
27                 y_N[i,j] = 0.5*(y_M[i,j] + y_M[i,j-1])
28             if j == (mJ-1) and i>0:
29                 x_N[i,j+1] = 0.5*(x_M[i,j] + x_M[i-1,j])
30
31     x_N[-1,:] = xL  # the x-coordinate of nodes on the right (east) boundary
32     y_N[:,-1] = yL  # the y-coordinate of nodes on the top (north) boundary
33
34     return
35
36
37 # GAUSS-SEIDEL FUNCTION
38 def solve_gs (nI, nJ, aE, aW, aN, aS, aP, T, Su):
39     # Solve for T using Gauss-Seidel
40     for i in range (1,nI-1):
```

```
        for j in range (1,nJ-1):
            T[i,j] = (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + aN[i,j] * T[i,j+1] + aS[i,j] * T[i,j-1]
                      + Su[i,j]) / aP[i,j]


# TDMA FUNCTION
def solve_tdma (nI, nJ, aE, aW, aN, aS, aP, T, Su, Px, Qx, Py, Qy):

    # x-direction
    for j in range (1,T.shape[1]-1):
        i = 1
        a = aP[i,j]
        b = aE[i,j]
        c = aW[i,j]
        d = aN[i,j] * T[i,j+1] + aS[i,j] * T[i,j-1] +Su[i,j]

        Px[i,j] = b / a
        Qx[i,j] = (d + c * T[i-1,j]) / a

        i = T.shape[0]-2
        a = aP[i,j]
        b = aE[i,j]
        c = aW[i,j]
        d = aN[i,j] * T[i,j+1] + aS[i,j] * T[i,j-1] + Su[i,j]

        Px[i,j] = 0
        Qx[i,j] = (d + c * Qx[i-1,j] + b * T[i+1,j]) / (a - c * Px[i-1,j])

        for i in range (2,T.shape[0]-2):
            a = aP[i,j]
            b = aE[i,j]
            c = aW[i,j]
            d = aN[i,j] * T[i,j+1] + aS[i,j] * T[i,j-1] + Su[i,j]

            Px[i,j] = b / (a - c * Px[i-1,j])
            Qx[i,j] = (d + c * Qx[i-1,j]) / (a - c * Px[i-1,j])


        for i in reversed (range (1,T.shape[0]-1)):
            T[i,j] = Px[i,j] * T[i+1,j] + Qx[i,j]

    # y-direction
    for i in range (1,T.shape[0]-1):
        j = 1
        a = aP[i,j]
        b = aN[i,j]
        c = aS[i,j]
        d = aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + Su[i,j]

        Py[i,j] = b / a
        Qy[i,j] = (d + c * T[i,j-1]) / a

        j = T.shape[1]-2
        a = aP[i,j]
        b = aN[i,j]
        c = aS[i,j]
        d = aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + Su[i,j]


        Py[i,j] = 0
        Qy[i,j] = (d + c * Qy[i,j-1] + b * T[i,j+1]) / (a - c * Py[i,j-1])

        for j in range (2,T.shape[1]-2):
            a = aP[i,j]
            b = aN[i,j]
```

```
            c = aS[i,j]
            d = aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + Su[i,j]

            Py[i,j] = b / (a - c * Py[i,j-1])
            Qy[i,j] = (d + c * Qy[i,j-1]) / (a - c * Py[i,j-1])

        for j in reversed (range (1,T.shape[1]-1)):
            T[i,j] = Py[i,j] * T[i,j+1] + Qy[i,j]


# RESIDUALS FUNCTION
def solve_res (nI, nJ, aE, aW, aN, aS, aP, T, Su, residuals):
    r = 0
    # F is the temperature flux at the boundaries
    F = 0
    F = np.sum(Su[:,:])

    for i in range (1, nI-1):
        for j in range (1, nJ-1):
            r += abs(aP[i,j] * T[i,j] - (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + aN[i,j] *
 T[i,j+1] \
              + aS[i,j] * T[i,j-1] + Su[i,j]))

    r = r/F
    residuals.append(r)


# MAIN FUNCTION
def poisson (nIterations, resTolerance):
    # GEOMETRIC INPUTS

    mI = 20 # number of mesh points X direction.
    mJ = 20 # number of mesh points Y direction.
    xL =  10 # length of the domain in X direction
    yL =  10 # length of the domain in Y direction

    # Allocate all needed variables
    nI = mI + 1                    # number of nodes in the X direction. Nodes
                                   # added in the boundaries
    nJ = mJ + 1                    # number of nodes in the Y direction. Nodes
                                   # added in the boundaries
    Su      = np.zeros((nI,nJ))    # source term for temperature
    T       = np.zeros((nI,nJ))    # temperature matrix
    k       = np.zeros((nI,nJ))    # coefficient of conductivity
    aE      = np.zeros((nI,nJ))    # coefficient for east node
    aW      = np.zeros((nI,nJ))    # coefficient for west node
    aN      = np.zeros((nI,nJ))    # coefficient for north node
    aS      = np.zeros((nI,nJ))    # coefficient for south node
    aP      = np.zeros((nI,nJ))    # coefficient for P node
    Px      = np.zeros((nI,nJ))
    Py      = np.zeros((nI,nJ))
    Qx      = np.zeros((nI,nJ))
    Qy      = np.zeros((nI,nJ))

    residuals = [] # List containing the value of the residual for each iteration

    # Generate mesh and compute geometric variables
        # Control volume size
    dx = xL/(mI - 1)
    dy = yL/(mJ - 1)

    # Allocate all variables matrices
    x_M       = np.zeros((mI,mJ)) # X coords of the mesh points/cell corners
    y_M       = np.zeros((mI,mJ)) # Y coords of the mesh points/cell corners
    x_N       = np.zeros((nI,nJ)) # X coords of the nodes (inner nodes, boundary nodes, ...)
    y_N       = np.zeros((nI,nJ)) # Y coords of the nodes (inner nodes, boundary nodes, ...)
```

```
171
172     # PARAMETERS
173     k = 10
174     Su[int(0.45*nI):int(0.55*nI), int(0.45*nJ):int(0.55*nJ)] = 100 * dx * dy
175
176     # Define coefficients
177     aE[1:nJ-1, 1:nJ-1] = k * dy / dx    # East
178     aW[1:nJ-1, 1:nJ-1] = k * dy / dx    # West
179     aN[1:nJ-1, 1:nJ-1] = k * dy / dx    # North
180     aS[1:nJ-1, 1:nJ-1] = k * dy / dx    # South
181     aP[1:nJ-1, 1:nJ-1] = aE[1:nJ-1, 1:nJ-1] + aW[1:nJ-1, 1:nJ-1] + aN[1:nJ-1, 1:nJ-1] + aS
        [1:nJ-1, 1:nJ-1] # P
182
183     # Dirichlet boundary condition
184     T[0,:]     = 0
185     T[nI-1,:] = 0
186     T[:,0]     = 0
187     T[:,nJ-1] = 0
188
189
190     # Call the mesh generation function
191     mesh_gen(mI, mJ, xL, yL, dx, dy, x_M, y_M, x_N, y_N)
192
193     for n in range(nIterations):
194
195         # Call a linear solver
196         solve_gs(nI, nJ, aE, aW, aN, aS, aP, T, Su)
197         # solve_tdma (nI, nJ, aE, aW, aN, aS, aP, T, Su, Px, Qx, Py, Qy)
198
199         # Compute residuals
200         solve_res(nI, nJ, aE, aW, aN, aS, aP, T, Su, residuals)
201
202         # print('iteration: %d \n' % (n))
203         # print('residual:', residuals[-1], '\n')
204
205         # Check convergence
206         if resTolerance>residuals[-1]:
207               break
208
209     return x_N, y_N, T
210
211 # Benchmark
212 start = time.time()
213
214 x_N, y_N, T = poisson(nIterations, resTolerance)
215
216 print('time =', time.time()-start)
217
218 # # Plot results
219 plt.figure()
220
221 # Plot temperature contour
222 # plt.subplot(2,2,2)
223 plt.contourf(x_N, y_N, T)
224 plt.colorbar()
225 plt.title('Temperature [ C ]')
226 plt.xlabel('x [m]')
227 plt.ylabel('y [m]')
228 #plt.axis('equal')
229
230 plt.show()
```

C 2D Diffusion for GPU

```
1 # Packages needed
2 import numpy as np
```

```
import matplotlib.pyplot as plt
import math
from numba import cuda, vectorize, jit
import time


# SOLVER INPUTS
nIterations  = 100 # maximum number of iterations
resTolerance = 0.001 # convergence criteria for residuals each variable


# MESH GENERATION FUNCTION
def mesh_gen(mI, mJ, xL, yL, dx, dy, x_M, y_M, x_N, y_N):
    # Fill the coordinates
    for i in range(mI):
        for j in range(mJ):
            # For the mesh points
            x_M[i,j] = i*dx
            y_M[i,j] = j*dy

            # For the nodes
            if i > 0:
                x_N[i,j] = 0.5*(x_M[i,j] + x_M[i-1,j])
            if i == (mI-1) and j>0:
                y_N[i+1,j] = 0.5*(y_M[i,j] + y_M[i,j-1])
            if j >0:
                y_N[i,j] = 0.5*(y_M[i,j] + y_M[i,j-1])
            if j == (mJ-1) and i>0:
                x_N[i,j+1] = 0.5*(x_M[i,j] + x_M[i-1,j])

    x_N[-1,:] = xL  # the x-coordinate of nodes on the right (east) boundary
    y_N[:,-1] = yL  # the y-coordinate of nodes on the top (north) boundary

    return


# GAUSS-SEIDEL FUNCTION
@cuda.jit
def solve_gs (aE, aW, aN, aS, aP, T, Su):

    # Global indices
    i,j = cuda.grid(2)

    # Thread check
    if 0 < i < T.shape[0]-1 and 0 < j < T.shape[1]-1:


        T[i,j] = (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] + aN[i,j] * T[i,j+1] + aS[i,j] * T
    [i,j-1]
                          + Su[i,j]) / aP[i,j]


# RESIDUALS FUNCTION
@cuda.jit
def solve_res_gpu (aE, aW, aN, aS, aP, T, Su, residuals, F):
    # Global indices
    i,j = cuda.grid(2)

    # Thread check
    if 0 < i < T.shape[0]-1 and 0 < j < T.shape[1]-1:

        residuals[i,j] = abs(aP[i,j] * T[i,j] - (aE[i,j] * T[i+1,j] + aW[i,j] * T[i-1,j] +
    aN[i,j] * T[i,j+1]
                + aS[i,j] * T[i,j-1] + Su[i,j]))

        cuda.syncthreads()
```

```
67
68
69
70 # MAIN FUNCTION
71 def poisson (nIterations, resTolerance):
72
73     mI = 1000 # number of mesh points X direction.
74     mJ = 1000 # number of mesh points Y direction.
75     xL =  10 # length of the domain in X direction
76     yL =  10 # length of the domain in Y direction
77
78     # Allocate all needed variables
79     nI = mI + 1                    # number of nodes in the X direction. Nodes
80                                    # added in the boundaries
81     nJ = mJ + 1                    # number of nodes in the Y direction. Nodes
82                                    # added in the boundaries
83     Su      = np.zeros((nI,nJ))    # source term for temperature
84     T       = np.zeros((nI,nJ))    # temperature matrix
85     aE      = np.zeros((nI,nJ))    # coefficient for east node
86     aW      = np.zeros((nI,nJ))    # coefficient for west node
87     aN      = np.zeros((nI,nJ))    # coefficient for north node
88     aS      = np.zeros((nI,nJ))    # coefficient for south node
89     aP      = np.zeros((nI,nJ))    # coefficient for P node
90
91     # Change the residual data structure (from list to array)
92     residuals = np.zeros((nI,nJ)) # List containing the value of the residual for each
       iteration
93
94     # Generate mesh and compute geometric variables
95         # Control volume size
96     dx = xL/(mI - 1)
97     dy = yL/(mJ - 1)
98
99     # Allocate all variables matrices
100    x_M       = np.zeros((mI,mJ)) # X coords of the mesh points/cell corners
101    y_M       = np.zeros((mI,mJ)) # Y coords of the mesh points/cell corners
102    x_N       = np.zeros((nI,nJ)) # X coords of the nodes (inner nodes, boundary nodes, ...)
103    y_N       = np.zeros((nI,nJ)) # Y coords of the nodes (inner nodes, boundary nodes, ...)
104
105    # PARAMETERS
106    k = 10
107    Su[int(0.45*nI):int(0.55*nI), int(0.45*nJ):int(0.55*nJ)] = 100 * dx * dy
108
109    # Define coefficients
110    aE[1:nJ-1, 1:nJ-1] = k * dy / dx   # East
111    aW[1:nJ-1, 1:nJ-1] = k * dy / dx   # West
112    aN[1:nJ-1, 1:nJ-1] = k * dy / dx   # North
113    aS[1:nJ-1, 1:nJ-1] = k * dy / dx   # South
114    aP[1:nJ-1, 1:nJ-1] = aE[1:nJ-1, 1:nJ-1] + aW[1:nJ-1, 1:nJ-1] + aN[1:nJ-1, 1:nJ-1] + aS
       [1:nJ-1, 1:nJ-1] # P
115
116    # Dirichlet boundary condition
117    T[0,:]     = 0
118    T[nI-1,:] = 0
119    T[:,0]     = 0
120    T[:,nJ-1] = 0
121
122    # Copy to GPU
123    Su_gpu = cuda.to_device(Su)
124    T_gpu  = cuda.to_device(T)
125    aE_gpu = cuda.to_device(aE)
126    aW_gpu = cuda.to_device(aW)
127    aN_gpu = cuda.to_device(aN)
128    aS_gpu = cuda.to_device(aS)
129    aP_gpu = cuda.to_device(aP)
130    residuals_gpu = cuda.to_device(residuals)
```

```
131
132     # Call the mesh generation function
133     mesh_gen(mI, mJ, xL, yL, dx, dy, x_M, y_M, x_N, y_N)
134
135     # Define block dimension
136     threadsperblock = (32, 32)
137     blockspergrid_x = math.ceil(T.shape[0] / threadsperblock[0])
138     blockspergrid_y = math.ceil(T.shape[1] / threadsperblock[1])
139
140     blockspergrid = (blockspergrid_x, blockspergrid_y)
141
142
143     # Normilization factor F
144     F = np.sum(Su[:,:])
145
146     for n in range(nIterations):
147
148         # Kernel launch for Gauss-Seidel
149         solve_gs[blockspergrid, threadsperblock](aE_gpu, aW_gpu, aN_gpu, aS_gpu, aP_gpu,
    T_gpu, Su_gpu)
150
151         # Kernel launch for residual computation
152         solve_res_gpu[blockspergrid, threadsperblock](aE_gpu, aW_gpu, aN_gpu, aS_gpu, aP_gpu
    , T_gpu, Su_gpu, residuals_gpu, F)
153
154         residuals = residuals_gpu.copy_to_host()
155
156         # Normalize the residual
157         residuals_norm = np.sum(residuals[:,:])/F
158
159         print('iteration: %d \n' % (n))
160         print('residual:', residuals_norm, '\n')
161
162         # Check convergence
163         if resTolerance>residuals_norm:
164               break
165
166     T = T_gpu.copy_to_host()
167
168     return T, x_N, y_N
169
170 # Benchmark
171 start = time.time()
172
173 T, x_N, y_N = poisson(nIterations, resTolerance)
174
175 print('time =', time.time()-start)
176
177 # # Plot results
178 plt.figure()
179
180 # Plot temperature contour
181 # plt.subplot(2,2,2)
182 plt.contourf(x_N, y_N, T)
183 plt.colorbar()
184 plt.title('Temperature [ C ]')
185 plt.xlabel('x [m]')
186 plt.ylabel('y [m]')
187 plt.savefig('initial_result.svg', format='svg')
188 #plt.axis('equal')
189
190 plt.show()
```