Chalmers University of Technology

Project in
Applied Mechanics

GPU Accelerated CFD using CUDA

- TME131 -

2022 / 05 / 19

David Andersson
daanders@student.chalmers.se

Robert Ranman
ranman@student.chalmers.se

Shisheer Shetty
Shisheer@student.chalmers.se

Frowin Winkes
frowin@student.chalmers.se

Abstract

The GPU programming extension CUDA, developed by NVIDIA and released in 2007, has over the last years lead to a numerous advances in compute capabilities. The CUDA API enables relatively easy access to the graphics card hardware, which enables the user to perform parallel computations with thousands of CUDA cores.

The following report investigates the methodology and advantages of using the CUDA API for CFD computations. To do this, an existing code written for the CPU was rewritten to run on the GPU. It has been evaluated in terms of the reduction in iteration time, as well as development difficulty. The code was rewritten using Numba, which is a Python module that allows the user to access the CUDA API in the Python programming language. Furthermore, a simpler CFD code has also been written from scratch, where the GPU computations were taken into consideration at the first implementation stage. This is used as small comparison.

Both implementations show significant improvements with regard to iteration time, independent of GPU model or architecture. For larger domain sizes the performance increased reached a factor 18. It has also been shown how the GPU can be maximized for up to an additional 25-30% performance by simply varying certain parameters, boosting the overall performance of the code to generate a performance increase of a factor 26. Another investigation dives into the overhead associated with programming memory intensive scripts to the GPU and shows what effect this has on the total times for the application.

These results are followed by a discussion about the implementation limitations and for what problems it would be considered worthwhile to target the code for execution on the GPU.

Abbreviations

CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
float32	32bit Floating Point Number (Data type)
float64	64bit Floating Point Number (Data type)
GMRES	Generalized Minimal Residual (Method)
GPU	Graphics Processing Unit
RANS	Reynolds Averaged Navier-Stokes
SM	Streaming Multiprocessor

Contents

1 Introduction

The increased capabilities and computational power of graphics cards has during the last decade spurred a transition to where more and more computations are transferred to the GPU. The main advantage of the GPU-architecture is that it facilitates parallel execution, which means that large amounts of calculations can occur simultaneously.

To facilitate programming for the GPU, the large GPU manufacturer; NVIDIA, has developed a GPU programming extension called CUDA. Using the extension, every compatible NVIDIA GPU is able to execute that code in a parallel manner.

Running a sequentially executed code in a CFD program for fine mesh resolutions is devastatingly slow. In order to take one step towards fully utilizing the power of the computational resources which are available, this project aimed to convert the in-house developed CFD-code *pyCalc-RANS* to run completely on the GPU [1].

1.1 Provided RANS Solver

The provided RANS solver is a fully vectorized code used to solve the two-dimensional, steady, incompressible momentum equations. It solves both the laminar the turbulent case, where in the turbulent case the $k - \omega$ turbulence model is used. The code does not solve for density, but uses the continuity equation to create a pressure correction equation, according to the SIMPLEC method. The discretized equations are solved with sparse matrix solvers using either a direct solver, or a GMRES-type solver.

The code is implemented in Python, and solves the lid driven cavity.

1.2 Lid Driven Cavity

Lid driven cavity is a common, and simple case used for many CFD simulations. It consists of a system confined by three stationary no-slip walls and one moving wall. The moving wall, which is usually placed at the top of the domain, acts as a boundary condition for the u-velocity which propagates into a rotational flow as the moving fluid interacts with the walls, see figure 1.1. The arrows denote the direction of the flow, while the contour defines the magnitude of the velocity.

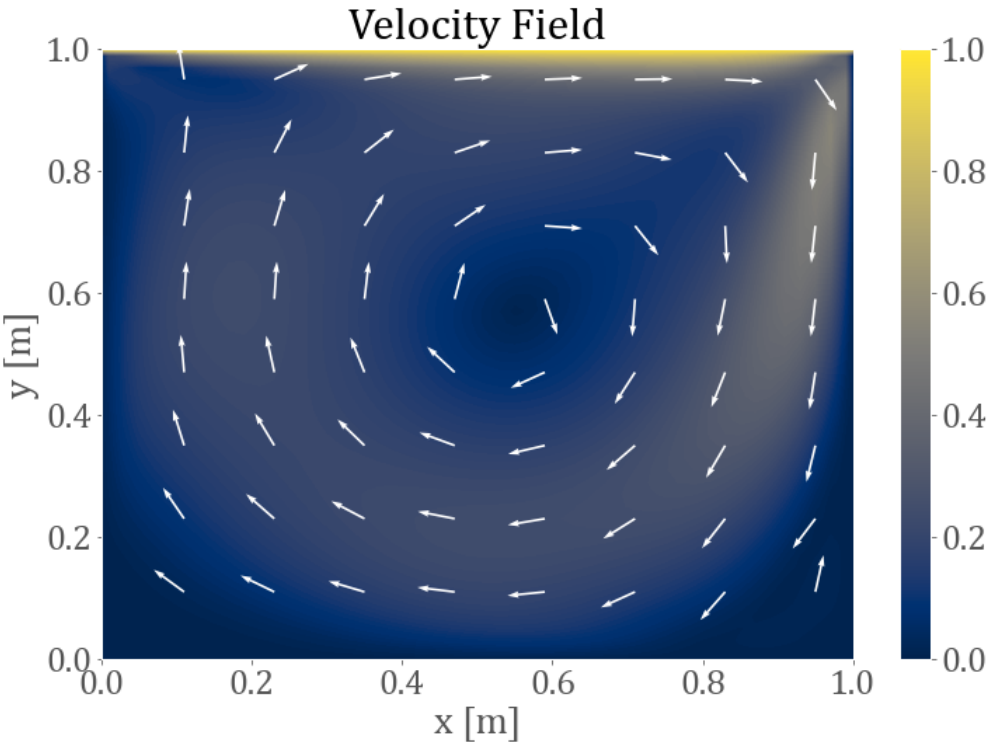

Figure 1.1: Velocity field of the lid driven cavity, with $u_{wall} = 1$ [m/s].

1.3 Delimitations

To delimit the project, the following boundaries will confine our work:

- Only the laminar flow type will be converted to run on the GPU
- The primary focus is on the CUDA conversion.

1.4 Limitations

The project is limited by the following factors

- Lid driven cavity is the only case which will be studied.
- GPU-brands other than NVIDIA will not be supported by this project.

2
Theory and Methodology

In this chapter, a brief summary of relevant theory behind the CUDA framework, as well as GPU memory architecture, has been presented. The theory presented in this chapter is selected and motivated by the implementation strategy of the code, as well as to enlighten the reader of the key components of the CUDA framework, or parallelization in general.

Since the CUDA framework is an extension of the C++ programming language, there were also short comparisons made between the implementation procedures using C++ and Python.

2.1 GPU Architecture

Before introducing CUDA and its implementation, the reader should have a basic understanding of the memory architecture of a normal CPU and GPU. Therefore, this first section aims to provide sufficient background information to facilitate a good understanding later on.

The CPU and GPU are two physically separate entities, with unique memory locations. There can be no implicit communication between the two entities, which means that the user must explicitly transfer specific data either from the CPU to the GPU, or the other way around. In practice, the programmer will need two copies of the data to be passed to the GPU. One copy on the host side (CPU), which is initialized in some pre-defined process, and one copy which has been allocated, and transferred to the device (GPU).

In figure 2.1 we can see a schematic view of the memory architecture for the host, on the left, as well as for the device, on the right. The GPU consists of multiple streaming multiprocessors (SM), which are the fundamental building blocks of every GPU [2]. A large amount of SMs will generally, simply put, create a more powerful GPU. The amount of SMs on modern GPUs today are on the order of 10.

Inside an SM there are a certain number of *cores*. How many cores that are available depends on the GPU model and architecture [3]. Inside the cores the coding instructions take place. Hence, having access to many SMs open up more cores, and thereby more places to execute code. Thus, having many SMs increase the upper limit to how much parallelization can occur at the same time.

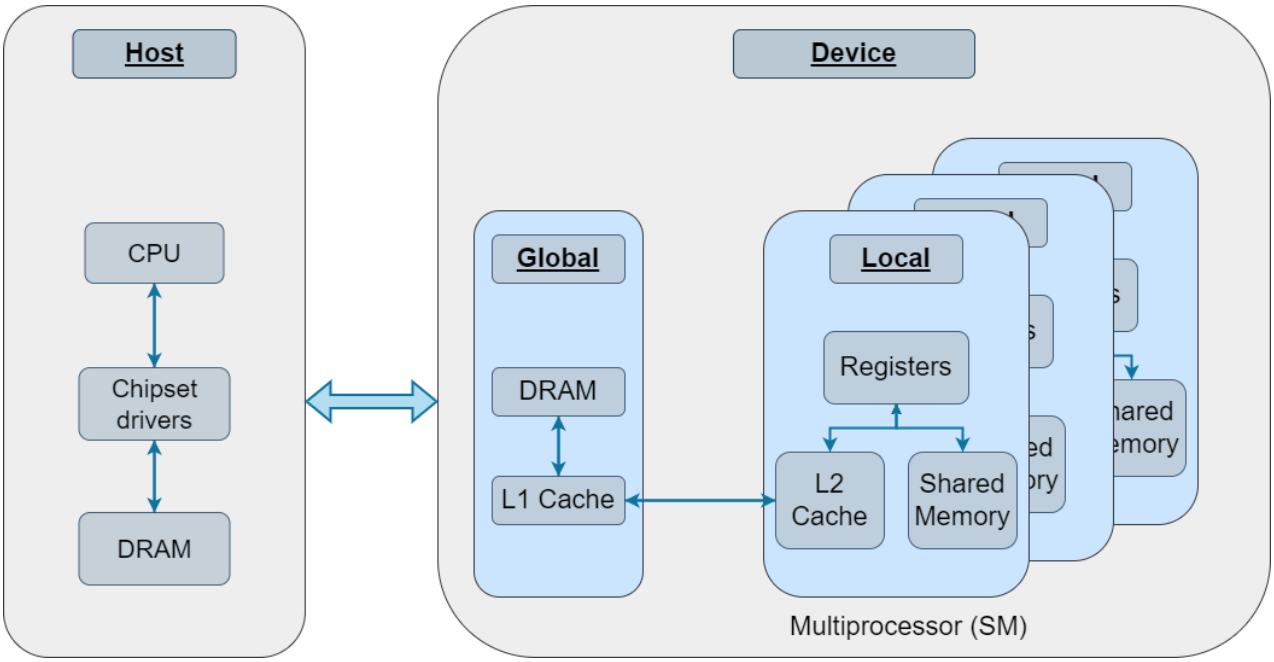

Figure 2.1: Memory architecture of the CPU (host) and the GPU (device)

The data transferred from the host to the device is stored in global DRAM memory, available for any thread to access. Recently accessed data requested from an SM, on the way to the registers, passes through two fast access caches [4]. Firstly, the global L1 cache, followed by the local L2 cache. The L1 cache is shared with the whole device, and the L2 cache is local to each SM. Their function is to reduce the number of times needed to perform a fetch from global DRAM, a process which is slow, and induces latency.
As the data reaches the registers, or the cores of the GPU, it can either be used for computations, or placed in shared memory. By placing the data in shared memory, we can ensure that the data is always accessible with minimal latency [5]. A deeper explanation of shared memory has been covered in section 2.2.4.

2.1.1 Caches

While not specific to either CUDA or GPUs, it is considered pertinent for the reader to understand something about how the caches work.

Memory requests do not occur on a byte to byte level. Instead, the total memory is divided into several *cache lines*. When requesting a specific value, the whole cache line is be pulled, and temporarily stored in some cached memory allocation [2]. This is done regardless of your initial request. While this seem like an odd feature, it's in fact, in part, created for the purpose arrays. We know that the raw definition of an array is a contiguous block of memory. Since many array requests usually occur within a short timeframe and a smaller memory subset, it is very likely that the data of a later memory request has already been pulled and stored at an earlier stage.

The size of a cache line varies, but most architectures uses a 64 byte cache line [2]. The size of the cache memory allocation can vary between a few kB to hundreds, or thousands of kB depending on what the cache is for. Important to note is that the caches are system managed, which means that the user has no control of what is stored in the cache at any given time.

2.2 CUDA

Implementing code on the GPU needs to be done inside special functions, which are typically called *kernels*. In order for the compiler to be able to differentiate between regular functions and GPU kernels, extra specifiers are needed. In C++ an additional keyword is added, and in Python a special decorator is used, see the following code snippet.

```
__global__ void A_C++_kernel (int *a_param)
{

        // Kernal Body
}
```

```
@cuda.jit
def A_Python_Kernel (a_param):

        # Kernel Body
```

We can, based on the C++ code, see that the kernels are *void*-typed, i.e. they return nothing. This is a characteristic every kernel must obey. The typical workflow in functions of void type is to modify the result directly in the memory. This means that an array with memory allocated for the result needs to be passed in as an argument and assigned in the function.

A typical C++ kernel also passes the parameters to the function by either reference or by pointer. This means that the actual array is not passed, but the memory location of the first element in the array. The process is handled implicitly by the CUDA extension to Python, since Python has no memory management for the user.

Calling the kernel from the main part of the program will also appear differently than regular functions. When calling the kernel, the *parallel discretization* parameters need to be specified so that the kernel knows which resources are needed. Parallel discretization will be covered in section 2.2.1.

```
A_C++_Kernel <<< BlocksInGrid, ThreadsPerBlock >>> (...);
```

```
A_Python_Kernel [BlocksInGrid, ThreadsPerBlock] (...)
```

2.2.1 Parallel Discretization

In general, running code on a CPU is done *serially*. This means that a certain process is performed in sequence with respect to the data. In a serial discretization of a CFD problem,

we can simply divide the domain into cells, and perform calculations for each cell in sequence.

For *parallel* computing, there are two steps in the discretization process. As can be seen in figure 2.2 the domain is divided into a certain number of *blocks*. The blocks contain *threads*, which perform the calculations, similarly to the serial case. To understand why there are two layers in a parallel discretization process, we can think of a block as a worker. Each worker controls the area of the domain it's physically occupying, which is done through its threads, or imagine, multiple arms. The arms perform the same tasks synchronously, but separately, which means that once a value has been computed for a certain cell, the value is unknown to every other cell in the domain. Luckily, and the reason for the "worker" metaphor, a worker has the possibility to track what its arms are computing. But this is limited to each worker, i.e. blocks are able to communicate internally, but not intercommunicate with each other.

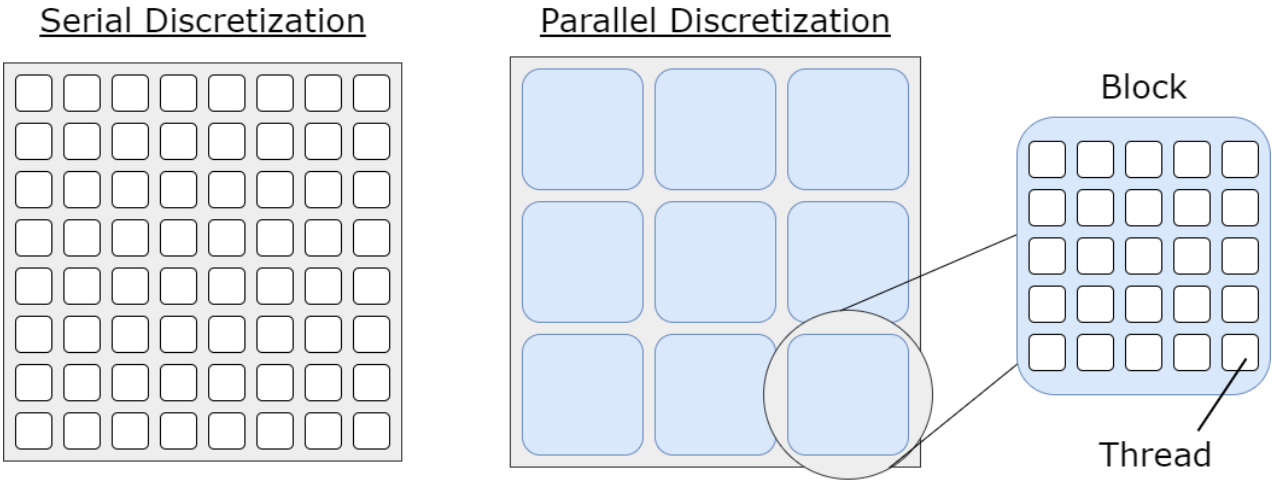

Figure 2.2: A serial discretization, compared to a parallel discretization

Observant readers might note a big problem the lack of intercommunication causes, as illustrated in figure 2.3. The intercommunication problem is especially prominent in a CFD program, where the process of performing computations for one cell often involve the neighboring cells. For cells which are on block boundaries, this causes issues since they cannot access all of their neighbors.

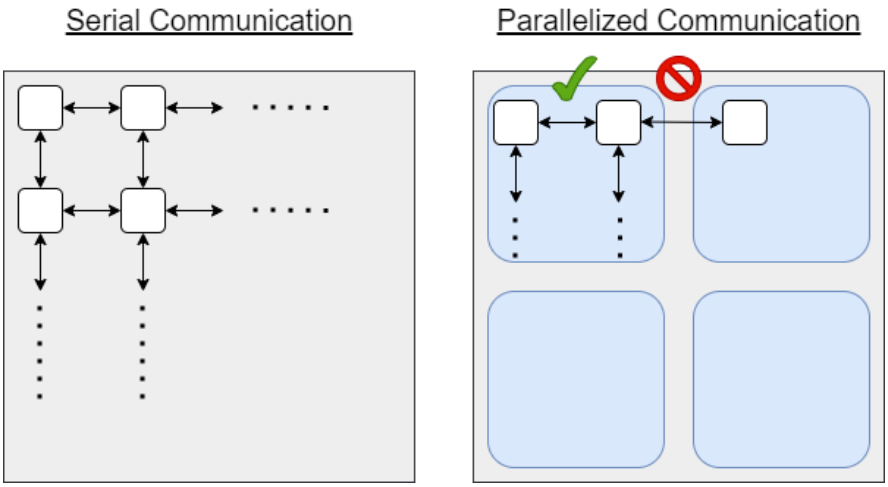

Figure 2.3: In the parallel kernels, the only valid communication between threads is within a thread block.

One way to resolve this problem in this project was to use a resource called *shared memory*, which can uniquely store a certain amount of data inside each block. More on how this was implemented in Section 2.2.4.

2.2.2 Indexing

The indexing for a parallel discretization is two-layered. Firstly, there is an index which identifies each block and inside a block the threads are indexed locally by their position in the block. Thread indices are thereby not unique, but the combination of thread index and block index results in a unique location in the total domain.

The thread and block indices are hidden *struct*-objects which can uniquely be accessed by every thread [6]. Getting the global location of each thread require some index mapping

$$\text{Global Index} = \text{Thread Index} + \text{Block Index} * \text{Block Size}$$

If the domain is two-dimensional as in this case, the struct-objects; thread index, block index and block size have an x and a y component. In that case, there will be two global indices, which are computed using the x or y components respectively.

This operation is so fundamental to the workflow in CUDA that Numba has added a supporting method called *grid* which will perform this operation behind the scenes. As an input argument, you specify how many dimensions your domain is defined by, see the following code snippet.

```cpp
__global__ void A_C++_Kernel (...) {

        // Thread Indices
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        // Block Indices
        int bx = blockIdx.x;
        int by = blockIdx.y;

        // Global indices
        int i = tx + bx * blockDim.x;
        int j = ty + by * blockDim.y;
}
```

```python
@cuda.jit
def A_Python_Kernel (...):

        i,j = cuda.grid(2)
```

Numba also supports a very similar syntax to that of C++, in the cases where the user does not want to use the grid method.

2.2.3 Block Sizes and Grid Sizes

A logical question at this point might be how to be able to decide, or determine, the best size of the blocks. As it turns out though, determining the optimal block size is not a straight forward process. It is both hardware, and problem dependent. The general guidelines are, a maximum of 1024 threads per block, i.e. 32x32, in two dimensions, and a block size evenly divisible by 32. The reason for the last guideline will be covered in Section 2.2.8.

It is the block size we care most about for optimization. But as previously mentioned, we would like to launch one thread per cell, which means we need to create the correct number of blocks to cover the domain completely. The set of all blocks in the domain is generally called a *grid*. The grid size is obtained by dividing the total domain size by the block size in each coordinate direction. Since this needs to be an integer, we can not round down and thereby risk missing cells. We therefore ceil the divisions.

$$\text{Grid Size} = \text{ceil}\left(\text{Domain Size} / \text{Block Size}\right)$$

However, this might instead lead to the kernel launching threads outside the domain, i.e. in unspecified memory. In order to make sure none of those threads are accessed, we need to add a conditional at the beginning of the kernel, see code snippet below.

```
@cuda.jit
def A_Python_Kernel ( ... ni, nj, ... ):

        i, j = cuda.grid(2)

        if i < ni and j < nj:

                # Perform kernel code
```

2.2.4 Shared Memory

The shared memory is a user managed memory allocation, unlike the caches, which are system managed allocations based on recently accessed data. This means that the user has full control of what and how data is stored [5].

In many general parallelization cases, the shared memory has the same size as the block and is used for memory optimization purposes. Generally, the optimization revolves around avoiding latency from global memory calls, if some values are to be used multiple times in a kernel.

But the size of the shared memory allocation is up to the user, which means that to solve the intercommunication problem presented previously, we can expand the shared memory so that it covers an area larger than the block. The expanded area will cover cells in neighboring blocks, which means that every thread in the current block will have access to all of their neighbors.

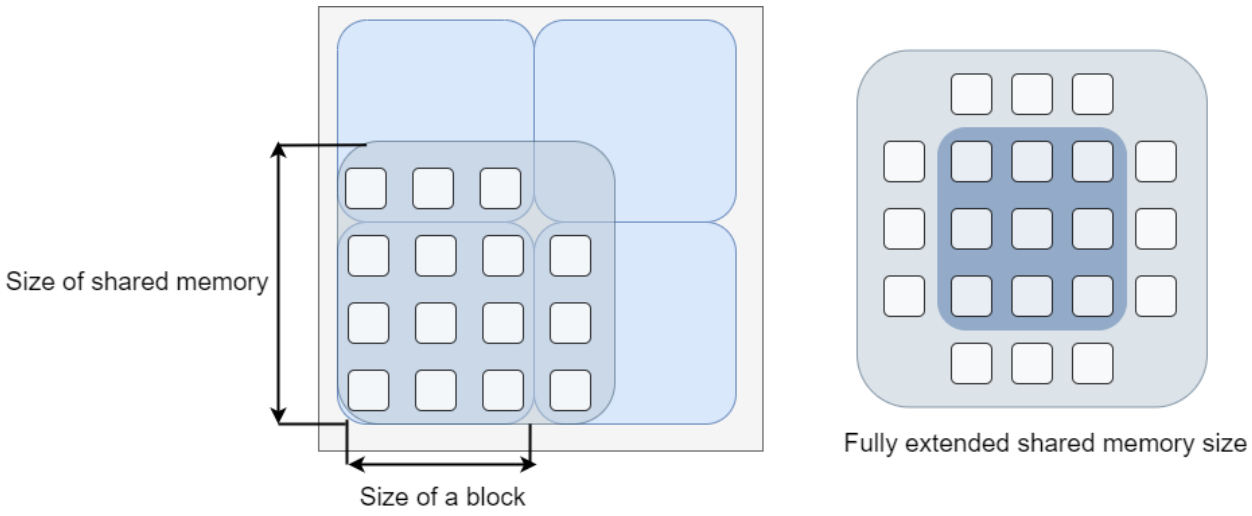

Figure 2.4: Sizes of the shared memory compared to the block size

In figure 2.4 to the left, we can see the area expansion of the shared memory for a block in the bottom left corner. For a general block, in the middle of the domain, the expansion region would look like the image to the right.

The process of actually computing the extra values needs to be managed. Recall that each block has a fixed number of threads, and each thread handle the calculations of one cell. Now that we have included additional cells for the block, extra data needs to be requested from memory. The job is assigned to the boundary cells, and implemented in code through simple conditional statements. Boundary cells will, in addition to their own work, also perform the same work for their neighboring cell outside the block. The Python syntax for this, looking at the left image in figure 2.4, see the following code snippet.

```
BlockSize = 32

# Create Globally defined shared memory size. In this case
# we just add one cell to the right and up. See figure 2.4 (left)
ShMemSize = BlockSize + 1

@cuda.jit
def A_Python_Kernel (Input, ni, nj, ...):

        i, j  = cuda.grid(2)

        if i < ni and j < nj:

                tx = cuda.threadIdx.x
                ty = cuda.threadIdx.y

                ShArr = cuda.shared.array(shape=(ShMemSize, ShMemSize), dtype=numpy.float32)

                # Every thread in the block loads in its corresponding global value
                ShArr[tx, ty] = Input[i,j]

                # Let block-boundary threads load in extra values

                # Right boundary
                if tx == cuda.BlockDim.x - 1:
                        ShArr[tx + 1, ty] = Input[i + 1, j]

                # Upper boundary
                if ty == cuda.BlockDim.y - 1:
                        ShArr[tx, ty + 1] = Input[i, j + 1]


                cuda.syncthreads()
```

Important to note here is that the shared memory arrays use the thread indices, tx and ty, in the block. This should make sense since it was previously mentioned that thread indices are local to each block, i.e. in every block, the first thread has location (0, 0), and the last thread has index (Block Size x - 1, Block Size y - 1). This means that the thread indices maps very nicely to the shared arrays.

2.2.5 Index Mapping for Shared Arrays

In some cases, when shared arrays are used in an extended form, we need to perform some simple index mapping.

But firstly, by observing the left image in figure 2.4, we can note that index mapping is not required in this case. If we imagine index (0,0) being the bottom left thread, then every index remains the same, even if we add threads to the sides.

On the other hand, the right image in figure 2.4 shows a case in which we do need to perform index mapping. We see that index (0,0) of the shared memory is offset from index (0,0) in the block. In fact in shared memory space, thread index (0,0) is index (1,1). This is the clue to the index mapping which needs to be done. The *shared index* is thread index + 1, and this is the indices which needs to be used when working with this type of shared array.

2.2.6 Boundary Control

Observant readers might have caught a dangerous coding pattern in the latest code snippet. When accessing the right and upper values for the boundary threads, we used no bounds checking. Since raw arrays in C++, which this code is compiled into, has no internal bounds checking, we will access unspecified memory for cells on the domain boundaries. Therefore, we need to create our own bounds check, to make sure that we stop at the domain limits. This is done through simple min and max functions.

Another form of boundary control, which is of great importance when working with shared memory, is block overflow at domain boundaries. If the total domain size and the block size are not evenly divisible, there will be some overflow at the edges of the domain. A schematic illustration of this issue can be seen in figure 2.5.

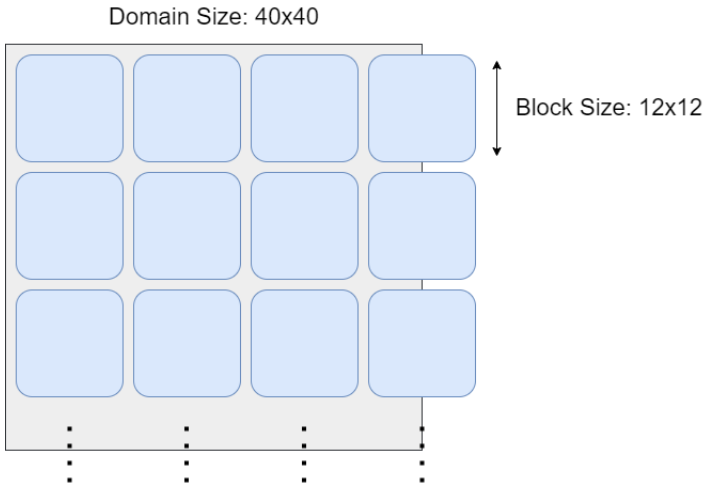

Figure 2.5: Block overflow at domain boundaries

Therefore, when using shared memory implemented the way this project does, extra care needs to be taken into consideration at the right domain boundary in figure 2.5. Normally it's the thread index which correspond to the size of the block that determines if the thread is positioned on the boundary of the block. Now we end up in a position where the domain ends in the middle of the block, and we can not use the block width to determine if a thread

is located on the boundary.

Solving this was done by adding an *or*- conditional to check if the thread is either at a block boundary, or at the domain boundary.

2.2.7 Synchronization

An important concept when working with shared memory is synchronization, or more particularly, synchronization of threads within a block. If we imagine several threads running through code in parallel, and in this process writing to shared memory, just like the previous code snippet, not every thread might be in synch at a given location in the code. This is a problem if a thread accesses shared memory for a cell for which another thread has not yet finished its calculations. The synchronization acts as a barrier, stopping the threads until every thread in that block has reached that point.

```
cuda.synchthreads()
```

The syntax cuda.synchthreads() is therefore placed directly after the shared memory array is filled up correctly, which means that it's safe for the code to continue.

The reason for threads in a block being out of synch could for example be conditional forks in the code, but is mostly caused by *warps*, see section 2.2.8.

As hinted before, the synchronization is only valid inside the threads blocks. There is no direct way to establish a global synchronization syntactically. However, different kernel calls will be called sequentially, which means that practically, global synchronization can be reached by splitting up code into two or more kernel calls.

2.2.8 Warps

Warps are a low level implementation feature in CUDA whereby a finite number of threads are launched in sequence. Although previously mentioned that the entire process takes place in parallel, this is not precisely true in the implementation. The execution of a block is structured into several warps, with a fixed *warp size*. For the CUDA framework, this size is 32, i.e. 32 threads are launched in sequence [7].

The graphics card hardware is able to switch between different warps with zero overhead, as a warp is currently inactive. Inactivity can be caused by latency in global memory calls, or through synchronization barriers.

The CUDA warp size is important to keep in mind in order to have the program run as efficiently as possible. Since the warp size is constant, it will always launch 32 threads at the same time. But if the block size is not evenly divisible by 32, we will at the last scheduler

have < 32 threads left in our domain, and therefore launch too many and cause inefficiency in the program.

2.2.9 Occupancy

For maximal performance of the GPU we want to maximize the *occupancy*. The occupancy of the GPU is determined by how many of all the SMs are put to work, i.e. how parallelized the code becomes in the execution stage. [2]. The code could for example run on one SM which would cause low occupancy, or it could utilize every SM, which should significantly enhance the performance.

The way the SM utilization is determined is through the blocks and their sizes. If the parallel discretization consists of very few blocks, then the code will only load a few of the SMs. Introducing many blocks of significant sizes will allow the GPU to maximize the amount of parallelization.

2.3 CUDA using Numba

As previously mentioned, CUDA is implemented in C++, but there exists multiple modules or libraries in other programming languages which allows the programmer to write functions to the GPU in that language. One example is the Numba module for Python.

Numba is an open-source compiler used to convert Python code into C++ code. Due to the conversion between the two languages, Python-specific data structures such as dictionaries or lists etc will not be able to be mapped over. Numba does support NumPy, which means that every array passed in to a kernel should be a NumPy array which has been transferred to the device.

One issue which was found was the use of characters in kernels compiled in Numba, or rather, character arrays passed into a kernel as a parameter. If the characters passed are used to check for equality against another given character, as can be seen in the following code snippet, the program will fail.

```
@cuda.jit
def A_Python_Kernel( A_Character, A_Dict, ... ):

        i, j = cuda.grid(2)

        # Will generate Errors
        Some_Variable = A_Dict[Key][i,j]

        # Will also generate Errors
        if A_Character == 'd':
                # Do something
```

Numba seem to auto-convert the character into a device-array, which makes it impossible to use for the comparison check against a pure character. The issue was handled by replacing the character arrays (which were used to define boundary types, for example) with numbers instead. Being extra careful in commenting this discrepancy in the source code.

2.4 Poisson CFD Solver

In order to implement the above outlined practices a code, previously written for serial compute on the CPU was rewritten to run in parallel on the GPU. Since this case was deemed to be of somewhat complex nature, another code was written from scratch solving the Poisson equation and was from the beginning targeted at running on the GPU. While the two codes did end up rather similar in structure, they still illustrate quite well how to, on one hand, implement GPU acceleration in a more rudimentary and basic form. On the other hand, it shows how a more complex code significantly increases the difficulty of implementing GPU acceleration.

2.5 Kernel Setup for Shared Memory usage

The following code snippet shows a complete kernel introduction whilst working with shared memory the way needed in this project. It includes all the important considerations, and has been validated to produce the exact same results as the serial implementation.

```
BlockSize = 32

# Create Globally defined shared memory size. In this case
# we add one cell to every direction. See figure 2.4 (right)
ShMemSize = BlockSize + 2

@cuda.jit
def A_Python_Kernel (Input, ni, nj, ...):

        i, j  = cuda.grid(2)

        i_left = max(i-1, 0)
        i_right = min(i+1, ni-1)
        j_down = max(j-1, 0)
        j_up = min(j+1, nj-1)

        if i < ni and j < nj:

                # Thread indices
                tx = cuda.threadIdx.x
                ty = cuda.threadIdx.y

                # Shared indices
                six = tx + 1
                siy = ty + 1

                ShArr = cuda.shared.array(shape=(ShMemSize, ShMemSize), dtype=numpy.float32)

                # Every thread in the block loads in its corresponding global value
                ShArr[six, siy] = Input[i, j]

                # Let block-boundary threads load in extra values

                # Right boundary
                if tx == cuda.BlockDim.x - 1 or i == ni - 1:
                        ShArr[six + 1, siy] = Input[i_right, j]

                # Left boundary
                if tx == 0:
                        ShArr[six - 1, siy] = Input[i_left, j]

                # Upper boundary
                if ty == cuda.BlockDim.y - 1 or j == nj - 1:
                        ShArr[six, siy + 1] = Input[i, j_up]

                # Lower boundary
                if ty == 0:
                        ShArr[six, siy - 1] = Input[i, j_down]

                cuda.syncthreads()

                # Main Kernel code goes here!
```

3
Result

The result presented in this chapter solely covers the parts of the script connected with CUDA. To make sure the CUDA conversion of the code was implemented correctly, the numerical results have been compared to the results on the CPU.

This project did not manage to successfully integrate a parallel linear solver in the CFD code, which meant that in order to solve the system, multiple data transfers back to the CPU were needed. Therefore, in order to create a fair comparison between the serial and the parallel implementation, the whole solution process was commented out for both implementations. The absolute times presented in this chapter are therefore not necessarily representative of the actual execution time. However, the main pattern of the time differences between the parallel and the serial implementations will still clearly be evident.

3.1 Performance Comparison between the CPU and GPU

Figure 3.1 shows the average iteration time of one iteration of the pyCalc RANS script, for both the parallel and serial implementation. The parallel implementation uses a block size of 32x32, and both implementations are run on a node on the Vera cluster which has an NVIDIA GP106GL graphics card of the Pascal architecture.

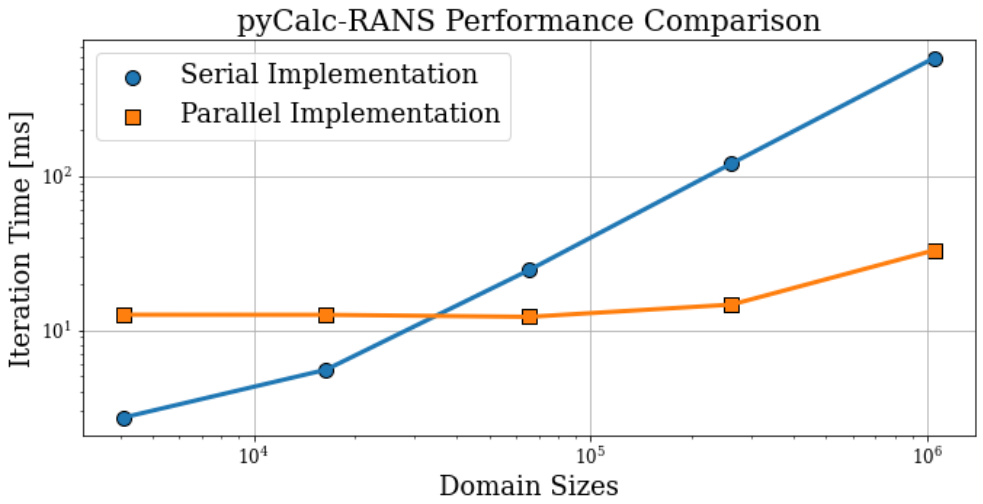

Figure 3.1: Speed comparison for the average iteration speed of the pyCalc RANS script using log scales.

To get a detailed view of the time difference between the parallel and serial implementation, table 3.1 shows, in milliseconds, the average iteration times.

Table 3.1: The measurement data from the speed comparison seen in figure 3.1. The parallel data is used with a block size of 32x32

	Serial	**Parallel**
64x64	2.7 [ms]	12.6 [ms]
128x128	5.53 [ms]	12.58 [ms]
256x256	24.55 [ms]	12.2 [ms]
512x512	121.5 [ms]	14.66 [ms]
1024x1024	591 [ms]	33.1 [ms]

As can be seen from table 3.1 for smaller sizes of the domain, the serial implementation is faster. This has to do with the overhead time for the kernels to launch. But after launching, the total times remain constant until the 1024x1024 mesh size. The serial implementation on the other hand sees significant increases, of a factor 5, in iteration time for every increase in domain size. For the largest domain size we see a performance increase of a factor 18 by executing the code on the GPU.

Similar trends were seen in the Poisson's CFD solver that was written from scratch. Difficulties with convergence with both the CPU and GPU implementation made it difficult to time the iterations seen in table 3.2. Further, inherent limitations in the way the reference CPU code was written limited the comparisons to only use small domains which give the CPU code a large advantage.

Table 3.2: Average iteration time comparing 1000 iterations for Poisson's CFD code running on CPU and GPU. A block size of 8x8 was used on the GPU

	Serial	**Parallel**
64x64	0.37 [ms]	11.84 [ms]
128x128	1.92 [ms]	11.82 [ms]
256x256	8.89 [ms]	11.83 [ms]
512x512	64.45 [ms]	16.27 [ms]

3.2 The Impact of Complexity

Since one of the major drawbacks of GPU targeted code is the required transfer of variables from the host to the device, one might think that an increase in the number of matrices, and

their size, would be an ever-growing bottleneck and a significant limitation to the problem size. While this might be true for significantly larger scale computations, we did not observe this in a prominent way in either of our codes. Interestingly, it appears as though the first copying of any variable to the device from the host requires a sort of synchronization or "handshake" between the CPU and GPU, resulting in noticeably longer transfer times for the first transfer. Subsequent transfers do not appear to suffer this delay, and are instead directly proportionate to the memory footprint of the transferred variable.

Note that there are significant differences in this behavior depending on the hardware that is used, with the initial transfer times being vastly different.

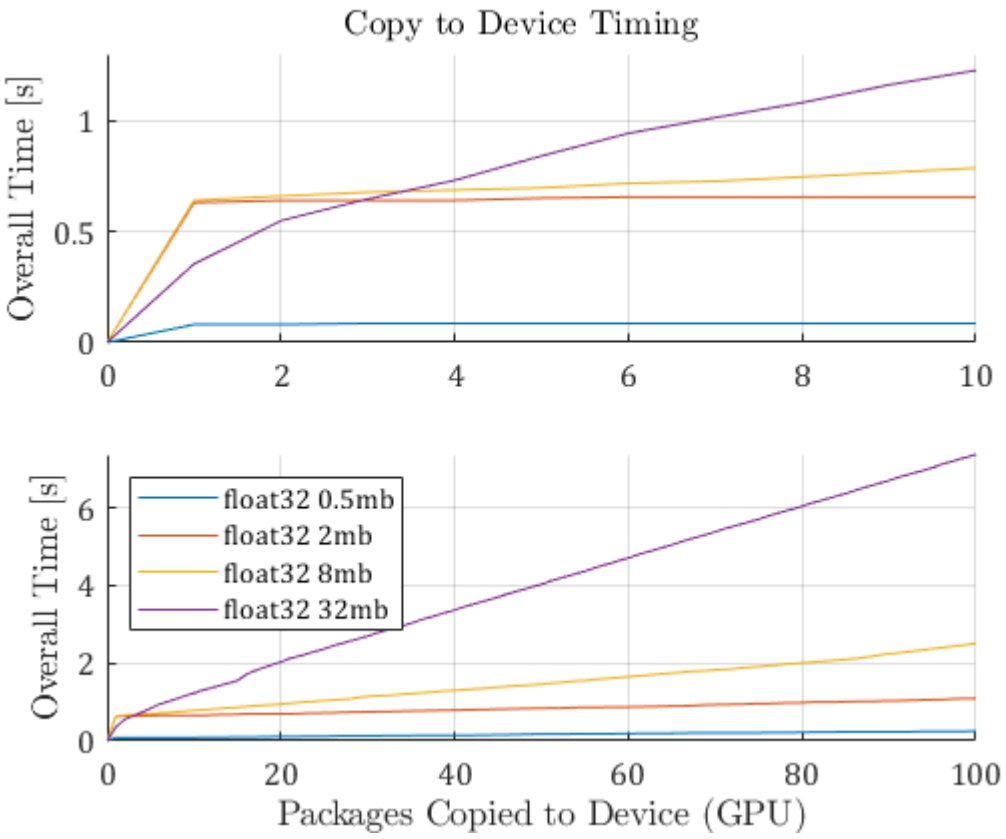

Figure 3.2: Cumulative time for the copying of float32 arrays containing random values in sizes of 1024x1024 (0.5MB), 2048x2048 (2MB), 4096x4096 (8MB) and 8196x8196 (32MB) to a GTX1060 (Pascal Architecture) GPU.

In figure 3.2 we see the cumulative time for copying large matrices to the device. The matrix sizes vary from 1024x1024 (0.5MB) to 8196x8196 (32MB). We note that for matrices of sizes up to 8 MB has a near instant copying time after an initial long waiting period. Zooming out, we can see that the graphs up to 8 MB grow slightly more linear as the amount of matrices copied over increases. The smaller slowdowns seen with regular intervals is suspected to be garbage collection and caching taking place. Note here that the 32 MB case is an outlier that behaves notably different to the other cases.

An argument can thus be made for repeated data transfers, while still unwanted, not being as bad as one might suspect from timing that initial transfer.

Another aspect to consider here is that the memory on the GPU is limited and as such the total memory footprint of all variables used in the calculations has to be considered. While a single matrix will not saturate the memory of a modern GPU, using unnecessarily large data types, such as float64 instead of float32 and copying unnecessary data to the device, might.

3.3 Performance Comparison between block sizes

As mentioned in Section 2.2.9, the parallel discretization is very important for maximal performance, mostly through maximizing occupancy. The following table shows a minor investigation in how the block sizes affect the average iteration times. The investigation was carried out using the 1024x1024 domain size, and averaged over four different runs. The investigation was also performed using three different GPUs; the GP106GL on Vera, an RTX 2070 and also on a GTX 1060.

Table 3.3: Investigation between the block size and iteration time for the pyCalc RANS code. Note that the domain size was set to 1024x1024 for all tests. The investigation was performed for three different GPUs.

	GP 106 GL (Pascal)	**GTX 1060 (Pascal)**	**RTX 2070 (Turing)**
4x4	32.32 [ms]	25.5 [ms]	15.0 [ms]
8x8	23.05 [ms]	19.0 [ms]	12.6 [ms]
16x16	27.15 [ms]	21.9 [ms]	12.9 [ms]
32x32	33.1 [ms]	26.44 [ms]	15.2 [ms]

From table 3.3 we firstly note that the three GPUs have vastly different performance, where the RTX 2070 is around two times faster than GP106GL. The GTX1060 is more similar to the GP106GL, which makes sense given the same card architecture. However, more interestingly, we note a similar pattern in the iteration speed as a function of block size. The 8x8 and 16x16 perform significantly better than the other two block sizes for every GPU. Both the GP106GL on Vera and GTX1060 see around 25-30% performance increase by reducing the block size to 8x8 from 32x32. If this result is compared to the performance increase received in Section 3.1, we will by using the 8x8 block size obtain a performance increase of a factor close to 26 instead.

This result seem to align well with the theory presented about occupancy, where a larger number of blocks launched allows the GPU to parallelize the code to a larger degree, and hence improves the performance.

3.4 Performance Comparison between C++ and Python

In order to investigate how much performance increase that could be expected from switching language to C++, a single function was implemented in three different ways. Firstly, it was implemented with a vectorized but serial approach in Python, in order to set a benchmark. Then it was implemented as a kernel in both Python and C++. Before the testing started, validation made sure that all versions produced the same result to a given initialization.

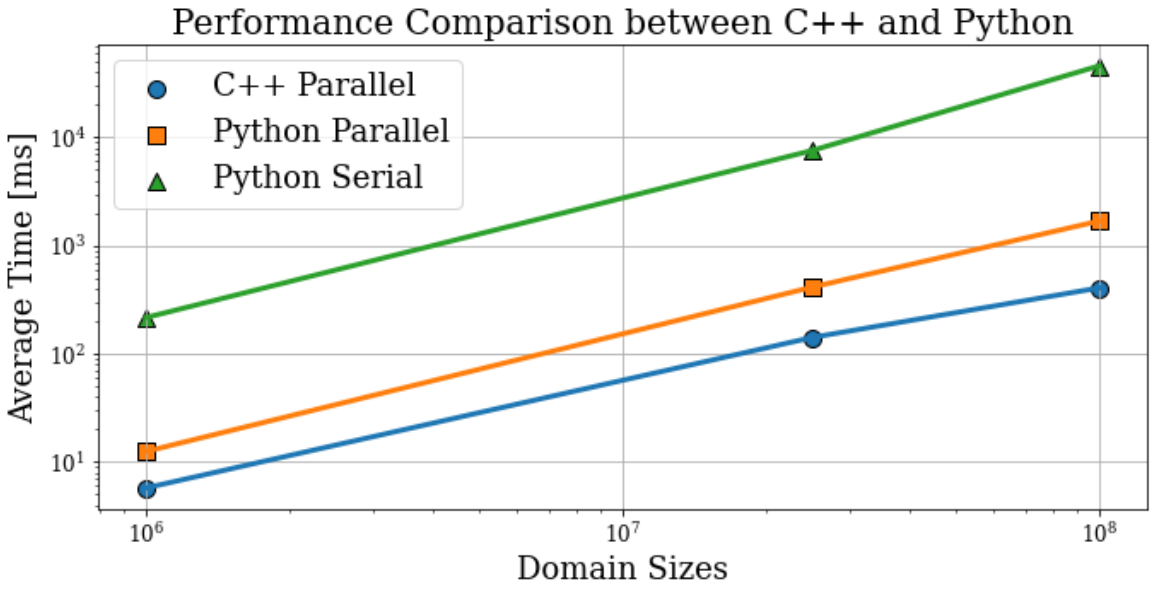

Figure 3.3: Speed comparison between parallelized C++ code, parallelized Python code and serial Python code for a single kernel plotted using log scales

Table 3.4: The measurement data from the speed comparison seen in figure 3.3

	Python Serial	**Python Parallel**	**C++ Parallel**
1000x1000	215.93 [ms]	12.5 [ms]	5.75 [ms]
5000x5000	7564.4 [ms]	412.87 [ms]	141.5 [ms]
10000x10000	45 797.1 [ms]	1686.15 [ms]	406.5 [ms]

From figure 3.3 and table 3.4 we see that both parallel implementations are vastly outperforming the serial implementation. Important to note is that the axis in figure 3.3 are log-scaled, meaning that there is not a parallel, but an exponential time relation between the three implementations.

The comparison between the two parallel approaches shows that the C++ kernel starts off as running about twice as fast, and then speeds up comparatively to three times as fast, and for the largest domain size, four times as fast.

4
Discussion

4.1 When CUDA is worth implementing

Rewriting, or targeting, your code for GPU computations is by no means a universal pathway to better overall times. Factoring in the time to re-target the code from a CPU implementation, this work might not be worth it for many applications. The benefit of using Numba, and thereby Python however, is that much of the complexity is handled for you. The simplicity offered by Python and Numba generally outweigh its drawbacks for small to medium size applications. It also makes it a good candidate for learning the concepts of the architecture and prototyping ideas. This can later be applied in a similar manner in more advanced, but also more flexible and powerful languages such as C++. One of the best ways to improve the performance of a kernel is to change the language, which is why C++ still remains the go-to option for commercial and finalized codes aimed at running on the GPU.

Another aspect in determining whether a parallel implementation will be worth the time and effort is the debugging difficulties associated with writing kernels. While it is easy to implement the general workflow, a CUDA-kernel is notoriously difficult to debug intermediate computations. Serial code allows the use of break points and slowly stepping over lines to check for unexpected events. A kernel does not support this sort of debugging. Instead, the programmer will have to find other creative ways to access intermediate results computed inside a kernel. One effective way, which was found in this project, was to write intermediate results to one of the input arguments of the kernel, and terminate the kernel early.

4.2 Implementation Issues

While re-writing the pyCalc-RANS into parallelized code, several costly and non evident bugs were introduced by accident. The following section will discuss some of these problems, why we believe they occurred and how they were fixed.

The first issue involves only passing a part of a whole array into a kernel. The coefficients a_E, a_W, a_N, a_S and a_P were collected into a common three-dimensional array where the third dimension separated the coefficients. For some kernels only a few of the coefficients were needed, in those cases they were passed in individually, by accessing the third dimension. This process made the script very slow, and increasingly slow for large domains. We believe the reason for this is that accessing the variable like that caused Numba to be unable to create a C++ code where the argument was passed in by reference (or by pointer). Instead, the arrays were

forced to be passed by value i.e. copied, which explains why this process became slower and slower with increased domain size. To solve the issue we passed the whole array, and accessed the desired coefficient inside the kernel. Passing in large arrays into a kernel does not result in a performance hit, since they are passed in by reference, i.e. a memory location.

A second long term issue was to define the shared memory size in a way which was adaptive to the block size the user specified. For improved structure and readability, the code was initially split into several files, where each kernel got its own file. But by doing this, the shared memory sizes were forced to be hard coded. The reason for this issue is that in C++ the shared memory size must either explicitly be a constant or a macro. In Python, the concept of constants does not exist, but variables created globally in the scope of a certain file will be treated as constants by Numba. This means that to define the size of the shared memory in an adaptive way, every kernel needed to be in a file where the block size was created in the global scope. The only way to achieve this was by adding all the kernels into the main Python file.

5
Conclusion

From the result, we can draw a few documented conclusions. Firstly, the cross-over point between the serial and parallel implementations occur for relatively small domain sizes. In the pyCalc-RANS code this cross over occurred at a domain size of around 256x256. For domain sizes larger than that, the parallel implementation quickly grows to be exponentially faster than the serial. Even by factoring in the extra overhead time of copying variables to the device, the decreased iteration time will make the parallel approach inherently more effective for a relatively moderate number of iterations. The performance increase for the largest domain size of 1024x1024 was measured to be a factor of 18.

We have also seen that for this particular case, a block size of 8x8 seem to be the optimum size for maximum performance universally over a number of GPUs. This particular block size improved the performance by up to 30% compared to a block size of 32x32. While smaller blocks mean that there are more block boundaries where threads need to perform extra computations, the added occupancy in the GPU seem to be the largest determining factor for performance. By using this block size, a performance increase of a factor 26 was obtained.

One last conclusion to draw is that, while Numba will convert Python code into C++ code for the programmer, it will not manage to perform better than a CUDA implementation originally written in C++. Although the entry level bar for CUDA in C++ is higher, this should be a real consideration if performance is of paramount interest.

References

[1] Lars Davidson. *pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow*. Tech. rep. Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, 2021.

[2] Robert Crovella. *Fundamental CUDA optimization (Part 1)*. Online Lecture. Oak Ridge National Laboratory. Mar. 2020. URL: https://vimeo.com/398824746.

[3] NVIDIA. *GeForece RTX 3080-Series*. 2022. URL: https://www.nvidia.com/sv-se/geforce/graphics-cards/30-series/rtx-3080-3080ti/.

[4] Robert Crovella. *Fundamental CUDA optimization (Part 2)*. Online Lecture. Oak Ridge National Laboratory. Apr. 2020. URL: https://vimeo.com/414827487.

[5] Robert Crovella. *CUDA Shared Memory*. Online Lecture. Oak Ridge National Laboratory. Feb. 2020. URL: https://vimeo.com/393552516.

[6] Robert Crovella. *Introduction to CUDA C++*. Online Lecture. Oak Ridge National Laboratory. Jan. 2020. URL: https://vimeo.com/386244979.

[7] Robert Crovella. *Atomics, Reductions and Warp Shuffle*. Online Lecture. Oak Ridge National Laboratory. May 2020. URL: https://vimeo.com/419029739.

[8] Lorena Barba. *12 steps to Navier–Stokes*. 2013. URL: https://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/.

Contributions

David Andersson

As I lacked the knowledge required for getting into the installation issues with AMGX, I instead focused on re-writing the pyCalc-RANS code. I also learned to use Vera, as well as worked on implementing a kernel in C++. After I made the pyCalc working, with the same result as the serial implementation, I focused more on theory for CUDA, which was presented in "Theory and Methodology".

Robert Ranman

With prior knowledge in linux initial focus was to install the linear solver AMGX and the corresponding python implementation pyAMGX. Building the AMGX library was initially troublesome but we got it working after tampering with the build files. Further the installation of pyAMGX was unsuccessful and either of us was able to get it working, thus AMGX was excluded from the project.

For CUDA programming i wrote a FVD Navier-Stokes solver and further tried to translate it to the GPU, but was unsuccessfully in my effort as i ran out of time to finish the solver.

Shisheer Shetty

As I did not initially have any experience with Linux, I was unable to install AMGX or pyAMGX as per the initial goal for the project. However, I did learn how to use Vera, and concentrated my efforts on learning more about Numba and CUDA in general. I also attempted to try and use the CPU version of the for the Poisson's equation solver for a lid driven cavity, to achieve consistent reliable results. However, I was rather unsuccessful at this.

Frowin Winkes

Initially the project was planned to use the linear solver AMGX and it's Python implementation pyAMGX to solve parts of pyRANS code when running on the GPU. However, both AMGX and pyAMGX have to be built locally in order for them to be used which turned out

to be more difficult than expected and while a build of both AMGX and pyAMGX finally succeeded in the end it was once more broken by and automated update in Ubuntu, at which point it was decided to move on with a pure CUDA implementation. During this process I learned a lot about building code libraries in Linux and managing their dependencies for both cMake and Pip.

As David was already working on the rewriting of the pyRANS code, I instead focused on writing a simple CFD code that solves Poisson's equation for a lid-driven cavity. A CPU code from [8] was used as a timing reference following a recommendation by our supervisor. This code turned out to be a limiting factor when analysing the gains in iteration speed since the scratch written GPU code turned out to be more capable in terms of domain size and more robust than the CPU reference so a significant amount of time was spent on reworking this code as well with only limited success. Changes to the code included amongst others implementation of under relaxation for u, v, and p since severe checker boarding was observed.

Finally I ran multiple speed comparisons for different GPUs for both the pyRANS code and Poisson's code as well as data transfer speed analysis for copying variables to the GPU and wrote about the results of these in the report.

Learning Process

This project has certainly reduced the entry level intimidation to start working towards the GPU for everyone in the group. As it turns out, Numba is very user-friendly for Python users, which makes it easy to transition. The general opinion of the group is that the knowledge created in this project will be of great benefit for the future, and perhaps a useful solution alternative for upcoming courses.

The following bullet list summarizes the key learnings obtained from this project.

- Basics and concepts of CUDA
- The CUDA API in Numba and C++
- Basic Linux commands
- Using vim programming interface in Linux
- Concepts of multi-process computing

From the group dynamic point of view we worked more in parallel than together. Since the project was coding based, which is only really effective doing alone, the work dynamics were very individual. The work product of every group member was shared with the rest, so that everyone could access any result.

While most of the intended work related to only CUDA was atleast looked at and worked on in some way, the goal of using pyAMGX and AMGX to solve the linear equation in pyRANS

was not achieved since the group was not able to get pyAMGX or AMGX working. While this was not defined as a risk in the planning report, one could in hindsight identify this as high risk but with low impact on the overall project since the remaining CUDA part could still be completed.