Technical University of Denmark

Special Course

Large-scale Computations on Modern GPUs

Authors:
Tim Felle Olsen (s134589)
Mathias Sorgenfri Lorenz (s134597)

June 22, 2018

Contents

1 Introduction

Large scale modelling is an important part of modern engineering, whether its for modelling stresses in airplane wings [Aage *et al.*, 2017], detection of underwater mines [Sullivan *et al.*, 2010] or simulating welding and laser forming problems [Sun and Michaleris, 2006]. Programming using parallel computations is what makes this possible. If we were unable to solve these huge problems in parallel then we would have to wait years to receive any useful results. Using distributed memory interfaces such as Message Parsing Interface (MPI) help achieve this by allowing computations to be run on how ever many threads we have access to and interchange the results from each thread through messages. A newer trend is to transfer the work to Graphics Processing Units or GPUs since these have very large numbers of compute cores that can complete tasks way faster than a CPU can.

Throughout this project we combine the two technologies by assigning several GPU to work together in a cluster environment using MPI as the method for synchronizing information between the different GPU. The problem we solve is the Poisson problem and we use the iterative Jacobi method to solve the problem.
The source code for this project is written in a combination of C and CUDA. In addition to this the library OpenMPI will be used to implement the distributed memory part of it. The code is available through Github:
https://github.com/MathiasLorenz/Large_scale_project. This report is based on the version 1.0.

Tim Felle Olsen (s134589) &
Mathias Sorgenfri Lorenz (s134597)

2 Mathematical theory

Our project builds on a couple of mathematical theories along with the computer science theory of the interfaces and languages we use. This section describes the basics behind the problem we use to test our implementation and the method we implement itself.

2.1 The Poisson Problem

The Poisson Problem is a partial differential equation (PDE) called the stationary heat equation, where the source term is non zero. It is defined as

$$\Delta u(\boldsymbol{x}) = -f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega,$$

where the domain $\Omega \subset \mathbb{R}^n$, Δ is the Laplacian operator, i.e. the sum of the second order partial derivatives, f is a source term and u is the heat distribution. The equation is called stationary as we have removed the dependency of time in the equation, i.e. we wait for the solution to settle into a state where it does not change. In order to ensure that this have a unique solution we must enforce boundary conditions (BC) onto the system. For Dirichlet BC the values on the boundary are known and for Neumann the derivatives on the boundary are specified. You can also have a mix of the mentioned conditions, which is called Robin conditions.

PDEs can be solved both analytically and numerically. One usually employs a technique called separation of variables to solve PDEs analytically. Here the equation is rewritten such that the solution u is represented as a product of the space $\boldsymbol{x}$ and time t. Using this form, one can solve each of the components individually and combine the results by multiplying them together in the end. This method has several drawbacks – one of the main being that acquiring a solution on some non-standard domain can be very tricky.

The other approach to solving PDEs is to use a numerical method. There are several different numerical methods for solving PDEs and we employ a finite difference method. Here the domain is first split into discrete grid points, where the solution is approximated. The solution is thus only obtained in these points, but for sufficiently dense grids, the solution can be approximated everywhere with numerical interpolation. The exact method is described in the following section. In order to ensure correctness of our numerical implementation, we need a problem where we know the analytic solution. With this we can both ensure correctness and observe the accuracy of our numerical scheme.

We solve a stationary Poisson Problem with homogeneous Dirichlet conditions defined as

$$\begin{aligned} \Delta u(\boldsymbol{x}) &= -f(\boldsymbol{x}), \\ u(\boldsymbol{x}) &= 0, \qquad \boldsymbol{x} \in \overline{\Omega}. \end{aligned} \tag{2.1}$$

In order to come up with a suitable test problem, we choose the solution to (2.1) as something 'nice', as we can easily compute the needed source term for the respective solution from the PDE itself. In 2D and 3D respectively we choose

$$u(\boldsymbol{x}) = \sin(\pi x)\sin(\pi y), \qquad \Omega \in \mathbb{R}^2, \tag{2.2}$$

$$u(\boldsymbol{x}) = \sin(\pi x)\sin(\pi y)\sin(\pi z), \qquad \Omega \in \mathbb{R}^3. \tag{2.3}$$

From (2.1) we then compute the source terms by applying the Laplacian operator to the solutions, which yields

$$f(\boldsymbol{x}) = 2\pi^2 \sin(\pi x)\sin(\pi y), \qquad \Omega \in \mathbb{R}^2,$$
$$f(\boldsymbol{x}) = 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z), \qquad \Omega \in \mathbb{R}^3.$$

We note that the domain Ω has to be $\Omega \equiv -1 \leq \{x, y, z\} \leq 1$ to satisfy the boundary conditions. Thus we have a box in 3D with all sides going from -1 to 1 and a similar square in 2D. This test problem is used for all methods throughout the report.

2.2 Finite Difference Methods for Solving PDEs

A finite difference method is perhaps the simplest numerical scheme for approximating the solution to PDEs. In general they do not expand gracefully to non-standard grids, but they are easy to implement and test, which is why we have chosen to employ them. We are going to explain and illustrate the method in 2D, but the method extends trivially into 3D. Finite difference methods require that the differential equation is satisfied at the discrete grid points. This is equivalent to solving the PDE in the so-called strong form (in contrast to the weak form that is solved in Galerkin type methods).

We use a centered finite difference stencil to approximate each of the second derivatives, here shown for some arbitrary 1D function v at $\bar{x}$,

$$\frac{\partial^2 v(\bar{x})}{\partial x^2} = \frac{v(\bar{x}-h) - 2v(\bar{x}) + v(\bar{x}+h)}{h^2},$$

where h is the grid spacing. We do this in each of the spatial dimensions (2- or 3D) and insert this into the differential equation (2.1). In 2D this yields

$$\frac{-4u(x,y) + u(x-h,y) + u(x+h,y) + u(x,y-h) + u(x,y+h)}{h^2} = -f(x,y).$$

This however introduces a significant restriction to our model which is that the distance between grid points h must be identical for all dimensions. If we are to include different distances for each of our dimensions we will end up with the following equation for the 2 dimensional case which can be extended to 3D by adding another term,

$$\begin{aligned} -f(x,y) =& \frac{u(x+h_x,y) + u(x-h_x,y) - 2u(x,y)}{h_x^2} \\ &+ \frac{u(x,y+h_y) + u(x,y-h_y) - 2u(x,y)}{h_y^2}. \end{aligned}$$

2.3 Jacobi Iterative Method

The finite differences can be used in a number of different ways when actually solving the PDE. We have decided to implement the Jacobi iterative method where we go through all interior points of our grid and compute the approximation of the equation at these points. We keep iterating over the array to refine the result. The approximation to the solution is computed by rearranging the terms in the finite difference method above. The notation

becomes very extensive by now so we introduce the notational abbreviation, $x_- = x - h_x$ and similarly for a positive grid increase and for the other dimensions. From this the Jacobi update is defined as the following equation in 2D,

$$u(x,y) = \frac{1}{4}\left(u(x_-,y) + u(x_+,y) + u(x,y_-) + u(x,y_+) + \frac{1}{2}(h_x^2 + h_y^2)f(x,y)\right).$$

For the 3 dimensional case this is readily extended with the fractions changed from $\frac{1}{2}$ and $\frac{1}{4}$ to $\frac{1}{3}$ and $\frac{1}{6}$, which becomes

$$\begin{aligned} u(x,y,z) =& \frac{1}{6}\Big(u(x_-,y,z) + u(x_+,y,z) + u(x,y_-,z) + u(x,y_+,z) \\ &+ u(x,y,z_-) + u(x,y,z_+) + \frac{1}{3}(h_x^2 + h_y^2 + h_z^2)f(x,y,z)\Big). \end{aligned}$$

This is the iteration that we have implemented in all our methods. This iteration is a regular finite difference method and the Jacobi method is a specific way of implementing this using 3 arrays. Two arrays are used for the current and previous iterations and one for the source term. Had we decided to use a single array to handle the solution and updated in-place the method would be the Gauss-Seidel iterative method.

3 Computer Science Theory

This section introduces a range of theories and technologies which are used in our implementations. The section focuses on parallel programming and introduces the libraries OpenMP, MPI and CUDA which are used in the implementations.

3.1 Parallel programming

Parallel programming is the process of doing several computations simultaneously on a computer. This way we can boost performance of a program simply by using more resources at the cost of additional constraints to how we write the program.

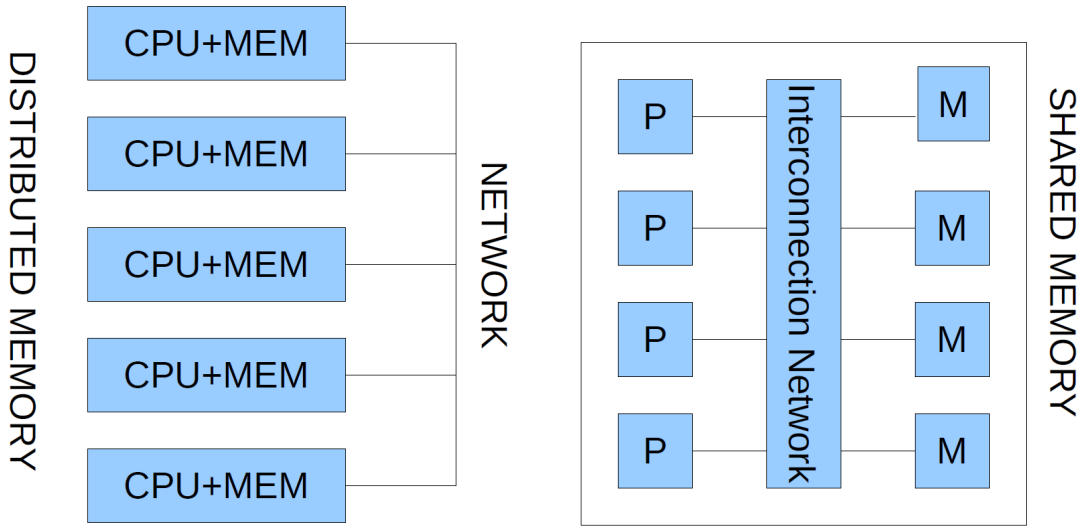
First of all the algorithm we implement must be parallel; that is we must have parts of the algorithm which can run independently from other parts of the algorithm. Not only does the algorithm need to contain parallel parts, but the proportion of parallelism must also be quite large in order justify spending additional resources to complete the computations. One of the first places to look is at loops in the code. Often loop iterations does not depend on other iterations.
This section focuses on methods and technologies used within parallel programming and how we intend to use it to solve the problem we defined earlier.

Memory Models – Shared and distributed memory

Parallel computing can be achieved with two different memory models; shared or distributed memory. In the shared memory model, the parallel pool has a single shared memory base that all processes can read and write to. No explicit memory management (send/receive) is required, because all processes can read the entire memory pool. The programmer must ensure that no race conditions are entered during execution, i.e. multiple processes writing to the same memory location, such that the output becomes dependent on the order of execution. OpenMP (Open Multi-Processing) is an API (application programming interface) for making shared memory multiprocessing in C, C++ and Fortran and is widely used in the industry. Programming in OpenMP is relatively easy and straight-forward. However, the shared memory model does not scale beyond the node, thus if you need more cores than your node has, you are out of luck with OpenMP and other shared memory models.

The distributed memory model on the other hand scales well beyond the node. The program is written in such a way that each process has its own memory space (hence distributed memory) and cannot read nor write to the memory space of other processes. Thus the programmer himself is responsible for exchanging information between processes, which can be e.g. sending messages between processes or broadcasting a message to all processes. There are many different libraries for doing shared memory parallel computing, but we have chosen to use the Message Passing Interface (MPI) protocol, which like OpenMP offers C, C++ and Fortran bindings. MPI offers send/receive operations between different processes as well as other collective communication techniques to ease message passing. In an MPI program each process (called rank) is spawned and all code in the

program is run by all processes. It is the responsibility of the programmer to send required messages between the processes and ensure all calculations are done correctly.



(a) Distributed memory system, also called a cluster.

(b) Shared memory system.

Figure 3.1: Comparison of the general structure of a shared and distributed memory system. Figure originally from Lazarov [2017]

Programming on the GPU – CUDA

A specialized way to do parallel computing is to use a Graphics Processing Unit (GPU) to do the computations for us. A GPU is a device designed and optimized to produce a vast number of outputs that are almost always embarrassingly parallel. The usual tasks are shading and rendering of graphics for use on a computer. But in recent years a move has been made towards what is often referred to as General Purpose GPU (GPGPU). The idea is utilizing these extremely powerful components to do other computations than just graphics. The reason being that a GPU is a unit consisting of thousands of processing units compared to the CPU which usually never contain more than 32 cores. The GPU is good at very parallel and fairly simple computations which makes them ideal for the problem we are trying to solve.
In order to communicate with the GPU for scientific computation the language CUDA was created. It is a modification of C++ and contains a number of library calls which allow us to give instructions to the GPU.

Running code on a GPU needs a bit of care when it comes to running loops. Normally in C, C++ and so on, a loop is run sequentially by invoking the `for` command or similar. On a GPU we skip the sequential loops and launches a thread for each step in the loop. These threads are organized into blocks which is organized into grids. Blocks and grids are used for grouping the computations such that each block is computed and then we move the computing group on to the next block. The GPU will then allocate resources to compute each part in parallel.
Computational units on the GPU handle several threads at the same time organized in

warps, these warps usually consists of 32 threads which all do computations at the same time. So if a block consists of 30 threads then 2 threads will always be idle since they are locked to the other 30 threads of the warp. Figure 3.2 shows this structure through a simple addition of two arrays.

Functions written to run on the GPU in the CUDA C++ language is called kernels and they are quite different from regular functions. They are always launched in a non-blocking way meaning the CPU part of the code will just keep going once the kernel has been launched. Therefore its important to maintain synchronization of the GPU and CPU manually in order to avoid errors, which makes usage of the function `cudaDeviceSynchronize` quite important. Additionally kernels are launched in several instances defined through call syntax `<<<Threads,Blocks>>>cudaKernel`, where `Threads` define how many threads should be part of each block and `Blocks` define how many blocks should be part of the grid for the kernel. These two variables can be multi-dimensional through the type `dim3` so that 3 dimensional problems are easier to organize. An example of this organization is included in subsection 5.2.

Even though kernels are launched asynchronously they are still not executed in parallel. Of course the kernel itself is computed in a parallel fashion, but a second kernel will wait the completion of the first before executing. These queues are called streams and we can have several streams running at the same time. Therefore if we want to have several kernels running at the same time we must assign them to different streams. `cudaDeviceSynchronize` will do a cross device synchronization but it is also possible to wait for just a single stream to complete its execution if other CPU based instructions require the data or result from a specific kernel. This waiting function is called `cudaStreamSynchronize`.

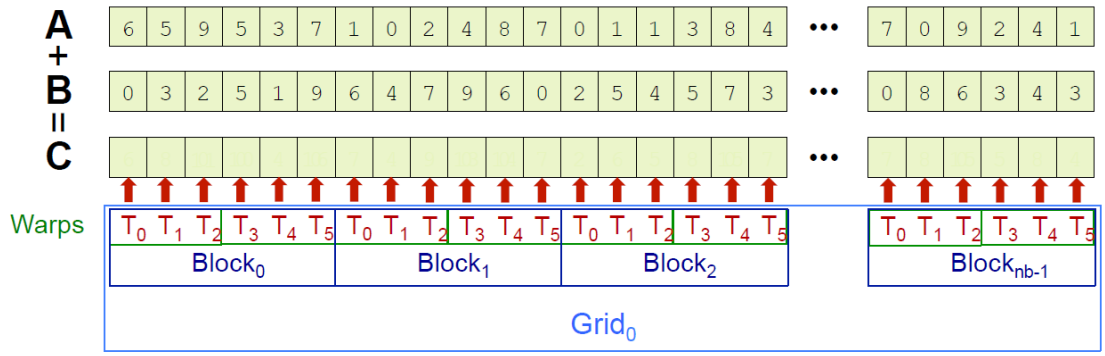


Figure 3.2: Organizational structure of the compute units in a GPU.

3.2 Domain Decomposition

In order to use MPI to solve our problem we must split our domain into parts and assign these to different computational units. Each point in the domain needs the information stored in the nearest neighbours in each direction. This mean we must exchange the information needed for the grid points close to our boundaries to ensure the algorithm runs correctly. Figure 3.3 shows this in a more visual fashion. The two grey lines are

representing the so called "ghost lines". These lines are a duplicate of the neighbour points on the other MPI thread which is copied over such that their values can be accessed when needed. Transferring data will be a large part of the execution time for the implementation. Thus hiding these transfers behind computations of other parts of the domain will be an important way to improve the performance of the implementation.

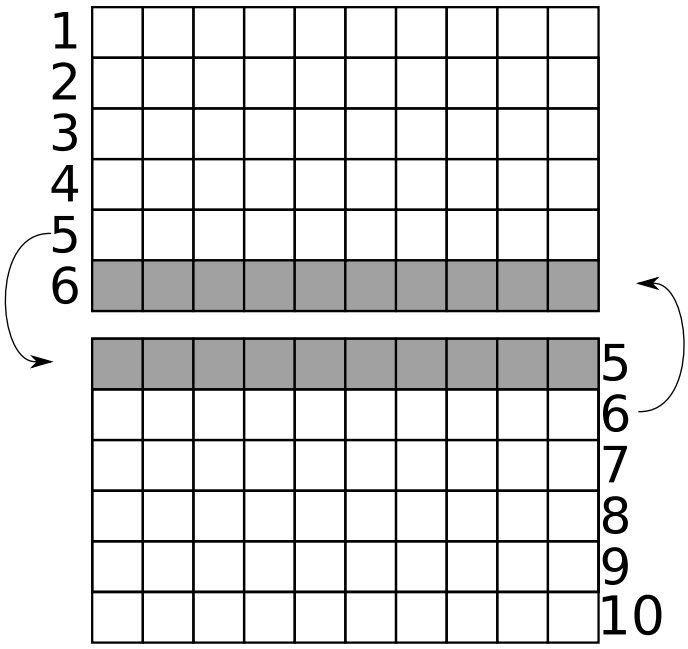


Figure 3.3: Image of the process needed when splitting the domain and interchanging values.

3.3 Memory Management, Macros and Organizing 3D Problems

We are solving a 3 dimensional Jacobi problem and, even though C natively support 3 dimensional arrays, we use a 1D array for all calculations. This is standard in scientific computing and the massively popular BLAS and LAPACK libraries use this approach. This popularity comes from the ease of determining how to traverse the data in memory. For a 3 dimensional array we must remember which dimension is contiguous in memory while for a 1d array its very simple to traverse it without creating unnecessary cache misses. Additionally changing between row and column major array access will be easier.

Say we had a pointer to a dynamically (or statically) allocated array `A` in 3 dimensions (with sizes I, J, K), we could access element (i, j, k) by `A[i][j][k]`. This would return the element in the k'th column from the j'th row in the i'th matrix. In the case where `A` is a 1D array with size $I \cdot J \cdot K$, we could access the same element as before with `A[i*J*K + j*K + k]`. Here we go i 'matrices' out in the array, followed by j rows and lastly k elements down the desired column. Doing this for all pointer accesses is certainly both cumbersome and error prone. Therefore we implemented the macro `IND_3D()` for a unified way of linear indexing in 3D arrays. The implementation is shown in Listing 3.1. A macro in C can be defined like a function, but instead of separate compilation and linking, the preprocessor simply replaces all instances of the macro in the code with the definition

before compilation begins. Note that all variables are wrapped in parentheses, which is done to avoid erroneous input expansion. All pointers in the code are therefore called like `A[IND_3D(i, j, k, I, J, K)]`, as the macro only returns the linear index for the pointer.

Listing 3.1: `IND_3D macro.`

```
1 #define IND_3D(i, j, k, I, J, K) ((i)*(J)*(K) + (j)*(K) + (k))
```

Using the linear indexing system does remove a bit of the connection to the mathematical problem we are attempting to solve. In Figure 3.4 we can see how the indexing scheme matches with the physical dimensions. This is done to preserve the matrix-like calling of the arrays, which we deemed to be the least error prone way of accessing elements. However, as the grid of our problem is usually specified in physical dimensions, we have to convert from that to our computational domain as per the figure. What we actually do in the code is shown in Listing 3.2. The code is then only utilizing the mathematical (I, J, K) and the physical sizes are hidden until the output of the program.

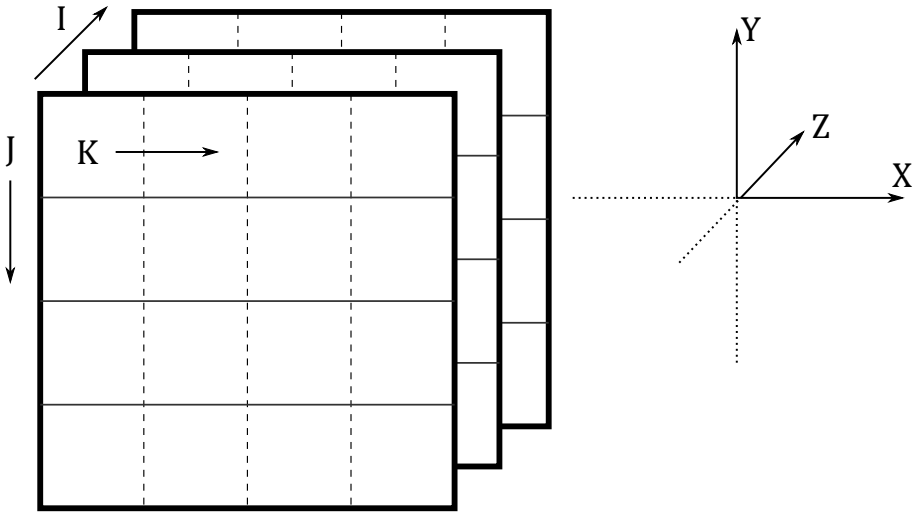


Figure 3.4: Figure showing how the allocated memory connects with the physical dimensions of the problem. Dashed lines are fast to traverse across, full lines are slower and thick lines are very slow.

Listing 3.2: `Converting from physical to computational domain.`

```
1 // Extract problem dimensions
2 int rank = information->rank;
3 int loc_Nx = information->loc_Nx[rank];
4 int loc_Ny = information->loc_Ny[rank];
5 int loc_Nz = information->loc_Nz[rank];
6 // Rewritting to C style coordinates
7 int K = loc_Nx, J = loc_Ny, I = loc_Nz;
```

3.4 Computing the Error of the Numerical Approximation

As described in section 2 we developed a test example with an analytical solution that we can confirm our numerical implementation against. The test solution itself is shown in Equation 2.2, but we need a way to compute the error of our numerical approximation. One way is to compute the solution, write it to a file, and read it into a higher level language like Matlab, where we can easily define the solution and compare the different errors. This is possible with the function `print_jacobi3d_z_sliced` in `Poisson/src/util/matrix_routines.c` that is called when the program is run with the environmental variable `OUTPUT_INFO=matrix_full` set. The result matrix is printed to `stdout` in a format where the first line specifies `Nx Ny Nz` and the next line is a continuous stream of the matrix values in row-major order. In order to print the array correctly when using MPI all prints are done by rank 0. Once rank 0 has printed the local data it then receive and print the data from the other ranks in turn.

The print-read-check approach is slightly cumbersome to use in practice and we wish to have methods to compute the error (or an approximation thereof) while running the program. For this we have developed to different methods. The first computes the analytical solution and compares it to the computed result, whereas the other uses a Frobenius norm approximation while iterating. The second method is also implemented as a stopping criterion for the iterations.

Error from Analytical Solution

This error calculation is invoked by calling the driver with `OUTPUT_INFO=error`.

The error is calculated after the called function has stopped iterating. The true solution is calculated from (2.2), where each MPI rank calculates its own contribution for all grid points. The true solution is generated with `generate_true_solution` in `Poisson/src/jacobi/jacobi_util.c` and the important parts are shown in Listing 3.3. First we determine how far in the z-direction we are depending on the rank. Then we iterate through all interior grid points to calculate the solution.

Listing 3.3: Snippet of `generate_true_solution` in `Poisson/src/jacobi/jacobi_util.c`.

```
1  // Setting up steps and variables
2  double hi = 2.0 / (Nz - 1.0);
3  double hj = 2.0 / (Ny - 1.0);
4  double hk = 2.0 / (Nx - 1.0);
5
6  // Determine how far we are in the z-direction
7  double z = -1.0;
8  for (int r = 0; r < rank; r++)
9    z += hi*(information->loc_Nz[r]-2.0);
10
11 for (int i = 0; i < I; i++)
12 {
13   double y = -1.0;
14   for (int j = 0; j < J; j++)
15   {
```

```
16      double x = -1.0;
17      for (int k = 0; k < K; k++)
18      {
19        A[IND_3D(i, j, k, I, J, K)] =
20          sin(M_PI * x) * sin(M_PI * y) * sin(M_PI * z);
21        x += hk;
22      }
23      y += hj;
24    }
25    z += hi;
26  }
```

After this step all threads own an array of the true solution for their portion of the grid. We wish to inform the user of the maximal error on the entire grid, which is accomplished with `compute_global_error` shown in Listing 3.4 and `compute_max_error` both found in `Poisson/src/jacobi/jacobi_util.c`. The latter function simply iterates over the array and saves the largest difference between the approximated and the true solution. The global error is calculated with Listing 3.4, where all threads computes their own contribution and the global error is computed with a call to `MPI_Reduce`. Here the maximum of the input values over all threads is computed and the result is saved in a `global_error` for rank 0. Of course the final print of the error is only carried out for rank 0 as well.

This approach could be improved as there is no reason for spending additional memory generating the true solution in a new array. What could have been done instead was to just loop through our approximation and compare with the theoretical equation at each grid point. This would save both memory (by not having to save the entire solution array) and computational effort.

Listing 3.4: `compute_global_error` in `Poisson/src/jacobi/jacobi_util.c`.

```
1  // Function to compute global error on solution.
2  void compute_global_error(Information *information, double *A, double *U,
3    double *global_error)
4  {
5    *global_error = 0.0;
6    double local_error = 0.0;
7
8    compute_max_error(information, A, U, &local_error);
9    MPI_Barrier(MPI_COMM_WORLD);
10   MPI_Reduce(&local_error, global_error, 1, MPI_DOUBLE, MPI_MAX,
11     0, MPI_COMM_WORLD);
12 }
```

Error from Relative Norm Approximation

This error calculation is invoked by calling the driver with the environment variable `USE_TOLERANCE=on`.

The approach outlined in the previous method is perfect for computing the absolute error of the approximation. But sometimes an approximation of the error is desired at run time to determine whether or not you need to keep iterating. To tackle this problem we implemented a calculation of the Frobenius norm to calculate the difference between iterations

to be used as a stopping criterion.

The Frobenius norm of a matrix A is given as

$$\|A\|_F = \sqrt{\sum_i \sum_j a_{i,j}}, \quad \forall i, j \in A.$$

So if U_i is the current iteration and U_{i+1} is the next iteration (that has just been calculated), we calculated the difference between these

$$\texttt{norm\_diff} = \|U_i - U_{i+1}\|_F .$$

We then stop iterating when `norm_diff` is below the given tolerance, that can be set with the environment variable `TOLERANCE` as described in README.txt in Listing A.1.

We have currently only implemented the relative norm stopping criterion for the MPI versions, as we did not have the time for the CUDA version. The norm calculations are carried out in the `jacobi_iteration`-functions. When to stop is calculated and decided with the function `norm_early_stop` again found in `jacobi_util` and Listing 3.5. Here we use `MPI_Allreduce` to sum the results over all threads and distribute the result back to all threads to ensure that they all break together.

Listing 3.5: `norm_early_stop` in `Poisson/src/jacobi/jacobi_util.c`.

```
1 // Calculate if norm criterion is reached
2 bool norm_early_stop(Information *information)
3 {
4   // Reduce and send global norm diff to all threads
5   MPI_Allreduce(&information->norm_diff, &information->global_norm_diff, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
6
7   // Returning boolean for stopping
8   return (information->global_norm_diff < information->tol);
9 }
```

The calculation of the norm error itself is done inside the Jacobi iteration functions. As can be seen in the code we put in `if`-clause inside the loop about whether or not to run the norm calculations. Usually it is unwise to put `if`-statements inside loops as it breaks the pipeline and can disrupt compiler optimizations. However, we ran some tests with the calculations inside the loop (with the `if`-statement) and outside the loop, where we evidently had to loop over the array once more. These tests showed that for small grid sizes moving the norm calculations into their own loop was fastest, but for larger grids having only one loop yielded the best performance. Thus we ended up placing the calculations inside the loop as we wish to calculate large grids as efficiently as possible.

3.5 Measuring performance

In order to determine the efficiency of our implementations we measure the performance of our code. One way is to measure how efficient our methods are in terms of computations against memory. We analyze how many computations our code is doing per second and plot this against the size of our arrays or how much memory is spent during the computations.

This way we can see how well our methods handle an increasingly large problem and hopefully we can find out if the improvements we make actually improve the performance. In order to measure the amount of floating point operations (FLOP) we executed throughout the program we add up the number of operations done in the Jacobi solver and in the initialization of the data. From Atkinson [2014] we have measurements on combined operations like sin and square root which costs 15 and 6 FLOP. Using those measurements we calculate $55 \cdot N$ FLOP for the initialization and $41 \cdot N \cdot I$ FLOP for the iterations, where N is the total number of elements and I is the number of iterations we compute.

4 Implementations

All code is on Github and can be cloned and forked here. The git repository contains a number of folders serving different purposes. The list bellow is an outline of the structure within the folder. As mentioned earlier this report is based on version 1.0 of the code.

- Poisson – Contains all files associated with the code and compilation of `jacobiSolver.bin`
  - lib, containing the header files.
  - src, containing the source code.
    - jacobi, contain the Jacobi implementations and specific tools for these.
    - util, containing the more general source files.
  - Makefile.
  - README.
- Testing
  - matlab, contain files used to create figures used in the report.
  - shell, contain files used for testing the code on the DTU cluster.

4.1 Driver

For the program we are writing here, we have decided to handle all interface with the user through a driver. This way we can make sure functions etc. are called correctly and objects are defined correctly before attempting to solve the problem desired. This helps avoid unintended deadlocks of the parallel processes and so on. The driver must be called through a terminal and has to receive a couple of inputs. First input defines which implementation is used to solve the problem, `omp2d`, `mpi3d_1` and so on. The next inputs are the sizes of each dimension. If a single number is given it is assumed all sizes are the equal, eg. `>./jacobiSolver omp3d 10` assumes the 3 dimensional problem must be solved using the openMP implementation and the problem is defined on a 10x10x10 grid.

Along with the regular inputs to the function additional functionality can be used through environmental variables set before calling the function. These environmental variables defines settings such as maximal number of iterations, tolerances and which types of output to show to the user. A full list of inputs and description of each of these can be seen in the Readme either in Listing A.1 or supplied with the code through Gihub.

One of the main functionalities which can be controlled through the environment variables is the type of output we want from the function. This variable is called `OUTPUT_INFO` and can be one of the following: `timing` (which is the default), `error`, `matrix_full` and `matrix_slice`. The two matrix output is useful for debugging and visualization purposes while error and timing is better suited for performance and validation.

Listing 4.1: An example of how the program is called using the MPI library while specifying which output is desired.

```
1 > OUTPUT_INFO=error mpiexec -q -n 2 ./jacobiSolver.bin mixed_3 200
2     Grid: 200 200 200, iter: 10000, error: 6.7542334e-03
```

Information structure

In all of our functions, we need a lot of meta information about the problem and how the problem is structured. In order to simplify the function calls and to help organizing the code in general we decided to implement a structure that is used to handling all of this meta data. The structure is called `Information` and contains a range of fields holding the needed data. The structure can be seen in Listing 4.2. The structure is allocated on the stack in our main function and a pointer is passed around the program. The structure is filled in the routine `write_information` located in `Poisson/src/jacobi/jacobi_util.c`. Most notably the structure contains information about all local sizes (the `loc_N*`-arrays). As we have only done splits in the z-axis we only use this array currently for computing the analytical solution and for printing the solution. But we have included the arrays for the x- and y-direction for completeness and for making the code suitable for splits in other directions. A function `free_information_arrays` is included in the same file to free the allocated arrays in the structure.

Listing 4.2: `Information` structure.

```
1  typedef struct Information {
2    // MPI information of size and local rank.
3    int rank;
4    int size;
5
6    // Global dimensions of the problem
7    int global_Nx;
8    int global_Ny;
9    int global_Nz;
10
11   // Arrays holding the local number of gridpoints for each rank.
12   int *loc_Nx;
13   int *loc_Ny;
14   int *loc_Nz;
15
16   // Variables for looping
17   int    maxit;
18   int    iter;
19
20   // For relative stop criterion
21   double tol;
22   bool   use_tol;
23   double norm_diff;
24   double global_norm_diff;
25 } Information;
```

4.2 Testing

All code has been tested on the DTU cluster[1], more specifically in the `gpuv100` queue. The cluster is a system of 6 nodes with 2 CPUs of type Intel Xeon Gold 12 cores. Connected to each node is also 2 GPUs of type Tesla V100 PCIE 16GB. More information on the CPU and GPU can be found in Appendix B.

[1] See more at https://www.hpc.dtu.dk/.

Tests are run on the cluster through the system called Platform Load Sharing Facility (LSF). This is the scheduling system used on the GPU nodes of the DTU cluster. We have implemented a number of submission scripts in order to test our program. Now a quirk with the LSF system is that the executions are run directly in the folder system of the user, therefore if a file is changed by the user while the job is being executed the output might be changed or the execution might fail.

A shell function called `submit.sh` has been written in order to streamline testing of the code and ensure problems with updated files are avoided as much as possible. This function accepts the name of a test which should be queued on the LSF system. All the submition files defining jobs which should be run is defined by the naming scheme `submit<TESTNAME>.sh`. The function then compiles the driver `jacobiSolver.bin` such that we ensure the newest version of the code is used. All necessary files are copied into the log folder under a temporary folder named after the test that is the working directory for the LSF job. All output data files and figures are kept in this folder until the entire computation is done and is then moved into the `data` and `figures` folders located in `Testing`.

Throughout our testing the size of problems we can solve is limited by the hardware. On the test system we have access to a very large amount of RAM, but since we are testing how to expand the tests to the GPUs on the system we have to stick with problems which can be solved on the GPU. The total amount of memory needed to solve the problem is about $3 \cdot Nx \cdot Ny \cdot Nz$ double precision floating point numbers. The 3 multiplier comes from the jacobi method needing 3 arrays holding the previous iteration U, the current iteration U_{new} and the source F. With the total amount of GPU RAM being 16GB per GPU we can expect to be able to solve problems of about 800^3 elements on a single GPU. The theoretical max as a function of the number of GPUs can be seen in Figure 4.1. Here it can be seen that even with all 12 GPUs the maximal number of elements is still only about 2000 for each dimension. This is a theoretical max though, there will be additional memory needed for buffers, constants and some that are reserved by the operating system.

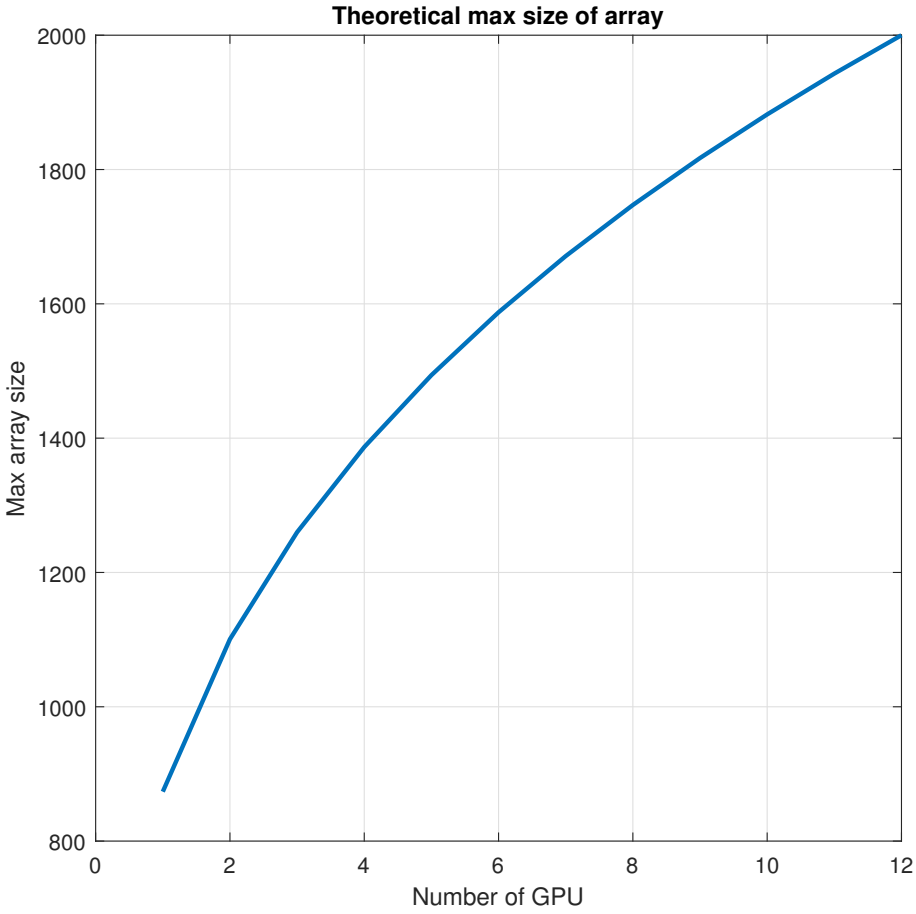


Figure 4.1: Theoretical maximal size of problem assuming the cubic decomposition ($Nx = Ny = Nz$) shown against the number of GPUs used in the computation.

5 Implementations of the Jacobi solver

Figure 5.1: Timeline of which order the implementations was completed.

This section deals with the different implemented versions of the Jacobi solver. Figure 5.1 shows the order in which the different versions of the solver were implemented. The very first version of the code is an OpenMP based 2 dimensional Jacobi method developed during the course High Performance Computing[1]. With this we already had a working framework that could be build upon. Before working on improving the code and including several new technologies we decided to expand the solver to handle 3 dimensional problems. As described in section 3 all arrays are written with linear indexing and accessed through the macros defined earlier. This means the addition of another loop around the original computation made for a very simple solution to expanding to 3D. This version of the code is our minimal working example and baseline for improving with the more advanced technologies.

After the 3D OpenMP version was done the first MPI version was made. MPI 1-2 were made such that a general framework for message passing between threads was established and tested. With this done we returned to the bare 3D version and rewrote it into CUDA 1 such that the iteration itself was done on the GPU. From there we started the mixed versions (1 through 5), where the computations were on the GPU and boundary sharing was accomplished with MPI.

All versions are introduced below arranged according to which technology was used.

5.1 MPI implementations

The MPI versions were developed in order to ensure the boundary sharing between threads was implemented correctly before we started calculating on GPUs with CUDA. The domain decomposition as described in subsection 3.2 is implemented with MPI in the current section.

MPI version 1

The first step in expanding the program to utilize the cluster of DTU is to make a single cut and place the two parts of the domain on different ranks. This is done as a cut half way up the z-dimension of our domain. A very important part of this version is to make sure the domain is split and initialized correctly. Additionally we now introduce the need of communication between the two ranks as they must interchange the boundary values as

[1] University course at DTU. Course number: 02612, see http://kurser.dtu.dk/course/2017-2018/02614 for more information.

described in subsection 3.2. This is done with the two commands `MPI_Send` and `MPI_Recv` which sends and receives messages from other ranks.

The reason for choosing the third dimension for our cut is that this is the dimension with the largest stride in our data, this means a boundary which is fixed in z will be contiguous in memory and therefore performance is maintained in terms of memory management.

MPI version 2

Now that a single cut can be made on our domain we expand this to support many cuts, still just along the 3rd dimension of our data.
All of these cuts must be done in a fair way so the data is seperated as evenly as possible. The simplest way to do this is to assign an even part to each rank $\frac{Nz}{size}$ where $size$ is the number of MPI ranks in total. Now this might not be possible depending on the amount of data e.g. 14 split in 5 will not be possible since the last rank will attempt to access data outside the array, therefore the last rank will just get whatever is left when all the other ranks have received their data. Additionally we should remember to include space for the ghost lines we described in subsection 3.2. Listing 5.1 shows how the splits are implemented in the code.

Listing 5.1: Computation of the local sizes.

```
1 if      (rank == 0)      {loc_Nz = (Nz/size) + 1; }
2 else if (rank == size-1) {loc_Nz =  Nz - (size - 1)*(Nz/size) + 1;}
3 else                     {loc_Nz = (Nz/size) + 2; }
```

MPI version 3

Now that we have a Jacobi solver written using MPI we start improving the performance of the solver instead of adding features. First step in improving the performance is to hide some of the send and receive latency behind computations. This is done by splitting the computations in two. First we compute the boundary of the section we want to complete. The boundaries are then posted for send and we post receive calls to get the solutions from our neighbours. These sends and receives are done asynchronously such that when the calls are posted we can compute the interior of our local part of the domain. Once all computations are completed we must ensure the boundary exchange has completed and insert the boundaries in the array. After this we continue with the next iteration.

Implementation-wise this is accomplished by modifying our `jacobi_iteration` function to the function `jacobi_iteration_separate`. This function now accepts a character stating whether the interior (i) or boundary (b) should be computed.
However, if we use the same method for sending and receiving data as we did earlier we would still pause the code. We must use non-blocking MPI communication by invoking the `MPI_Isend` and `MPI_Irecv`. These two functions need a request which we can use in the function `MPI_Waitall` to create a barrier for the code which halts execution until the local sends and receives are completed, but we do not have to wait for all other ranks like we do with `MPI_Barrier`.

Performance

Each step in the implementation should improve the performance of the solver. The first version of our MPI implementation had only a single split in the domain, therefore we are limited to only 2 ranks for our solver. This will severely limit the performance compared to versions which can utilize more resources. The two other implementations have been tested using 6 nodes each housing 2 cores in order to compare the results to the mixed versions.

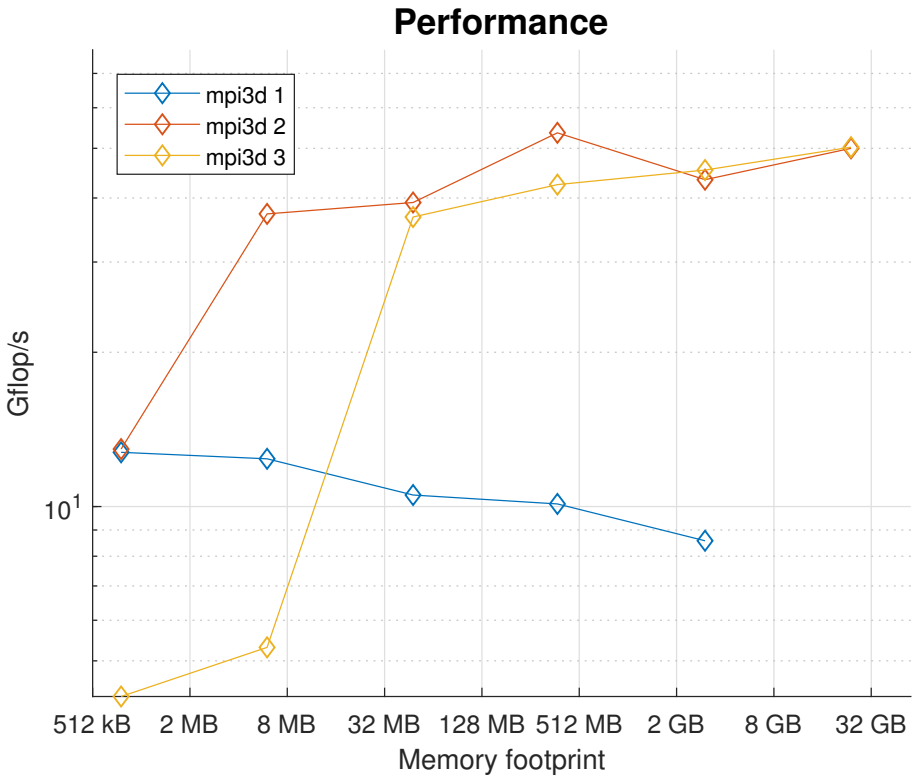


Figure 5.2: Figure showing the performance of the different versions of the code using MPI. Both axes are logarithmic (x-axis log2 and y-axis log10) to help display the results. Version 1 is tested on 2 cores while version 2 and 3 is tested on 12 cores.

In Figure 5.2 we see how well the different versions of the code performs. It is worth noting how much it matters that the computations in version 3 have been split into a boundary and an interior computation. For small problems the performance is quite a bit lower than the other versions because we do not get to use the cache in an efficient way, but when the problem size grows so does the performance. Overlapping the computations and the receiving of data does not seem to provide a very distinct advantage for these versions of the code. The 2nd version is only marginally worse for the large problems than the 3rd version of the code, however the increased amount of resources available to version 2 and 3 clearly gives a big advantage compared to the first version of the code.

5.2 CUDA

Before making a solver utilizing both CUDA and MPI we had to develop a standalone CUDA solver. This solver only uses a single GPU for all calculations. The problems and all arrays are initialized on the CPU, copied to GPU and the result copied back to the CPU. We could have optimized this version further by initializing on the GPU and saving some memory copies, but as the scope of this report was to combine GPUs with MPI, we did not explore this any further.

The CUDA kernel used for computing the iterations is called `jacobi_iteration_kernel` and is found in `Poisson/src/jacobi/jacobi_util_cuda.cu`. As described in section 3.1 kernels are launched with grid-specifications. As our model is 3-dimensional by nature we use the built-in `dim3`-vector class in CUDA to specify our problem. With this we can identify grid points in 3D space much like we usually do in the sequential case with three for loops. We identify the thread inside the kernel with the first part of Listing 5.2. Notice how we index as was described in subsection 3.3. Each thread then calculates the contribution exactly like was done in the sequential case but without the loops, as each thread already has a position in 3D-space.

This way of indexing the threads do not utilize the threads perfectly. Some threads will be idle and therefore the resources could be utilized in a better way. A change could be to take advantage of the linear indexing of our implementations and just spawn a 1 dimensional grid for the kernel instead of a 3 dimensional one. Another way of solving the problem with idle threads could be to push the idle threads further down to another index in the array, this would however be cumbersome.
The performance gain of using the idle threads would not be significant. It is only a couple of threads each time the block reaches the boundary of the problem so the computational loss is fairly minimal.

Listing 5.2: Snippets from `jacobi_iteration_kernel` kernel.

```
1  // Determine where the thread is located
2  int k = threadIdx.x + blockDim.x*blockIdx.x;
3  int j = threadIdx.y + blockDim.y*blockIdx.y;
4  int i = threadIdx.z + blockDim.z*blockIdx.z;
5
6  ...
7
8  // Compute grid points
9  if (
10   ( (i > 0) && (j > 0) && (k > 0))
11   && ((i < (I-1)) && (j < (J-1)) && (k < (K-1) )) )
12 {
13 ...
14 }
```

Copying information

Now a big caveat for this method was the `Information` structure we have defined. This structure contains arrays which we also transfer to the GPU since all size information is

stored here.
In order to use the information structure on the device we must first copy it over. This was a challenge since we have the pointer to a structure which contains arrays. How we solved it is based on a solution found on Stackoverflow [2015].
The copying is done by first creating a temporary information structure on the CPU, but the arrays are allocated as device arrays through `cudaMalloc`. A memory copy used to get the arrays from the CPU information to the temporary information. Lastly we copy the temporary information structure to the device. This way we avoid problems with attempting to access memory on the device from the host and vice versa.

Performance

The performance of our CUDA implementation depends very much on the size of our problem. It is very clear that we do not fully take advantage of the computational power of the GPU in this program. The bigger our array gets the better performance we get. This is clearly seen since the performance does not stagnate in Figure 5.3.

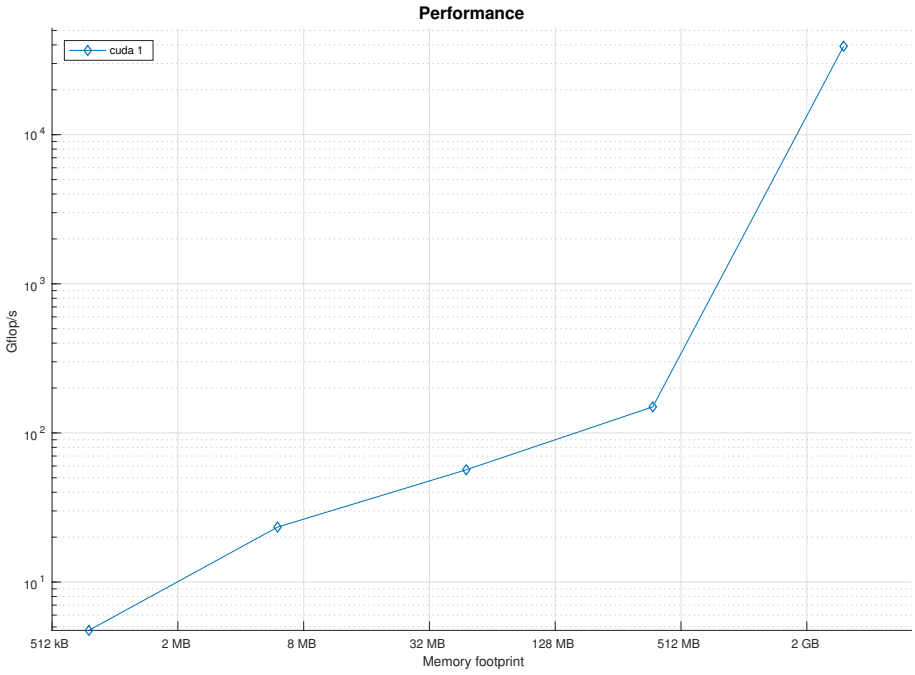


Figure 5.3: Figure showing the performance of the CUDA version of our code.

5.3 MPI and CUDA

Now we turn to the mixed versions, where CUDA is used for the computations and MPI for the message passing of the boundary arrays.

A compilation problem arises when CUDA and MPI is to be combined. CUDA is written in the language C++ and we use the MPI bindings written in C. So in order to use the CUDA functions we must create a number of wrapper functions which can be compiled by nvcc, the NVIDIA CUDA compiler[1]. All of these functions are located in the file `cuda_routines.cu` and include wrappers for functions like `cudaMalloc`, `cudaMemcpy` and

so on. Additionally we must specify for the nvidia compiler that all other code is written in C and not in C++, this is done in the header files by specifying `extern "C"` if we attempt to link the code with a C++ compiler.

The system used to test the code contains nodes with two GPUs. Because of this we must explicitly tell each MPI rank which GPU is available for them to use. Thus we wrote the function `cuda_set_device` that receives the current MPI rank as an input, reads how many devices are available through `cudaGetDeviceCount` and assigns a GPU to the ranks. If only 1 GPU is available for each node then each MPI rank should communicate with device 0, if the number of GPUs are 2 then all even ranks get device 0 and uneven ranks will get device 1 assigned.

Mixed version 1

The first implementation was based on the second MPI implementation, meaning several splits along the z-dimension and the first CUDA implementation for how we compute each iteration. This means we transfer the array to the GPU in every iteration, compute the solution on the GPU and transfer the result array back to the CPU. MPI then takes care of transferring the boundaries to the correct neighbours. This system repeats for each iteration of our scheme. This way of implementing the code is ideal, but it gives a simple working version of the code and shows that it is possible to combine the two technologies.

Mixed version 2

A large part of performance comes from improving how we exchanging information, both between nodes through MPI and between each node and the attached GPU. The first optimization we decided to do was to minimize the amount of data transferred between the GPU and the CPU. This was done by writing directly from the GPU to the send buffers we use in MPI. This way only the data which have to be transferred to other nodes is saved on the CPU memory, but we avoid copying over the entire solution in each iteration.

Mixed version 3

Now that the GPU version is working along with the MPI implementation we have used the methods we implemented for the 3rd version of the MPI code to hide the latency. This is done by rewriting our `jacobi_iteration_cuda` such that we can compute the boundaries separately. This ensures the possibility to asynchronously transfer data between the MPI threads while the GPU is working on the interior of our domain. The implementation is done similarly as to what was described in MPI version 3 subsection 5.1 with a wrapper function that takes in a character parameter that determines whether to launch the boundary or the interior kernel.

Mixed version 4

After using the Nvidia Visual Profiler we realized that the code is still not executing simultaneously on the node itself. Therefore we implemented a stream based waiting

[1] See https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

system. CUDA kernels are by default launched in an asynchronous manner which means we can post the computation of both the boundary and interior parts of the domain. Once the boundary is complete we can start transferring the data. When the sends and receives are posted we wait until the transfers and the interior computation are complete. This way we overlap all of our computations with out data transfers and minimize the amount of idle resources.

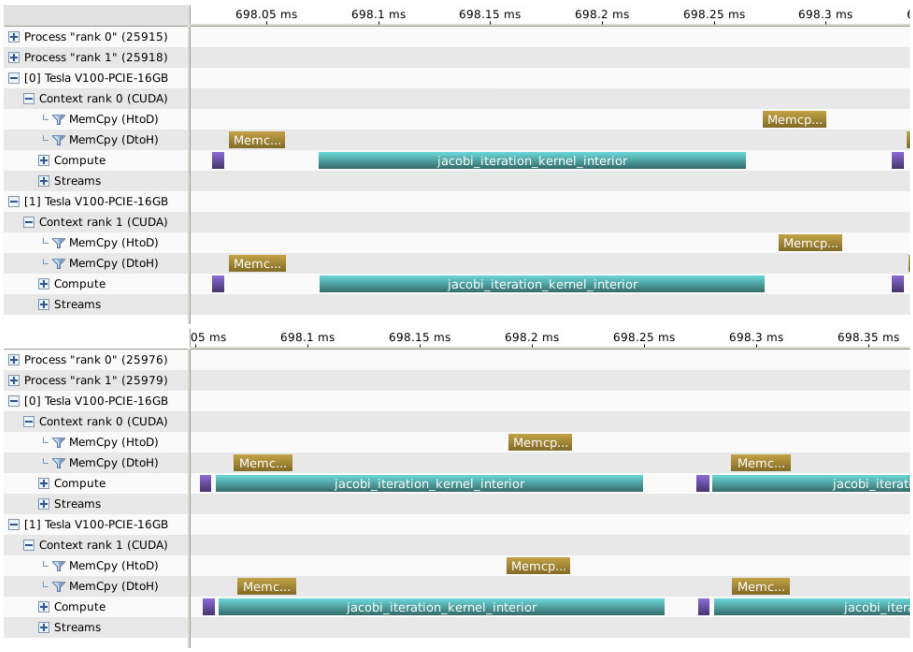


Figure 5.4: This figure show the analysis from Nvidia Visual Profiler. The top image is the analysis of the 3rd mixed version while the bottom image is mixed version 4. It is clear that parts of the code are now running asynchronously.

Listing 5.3: Snippet showing the usage of CUDA streams.

```
1  void cuda_create_stream(void **stream)
2  {
3    *stream = malloc(sizeof(cudaStream_t));
4    cudaStream_t streamT;
5    cudaStreamCreate(&streamT);
6    memcpy(*stream,&streamT,sizeof(cudaStream_t));
7  }
8  void jacobi_iteration_cuda_separate_stream( ... ,void *stream)
9  {
10   cudaStream_t *s;
11   s = (cudaStream_t*)stream;
12     ...
13   // interior or boundary
14   if (strcmp(ver, "i") == 0)
15     jacobi_iteration_kernel_interior<<< ... ,*s>>>( ... );
16
17   if (strcmp(ver, "b") == 0)    // boundary
18     jacobi_iteration_kernel_boundary<<< ... ,*s>>>( ... );
19   ...
```

```
20 }
```

The main body of our code is written in C. Because of this we cannot create an object of the type `cudaStream_t` outside the cu files. In order to create separate streams for the interior and boundary computations we had create a `void *` and then in a cu function create a `cudaStream_t` and copy the stream to the pointer location. Every time we need to use the stream we then have to do a typecast to ensure readability for the program. Listing 5.3 show how we create the streams and an example on how they are used.

Mixed version 5

The next step in our optimization is to assist the transferring of data between the different GPUs. Since OpenMPI 1.7 it has been possible to combine MPI and CUDA directly by passing GPU pointers to `MPI_Send` and `MPI_Recv` [Kraus, 2013]. MPI will then handle transferring the data and create a streamlined pipeline for the data as shown in Figure 5.5. This should help with both the readability of the code, since we can cut out several lines of code by using this method and it should increase the performance of the code by using the memory bandwidth much more efficiently.

(a) Regular MPI implementation (b) CudaAware MPI implementation

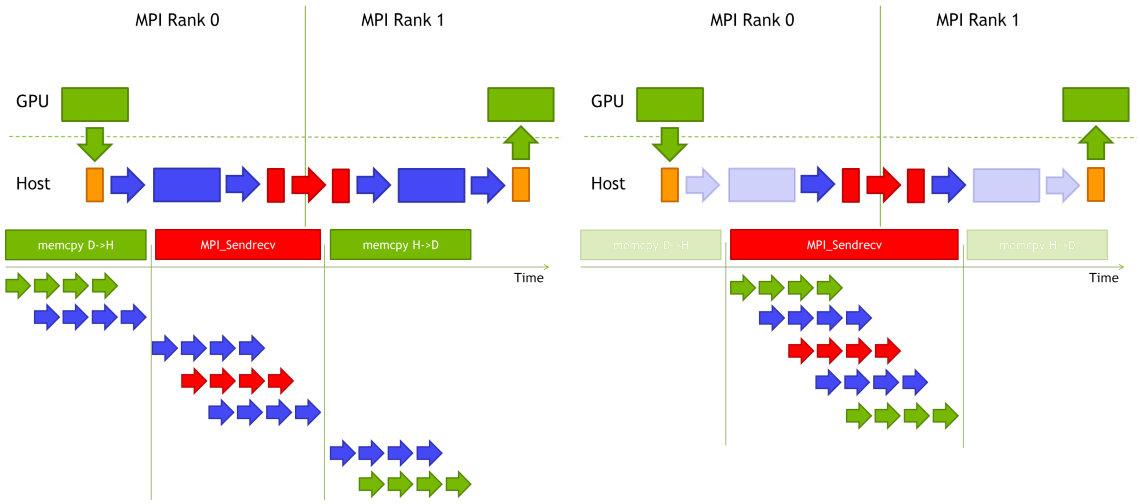


Figure 5.5: The two figures here shows how a CUDA Aware MPI library is able to create a more streamlined transfer from GPU to GPU by packing the data and send it on as soon as there is room on the connection. The regular implementation will stall the pipeline since all data must be on the host before mpi can send it to the neighbour. Figures are from at [Kraus, 2013].

Using this CUDA aware approach might in theory yield an improvement, but none of the OpenMPI libraries we have available at DTU is compiled with the CUDA aware setting on. Due to this we have been unable to actually test if the implementation is correct or how big the performance benefit of this approach is.

Performance

The performance gains we got from minimizing the amount of data which is transferred between the GPU and the CPU provided an enormous speedup relative to the regular MPI implementation as seen in Figure 5.6. For the mixed code we measured hundreds and even thousands of GFLOPs where the MPI version maxed out below 100 GFLOPs.

Compared to MPI the first version of the mixed code does provide some speedup and better consistency in the performance. It scales well through our testing even though it does seem to approach a maximum performance slightly above 100 GFLOPs. This is clearly caused by the huge amount of data we are transferring from the GPU to the CPU. Reducing the amount of data transferred between the CPU and GPU improves the performance greatly as can be seen in Figure 5.6 when comparing version 1 to the rest.
The effect of compute streams is clearly visible as the performance for small problems are greatly improved from version 3 to version 4. For even for small problems it is a clear advantage to use the split computations of interior and boundary since we have full overlap of computations and the sending and receiving of data.
As the problem size increases the two methods 3 and 4 converge since the interior computations will dominate the computational effort.

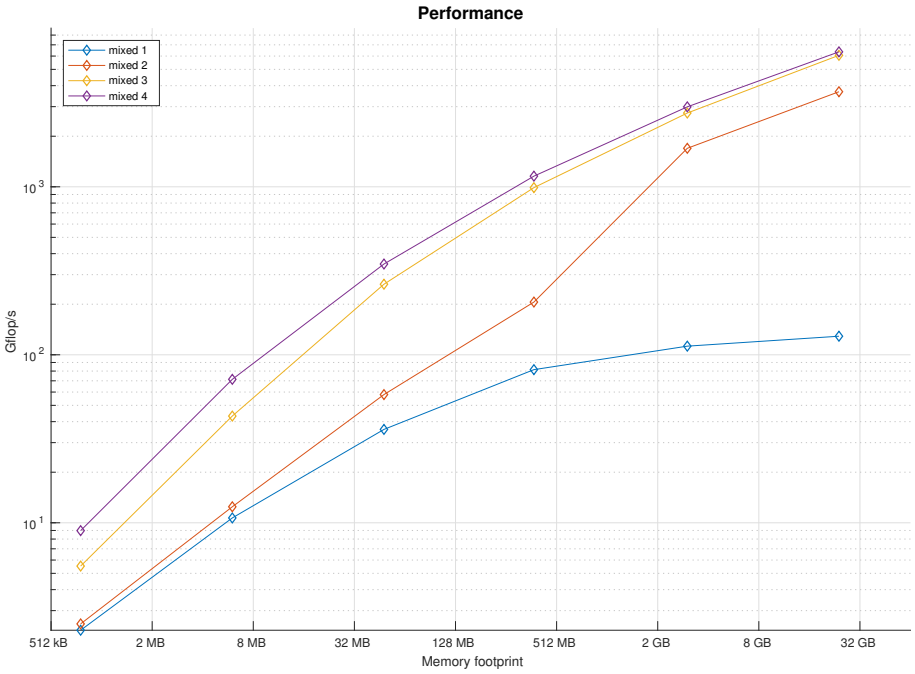


Figure 5.6: Figure showing the performance of the different versions of the code using a combination of MPI and CUDA.

6 Comparisons and performance

For comparing all implemented methods we decided to focus on the best performing method for each type of implementation. Thus the methods compared here are `cuda_1`, `mixed_4`, `mpi3d_3` and `omp3d`. The performance of these 4 methods is shown in Figure 6.1.

It's not surprising to see small problems being quite efficient to compute using the simple OpenMP solver as the overhead for this method is very low. As long as a single CPU can cope with the computations this method is quite efficient. However it is quickly overtaken by the methods using the GPU since the computational power of the GPU is much larger than the CPU. The overhead for sending data between nodes is clearly limiting the performance of the MPI version compared to the OMP version, but with a big enough problem this is the best to use if a GPU is not available. Another advantage of the MPI version is that we often have quite a lot of CPU cores available compared to the number of GPUs. Here the MPI version is run on 12 CPU cores, but with more CPUs come more performance. Assigning more CPUs to the program will only help cut down execution time, it will not make the program scale better to bigger problems.

A very note-able part of the performance is the relation between the mixed model and the pure CUDA implementation. The mixed model outperforms the CUDA implementation for a large part of the domain, but when the size of the problem approaches the memory limit of the single GPU we have a big spike in performance. The computational load for each iteration is not large enough to saturate the GPU. The lack of saturation explains why the performance of the GPU implementation keeps rising. However the GPU will fail on another account, it cannot compute problems larger than a certain size. Beyond this point we don't have enough memory available on the GPU to hold the arrays. Here the mixed version is again the best version we can use.

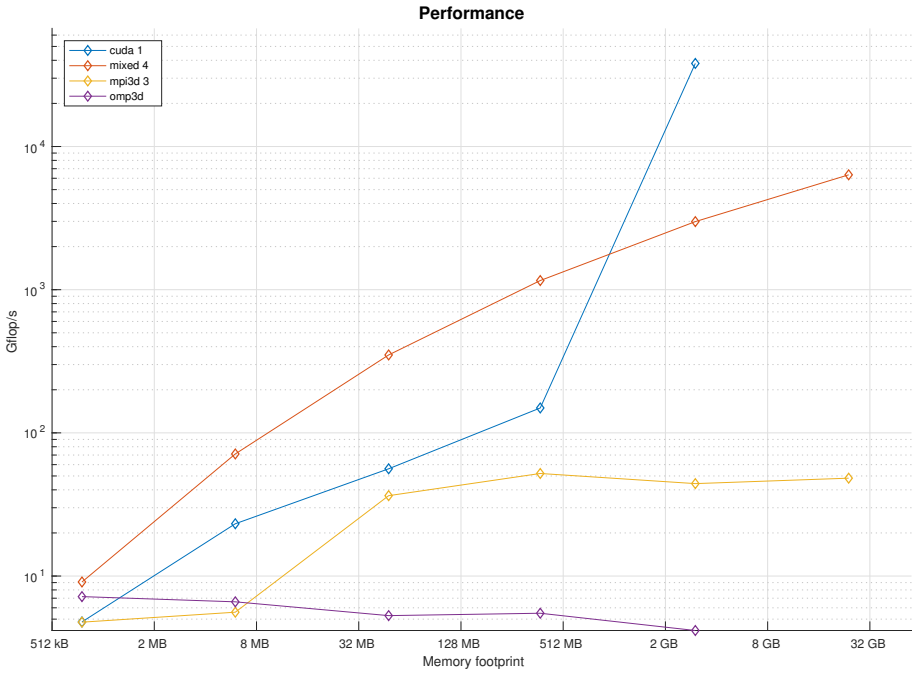


Figure 6.1: Figure showing the performance of the different versions of our implementations. The versions shown are compared in section 6.

In parallel programs another important measure of performance is scale-ability. Scale-ability describes how performance of the implementation changes relatively when more resources are added. Resource scaling can be seen in Figure 6.2 for some of the implemented methods. If our program had perfect scale-ability we would see 2 times performance when the number of computational units are doubled.

Figure 6.2 was made from solving a $512 \times 512 \times 512$ problem. The problem size was chosen to insure the problem could be solved on 2 GPU.
Mixed version 1 and both MPI implementations have a fair scale-ability the two mixed version 2 and 3 does not. The two MPI versions get a better performance when the amount of computational units increase since all computational units are saturated with the problem. Mixed 1 have a more subtle reason behind the scale-ability. In the mixed implementation version 1 we transfer the entire solution from the GPU to the CPU in each iteration. So when the number of computational units is increased the amount of data transferred from GPU to CPU is reduced which decreases execution time. That the performance comes from the decrease in transferred data is clearly visible when this is minimized in the other mixed versions. To determine the scale-ability of the mixed versions we have to run problems that are much larger than what has been done in this report.

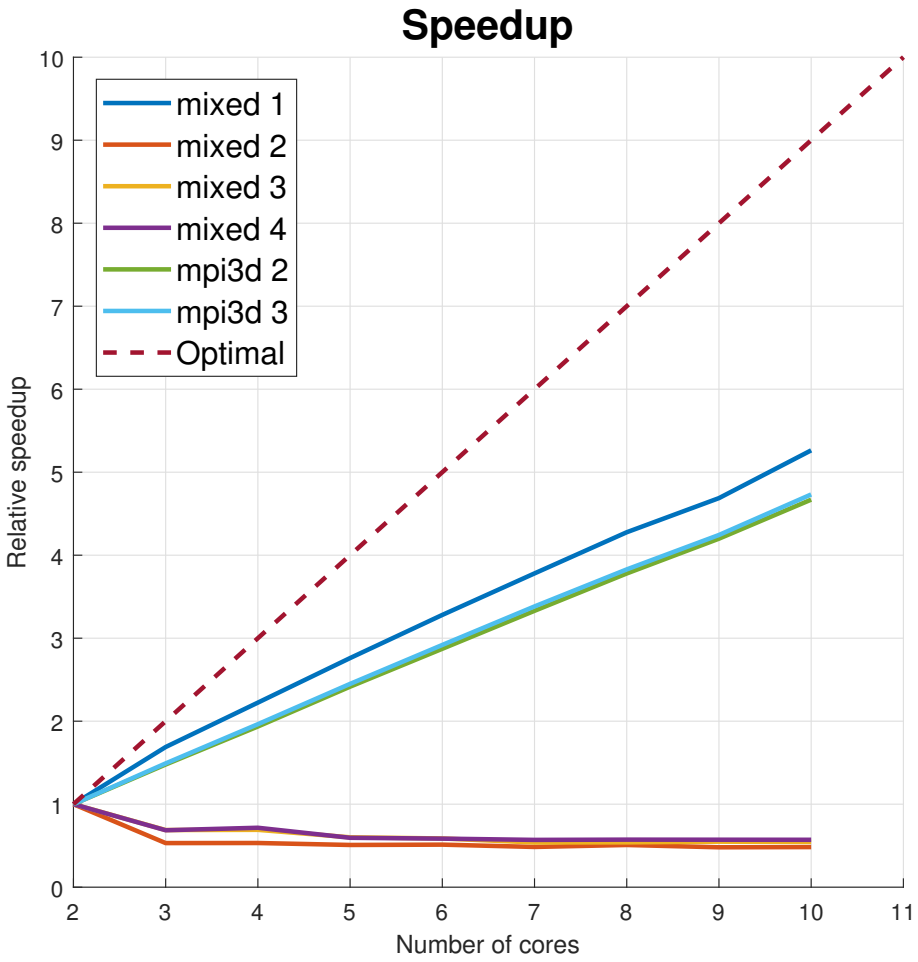


Figure 6.2: Scale-ability of the implementations which can be up-scaled. For MPI implementations this mean how many CPU cores are assigned to the program. For mixed the number of cores is the number of GPU used to solver the problem.

7 Conclusion

Throughout this project we have succeeded in implementing both CUDA, MPI and mixed versions of the Jacobi method for solving steady state partial differential equations. In terms of performance even the simple CUDA method outperforms the mixed and the MPI versions when the problem size is close to the capacity of the GPU memory. For relatively small problems the mixed version is the most efficient method and once the GPU memory is saturated, where we are forced to use multiple GPUs, the mixed versions reign supreme.

We currently have an issue that `MPI_Waitall` does not wait for the computations as we believe it should. Therefore we have been forced to use the function `MPI_Barrier` several times to ensure sends and receives are completed in each iteration. This is a hit to our performance and should be addressed.

7.1 Extended work

Based on this report a number of additional problems can be examined. The matter of testing the CUDA aware MPI implementation as described in subsection 5.3 should be addressed. Theoretically this should improve performance based on the streamlined pipeline. Not only do we expect performane to improve, but the code is also easier to write and debug when CUDA pointers are passed directly to MPI.

The implementations involving GPUs are currently limited in terms of how big problems they are able to solve. To overcome this one could place the large problem on the CPU and solve it in parts sequentially on a single GPU. If sending and receiving from the CPU could be overlapped such that the GPU calculates constantly, then we expect this solver could be very efficient. This version is, of course, limited by the memory bank of the host.

CUDA can handle several GPUs on a single node without including MPI or other external libraries. Furthermore the bandwidth between these are usually higher than what can be achieved by MPI through hosts. This might be used to increase the performance for GPUs connected on a single node. Furthermore one could have GPUs on the same node communicate directly and through MPI for other nodes.

MPI could be combined with OpenMP in order to make an efficient solver when GPUs are not available. This is done by using OpenMP for all cores on a node and MPI to communicate between nodes.

Additionally one could examine which method is optimal to use for which size of problem based on the resources available. This could lead to a very versatile solver which just gets presented a problem and then determines how to solve it. Examining this will require a much deeper look into the methods and how the test setup is organized.

The splits are made in one dimension, namely the z-dimension. This could be done in all 3 dimensions instead to help optimize the boundary to interior ratio for the problem. We want as big an interior compared to boundary as possible, which should be achieved by using a cubic decomposition.

References

Niels Aage, Erik Andreassen, Boyan S. Lazarov, and Ole Sigmund. Giga-voxel computational morphogenesis for structural design. *Nature*, 550(7674):84–86, 2017.

Lincoln Atkinson. A simple benchmark of various math operations, 2014. Last checked: 14-06-2018,
url: http://www.latkin.org/blog/2014/11/09/a-simple-benchmark-of-various-math-operations/.

Jiri Kraus. An introduction to cuda-aware mpi, 2013. Last checked: 11-06-2018,
url: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/.

Boyan S. Lazarov. 02616 large-scale modelling. University course at Technical University of Denmark, url: http://kurser.dtu.dk/course/2016-2017/02616, 2017.

Stackoverflow. Structure copying, 2015. Last checked: 29-05-2018,
url: https://stackoverflow.com/questions/31133522/simple-operation-on-structure-in-cuda-segmentation-

Dennis M. Sullivan, Yang Xia, and Alireza Mansoori. Large scale underwater fdtd elf simulations using acceleware and mpi parallel processing. *Symposium Digest - 20th Ursi International Symposium on Electromagnetic Theory, Emts 2010*, pages 5637133, 104–106, 2010.

Jun Sun and Pan Michaleris. Mpi implementation of the feti-dp-rbs-lna algorithm and its applications on large scale problems with localized nonlinearities. *Trends in Welding Research, Proceedings*, pages 553–558, 2006.

A Readme

VIGTIGT:
Remember to update the readme

Listing A.1: `README` file for the project

```
README for jacobiSolver
-------------------------------------------------------------------------------

The Jacobi solver have been implemented into the jacobiSolver executable.

Default call to the function will be as bellow, please be adviced mpiexec
    or
mpirun must be used for all mpi based versions.
  > ./jacobiSolver METHOD NX NY NZ

where:
  METHOD :
    omp2d : Is the Jacobi method using OpenMP for a 2D problem.
    omp3d : Is the Jacobi method using OpenMP for a 3D problem.
    mpi3d_1 : Is the Jacobi method using MPI with a single split along Z.
    mpi3d_2 : Is the Jacobi method using MPI with multiple split along Z.
    mpi3d_3 : This version overlaps the sending and receiving of data with
          computation of the interior of the domain.
    cuda_1  : First implementation using cuda for a single GPU.
    mixed_1 : First implementation using both MPI and CUDA. Very naive.
    mixed_2 : Second implementation using both MPI and CUDA. Copies only
          needed data.
    mixed_3 : First attempt in overlapping computations.
    mixed_4 : Stream based overlapping of computations.
    mixed_5 : (REQUIRES CUDA AWARE MPI) Passes device pointers directly to
          MPI for streamlined pipelining. This version have not been
          tested.

  NX NY NZ : Integer numbers.
    The number of points in the problem, fx. 7 by 7. Nz is the size in the
    3rd dimension. NY and NZ can be omitted for cubic domains.
      B B B B B B B
      B X X X X X B
      B X X X X X B
      B X X X X X B
      B X X X X X B
      B X X X X X B
      B B B B B B B
    Where the boundary is set during the function and the elemnets in X is
    computed.

-------------------------------------------------------------------------------

The solver will read for several environment variables and change behavior
based on these. The environment variables supported are listed bellow.

OpenMP: All default OpenMP controls such as setting number of threads
    are present and the OpenMP versions have been implemented such that
    the schedule can be set during runtime using OMP_SCHEDULE.

```

```
48 Extra environment variables: ( > ENV_NAME=value ./jacobiSolver ...)
49   PROBLEM_NAME: [default: sin]
50     Defines the problem that should to be solved.
51
52     sin :   The well defined example where the Boundary condition and
53         source is defined as:
54         (2D version):
55           f(x,y)   = 2*pi^2*sin(pi*x)*sin(pi*y)
56           u(x,y)   = 0, for (x,y) on boundary.
57         (3D version):
58           f(x,y,z) = 3*pi^2*sin(pi*x)*sin(pi*y)*sin(pi*z)
59           u(x,y,z) = 0, for (x,y,z) on boundary.
60     rad :   [ONLY OMP2D] The radiator problem with Boundary and source
61         is defined as:
62                     f(x,y) = 200 for x in [ 0, 1/3 ] and y in [ -2/3, -1/3
                           ]
63                     f(x,y) = 0 otherwise.
64                     u(x,1) = 20, u(x ,-1) =  0,
65                     u(1,y) = 20, u(-1, y) = 20.
66
67   OUTPUT_INFO: [default: timing]
68     Defines the output type.
69
70     matrix_slice:
71       Defines that the result matrix should be printed as the
72       output. This will print the slice at the center of z for,
73       3 dimensional problems.
74     matrix_full:
75       Defines that the result matrix should be printed as the
76       output for the full 3 dimensional problem. The format of
77       the output will be:
78         Nx Ny Nz
79         U(0,0,0) U(0,0,1) ....
80     error:
81       Prints an output of the dimension sizes in terms of number of
82       gridpoints in each and the maximal absolute difference between
83       the analytical solution and the solution calculated in the program.
84     timing:
85       Defines that the output should be the timing info. Giving
86       the memory footprint (kB), performance (Mflops) and walltime (s)
87       for the main loop.
88
89   TOLERANCE:  [default: 1e-6]
90     Sets the tolerance for the program to terminate when the maximal
91     absolute error falls under this number.
92
93   USE_TOLERANCE: [default: off]
94     Defines if the tolerance should be used or not.
95
96     on :  Use the tolerance as stop criterion.
97     off:  Do not use the tolerance. Force to do MAXITER iterations.
98
99   MAX_ITER: [default: 1e4]
100    The maximal number of iterations done by the solver.
101
102
103 % = = EOF = = %
```

B Computer information

Listing B.1: Information output from `lscpu` for the test system.

```
1  Architecture:          x86_64
2  CPU op-mode(s):        32-bit, 64-bit
3  Byte Order:            Little Endian
4  CPU(s):                24
5  On-line CPU(s) list:   0-23
6  Thread(s) per core:    1
7  Core(s) per socket:    12
8  Socket(s):             2
9  NUMA node(s):          2
10 Vendor ID:             GenuineIntel
11 CPU family:            6
12 Model:                 85
13 Model name:            Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
14 Stepping:              4
15 CPU MHz:               999.914
16 BogoMIPS:              5215.09
17 Virtualization:        VT-x
18 L1d cache:             32K
19 L1i cache:             32K
20 L2 cache:              1024K
21 L3 cache:              19712K
22 NUMA node0 CPU(s):     0-11
23 NUMA node1 CPU(s):     12-23
```

Listing B.2: Information output from `nvidia-smi` for the test system.

```
==============NVSMI LOG==============

Timestamp                           : Wed Jun  6 11:47:34 2018
Driver Version                      : 390.46

GPU Name                            : Tesla V100-PCIE-16GB
    Memory
        Total                       : 16160 MiB
    Clocks
        Graphics                    : 135 MHz
        SM                          : 135 MHz
        Memory                      : 877 MHz
        Video                       : 555 MHz
    Applications Clocks
        Graphics                    : 1245 MHz
        Memory                      : 877 MHz
    Default Applications Clocks
        Graphics                    : 1245 MHz
        Memory                      : 877 MHz
    Max Clocks
        Graphics                    : 1380 MHz
        SM                          : 1380 MHz
        Memory                      : 877 MHz
        Video                       : 1237 MHz
    Memory Clock Samples
        Duration                    : 18434.14 sec
        Number of Samples           : 100
        Max                         : 877 MHz
        Min                         : 877 MHz
        Avg                         : 877 MHz
```