

TECHNICAL UNIVERSITY OF DENMARK

SPECIAL COURSE

---

## Scaling Analysis of a Multi-GPU Poisson Solver

---

*Author:*

Mathias Sorgenfri Lorenz (s134597)

September 4, 2018



**Contents**

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                            | <b>1</b>  |
| <b>2</b> | <b>Hardware and Software Specifications</b>    | <b>2</b>  |
| <b>3</b> | <b>CUDA Performance</b>                        | <b>4</b>  |
| 3.1      | New Performances . . . . .                     | 4         |
| <b>4</b> | <b>Stopping criterion</b>                      | <b>6</b>  |
| <b>5</b> | <b>CUDA Using Peer-to-Peer Communication</b>   | <b>9</b>  |
| 5.1      | Accessing Devices using Peer-to-Peer . . . . . | 9         |
| 5.2      | Implementation . . . . .                       | 10        |
| <b>6</b> | <b>CUDA Aware MPI</b>                          | <b>11</b> |
| <b>7</b> | <b>Performance Scaling</b>                     | <b>12</b> |
| 7.1      | Sectioning . . . . .                           | 12        |
| 7.2      | GPU Placement on Nodes . . . . .               | 13        |
| <b>8</b> | <b>Conclusion</b>                              | <b>14</b> |

## 1 Introduction

This report is a continuation of [Lorenz and Olsen, 2018], where everything is introduced and the theory, both mathematical and computer science, is explained. In the end of the first report a number of proposals were made on how to expand the functionality and the performance of the implemented Jacobi method. In this continuation we describe some of these further and expand the program to accommodate the changes.

All of the code is located in the github repository: [https://github.com/MathiasLorenz/Large\\_scale\\_project](https://github.com/MathiasLorenz/Large_scale_project). Where every modification described in this report is present in Version 2.

## 2 Hardware and Software Specifications

All tests were run on the DTU High Performance Computing Cluster [DTU, 2018]. Specifically, the `gpub100` queue was used, which is fitted with 12 Nvidia Tesla V100 GPUs (2 per node) and Intel Xeon Gold 6126 CPUs (2 per node). The specifications for the CPUs can be found below and the GPUs in Figure 2.1. All tests have been run with 10 GPUs (and one core for each GPU) unless otherwise stated.

```

Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):            24
On-line CPU(s) list: 0-23
Thread(s) per core: 1
Core(s) per socket: 12
Socket(s):         2
NUMA node(s):      2
Vendor ID:         GenuineIntel
CPU family:         6
Model:             85
Model name:        Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
Stepping:          4
CPU MHz:           2600.000
BogoMIPS:          5204.91
Virtualization:     VT-x
L1d cache:         32K
L1i cache:         32K
L2 cache:          1024K
L3 cache:          19712K
NUMA node0 CPU(s): 0-11
NUMA node1 CPU(s): 12-23

```

Code has been compiled with CUDA version 9.2 and CUDA aware MPI version 3.1.1, build on gcc 6.4.0. This can be loaded on the DTU cluster with the command `module load CUDA/9.2 mpi/3.1.1-gcc-6.4.0-cuda-9.2-without-mxm`. The Makefile for compiling the project can be inspected in the GitHub repository in the `Poisson` folder.

## SPECIFICATIONS



|                              |   |  |
|------------------------------|---|--|
|                              |  |  |
|                              | <b>Tesla V100<br/>PCIe</b>  | <b>Tesla V100<br/>SXM2</b>   |
| GPU Architecture             | <b>NVIDIA Volta</b>   |  |
| NVIDIA Tensor Cores          | <b>640</b>  |  |
| NVIDIA CUDA® Cores           | <b>5,120</b>  |  |
| Double-Precision Performance | <b>7 TFLOPS</b>   | <b>7.8 TFLOPS</b>  |
| Single-Precision Performance | <b>14 TFLOPS</b>  | <b>15.7 TFLOPS</b>   |
| Tensor Performance           | <b>112 TFLOPS</b>   | <b>125 TFLOPS</b>  |
| GPU Memory                   | <b>32GB /16GB HBM2</b>  |  |
| Memory Bandwidth             | <b>900GB/sec</b>  |  |
| ECC                          | <b>Yes</b>  |  |
| Interconnect Bandwidth       | <b>32GB/sec</b>   | <b>300GB/sec</b>   |
| System Interface             | <b>PCIe Gen3</b>  | <b>NVIDIA NVLink</b>   |
| Form Factor                  | <b>PCIe Full Height/Length</b>  | <b>SXM2</b>  |
| Max Power Consumption        | <b>250 W</b>  | <b>300 W</b>   |
| Thermal Solution             | <b>Passive</b>  |  |
| Compute APIs                 | <b>CUDA, DirectCompute, OpenCL™, OpenACC</b>                                      |  |

Figure 2.1: Specifications for the Nvidia Tesla V100 GPU. Taken from [here](#). Note that the card we have been using is the one in the first column, i.e. the one without Nvidia NVLink.

### 3 CUDA Performance

The report [Lorenz and Olsen \[2018\]](#) contained plenty of performance measurements for all versions of the code. It was concluded that the version `cuda_1` was faster than all other methods (see Sections 5 and 6 of the report). However, after reviewing the code for this version some errors in the kernel calls were discovered. Most significantly the kernels were for large grids launched in an incorrect manner, such that the computations simply were not carried out. A silent error was returned instead making our measurements wrong.

The error occurred for all CUDA versions and was caused by trying to spawn too many threads pr block. As can be seen in<sup>1</sup> only 1024 threads per block can be spawned (regardless of the version of our GPU), but we tried to spawn more than the allowed. However only `cuda_1` solved grid large enough for the error to occur. The mixed versions, by luck, didn't get to reach the error. The problem was caused by mixing up the order of the kernel launch, see Listing 3.1. As our `numBlocks` grew with problem size we eventually ended up trying to spawn too many threads per block (as is specified in the second entry) - ultimately making our program fail.

Listing 3.1: Correct and incorrect kernel launching.

```
1 // Incorrect call (as we had done previously)
2 my_kernel<<<threadsPerBlock, numBlocks>>>();
3
4 // Correct call
5 my_kernel<<<numBlocks, threadsPerBlock>>>();
```

The reason we reported our findings in [Lorenz and Olsen \[2018\]](#) as we did was that our faulty kernel launching does not return an error. Therefore we interpreted the performance as a speed-up when we started to reach the capacity of the GPU such that we streamed efficiently through the data. However, this was not the case, as all the launched kernel simply did not do anything. A contributing problem is also that we were unable to measure the error efficiently for the (very) large grids as we needed more iterations to converge than was tested. Thus we could only rely on the extensive testing we did for smaller grids, which was successful in all cases.

#### 3.1 New Performances

Our new scaling results can be seen in Figure 3.1 and can be compared to Figure 6.1 in [Lorenz and Olsen \[2018\]](#). From the updated figure its very clear to see the benefit of having a single GPU. The single GPU version outperforms all the other methods until the problem reaches a quite large size. The performance increase stagnates close to a teraflop where the mixed version keeps scaling. This is expected since the compute units of the single GPU will be saturated while the mixed suffers from communication delays. For problems just under 512MB we see the critical point for the two methods, where the saturation and the communication have an equal impact on the two methods.

---

<sup>1</sup><https://en.wikipedia.org/wiki/cuda>

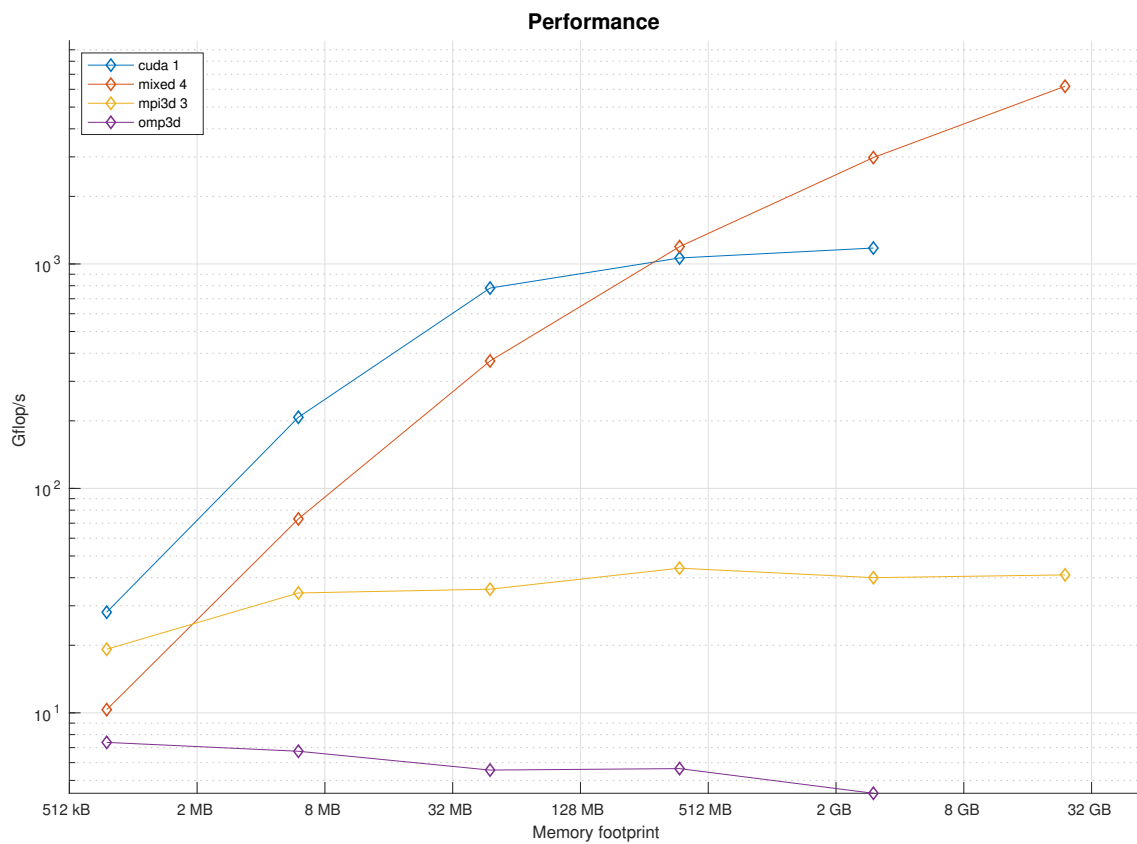


Figure 3.1: New performance plot with fixed CUDA implementation.

## 4 Stopping criterion

In [Lorenz and Olsen, 2018] section 3.4 stopping criteria are discussed. For early stopping at runtime only a CPU version was implemented. In this section we expand on the original work with some more general code and we supply a working GPU version of the stopping criterion using norm difference.

The original program always run a set number of iterations. A more useful way to stop the computations is to implement a way to decide when the result is good enough. Often this is done by examining the relative change between each iteration,

$$D = \|U[n+1] - U[n]\|,$$

where  $n$  represents the iteration number. This is a norm difference between the current and the previous iteration is computed. When this change is smaller than a predefined threshold the computations are stopped since iterating further would only marginally change the result. As the computations are matrix based the Frobenius norm is used. The Frobenius norm is defined by,

$$\|X\|_F = \sqrt{\sum_{i=1}^I \sum_{j=1}^J \sum_{k=1}^K |x_{ijk}|^2},$$

where  $X$  is a 3 dimensional matrix and  $x_{ijk}$  is the element at position  $(i, j, k)$  in the matrix. The program is set up so a stopping criterion can be implemented and controlled through the environment variables `USE_TOLERANCE` and `TOLERANCE` so no structural changes must be made to the program.

Additional elements were added to the information structure to handle the local and global contribution to the Frobenius norm difference as well as a Boolean used to determine whether the stop criterion should be used or not.

### CPU implementation

Implementing the method for the different CPU based methods were simple. The computation is done directly as part of the iteration. Listing 4.1 shows the modification made to the implementations of the Jacobi iteration. The function described in Listing 4.2 shows how the different contributions are collected for MPI through a reduction.

Listing 4.1: Implementation of stop criterion for CPU based code. The implementation is part of the existing Jacobi iteration loop.

```

1 // Loop over all interior points
2 for (int i = 1; i < I - 1; i++) {
3     for (int j = 1; j < J - 1; j++) {
4         for (int k = 1; k < K - 1; k++) {
5             /* Compute the new iteration Unew[ijk] */
6
7             // Tolerance criterion
8             if (information->use_tol)
9             {
10                 double uij      = U[ijk];
11                 double unewij = Unew[ijk];

```



```

12         information->local_frobenius += (uij - unewij)*(uij - unewij);
13     }
14 }
15 }
16 }

```

Listing 4.2: Function used to collect local contributions.

```

1 // Calculate if norm criterion is reached
2 bool norm_early_stop(Information *information)
3 {
4     int size = information->size;
5
6     // Reduce and send global norm diff to all threads
7     if (size > 1)
8         MPI_Allreduce(
9             &information->local_frobenius, &information->frobenius_error,
10            1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
11     else
12         information->frobenius_error = information->local_frobenius;
13
14     information->local_frobenius = 0.0;
15     information->frobenius_error = sqrt(information->frobenius_error);
16
17     // Returning boolean for stopping
18     return (information->frobenius_error < information->tol);
19 }

```

## CUDA implementation

Implementing the computation described above for the CUDA versions of the code was quite different. Since the iteration is done in parallel and not in a loop adding a simple computation to the original iteration was not possible. To circumvent this a new kernel was implemented as shown in Listing 4.3. This new kernel is called through the wrapper also shown in Listing 4.3. The kernel is called with a single block consisting of 1024 threads. These threads run through a loop with a stride of 1024 to distribute the computations to all threads. When a thread is done its added to the existing result with a `atomicAdd`. The `atomicAdd` function is used because it is safe when it comes to data races. The atomic function will add all contributions correctly even though other threads might be accessing the element at the same time.

This implementation is slightly problematic, it is quite inefficient. For a problem of the size 128 the original version of the code spent about 6 seconds to complete. But solving the same problem and computing the relative norm difference each iteration spent 17 seconds which is almost 3 times as much. We have considered the possibility of integrating the computation in the Jacobi iteration like it was done with the CPU versions. However it would require quite extensive work to make sure blocks and threads do not interfere with each other. And some kind of reduction might be needed on the GPU which is quite advanced. The code in Listing 4.3 is a proof-of-concept that early stopping criteria can be implemented on the GPU.

Listing 4.3: Implementation of norm computation on the CUDA device.

```

1 void compute_relative_norm_cuda(
2     Information *information, Information *information_cuda,
3     double *U_cuda, double *Unew_cuda)
4 {
5     int N = 1024;
6     frobenious_kernel<<<1,N>>>(information_cuda,U_cuda,Unew_cuda);
7
8     copy_from_device(
9         &information->local_frobenius, sizeof(double),
10        &information_cuda->local_frobenius
11    );
12 }
13
14 __global__ void frobenious_kernel(
15     Information *information_cuda, double *U_cuda, double *Unew_cuda)
16 {
17     double local_frobenius=0.0;
18     int rank = information_cuda->rank;
19     int loc_Nx = information_cuda->loc_Nx[rank];
20     int loc_Ny = information_cuda->loc_Ny[rank];
21     int loc_Nz = information_cuda->loc_Nz[rank];
22
23     int I, J, K;
24     I = loc_Nz; J = loc_Ny; K = loc_Nx;
25
26     // Loop over all interior points
27     for (int ijk = threadIdx.x; ijk < I*J*K; ijk += blockDim.x) {
28         //int ijk = IND_3D(i, j, k, I, J, K);
29         double uij = U_cuda[ijk];
30         double unewij = Unew_cuda[ijk];
31         local_frobenius += (uij - unewij)*(uij - unewij);
32     }
33
34     atomicAdd(&information_cuda->local_frobenius,local_frobenius);
35 }

```

## 5 CUDA Using Peer-to-Peer Communication

For all work done in the previous report [Lorenz and Olsen, 2018] we used MPI for communicating between threads and GPUs. As our compiled MPI version is not CUDA aware, we had to transfer arrays from the GPU -> CPU -> MPI message to other CPU -> GPU, which is both expensive and cumbersome. But for communicating directly between GPUs on the same node a more efficient alternative exists as they have peer-to-peer access. This allows the GPUs to send data directly between without using the CPU. This section is about our implementation of direct peer-to-peer access between GPUs.

### 5.1 Accessing Devices using Peer-to-Peer

With `cudaDeviceCanAccessPeer` you can query for peer-to-peer access between devices. If this is possible, the peer-to-peer can be setup with `cudaDeviceEnablePeerAccess` for both devices. We made a small wrapper (Listing 5.1) that sets up the peer-to-peer access and ensures the current device is reset after the setup. Similarly we made a wrapper for synchronizing the two devices as can be seen in Listing 5.2.

Listing 5.1: Implementation of enabling peer access between two GPUs on the same node. Note that the second input in `cudaDeviceEnablePeerAccess` is a flag for future use that has to be 0.

```
1 // Enable peer access between GPU 0 and 1. Restores currently chosen GPU
2 void cuda_enable_peer_access()
3 {
4     // Get the current device, s.t. it can be set afterwards.
5     int current_device;
6     cudaGetDevice(&current_device);
7
8     // Enable peer access.
9     cudaSetDevice(0);
10    checkCudaErrors(cudaDeviceEnablePeerAccess(1, 0));
11    cudaSetDevice(1);
12    checkCudaErrors(cudaDeviceEnablePeerAccess(0, 0));
13
14    // Set current device back
15    cudaSetDevice(current_device);
16 }
```

Listing 5.2: Implementation of device synchronizing on the same node.

```
1 // Synchronize two GPUs on the same node
2 void cuda_peer_device_sync()
3 {
4     // Get the current device, s.t. it can be set afterwards.
5     int current_device;
6     cudaGetDevice(&current_device);
7
8     // Synchronize
9     cudaSetDevice(0);
10    checkCudaErrors(cudaDeviceSynchronize());
11    cudaSetDevice(1);
12    checkCudaErrors(cudaDeviceSynchronize());
13 }
```

```
13
14 // Set current device back
15 cudaSetDevice(current_device);
16 }
```

## 5.2 Implementation

Implementing this turned out to be non-trivial to say the least. It turned out the code got pretty hard to write, as we now had to have both splits onto nodes (as has been done until now) and on the node itself (to use the two GPUs). This was cumbersome and error prone and we ultimately did not succeed in creating a working version. We kept running into strange and hard to debug segmentation faults. The method discussed in the next section with CUDA aware MPI is should, in our opinion, be the most optimal solution to our problem. With that we get the optimal bandwidth utilisation on the node and between nodes together with the most streamlined code.

The function is called `jacobi_cuda_2` and the (still not properly working) code is located in `Poisson/src/jacobi/`.

## 6 CUDA Aware MPI

In [Lorenz and Olsen, 2018, Section 5.3] an implementation using the CUDA-Aware MPI installation was described. However this version of MPI was not installed on the test server so the implementation had not been tested. These tests have now been performed since the required installation of MPI is now available. CUDA aware MPI was added in OpenMPI v1.7.0, but our modules have not been compiled with this option until now.

We tested the code extensively on small problems on the interactive node without any issues. But the GPU queue has been under a large load over the last couple of weeks and due to the nature of our program we require a large amount of resources to test the program. These resources have not been available for the majority of the time. The test commenced once but even though extensive tests had been made on the interactive queue, some bug caused the program to hang and it did not complete within the 24 hour allowance. Due to the load on the system we have been unable to run a new test in order to attempt debugging the code.

## 7 Performance Scaling

### 7.1 Sectioning

In order to analyse where in our code time is spent and to measure the performance gain of overlapping the computations and communication, we decided to run some tests on different sections of the code. In order to measure the time spent on communication and computation separately we used the latest version of the code that did not run those two sections in parallel which is Mixed 2. The parallel performance is measured in Mixed 5 since this is the version that have the biggest overlap between communication and computation.

Unfortunately the cluster have been unavailable for a while because of long queues so we had to run on the interactive server `gpuv100i`. This is a problem since the resources are shared between all users on the server and at the time of running the tests 2 other persons were using the system. An attempt to make up for the inconsistencies a shared system gives we tested each method 5 times for all the test problems and we then use the average of the measurements.

| N             | 32   |      |      |      |      | Average |
|---------------|------|------|------|------|------|---------|
| Communication | 0.44 | 0.35 | 0.29 | 0.31 | 0.31 | 0.34    |
| Computation   | 0.17 | 0.16 | 0.12 | 0.12 | 0.11 | 0.14    |
| Parallel      | 0.88 | 0.66 | 0.69 | 0.63 | 0.60 | 0.69    |
| N             | 64   |      |      |      |      | Average |
| Communication | 0.48 | 0.57 | 0.57 | 0.50 | 0.47 | 0.52    |
| Computation   | 0.15 | 0.23 | 0.21 | 0.14 | 0.14 | 0.17    |
| Parallel      | 1.03 | 0.70 | 0.76 | 0.62 | 0.71 | 0.76    |
| N             | 128  |      |      |      |      | Average |
| Communication | 1.25 | 1.19 | 1.34 | 1.26 | 1.25 | 1.26    |
| Computation   | 0.44 | 0.44 | 0.53 | 0.43 | 0.43 | 0.45    |
| Parallel      | 0.87 | 0.88 | 0.88 | 0.91 | 0.88 | 0.88    |
| N             | 256  |      |      |      |      | Average |
| Communication | 4.26 | 3.94 | 4.26 | 4.33 | 4.33 | 4.22    |
| Computation   | 3.56 | 3.71 | 3.57 | 3.21 | 3.21 | 3.45    |
| Parallel      | 4.31 | 4.32 | 4.24 | 4.30 | 4.33 | 4.30    |

Table 1: Raw and average data for the analysis of time spent calculating and how much we could overlap.

Table 1 shows the measurements made on the program for different sized problems. Here it is easy to see the benefit of running the communication and computation in parallel. All problem sizes are solved at the same speed or faster for the parallel code except for the smallest tested problem. This might have to do with additional overhead included for the CUDA-aware MPI we are using for Mixed 5. For the bigger problem sizes it is clear that all computations are hidden under the communication and our program is as fast as the slowest of the two parts. As the problem size grow we will expect the computations to dominate the computational time needed. But the communication should still be hidden

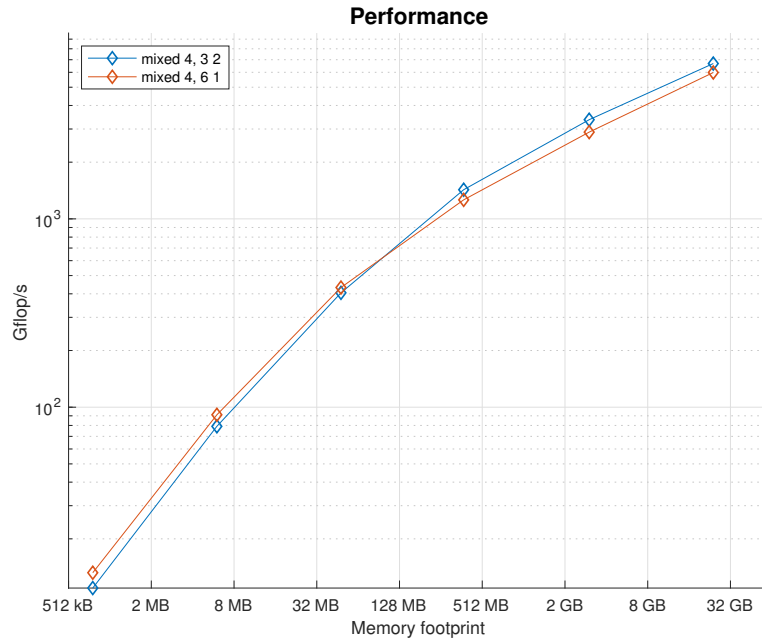


Figure 7.1: Comparing runs with 6 GPUs placed on 6 nodes (one GPU each) or 3 nodes (2 GPUs each). Is done with version `mixed_4`.

under the computations.

## 7.2 GPU Placement on Nodes

Here we experiment with how GPU placement on the nodes affect performance. We have nodes with two GPUs on each, but the question is whether we gain performance by using both GPUs on the node or just one GPU per node. The experiment is shown in Figure 7.1. Note that the experiment is done with version `mixed_4`.

Interestingly, the version with 3 nodes and 2 GPUs on each is faster for small problems and slower for small problems. The difference is not large but still significant. We do not have an answer for this behavior, but it is interesting and could be explored in more detail.

## 8 Conclusion

In this report we fixed an important error in the CUDA code such that the performance plot (Figure 3.1) is generated correctly. Furthermore a relative error stopping criterion on the GPU and CUDA aware MPI was successfully implemented. As described we had some trouble getting `mixed_5` (with CUDA aware MPI) to run properly on `gpub100` and that we basically could not test it because the queue was constantly filled with jobs by `kavsole`. The version `mixed_5` did however pass all our tests for small grids and we successfully showed that (for smaller grids) we could hide most of the time spent sending arrays.

We spent a good amount of time on CUDA peer-to-peer, but were ultimately unable to make a properly working version. The code also got significantly more bloated, thus we concluded that CUDA aware MPI is the better choice, if MPI is already included in the project.



## References

DTU. Dtu computing center webpage, 2018. Last checked: 16-08-2018,  
url: <https://www.hpc.dtu.dk/>.

Mathias Lorenz and Tim F. Olsen. Large-scale computations on modern gpus. University course at Technical University of Denmark, See [https://github.com/MathiasLorenz/Large\\_scale\\_project/blob/master/Large\\_Scale\\_Project\\_Report.pdf](https://github.com/MathiasLorenz/Large_scale_project/blob/master/Large_Scale_Project_Report.pdf) for the report, 2018.