DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

02614 - High Performance Computing

3. GPU Matrix Multiplication and GPU Poisson Problem

Authors:
Maria Garofalaki - s202378
Jorge Sintes Fernández - s202581
Klara Maria Ute Wesselkamp - s202382

Course responsible:
Bernd Dammann

January 22, 2021

Individual responsibilities

All the parts in this assignment were solved equally between the three group members.

Part I. GPU Matrix Multiplication

Our purpose in this part will be to implement a matrix multiplication algorithm that runs on the GPU. As in previous assignments, we will start really simple, and make slight modifications on our code step by step, doing some performance analysis on every iteration, in order to find a better tuned version of the original algorithm.

Exercise 1. GPU1 and GPU2

We'll start by implementing a single threaded version of the simpler Matrix Multiplication Algorithm we did for assignment one. The code is as follows:

```
extern "C" {

  #include <stdio.h>
  #include <omp.h>

  __global__  void
  matmultgpu1(int m, int n, int k, double *A, double *B, double *C) {

    int i1, i2, i3;

    for (i1 = 0; i1 < m; i1++) {
      for (i2 = 0; i2 < n; i2++) {
        C[i1 * n + i2] = 0;
        for (i3 = 0; i3 < k; i3++) {
          C[i1 * n + i2] += A[i1 * k + i3] * B[i3 * n + i2];
        }
      }
    }

  }

  void matmult_gpu1(int m, int n, int k, double *A, double *B, double *C) {

    double * d_A, * d_B, * d_C;

    int sizeA = m * k * sizeof(double);
    int sizeB = k * n * sizeof(double);
    int sizeC = m * n * sizeof(double);

    double time1, time2, elapsed;

    //Alloc memory on the device
    cudaMalloc((void **) & d_A, sizeA);
    cudaMalloc((void **) & d_B, sizeB);
    cudaMalloc((void **) & d_C, sizeC);

    time1 = omp_get_wtime();

    cudaMemcpy(d_A, A, sizeA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeB, cudaMemcpyHostToDevice);

    time2 = omp_get_wtime();

    matmultgpu1<<<1, 1>>> (m, n, k, d_A, d_B, d_C);
    cudaDeviceSynchronize();

    elapsed = omp_get_wtime() - time2;
    printf("Kernel time: %f\n", elapsed);

    cudaMemcpy(C, d_C, sizeC, cudaMemcpyDeviceToHost);
```

```
51
52      elapsed = omp_get_wtime() - time1;
53      printf("Kernel+copy time: %f\n", elapsed);
54
55      cudaFree(d_A);
56      cudaFree(d_B);
57      cudaFree(d_C);
58
59    }
60  }
```

Listing 1: GPU single threaded version

The way the function works is pretty straight forward, we copy the matrices `A` and `B` into the GPU and then call a kernel with only one block and 1 thread per block. We also include some OpenMP timers to be able to time the kernel and the copies of the matrices. We then will make use of the executable `matmult_f.nvcc` provided to analyze the performance of the algorithm. Also, as suggested in the assignment, we will compare this GPU version with the best CPU implementation we got in last assignment, that is, the `CBLAS DGEMM` function. To do the tests, we'll use the following bash script:

```
1  #!/bin/bash
2
3  #BSUB -J time_it
4  #BSUB -o time_it_%J.out
5  #BSUB -q hpcintrogpu
6  #BSUB -gpu "num=1"
7  ###:mode=exclusive_process"
8  #BSUB -W 15
9  #BSUB -R "rusage[mem=2048]"
10 #BSUB -N
11
12 # define the driver name to use
13 EXECUTABLE="matmult_f.nvcc"
14
15 # define the size values in the SIZE variable
16 SIZES="100 150 200 250 500 750 1000" #1500 2000 3000 4000 6000"
17
18 # define the function type in FUNC
19 FUNC="lib gpu1"
20 OUT="times_1.out"
21
22 # enable(1)/disable(0) result checking
23 export MATMULT_COMPARE=1
24 export MFLOPS_MAX_IT=5
25 export MKL_NUM_THREADS=1
26
27 rm -f ./$OUT
28
29
30 # start the collect command with the above settings
31 for F in $FUNC
32 do
33     for S in $SIZES
34     do
35         echo "Library: $F. Size: $S." >> ./$OUT
36         ./$EXECUTABLE $F $S $S $S $BLKSIZE >> ./$OUT
37         echo "" >> ./$OUT
38     done
39 done
40
41
42
```

```
for F in $FUNC
do
    rm -f ./$F.out
    grep "matmult_$F" $OUT >> ./$F.out
done
```

Listing 2: Bash script to test functions

We can see the following results in performance:

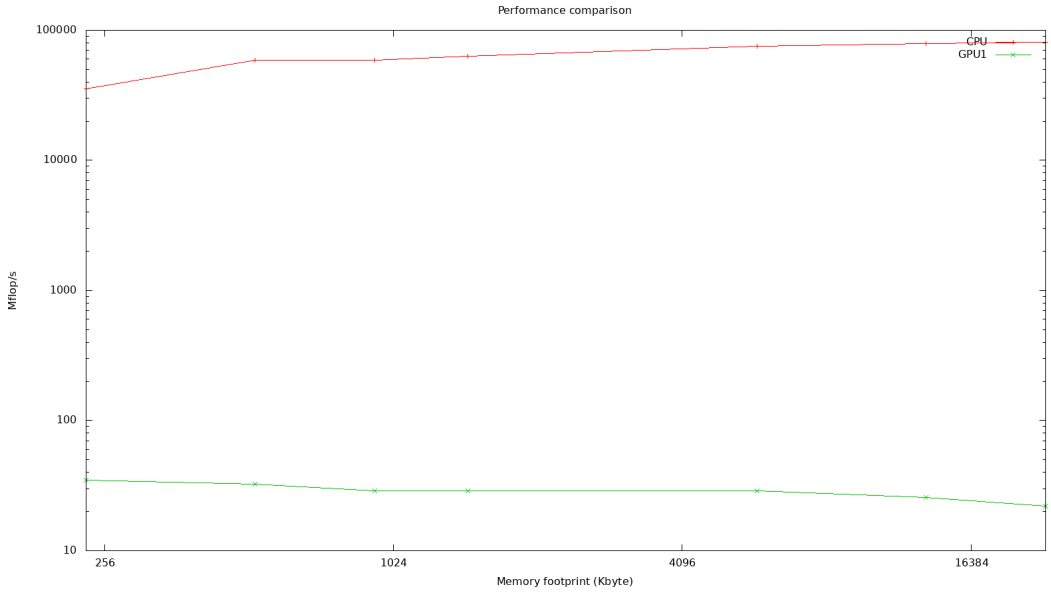

Figure 1: GPU1 vs. CPU

	100		250		500		750		1000	
	K	K+C	K	K+C	K	K+C	K	K+C	K	K+C
CPU	0.000042		0.0000452		0.003139		0.01029		0.024228	
GPU1	0.054362	0.054416	1.078200	1.078370	8.652178	8.652703	33.071155	33.072278	90.869593	90.871559

Table 1: Times in seconds for GPU1 vs. CPU

The timing we did of the functions is shown in the table for different matrix sizes. For GPU1, we state the time it takes only computing the kernel (K), and the same plus the time it takes to copy the matrices from the host to the device and back (K+C). It can be seen that, as expected, this version behaves pretty badly when comparing it to the CPU version. This is due to the fact we're running all the calculations in a single GPU core one after another.

Also, in the table, we observe the values for K and K+C within each matrix size are pretty similar: in other words, most of the time is spent in the calculation within the kernel. This is the sign of a computing bound problem. We'll be able to see this more in depth when we use the profiler.

The natural next step will be to calculate each element of matrix C with a different core. We'll call this function `gpu2` :

```
__global__ void
matmultgpu2(int m, int n, int k, double *A, double *B, double *C) {

  double Cvalue = 0.0;

  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  int e;

  if (row < m && col < n) {
    for (e = 0; e < k; ++e)
      Cvalue += A[row * k + e] * B[e * n + col];

    C[row * n + col] = Cvalue;
  }

}

// Declare the number of threads (INSIDE HOST FUNCTION)
dim3 numOfThreadsPerBlock;
numOfThreadsPerBlock.x = BLOCK_SIZE;
numOfThreadsPerBlock.y = BLOCK_SIZE;

dim3 numOfBlocks;
numOfBlocks.x = (n + numOfThreadsPerBlock.x - 1) / (numOfThreadsPerBlock.x);
numOfBlocks.y = (m + numOfThreadsPerBlock.x - 1) / (numOfThreadsPerBlock.y);
```

Listing 3: matmult_gpu2

To keep it simple we'll, from now on, just include the kernel and the definition of the thread blocks used to call it. Notice that if the number elements is not divisible by the `BLOCK_SIZE`, we take one extra block. For this reason, we need to have an `if` clause in the kernel that makes sure no thread outside the matrix tries to access it therefore causing a segmentation fault.

It's also worth noticing that every test will be done with a `BLOCK_SIZE` of 16. Let's see the results for this algorithm:

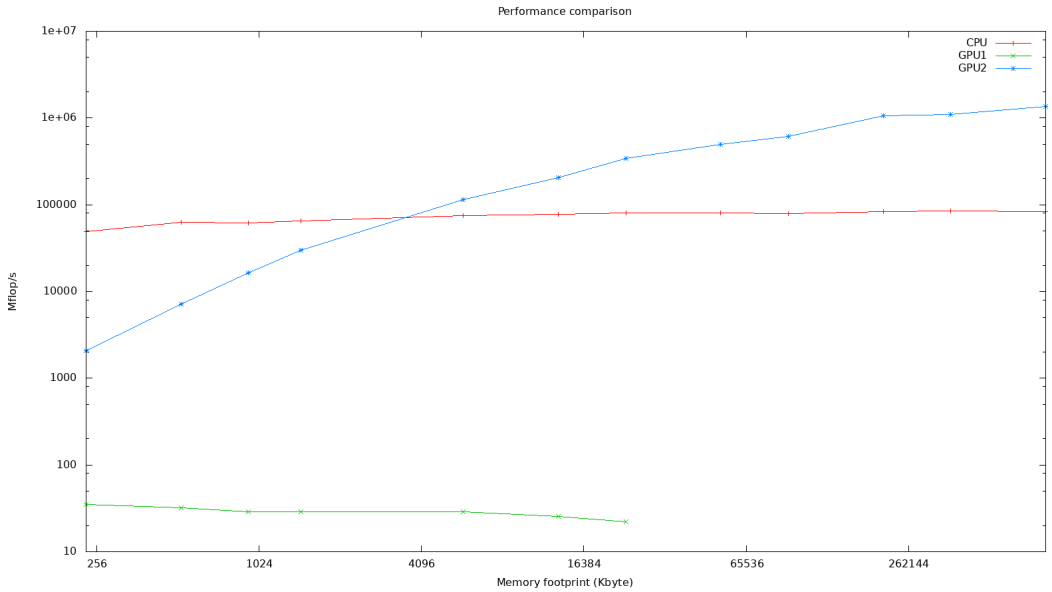

Figure 2: GPU2 vs. CPU

By computing every element with a single thread the performance improvement is huge! For small matrices the CBLAS function still behaves better. However, when we get to around 4000 kB in memory footprint, which corre-

	1000		2000		3000		4000		6000	
	K	K+C	K	K+C	K	K+C	K	K+C	K	K+C
CPU	0.024420		0.200668		0.647929		1.509835		5.132344	
GPU2	0.001646	0.003594	0.012856	0.020524	0.024826	0.043650	0.063494	0.101227	0.213298	0.296599
Speed-Up	14.84×	6.71×	15.61×	9.78×	26.01×	14.84×	23,78×	14.92×	24.07×	17.30×

Table 2: Speed-up GPU2 vs CPU

sponds to a matrix size of around 350∼400 elements, `gpu2` starts outperforming the CBLAS function. **Important:** The CPU version we use in this comparison and further ones is single threaded. We measured the performance changing the number of threads available with the environmental variable `MKL_NUM_THREADS` and it only made it worse. The single threaded version was the one with better performance for us, which doesn't make much sense. However, as it was the one with better behaviour, we chose this ones to be as fair as possible.

Taking a look at table 2, we can see that now the times for K and K+C are pretty different. Around 1/3 of the time is spent on passing the matrices from CPU to GPU and the other way around. We no longer have such an exaggerated computing problem. We will therefore take `gpu2` as the naive version (or baseline) we'll try to improve in the following exercises.

Exercise 2. GPU3 and GPU4

We'll try to improve our naive kernel by making each thread compute more than one element of C. That way, in some direction of the blocks we will need half as many threads. From the lectures, we already now that the optimal way of calculating the elements will be row-wise as in figure , to have a coalesced memory access. We will implement both versions to observe this behaviour in our implementation:

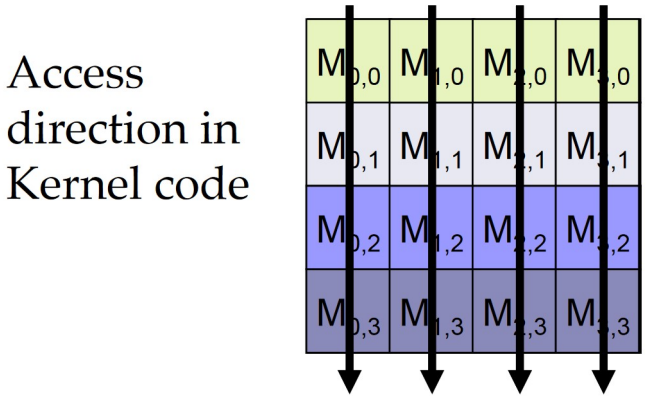

Figure 3: Row-wise access direction of a matrix

```
__global__ void
matmultgpu3_colwise(int m, int n, int k, double * A, double * B, double * C) {
  // Bad one
  double Cvalue1 = 0.0,
         Cvalue2 = 0.0;

  int col = 2 * (blockIdx.x * blockDim.x + threadIdx.x);
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  int e;

  if ((row < m) && (col < (n - 1))) {
    for (e = 0; e < k; ++e) {
      Cvalue1 += A[row * k + e] * B[e * n + col];
```

```
      Cvalue2 += A[row * k + e] * B[e * n + col + 1];
    }

    C[row * n + col] = Cvalue1;
    C[row * n + col + 1] = Cvalue2;
  }
  else if ((row < m) && (col == (n - 1))) {
    for (e = 0; e < k; ++e)
      Cvalue1 += A[row * k + e] * B[e * n + col];

    C[row * n + col] = Cvalue1;
  }

}

__global__ void
matmultgpu3_rowwise(int m, int n, int k, double * A, double * B, double * C) {
  // This is the good one!!!

  double Cvalue1 = 0.0,
         Cvalue2 = 0.0;

  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = 2 * (blockIdx.y * blockDim.y + threadIdx.y);

  int e;

  if ((row < m - 1) && (col < n)) {
    for (e = 0; e < k; ++e) {
      Cvalue1 += A[row * k + e] * B[e * n + col];
      Cvalue2 += A[(row + 1) * k + e] * B[e * n + col];
    }

    C[row * n + col] = Cvalue1;
    C[(row + 1) * n + col] = Cvalue2;
  }
  else if ((row == m - 1) && (col < n)) {
    for (e = 0; e < k; ++e)
      Cvalue1 += A[row * k + e] * B[e * n + col];

    C[row * n + col] = Cvalue1;
  }
}

  // Declare the number of threads
  dim3 numOfThreadsPerBlock;
  numOfThreadsPerBlock.x = BLOCK_SIZE;
  numOfThreadsPerBlock.y = BLOCK_SIZE;

  // Initializing for colwise
  // blocky = (n+numOfThreadsPerBlock.x-1)/(numOfThreadsPerBlock.x);
  // dim3 numOfBlocks;
  // numOfBlocks.x = (blocky+1)/2;
  // numOfBlocks.y = (m+numOfThreadsPerBlock.y-1)/(numOfThreadsPerBlock.y);

  // Initializing for rowwise
  blocky = (m + numOfThreadsPerBlock.y - 1) / (numOfThreadsPerBlock.y);
  dim3 numOfBlocks;
  numOfBlocks.x = (n + numOfThreadsPerBlock.x - 1) / (numOfThreadsPerBlock.x);
  numOfBlocks.y = (blocky + 1) / 2;
```

Listing 4: Both versions of matmult_gpu3

Notice the way we are declaring the blocks. We take half as many as needed in the direction we are taking the element. As we are taking two elements at a time we need to include an `if` and `else if` clause inside the kernel, to avoid segmentation faults. The results are as follows:

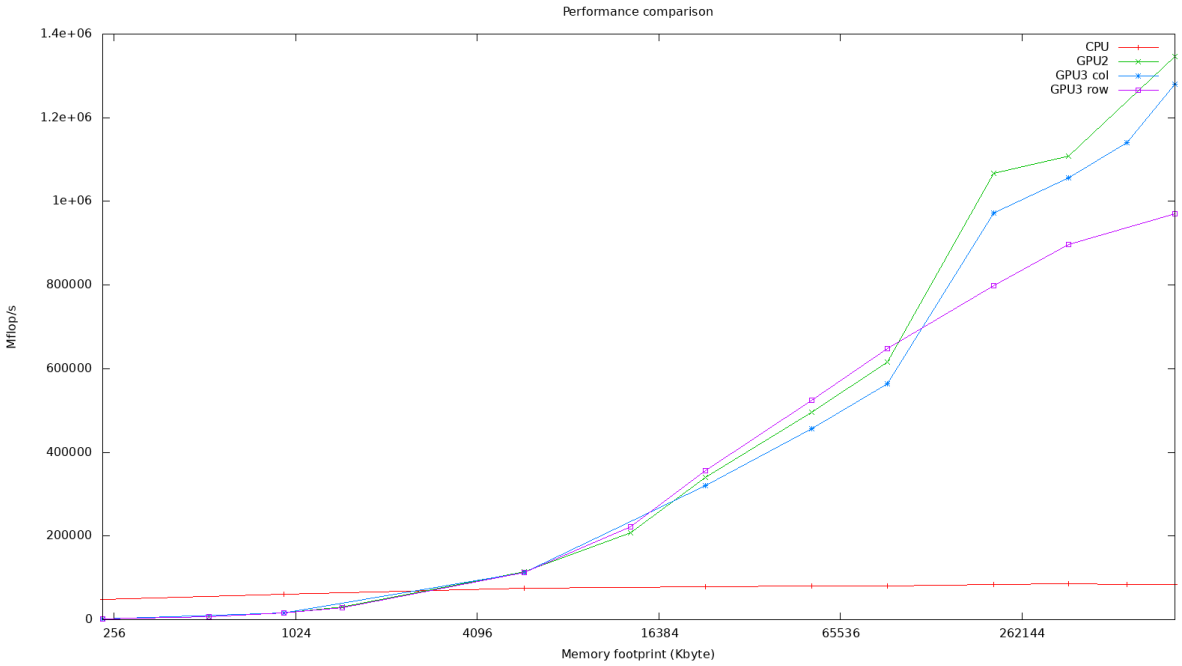

Figure 4: GPU3 vs. CPU

We see that the behaviour is pretty similar for the three algorithms at first. Weirdly, at some point, the col-wise version out-performances the row-wise version at some point. The difference is pretty small and could be caused by the fact that the `hpcintrogpu` queue was really busy at the time we made this tests. For big matrix sizes it's indisputable that the row-wise version stands better than the other. However, they do not manage to outperform the original version.

For `matmult_gpu4` we'll modify `matmult_gpu3` to make each thread calculate more than 2 elements at a time. In fact, we'll make a different function for 3, 4, 5 and 6 elements. For the sake of simplicity we won't include the codes in this pdf, as they are pretty similar to the 2 element version. They only differ in the amount of `else if` clauses needed inside the kernel and in the declaration of the number of blocks. The results are shown in figure 5.

Again, we see this weird result that none of the implementations behave better than `gpu2`. At the point when we were writing this comments of the results we decided to re-run `gpu2` several times and we observed that the behaviour we observe in the plot is really optimistic. Apparently, the queue system was less loaded when we took its results and the performance for big matrices shouldn't be taken too seriously.

However, we can see that the behaviour for all implementations is similar at first, but they diverge for large memory footprints. It clearly stands out the more elements we calculate at a time, the better the performance in this region (although there's a little fall in performance from 3 to 4 elements). The reason this increase is happening is because we access the matrix in a coalesced way and we take advantage of the registers. When compiling, we can see that the more elements we add to the calculation, the more registers are being used at these kernels.

This increase will reach a maximum and then start falling down if we keep increasing the number of elements. Sadly, we didn't reach a number high enough to observe this behaviour. We made a function `matmult_gpu_4_7` but due to lack of time we couldn't implement it in the analysis. In table 3 we show the timings for the two fastest functions.

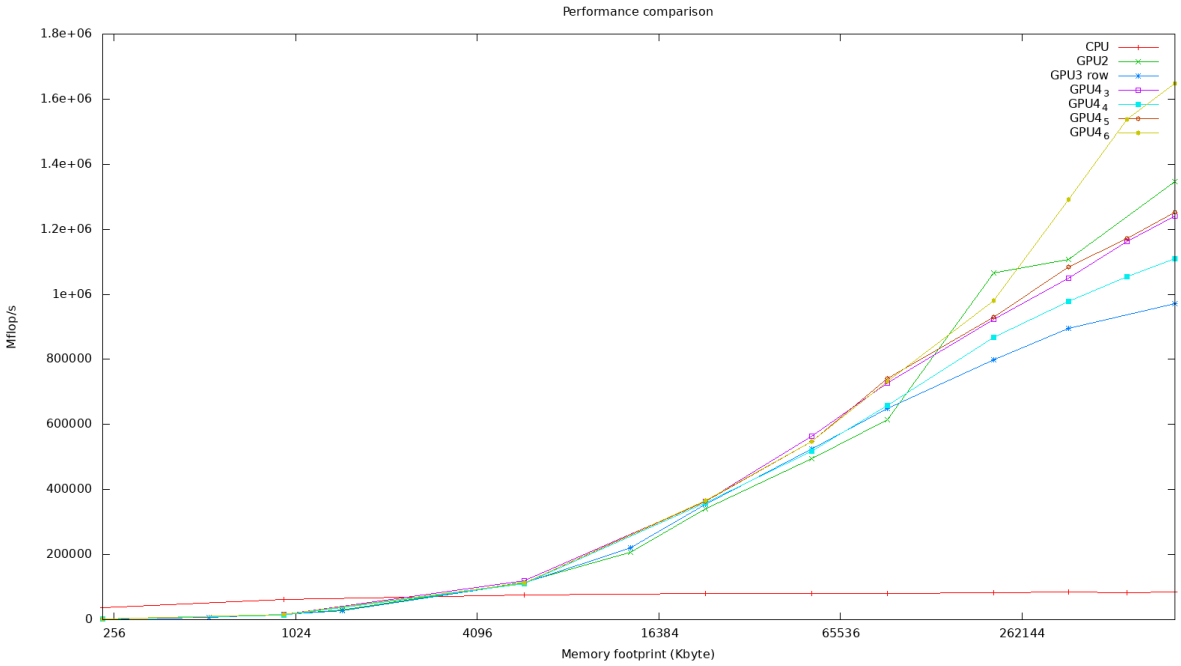

Figure 5: GPU4 vs. CPU

	1000		2000		3000		4000		6000	
	K	K+C	K	K+C	K	K+C	K	K+C	K	K+C
CPU	0.024420		0.200668		0.647929		1.509835		5.132344	
GPU3_row	0.001359	0.003364	0.011709	0.019370	0.040232	0.062496	0.095328	0.136882	0.340657	0.442785
Speed-Up	17.97×	7.26×	17.14×	10.36×	16.10×	10.37×	15.84×	11.03×	15.07×	11.59×
GPU4_6	0.001313	0.003263	0.009005	0.016691	0.031086	0.050423	0.074143	0.116056	0.155179	0.237939
Speed-Up	18.60×	7.48×	22.28×	12.02×	20.90×	12.86×	20.36×	13.01×	33.07×	21.57×

Table 3: Speed up of 2 elements and 6 elements

Exercise 3.

For the next implementation, we will adapt an example found in NVIDIA's "CUDA Programming guide" that uses shared memory for reading subblocks of `A` and `B` matrices in order to improve the performance. Their code uses structure and function definitions we won't implement, we will just use a direct way of indexing. Also, for this code, we will assume that `m`, `n` and `k` are multiples of the `BLOCK_SIZE`. The results of the analysis are shown in 6.

```
__global__ void
matmultgpu5(int m, int n, int k, double *A, double *B, double *C) {
  int blockRow = blockIdx.y,
    blockCol = blockIdx.x,
    row = threadIdx.y,
    col = threadIdx.x;
  int i, j, Asrow, Ascol, Bsrow, Bscol, Crow, Ccol;

  double Cvalue = 0.0;

  for (i = 0; i < (k / BLOCK_SIZE); i++) {

    __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];

    Asrow = blockRow * BLOCK_SIZE + row;
    Ascol = BLOCK_SIZE * i + col;

    As[row][col] = A[Asrow * k + Ascol];

    Bsrow = BLOCK_SIZE * i + row;
    Bscol = blockCol * BLOCK_SIZE + col;

    Bs[row][col] = B[Bsrow * n + Bscol];

    __syncthreads();

    for (j = 0; j < BLOCK_SIZE; j++) {
      Cvalue += As[row][j] * Bs[j][col];
    }
    __syncthreads();
  }

  Crow = blockRow * BLOCK_SIZE + row;
  Ccol = blockCol * BLOCK_SIZE + col;

  C[Crow * n + Ccol] = Cvalue;

}
```

Listing 5: matmult_gpu5

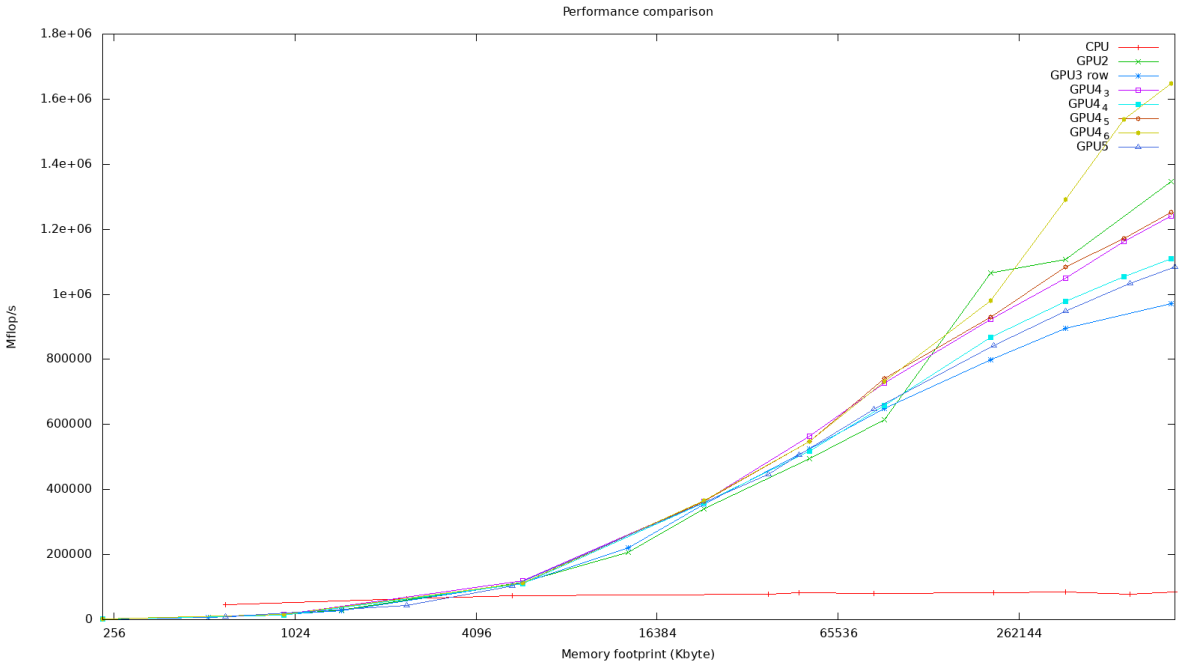

Figure 6: GPU5 vs. CPU

Just as before, we ran another test at the moment of writing this notes and observed that the values shown for `gpu_2` are pretty optimistic. In our later runs it's actually the worst of them all and this weird performance improvement in large memory footprints must be taken as an outlier. The reason of this outlier is the massive work load the queue `hpcintrogpu` was getting at the time of these analysis. Sadly, due to lack of time, we've not been able to reproduce again the experiments.

Addressing the results for `gpu_5` , it performs better than the naive version, and the 2 elements one (`gpu_3`), due to the use of shared memory. However, all implementations that take more than 2 elements outperform it, specially the 6 elements one.

	1280		1920		3040		4000		6080	
	K	K+C	K	K+C	K	K+C	K	K+C	K	K+C
CPU	0.052827		0.173276		0.687181		1.518741		5.311556	
GPU5	0.002939	0.006103	0.009772	0.016834	0.038537	0.056492	0.087443	0.128921	0.306797	0.377730
Speed-Up	17.9745×	8.6559×	17.7319×	10.2932×	17.8317×	12.1642×	17.3684×	11.7804×	17.3129×	14.06178×

Table 4: Speed up for gpu5

As we can see in the table 4 comparing speed-ups for different Matrix-sizes, we have an incredibly high speed up. In this case, incredibly can be taken literally, because this speaks probably more against our CPU-version than for our GPU implementation. An improvement in the latter would surely bring up the efficiency and down the speed-up, to a more realistic number in a range up to 10 or 11.

Exercise 4.

For the last exercise, we will compare the DGEMM function in the CUBLAS library with the CBLAS function. The final results are plotted in figure 7.

```
void matmult_gpulib(int m, int n, int k, double *A, double *B, double *C) {
  cublasStatus_t stat;
  cublasHandle_t handle;
  const double alpha = 1.0,
    beta = 0.0;
  double *d_A, *d_B, *d_C;

  double time1, time2, elapsed;

  stat = cublasCreate( & handle);
  if (stat != CUBLAS_STATUS_SUCCESS) {
    printf("CUBLAS initialization failed\n");
  }

  int sizeA = m * k * sizeof(double);
  int sizeB = k * n * sizeof(double);
  int sizeC = m * n * sizeof(double);

  // Allocate memory on the device
  cudaMalloc((void ** ) & d_A, sizeA);
  cudaMalloc((void ** ) & d_B, sizeB);
  cudaMalloc((void ** ) & d_C, sizeC);

  time1 = omp_get_wtime();

  // Copy the values over
  cudaMemcpy(d_A, A, sizeA, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, sizeB, cudaMemcpyHostToDevice);

  time2 = omp_get_wtime();

  /*
  CUBLAS only reads matrices in column major form, and we hace A, B and C stored in row
    major.
  This will make CUBLAS understand our matrix as the transpose version if we input them
    normally.
  However, we want the output of C to be in rowmajor form too, so we need CUBLAS to
    calculate C^T.
  C^T = (AB)^T = B^T * A^T. Good news! We just need to swap the matrices out in the
    arguments to make
  CUBLAS output our C matrix in rowmajor form!
  */

  cublasOperation_t transa = CUBLAS_OP_N;
  cublasOperation_t transb = CUBLAS_OP_N;

  cublasDgemm(handle, transa, transb, n, m, k, & alpha, d_B, n, d_A, k, & beta, d_C, n);
  cudaDeviceSynchronize();

  elapsed = omp_get_wtime() - time2;
  printf("Kernel time: %f\n", elapsed);

  cudaMemcpy(C, d_C, sizeC, cudaMemcpyDeviceToHost);

  elapsed = omp_get_wtime() - time1;
  printf("Kernel+copy time: %f\n", elapsed);

```

```
54    cudaFree(d_A);
55    cudaFree(d_B);
56    cudaFree(d_C);
57
58 }
59 }
```

Listing 6: matmult_gpulib

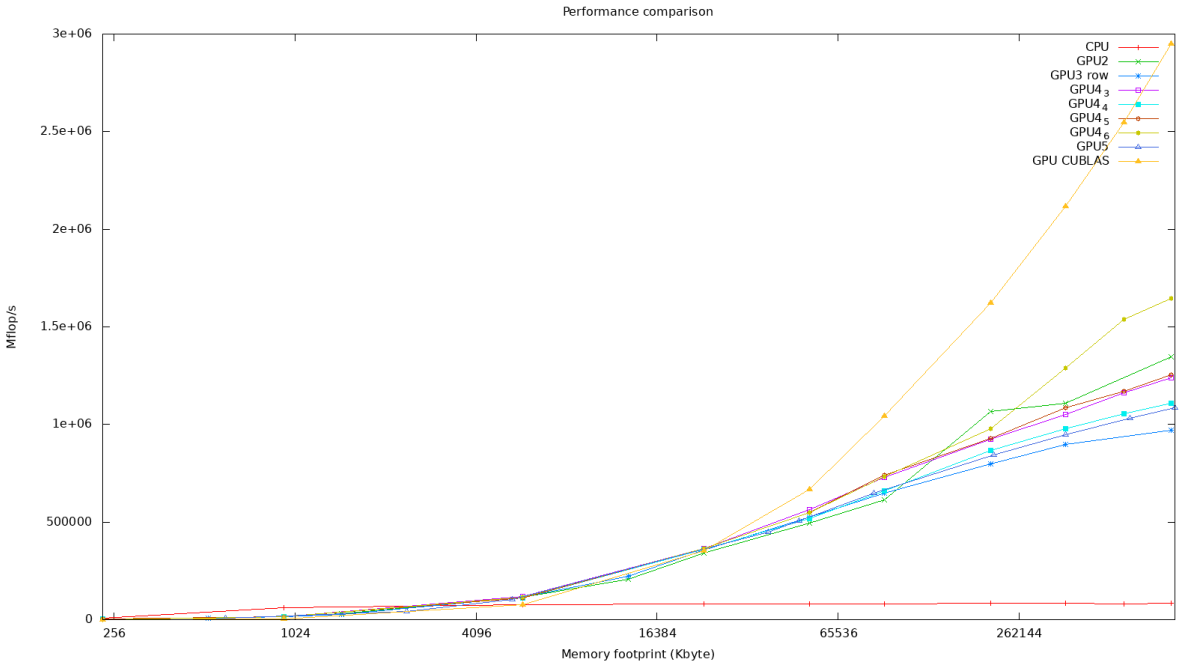

Figure 7: GPU6 vs. CPU

As expected, the CUBLAS implementation outperforms all our previous attempts. However, this only happens at greater matrix sizes. It also outperforms the CPU version but, as stated in exercise 2, we didn't use a multithreaded version of CBLAS. Somehow the single threaded version was the one that worked better for us, maybe because we didn't implement it correctly. Adding that to the fact the queue was really unstable, makes the results shown in our plots not as clear as we would like them to be.

	1000		2000		3000		4000		6000	
	K	K+C	K	K+C	K	K+C	K	K+C	K	K+C
CPU	0.024420		0.200668		0.647929		1.509835		5.132344	
CUBLAS	0.000273	0.002232	0.001763	0.009443	0.005480	0.023550	0.012556	0.053077	0.041486	0.141659
Speed-Up	89.4505×	10.9408×	113.8219×	21.2505×	118.2352×	27.5129×	120.2480×	28.446×	123.7126×	36.2303×

Table 5: Speed up for CUBLAS

Taking a look at table 5, we can see that now the times for K and K+C a big difference. So plenty of time is spend on passing the matrices from CPU to GPU and the other way around. The speed-up value is really increased but that was an expected behaviour as matmult gpulib has the best performance from the other kernels. In the next part, we will analyze the codes with the profiler.

Part II. Use the Profiler tool for GPU Matrix multiplication

We used **nvvp** profiler commant line interface to analyze our kernels in order to understand the limits in their performance. For that reason we used the profiler for all our kernels for the same matrix dimensions (mxnxk 2048x2048x2048) . One of the most important fields of our analysis are: the SM[%] and Memory[%] which indicate the compute-bound and memory-bound accordingly.

A kernel is memory-bound if the most of kernel time is spent in executing memory instructions.On the other hand a **SM utilization**(streaming multiprocessors) indicates whether the kernel is compute bound.

Table for analyzing the different Kernels for GPU Matrix multiplication for dimensions 2048x2048x2048

.	gpu2	gpu3	gpu4_3	gpu4_4	gpu4_5	gpu4_6	gpu5	gpulib
SM[%]	47.04	38.03	43.91	34.69	36.31	48.24	36.40	92.25
Memory[%]	94.01	76.04	87.73	69.34	72.21	97.84	72.27	59.15

Where versions gpu4_3 gpu4_4 gpu4_5 gpu4_6 are all gpu4 , and each thread computes¿ 2 elements of C. (3,4,5 and 6 elements accordingly)

Some important observations we can make from the above table:

- As we can observe from the above table the percentage of memory bound is really high for almost all our kernels. However **gpu5** has a relatively low Memory percentage around 72.27% . That was an expected behaviour as in that version we used shared memory for reading the A and B matrices in order to improve the performance. That has a big effect in Memory as it drops down from 94.01%(gpu2) to 72.27%.
- In addition versions **gpu3** and **gpu4** has lower memory latency than **gpu2**, that was also expected as in those kernels we reduce the number of threads we are using as one thread computes one or more elements of C.
- Ofcourse **gpulib**, DGEMM function for GPUs provided by Nvidia in the CUBLAS has the lowest percentage in memory use, as it optimizes the use of shared memory and registers in order to use as less as possible the really slow gpu DRAM.
- On the other hand as we can see **gpulib** is really compute bound 92.25% that was also something to expect as it performs ALU-FPU instructions (calculate an arithmetic result, compute a memory address, or perform a comparison for a branch and operations on floating-point numbers)
- Moreover we can see a decrease in the percentage of SM in **gpu5** from **gpu2** and that happens because in **gpu5** we are balancing shared memory versus register usage.
- Finally **gpu3** and **gpu4** have in general (except from gpu4_6) lower SM percentage as we have less registers because of the reduction of threads.

Moreover we used the Nsight profiler via user interface in order to get a better intuition for the analysis of our Kernels. Below we are going to present some of the most worth mentioning results which where analyzed before(gpu2 our initial GPU version compared to the gpulib):

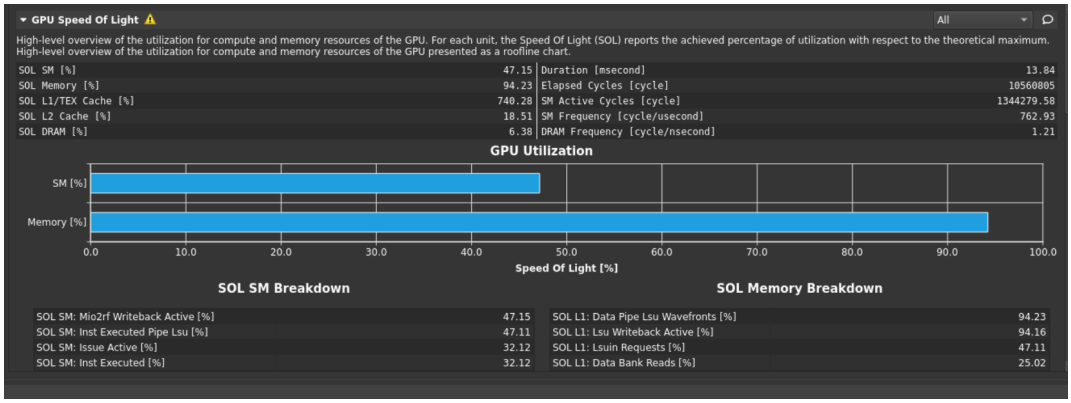

*Figure 8: Memory bound of **version gpu2***

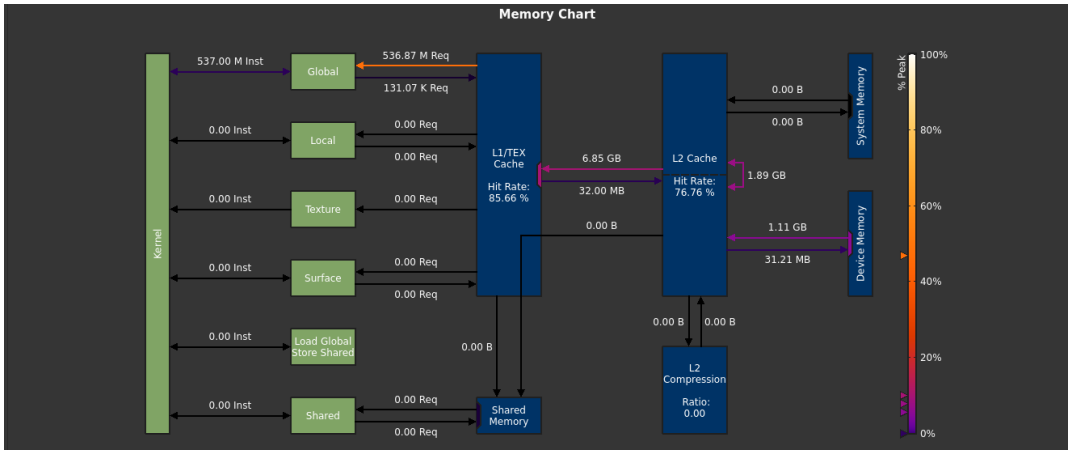

Figure 9: Memory Chart for ***version gpu2***

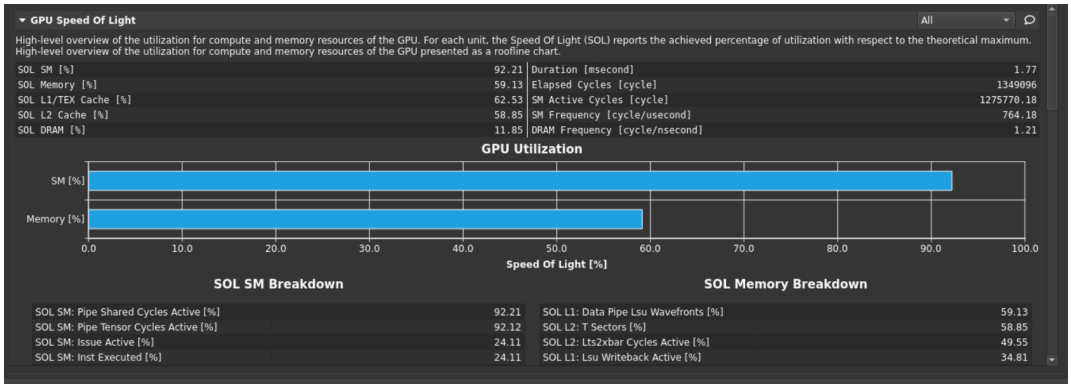

Figure 10: Compute bound of ***version gpu2***

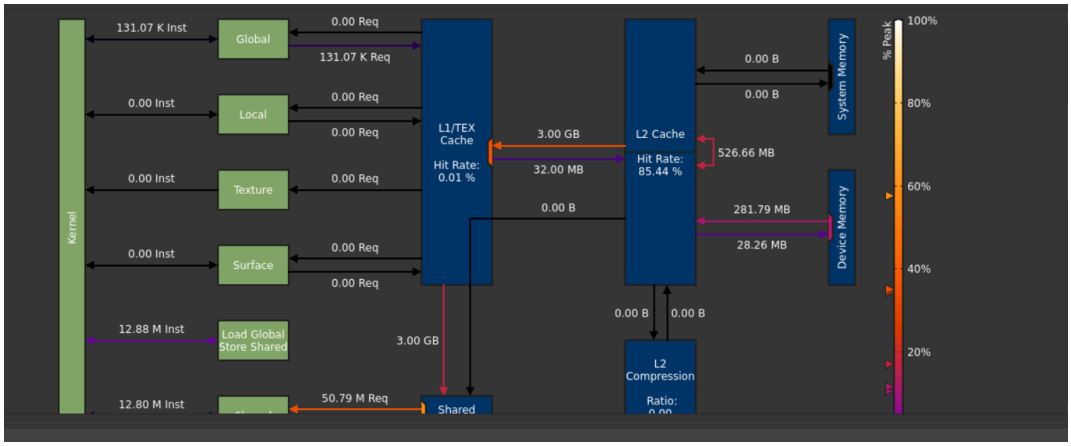

Figure 11: Memory Chart for ***version gpulib***

As we can see from the memory chart of version gpu2 the top left arrow has a really big value and that confirms that this version is memory bound as it uses global memory a lot. On the other hand gpulib uses the memory way mach more efficiently as we can see a high value in the yellow arrow in the bottom left corner which indicates the usage of the shared memory.

If we had time to improve our algorithm maybe we will tray a compination od the versions gpu4 and gpu5, in that way we will reduce the registers are reduced through variable reuse via shared memory and less threads.

Our fastest GPU version

Part III. GPU Poisson Problem

Exercise 5. First GPU-Version

We begin by implementing a single threaded version of the Jacobi-Algorithm. We initialize all arrays on the CPU and copy the over with the *transfer_3d*-function. Since a single thread is executing the whole operation, we will not need to change any of the for-loops in the iteration yet. We therefore include here only the code for the launching of the kernel. We remark that the function has two addition input arguments: step_width and denominator: The first one is $\frac{2}{N}$ and the second one is $\frac{1}{6}$, both operations that otherwise need to be executed inside the iteration, and, in the future, also by every single thread. By handing them over as values, we can thus save a considerable number of Flops.

```
    cudaEventRecord(start,0);
    for (i=0; i< iter_max; i++){
    jacobi_gpu1<<<1,1>>>(u_d,prev_u_d,f_d,N, step_width, denominator);
    cudaDeviceSynchronize();

    swap = u_d;
    u_d = prev_u_d;
    prev_u_d = swap;
}
  //stop timing
  cudaEventRecord(stop,0);
```

Listing 7: Launching of kernel

We remark that we have timed the kernel, using a CUDA event, which records the time used in performing an action in milliseconds. We compare the performance of this function for small matrix sizes (here up to N=55). We have the same number of iterations (here: 100) and the same number of floating point operations per iteration, and can thus express this as Memory-Footprint vs Flop/s. We see that this function is doing remarkably bad. This is not very surprising, as it would be a very inefficient way of calculating, considering the lower clock-rate of a GPU processor vis-à-vis a CPU processor.

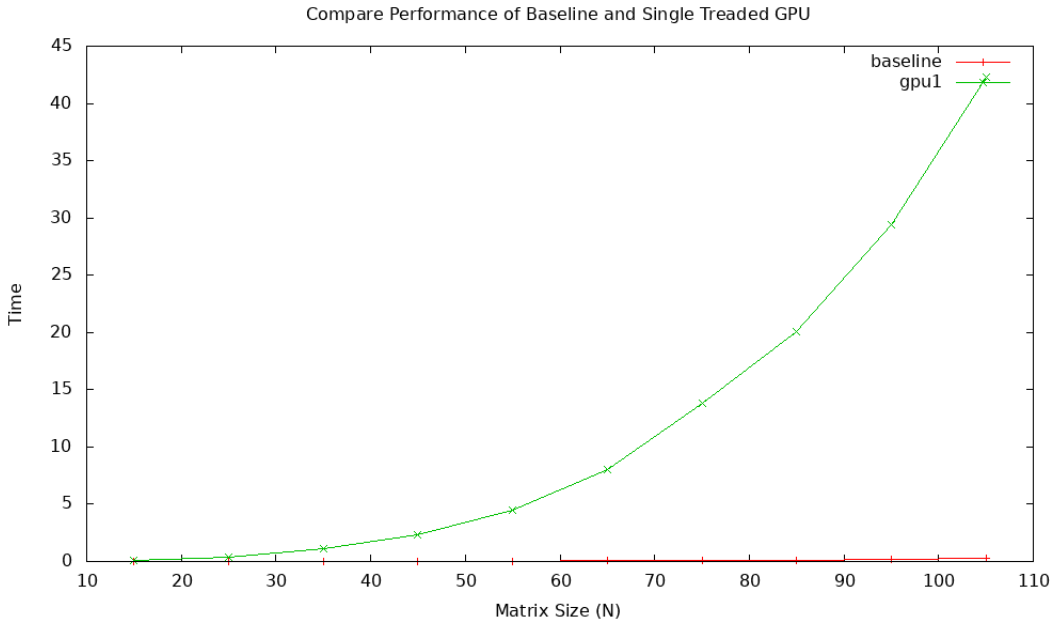

Figure 12: Compare Baseline and Single Threaded Version

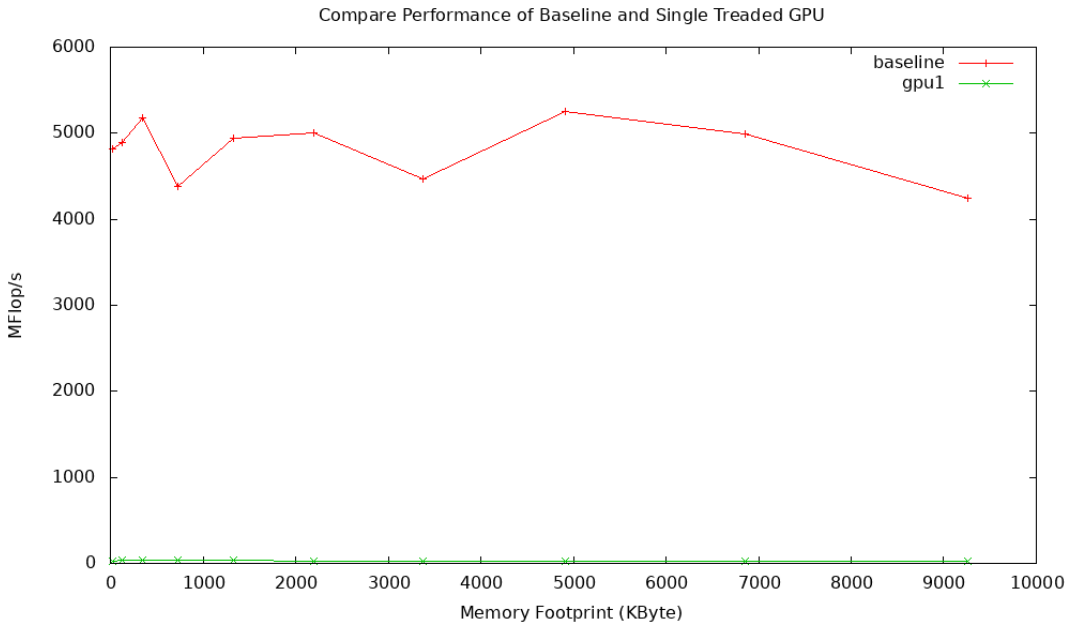

Figure 13: Compare Baseline and Single Threaded Version

Exercise 6. Multi-Threaded Version

We thus take this exercise as a stepping stone to a multi-threaded version, which, we hope will yield considerately better results, even largely outperform our quickest CPU-implementation of the Jacobi-Algorithm. We first need to create the grid 11. As we can only have blocks of at least 1024 threads, we temporarily set the block size in each dimension to 8 (equaling blocks of size $8 \times 8 \times 8$). We also remark, that the actual number of datapoints, that we want to iterate over is not actually N, but rather N-2 in each direction. We thus divide $N-2$ by 8 in order to obtain the number of blocks in the grid. If this is not a real number we round it up (or here: add a block to the automatically rounded down value). This method is also applied in part 1 of the assignment.

```
    dim3 block_size(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid_size((N-2+block_size.x-1)/block_size.x, (N-2+block_size.y-1)/ block_size.y
   , (N-2+block_size.z-1)/block_size.z);
```

Listing 8: Setting Grid

We then launch the kernel analogous to exercise 5, only replacing grid values. The codes have been spared here because they're so similar to exercise 5, but can be found in the other documents handed in.
In the global function, which is executing the iteration, we first define the indices. Later, the matrix u is going to be overwritten, and the matrix prev_u accessed in the following index. This is due to the observation that we only start our update at index 1. We also remark that there's an if-clause, making sure, that the additional but unused threads which were needed to produce an integer for the block number, stay idle.

```
__global__ void jacobi_gpu2(double*** u, double***prev_u, double*** f, int N, double
    step_width, double denominator) {

  double temp;

  int j_index=threadIdx.y + blockIdx.y*blockDim.y;
    int k_index= threadIdx.x + blockIdx.x*blockDim.x;
  int i_index=threadIdx.z + blockIdx.z*blockDim.z;


  if ((j_index<N-2) && (k_index<N-2) && (i_index<N-2)){
    temp=prev_u[i_index][j_index+1][k_index+1] + prev_u[i_index+2][j_index+1][k_index+1]
     + prev_u[i_index+1][j_index][k_index+1] + prev_u[i_index+1][j_index+2][k_index+1] +
```

```
     prev_u[i_index+1][j_index+1][k_index] + prev_u[i_index+1][j_index+1][k_index+2] +
    step_width*step_width*f[i_index+1][j_index+1][k_index+1];
12   u[i_index+1][j_index+1][k_index+1]=temp*denominator;
13   }
14 }
```

Listing 9: Setting Grid

We compare this new version with our baseline. For our baseline we chose the number of 16 threads, which, in the last assignment had a good performance, and seems to be a reasonably high number to compare to. We quickly see that the trend has been inversed. The GPU-Implemented version is doing much better. This should be hoped and expected, as as many as 6000 calculations can work in parallel, thus greatly outperforming the CPU-Version.

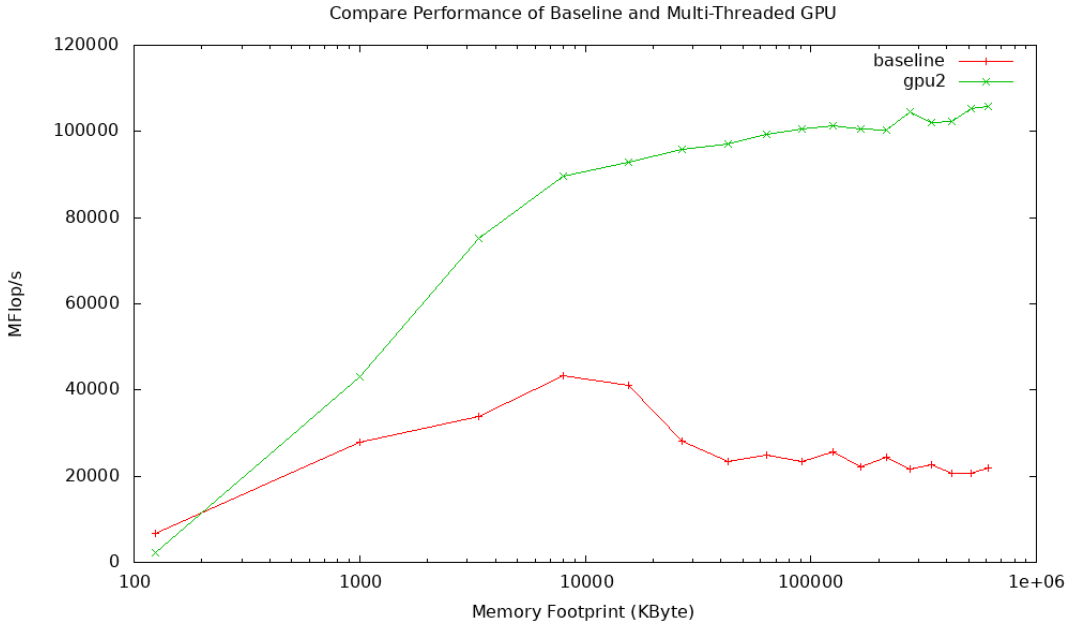

Figure 14: Compare Baseline and Second Version

We want to emphasise the fact, that this huge advantage stays, even if we consider the time using to write the matrices from one device to another, as can be seen in 15. This, means, that no matter if we keep working with the matrix on the GPU, or if they're later going to be used on the CPU, the implementation on the GPU proves it's value.

The calculated speedup (an average taken over 5 different matrix dimensions ranging from 200 to 700) is around 3.9×.

Profiler tool for the analyse of the Kernel We begin our analysis of the kernel with nvvp profiler command line interface and arguments 300 100 0 0:

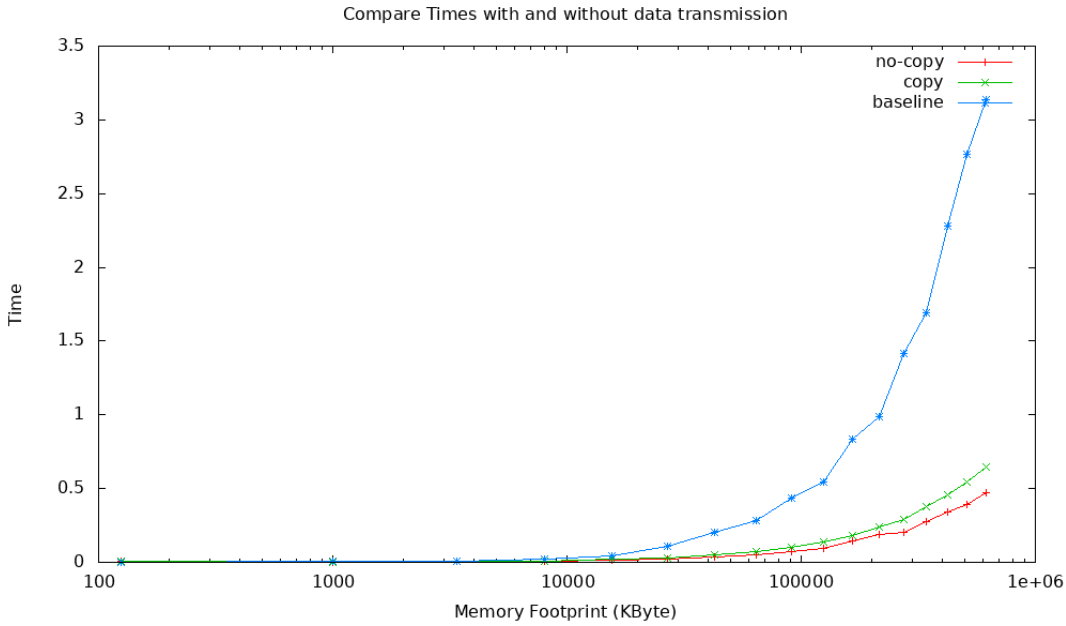

Figure 15: Time used on copying

```
_Z11jacobi_gpu2PPPdS1_S1_idd, 2021-Jan-22 21:47:18, Context 1, Stream 7
  Section: GPU Speed Of Light
  ---------------------------------------------------------------------- --------------- ------------------------------
  DRAM Frequency                                                         cycle/nsecond                           1.21
  SM Frequency                                                           cycle/usecond                         760.71
  Elapsed Cycles                                                                 cycle                         646327
  Memory [%]                                                                         %                          81.93
  SOL DRAM                                                                           %                          49.36
  Duration                                                                     usecond                         849.63
  SOL L1/TEX Cache                                                                   %                          82.25
  SOL L2 Cache                                                                       %                          92.45
  SM Active Cycles                                                               cycle                      643800.73
  SM [%]                                                                             %                          28.37
  ---------------------------------------------------------------------- --------------- ------------------------------
  OK    The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To
        further improve performance, work will likely need to be shifted from the most utilized to another unit.
        Start by analyzing workloads in the Memory Workload Analysis section.

  Section: Launch Statistics
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Size                                                                                                      512
  Grid Size                                                                                                     54872
  Registers Per Thread                                                   register/thread                           28
  Shared Memory Configuration Size                                                 Kbyte                        16.38
  Driver Shared Memory Per Block                                             Kbyte/block                         1.02
  Dynamic Shared Memory Per Block                                            byte/block                            0
  Static Shared Memory Per Block                                             byte/block                            0
  Threads                                                                        thread                     28094464
  Waves Per SM                                                                                                127.02
  ---------------------------------------------------------------------- --------------- ------------------------------

  Section: Occupancy
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Limit SM                                                                 block                           32
  Block Limit Registers                                                          block                            4
  Block Limit Shared Mem                                                         block                          164
  Block Limit Warps                                                              block                            4
  Theoretical Active Warps per SM                                                 warp                           64
  Theoretical Occupancy                                                              %                          100
  Achieved Occupancy                                                                 %                        88.11
  Achieved Active Warps Per SM                                                    warp                        56.39
  ---------------------------------------------------------------------- --------------- ------------------------------
```

Figure 16: nvvp profiler for Poisson problem using one thread per grid poin

As we can see from the results above the kernel is memory bound as there is a really high percentage 81.93% of memory,so most of the kernel time is spent in executing memory instructions . That was an expected behaviour as we are not using at all the shared memory which helps threads within the block to communicate with each other. That has a big effect in our memory as we use the local memory a lot which is really slow.

So to improve the kernel further I would suggest the usage of the shared memory.

Exercise 7. Implementation on 2 GPUs:

In this version the central issue is to implement the Jacobi-Algorithm to run on two GPUs. In order to do this we must initialize threads on both GPUs, where on each GPU we only initialize the half it will be calculating. It is important, that the GPUs have access to each other's memory, as there is going to be an access crossover, where each GPU needs to access the other half. New cuda versions, like the one we're using allow us to enable peer-access between the nodes. However this must be mutually set, as can be seen in the code. We then need to initialize the two kernels, switching between devices.

```
    dim3 block_size(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
    //reduce size of grid for smaller matrix
    dim3 grid_size(((N-2)*0.5+block_size.x-1)/block_size.x, ((N-2)+block_size.y-1)/
   block_size.y, ((N-2)+block_size.z-1)/block_size.z);

  //enable peer access between devices
  cudaSetDevice(0);
  checkCudaErrors(cudaDeviceEnablePeerAccess(1,0));
  cudaSetDevice(1);
  checkCudaErrors(cudaDeviceEnablePeerAccess(0,0));

    //start iteration
    for (i=0; i< iter_max; i++){
    printf("At iteration Nr %d \n", i);
    checkCudaErrors(cudaSetDevice(0));
    jacobi_gpu2<<<grid_size,block_size>>>(u_d,prev_u_d,f_d,N, step_width, denominator,
   0);
    cudaDeviceSynchronize();

    checkCudaErrors(cudaSetDevice(1));
    jacobi_gpu2<<<grid_size,block_size>>>(u_d,prev_u_d,f_d,N, step_width, denominator,
   1);
    cudaDeviceSynchronize();

    swap = u_d;
    u_d = prev_u_d;
    prev_u_d = swap;

  }
```

Listing 10: Enabling Peer Access

If we look into the Jacobi-function itself, we notice that we need to adapt the indices only in the direction in which we separated the matrix (for us the x-direction). Additionally, the incrementation of 1, which was done to compensate the fact, that we start the update for $N = 1$, is only valid for the first half, as the second half starts with the index $\frac{N}{2}$ immediately. This is reflected in the $devideID \times N \times 0.5 + (1 - deviceID)$- clause:

```
    int j_index=threadIdx.y + blockIdx.y*blockDim.y +1;
    int k_index= threadIdx.x + blockIdx.x*blockDim.x + deviceID*N*0.5 +(1-deviceID)*1;
  int i_index=threadIdx.z + blockIdx.z*blockDim.z+1;
```

Listing 11: Two GPUs - indices

When checking the performance of this specific function we chose not to examine the two kernels separately. First of all, we want to examine the performance of the whole Jacobi-Iteration, not just one half, secondly both GPUs that we're working on are of the same size. They're therefore expected to have the same performance anyways.

As we can see in 17, the version on two GPUs is doing tragically bad. The calculated average "speedup", compared to a 16-threaded baseline is 0.04×. The most probable reason for this, is that the peer-access is quite costly, and the two GPUs need to access each other's memory constantly. Some ideas for improvement of this would be slicing the matrix into two parts, only one of which needs to be accessed by both GPUs (more specifically the two middle-lines). Another idea would be to tweak the synchronization between the two kernels, allowing overlapping data transfer and computation.

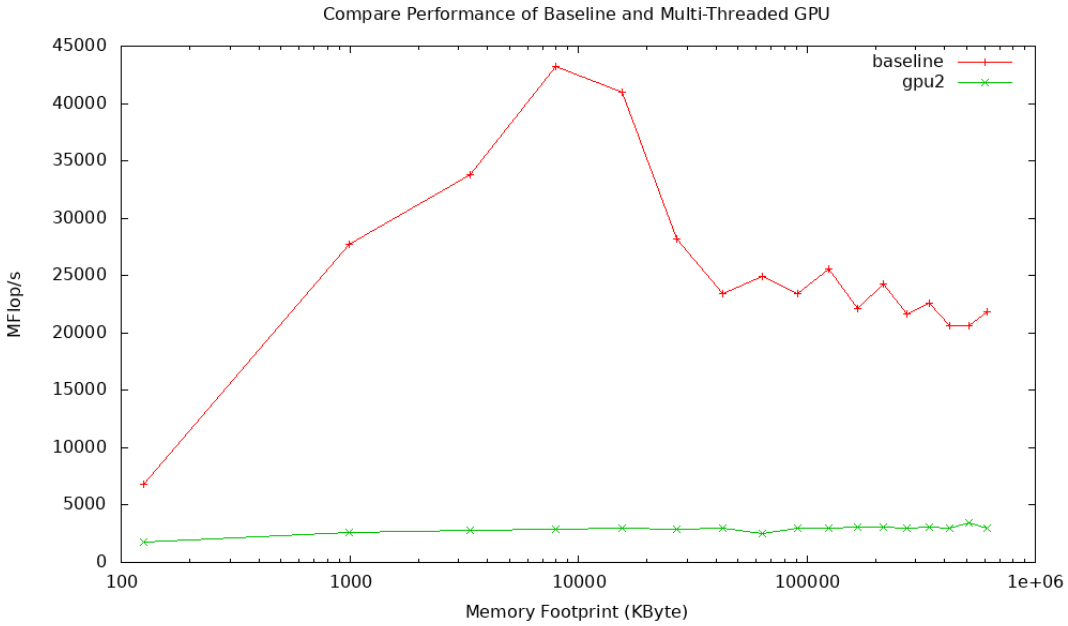

Figure 17: Baseline vs. Jacobi_GPU3

Exercise 8. Introducing the stopping condition

We re-introduce the stopping-condition that was spared out from the calculation before. The central challenge here is to have several, if not all threads write to the same pointer, in order to give us the norm. This is why we need to use the operation atomicAdd. The operation slows down the calculation enormously. We thus want to try and keep the application to a minimum. In the lecture we became familiar with several sum reduction techniques in varying sophistication.
In the following graph their performance can be seen. Version one, in which each thread is doing an atomicAdd is doing way worse, whereas the warp reduction is doing unexpectedly better, than our block reduction. The issue is in the blockReduceSum function: It iterates through the length of one direction, which is only a block size of 8:

It should be mentioned that there was a third sum reduction-technique introduced in the lecture. However, this technique saves time by initializing less threads on the GPU, something that was not possible in our task. This calculation is therefore spared here.

As we can see in 19, even the inefficient third version 3 of our sum reductions still yields better results than the baseline version (here with 16 threads), which is good news.

Average speed ups are the following:

Version	atomicAdd	Version1	Version2
SpeedUp	0.01×	6.2×	2.34×

Table 6: Speed-Up for different sum reductions

We want to know the overhead introduced by the calculation of the norm. We therefore compare our function jacobi_2 with version 2 (the best) of our sum-reduction variants, see table 7.

Size	With Norm	Without Norm	Difference	% Of without-norm to norm
100	0.016449	0.007208	0,0092	43%
200	0.234954	0.053569	0.182	22%
300	1.404525	0.163016	1.237	11%

Table 7: Overhead of Norm for GPU-Version

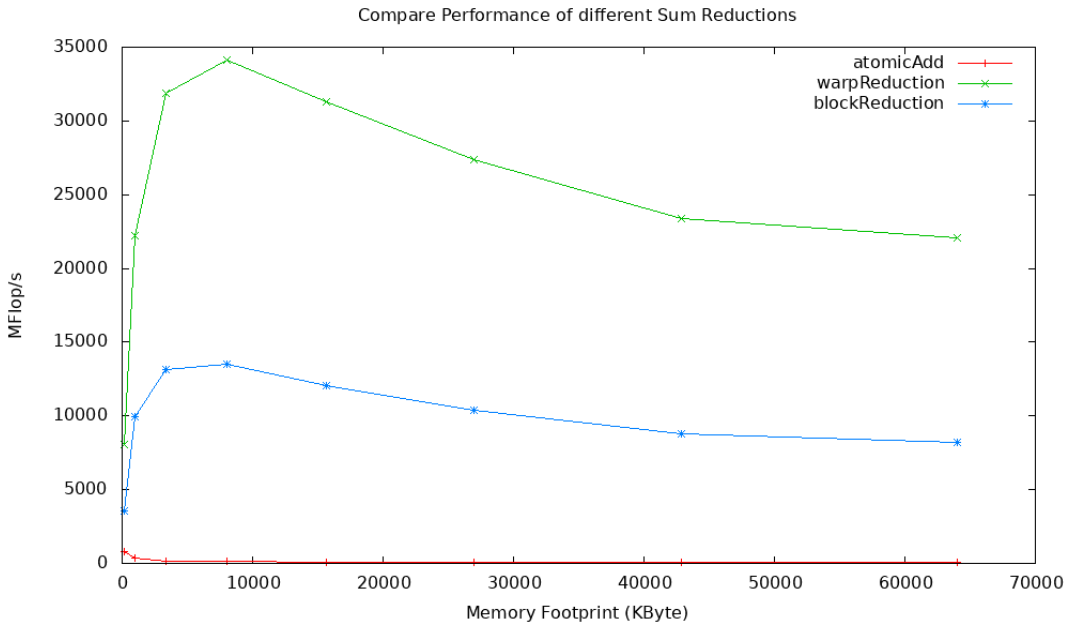

Figure 18: Three different Sum Reductions

We do the same comparison for our multi-threaded versions on the CPU, see table 7

Size	With Norm	Without Norm	Difference	% Of without-norm to norm
100	0.139187	0.019348	0.12	13%
200	1.620868	0.296793	1,3241	18%
300	4.866351	0.988948	3,88	20%

Table 8: Overhead of Norm for CPU-Version

It seems like the overhead produced by the sum varies from matrix site to matrix size, but, in both cases makes for around 80% of the total time, but seems to be a little higher for the CPU-Version.

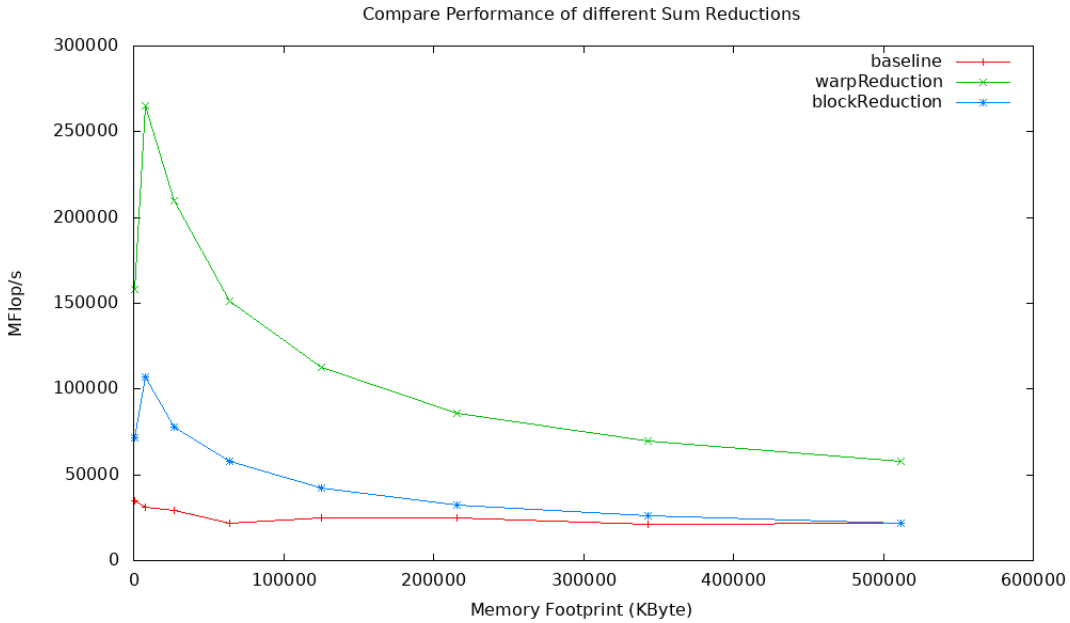

Figure 19: Sum reduction compared to Baseline