High-Performance Computing

Student name and id: Andreas Vedel Jantzen {s162858}

Collaborators: Anja Liljedahl Christensen{s162876} Marie Mørk {s112770} Anders Launer Bæk {s160159}

Hand-in: GPU Computing

1 Summary

In the report, the problems from assignment 1 and 2 are solved by implementing algorithms that make use of GPUs. Six different algorithms for performing matrix multiplication are implemented. This includes a GPU library function. The performance of the algorithms are compared and a speed-up calculated using the cBLAS DGEMM subroutine as reference. The best performing algorithm is found to be `gpu5`, which one thread on the device to calculate each element in the resulting matrix C, and exploits shared memory. In the chosen range of problem sizes, this algorithm performs even better than the GPU library function, cublasDgemm.
Three Jacobi methods for solving the Poisson problem, with different levels of utilization of the GPU have also been implemented. The performance of these has been compared to the fastest CPU implementation from assignment 2. The best performing GPU implementation has a speed-up of $\approx \times 16$.

2 Statement of the problem

In this assignment GPU computing is used to solve the problems from the two earlier reports in order to see if GPU computing enhance the performance. The report is therefore split in two parts; one concerning matrix-matrix multiplication, and the other on the Poisson problem.

Matrix-matrix multiplication
In the part about matrix-matrix multiplication, the dimensions of the matrices are the same as in Assignment 1, see figure 1. The performance of the implemented algorithms are to be compared with the cBLAS DGEMM subroutine used in Assignment 1.

Figure 1: Visualization of the matrix orientation. The picture is borrowed from the description of Assignment 1.

Six different algorithms for matrix-matrix multiplications using GPUs are to be implemented. In the following, a short description of the algorithms is provided listed by their function calls.

- `lib`: Multithreaded cBLAS DGEMM subroutine as reference.
- `gpu1`: Using a single thread.
- `gpu2`: Using one thread pr. element in matrix C.
- `gpu3`: Using one thread pr. 2 elements in matrix C. The optimal placement of the elements relative to each other is to be found.
- `gpu4`: Computing more than two elements in the C matrix pr. thread. The optimal placement of the elements relative to each other and number of elements is to be found.
- `gpu5`: Based upon the implementation linked to in the assignment description and modified to be compatible with the driver.
- `gpulib`: Implementation of the cuBLAS DGEMM subroutine.

Poisson Problem
The Poisson problem from assignment 2 will be solved using the Jacobi method. Three different algorithms will be implemented:

- `jac_cpu`: Best OpenMP function of the Jacobia function as reference.
- `jac_gpu1`: A sequential version using one thread and doing one iteration pr kernel launch.
- `jac_gpu2`: A naive version using one thread pr. grid point and only rely on global memory.
- `jac_gpu3`: Multiple GPU version, in which the interior points are to be updated from global memory and the boarder points between the two regions is read as peer values from the other GPU.

The kernels used are analyzed by the NVIDIA Visual Profiler (`nvvp`) in both parts of the assignment.

3 Hardware and software

Specifications of the test environment are listed below:

- CPU information
  - CPU(s): 24
  - Thread(s) per core: 1
  - Core(s) per socket: 12
  - Vendor ID: GenuineIntel
  - CPU family: 6
  - Model: 85

  - Model name: Intel(R) Xeon(R) Gold 6126 CPU @
  - 2.60GHz
  - CPU MHz: 2600.000
  - L1 (d / i) cache: 32K
  - L2 cache: 1024K
  - L3 cache: 19712K

- GPU information
  - NVIDIA TESLA V100 FOR PCIe x2
  - NVIDIA-SMI 387.26
  - Driver Version: 387.26

- Compilers
  - The SunCC compiler have been applied with followings flags: `-fast -xopenmp -xrestrict` for OpenMP reference in the Poisson problem.
  - `nnv` is used for default compilation of the cuda code.

4 Theory

One of the advantages of using a GPU instead of a CPU is that while a CPU has few cores, a GPU has thousands of smaller more efficient cores, that can work simultaneously. This results in a much higher amount of floating point operations and a much higher bandwidth, though the GPU cores can only perform simple operations, and the CPU cores can be assigned to different and more complicated operations [1].

The computations will be performed on a NVIDIA TESLA V100 GPU, hence the NVIDIA GPU optimized language CUDA will be used.

For medium to large matrices, matrix-matrix multiplication has the potential to be a compute-bound operation. The Jacobi method, on the other hand, is a memory-bound operation.

5 Matrix-matrix multiplication

In this section the algorithms, results, and analysis of the kernels used for matrix-matrix multiplication will be presented. The performance of the algorithms will be compared with each other and the CBLAS library DGEMM subroutine, which was implemented in the function `matmult_lib` in Assignment 1. The implementation of this algorithm will not be described again in this report, but the code is listed in the appendix, see algorithm 14.
Please note that the speed-ups are calculated based on the old driver on DTU Inside, which only uses 4 threads. Due to this, the calculated speed-ups are overestimated. Unfortunately, there was not sufficient time to update all the results.

5.1 gpu1

As already mentioned, in the implementation of `matmult_gpu1` the kernel is launched with a single thread. The implementation of the algorithm (both the CPU function and the kernel) is listed in algorithm 1.

```
1  __global__ void gpu1_kernel(int M, int N, int K, double *d_A, double *d_B, double
       *d_C){
2      //Set C entries equal to zero
3      for(int m=0; m<M; m++){
4          for(int n=0; n<N; n++){
5              d_C[m*N + n] = 0.0;
6          }
7      }
8
9      for(int m=0; m<M; m++){
10         for(int k=0; k<K; k++){
11             for(int n=0; n<N; n++){
12                 d_C[m*N + n] += d_A[m*K + k] * d_B[k*N + n];
13             }
14         }
15     }
16 };
17
18 extern "C" {
19 void matmult_gpu1(int M, int N, int K, double *A, double *B, double *C) {
20
21     //Define variables on device
22     double *d_A, *d_B, *d_C;
23
24     //Get sizes of matrices
25     int size_A = M*K*sizeof(double);
26     int size_B = K*N*sizeof(double);
27     int size_C = M*N*sizeof(double);
28
29     //Allocate memory on device
30     cudaMalloc((void**)&d_A, size_A);
31     cudaMalloc((void**)&d_B, size_B);
32     cudaMalloc((void**)&d_C, size_C);
33
34     //Copy memory host -> device
```

```
35      cudaMemcpy(d_A, A, size_A, cudaMemcpyHostToDevice);
36      cudaMemcpy(d_B, B, size_B, cudaMemcpyHostToDevice);
37
38      /* */
39      gpu1_kernel<<<1,1>>>(M, N, K, d_A, d_B, d_C);
40      /* */
41
42      //Synchronize
43      cudaDeviceSynchronize();
44
45      //Transfer C to host
46      cudaMemcpy(C, d_C, size_C, cudaMemcpyDeviceToHost);
47
48      // Free device memory
49      cudaFree(d_A);
50      cudaFree(d_B);
51      cudaFree(d_C);
52  }
53  }
```

Algorithm 1: `matmult_gpu1`, CPU function and kernel.

The performance of `matmult_gpu1` will be compared to the CPU CBLAS subroutine DGEMM. Performance is only measured for small matrix sizes, in this case four square matrices of sizes N = M = K = 32, 64, 96, and 128. Figure 2 shows the number of floating point operations pr. second in Mflops/s for the four matrix sizes.

The figure shows that `matmult_gpu1` is significantly slower than the optimized CPU library function DGEMM. This is as expected, as the GPU version only uses one thread, which means that the version is sequential and thereby does not take advantage of parallelism of the threads in the GPU. In addition to this, `gpu1` uses time for copying memory between the host and the device. The CPU function implementation on the other hand, is optimized using parallel programming ect. which makes it much faster.

In figure 3, the speed-up of `gpu1` compared to `lib` is shown. As it is already apparent from figure 2, the speed-up is below 1, meaning that the CPU version is faster than the GPU.

Performance of the GPUs is also evaluated using the `nvprof` command in the shell. For `gpu1`, the GPU summary shows that $99.98\% - 100\%$ of the time used for executing the GPU function is used in the kernel, whereas very little time is used for copying between host and device. This shows that in order for the algorithm to have a better performance, the computations within the kernels should be optimized.

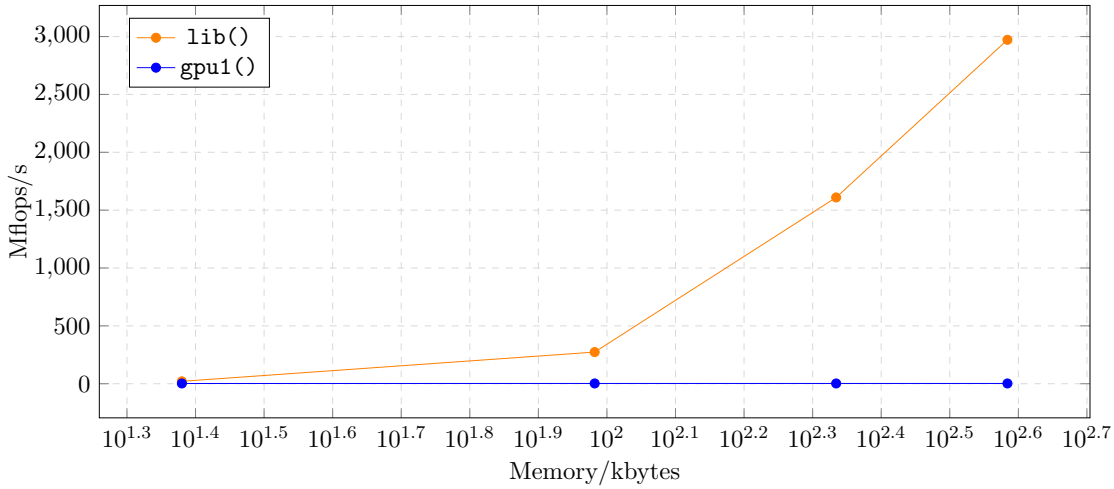


Figure 2: Comparison of `gpu1` and `lib`.

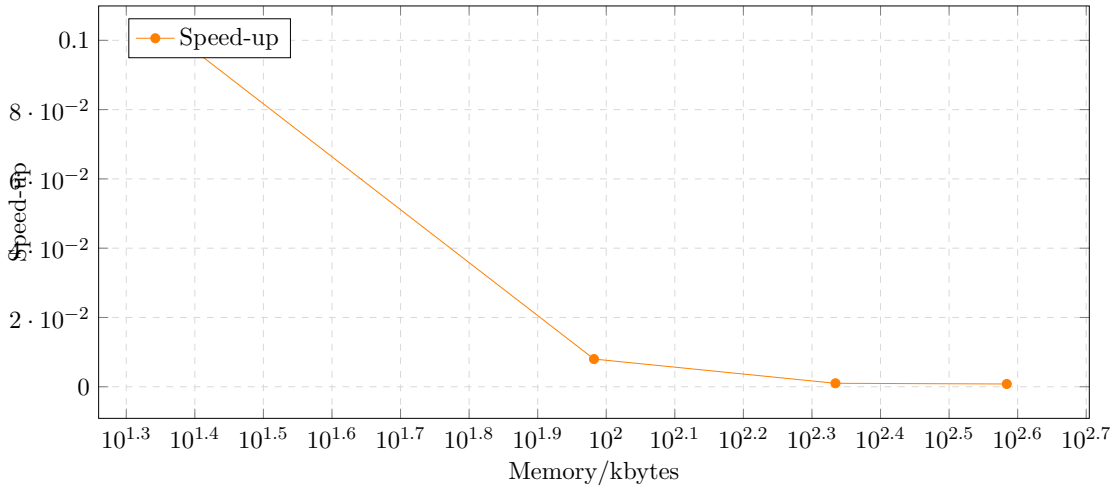


Figure 3: Speed up of `gpu1` compared to `lib`.

5.2 gpu2

The kernel for `matmult_gpu2` is listed in algorithm 2, while the changes in the launch from the CPU function is listed in 3. This time each thread computes one element of C: the thread with global thread id (i, j) computes element (i, j) in C. If there are more threads than elements in C the remaining threads will not compute anything.

```
1 void __global__ gpu2_kernel(int M, int N, int K, double *d_A, double *d_B, double
    *d_C){
2     //Get threads
3     int j = blockIdx.x * blockDim.x + threadIdx.x;  // In x
4     int i = blockIdx.y * blockDim.y + threadIdx.y;  // In y
5
```

```
    if(i < M && j < N) {
        //Set initial value to 0
        d_C[i*N + j] = 0.0;
        //Computing element
        for(int k = 0; k < K; k++){
            d_C[i*N + j] += d_A[i*K + k] * d_B[k*N + j];
        }
    }
};
```

Algorithm 2: `matmult_gpu2` kernel.

```
    int blocks = 16;
    int grid_m = (M + blocks - 1) / blocks;
    int grid_n = (N + blocks - 1) / blocks;
    gpu2_kernel<<<dim3(grid_n,grid_m), dim3(blocks,blocks)>>>(M, N, K, d_A, d_B,
    d_C);
```

Algorithm 3: `matmult_gpu2` launch of kernel.

Matrix sizes used for the comparison between `gpu2` and the CPU library function is set to be multiples of 16, such that the same matrix sizes can be used to compare with `gpu5` in a later section. We choose square matrices with dimensions 800, 1600, 2400, 3200, 4000, 4800, and 5600. In figure 4, Mflops/s as a function of problem size is shown for `gpu2` and `lib`. Comparing the two graphs, the advantages of using GPUs is now clearly visible and `gpu2` is faster than `lib` for all matrix sizes considered.

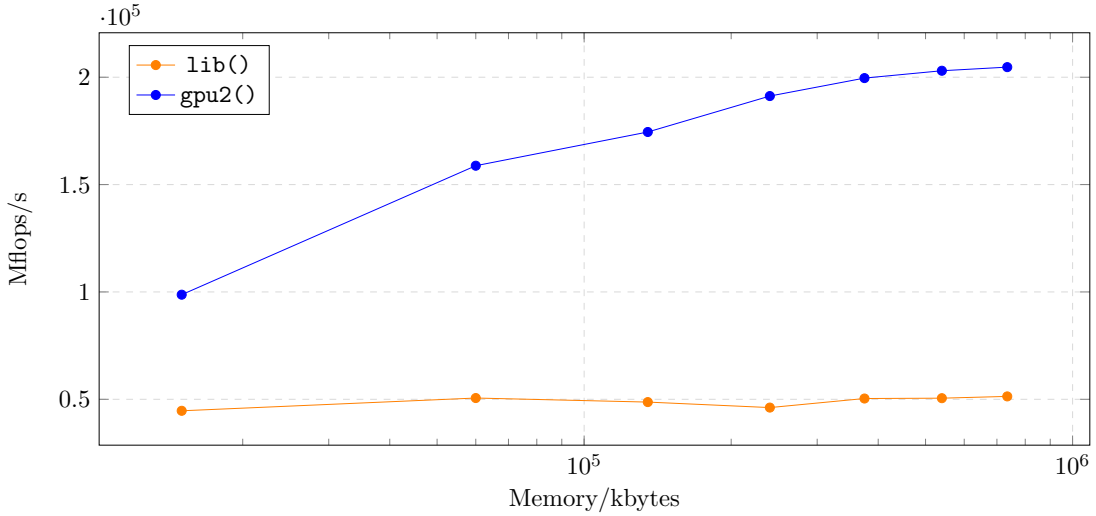


Figure 4: Comparison of `gpu2` and `lib`.

The speed-up is computed from mflops/s and reported in figure 5 for the chosen matrix sizes. It is seen that the speed-up rises with problem size up until matrices with dimensions higher than 3200. For larger matrices, the speed-up stays constant at around 4x.

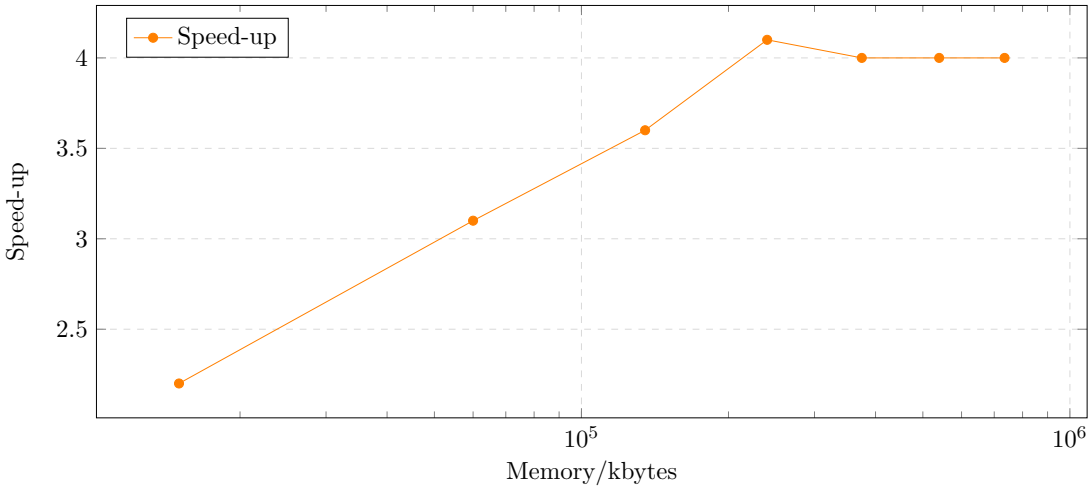


Figure 5: Speed up of `gpu2` compared to `lib`.

Using the call `nvprof --print-gpu-summary` in the shell, the different parts of the GPU function is timed. In table 1, the percentage used in the kernel, on copying from host to device, and from device to host is listed. For smaller problem sizes, the percentage of the total execution time used in the kernel is smaller than for large problems. It is to expect, as fewer calculations are needed for small matrices, while the memory still needs to be transferred between host and device.

Problem size	Kernel	HtoD	DtoH
800	81.63%	12.55%	5.82%
1600	89.86%	6.92%	3.23%
2400	92.93%	4.82%	2.25%
3200	94.11%	4.01%	1.88%
4000	95.19%	3.28%	1.53%
4800	95.95%	2.76%	1.29%
5600	96.51%	2.38%	1.11%

Table 1: Time spend on Kernel, HtoD and DtoH in the kernel of `gpu2` for the chosen problem sizes.

5.3 `gpu3`

In `gpu3`, each thread is to compute 2 elements in C. To do this, a stride introduced in the algorithm. The function is tested for the stride in both the x and the y direction for large matrices. A stride in y is found to be the fastest, hence the second element in C which is computed by the thread is the neighbor below the first element. This makes sense since in this way the threads access memory coalesced. Furthermore this way a single thread will access one column in B and two rows in A in order to compute the two elements in C. As memory storage in C is row-major, it is expected that accessing two rows and one column is faster than the opposite for each thread. The kernel for this implementation is listed in algorithm 4, while the changes in the launch from the CPU function is listed in algorithm 5.
Compared to the previous two kernels `gpu3_kernel` takes an extra argument which is the stride in the y (i) direction. Since each thread has to compute 2 elements, `stride` is set to 2 when the

kernel is launched, see algorithm 5. In algorithm 4 the thread with global thread id (i, j) will then compute element $(i \cdot 2, j)$ and $(i \cdot 2 + 1, j)$ of C.

```
void __global__ gpu3_kernel(int M, int N, int K, double *d_A, double *d_B, double
    *d_C, int stride){
    int i, j, k, s, is;

    i = (blockIdx.y * blockDim.y + threadIdx.y)*stride;
    j = blockIdx.x * blockDim.x + threadIdx.x;

    for (s = 0; s < stride; s++){
        is = i + s;
        if (is < M && j < N){
            d_C[is*N + j] = 0.0;

            for (k = 0; k < K; k++)
                d_C[is*N + j] += d_A[is*K + k] * d_B[k*N + j];
        }
    }
};
```

Algorithm 4: gpu3 kernel.

```
    int stride = 2, blocks = BLOCK_SIZE;
    int grid_m = (M-1)/blocks + 1;
    int grid_n = (N-1)/(stride*blocks) + 1;
    gpu3_kernel<<<dim3(grid_n,grid_m),dim3(blocks,blocks)>>>(M, N, K, d_A, d_B,
    d_C, stride);
```

Algorithm 5: gpu3 launch of kernel.

Again, the performance of the algorithm is compared to that of `lib`. In figure 6, Mflops/s is shown for different problem sizes. For the largest problem size, gpu3 computes almost twice as many Mflops/s as gpu2. This means that the speed up compared to the CPU version is higher for gpu3.

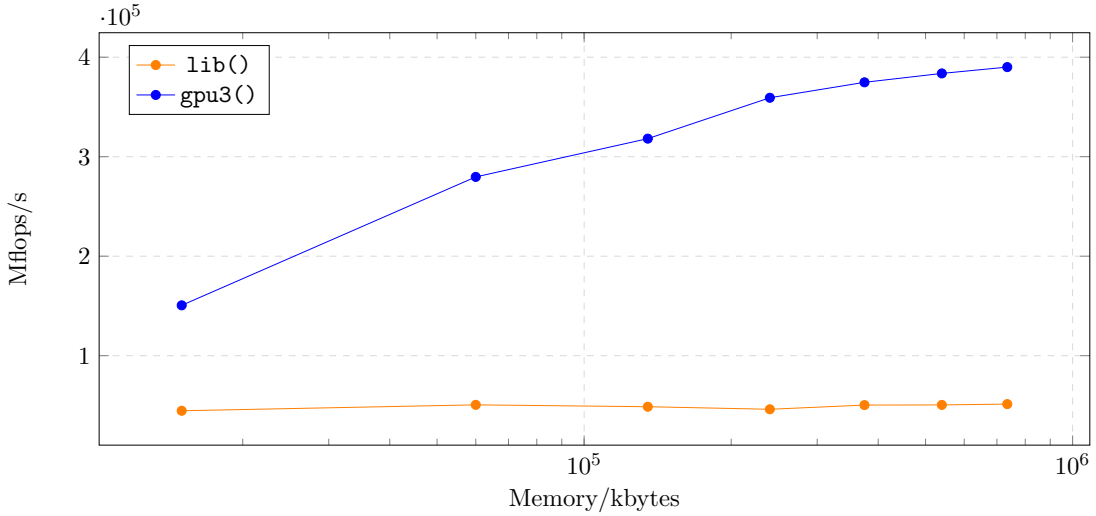


Figure 6: Comparison of gpu3 and `lib`.

In figure 7, the calculated speed-ups are shown for the chosen problem sizes. The graph shows the same tendency as the speed-up for `gpu2` namely that the speed up cease to increase for problem sizes larger than 3200. The speed-up is almost doubled compared to `gpu2`.

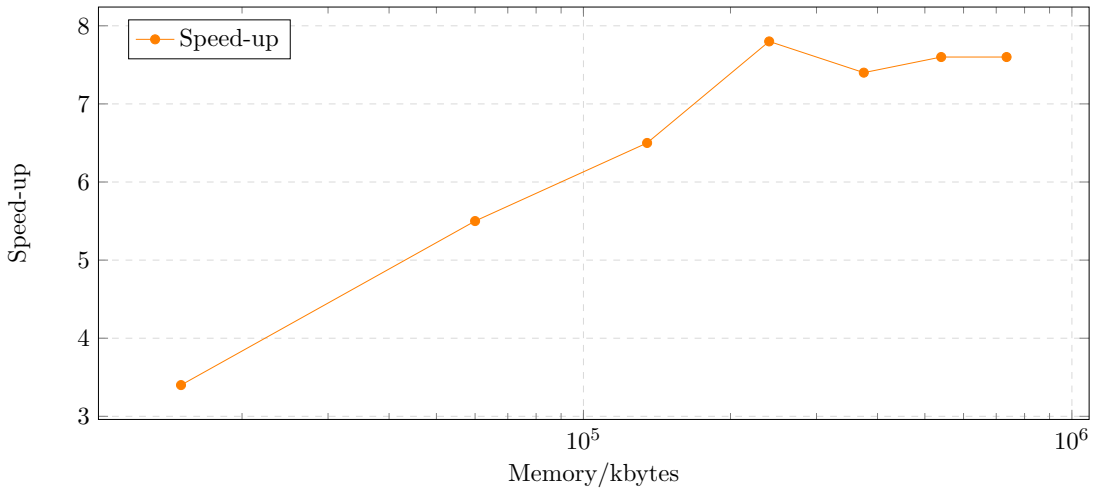


Figure 7: Speed up of `gpu3` compared to `lib`.

5.4 gpu4

In the previous version each thread computed 2 elements of C, but now the number of elements for each thread has to be more than 2. It is immediately ruled out to assign elements in more than 1 column of C for each thread (since it is faster for the threads to access memory coalesced), hence the question is: how many element in 1 column (directly on top of each other) should each thread compute. This is investigated by using the matrix sizes $M = N = K = 5000$ and trying out different strides in the y direction. The kernel is listed in algorithm 6, while the changes in the launch from the CPU function is listed in algorithm 7.

```
1  void __global__ gpu4_kernel(int M, int N, int K, double *d_A, double *d_B, double
       *d_C, int stride_n, int stride_m){
2      int i, j, k, sn, sm, js, is;
3
4      i = (blockIdx.y * blockDim.y + threadIdx.y)*stride_m;
5      j = (blockIdx.x * blockDim.x + threadIdx.x)*stride_n;
6
7      for (sn = 0; sn < stride_n; sn++){
8          js = j + sn;
9
10         for (sm = 0; sm < stride_m; sm++){
11             is = i + sm;
12
13             if (is < M && js < N){
14                 d_C[is*N + js] = 0.0;
15
16                 for (k = 0; k < K; k++){
17                     d_C[is*N + js] += d_A[is*K + k] * d_B[k*N + js];
18                 }
```

```
            }
        }
    }
};
```

Algorithm 6: `gpu4` kernel.

```
    int stride_m = 6, stride_n = 1, blocks = BLOCK_SIZE;
    int grid_m = (M-1)/(stride_m * blocks) + 1;
    int grid_n = (N-1)/(stride_n * blocks) + 1;
    gpu4_kernel<<<dim3(grid_n,grid_m),dim3(blocks,blocks)>>>(M, N, K, d_A, d_B,
    d_C, stride_n, stride_m);
```

Algorithm 7: `gpu4` launch of kernel.

In figure 8 below the GPU time is recorded for small values of `stride_m`, i.e. from 3 to 64, and in figure 9 the GPU time is recorded for larger values of `stride_m`. The size of both strides (`stride_n` in the x direction and `stride_m` in the y direction) are given as arguments from the command line. This might explain the strange behavior of the GPU time. When the strides are not predefined at compile time, memory is not allocated optimally. Hence, the registers of the threads are not fully exploited. From the plots below, the GPU time is minimal when each thread computes 6 elements, though it is hard to see.

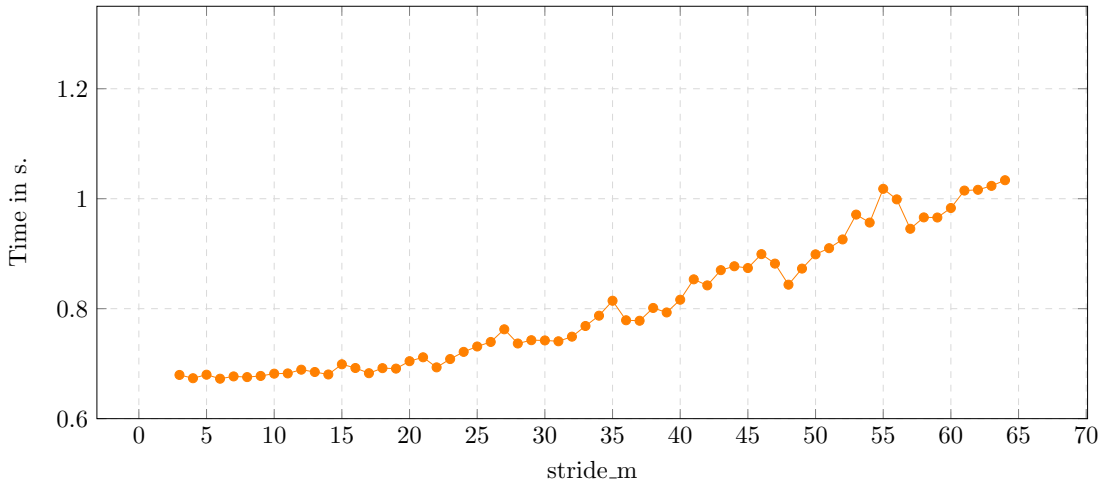


Figure 8: GPU time for `matmult_gpu4` for different small sizes of `stride_m`.

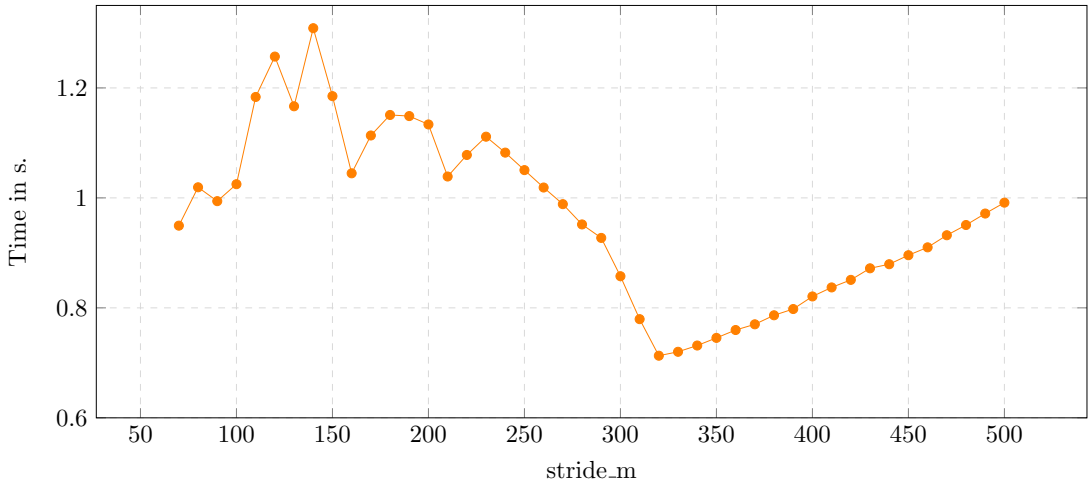


Figure 9: GPU time for `matmult_gpu4` for different large sizes of `stride_m`.

Below in algorithm 8 is the altered kernel for `matmult_gpu4` where the outer stride for-loop has been removed (since `stride_n` is always chosen to be 1), and the variable `stride_m` has been replaced by a fixed integer (6 in this case). This means that both the kernel and `matmult_gpu4` take 2 arguments less than the original versions.

"Hardcoding" the stride for different sizes did surprisingly and unfortunately not result in better GPU times, and `stride_m` $= 6$ is still the optimal choice.

```
1  void __global__ matmatgpu4(int M, int N, int K, double *d_A, double *d_B, double *
       d_C)
2  {
3      int i, j, k, sm, is;
4
5      i = (blockIdx.y * blockDim.y + threadIdx.y)*6;
6      j = blockIdx.x * blockDim.x + threadIdx.x;
7
8      for (sm = 0; sm < 6; sm++){
9          is = i + sm;
10         if (is < M && j < N){
11             d_C[is*N + j] = 0.0;
12
13             for (k = 0; k < K; k++)
14                 d_C[is*N + j] += d_A[is*K + k] * d_B[k*N + j];
15         }
16     }
17 };
```

Algorithm 8: Optimized gpu4 kernel.

In figure 10 below the number of floating points operations per second of `matmult_gpu4` is compared to `matmult_gpu3` and `matmult_lib`, and in figure 11 the speed-up of `matmult_gpu4` is compared to `matmult_gpu3`. The efficiency of `matmult_gpu4` is actually worse than the efficiency of the previous version, i.e. according to the implementations of these kernel-functions the best efficiency and

speed-up is achieved when each thread computes 2 elements of C instead of 1 or > 2. This is not as expected and is probably also due to the above issue with memory allocation at compile time.

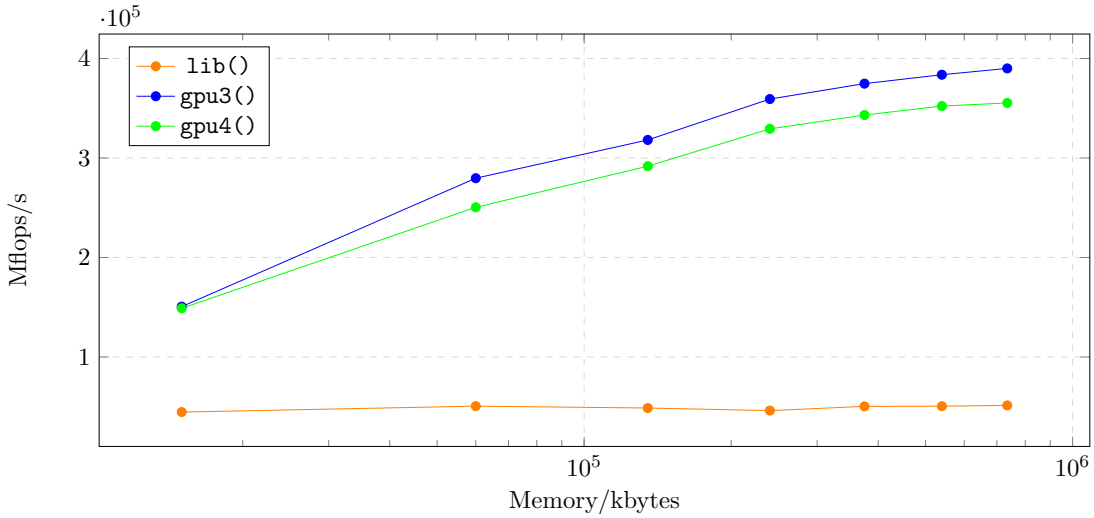


Figure 10: Comparison of `gpu4` and `lib`.

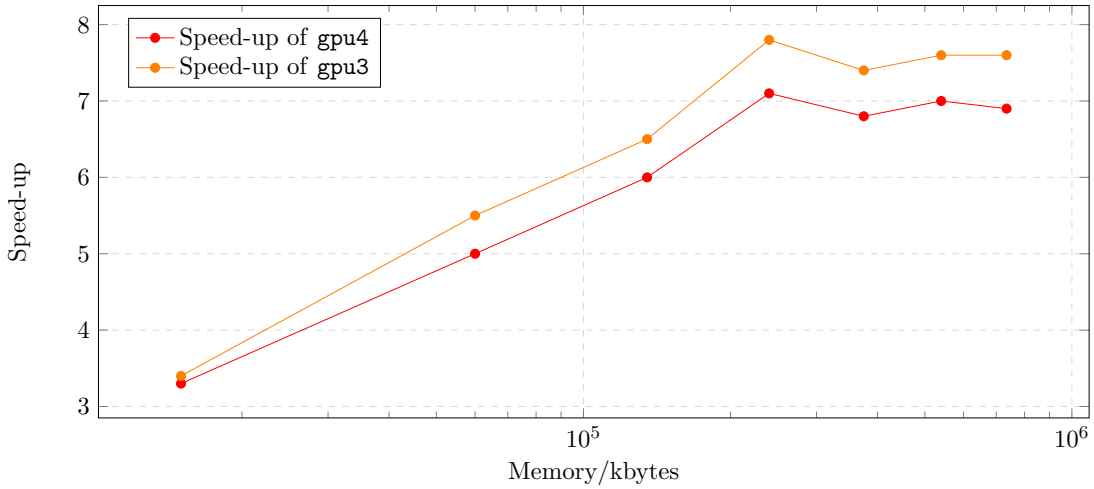


Figure 11: Speed up of `gpu4` compared to `lib`.

5.5 `gpu5`

The `gpu5` version is based on the shared memory matrix-matrix multiplication algorithm given on `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory`. Several things had to be changed in order for this version to be compatible with the driver and for it to provide the correct result. **Please note** that since the original version assumes that M, N, and K are integer multiples of the block thread size (i.e. 16×16), the modified version also assumes that M, N, and K are multiples of 16.

The issue with the compatibility is solved by simply changing the input arguments of `matmult_gpu5`. This means that the way the matrices A, B and C are loaded to the device memory also has to be changed. This is done by using the matrix sizes M, N, and K directly as well as the matrices A, B, and C, instead of first creating a Matrix struct for each of the host matrices and then use these to create the device Matrix structs. The modified version is given in algorithm 9 below.

Lastly all variables and help functions of type `float` have to be changed to `double` such that there are no round-off errors.

```
void matmult_gpu5(int M, int N, int K, double *A, double *B, double *C) {
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = K;
    d_A.height = M;
    size_t size = M * K * sizeof(double);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A, size, cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = N;
    d_B.height = K;
    size = K * N * sizeof(double);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B, size, cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = N;
    d_C.height = M;
    size = M * N * sizeof(double);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(N / dimBlock.x, M / dimBlock.y);
    gpu5_kernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C, d_C.elements, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
}
```

Algorithm 9: `gpu4` launch of kernel.

In figure 12 the efficiency of this version is compared to the dgemm subroutine. This is by far the fastest version and contrary to the previous versions Mflops/s is increased linearly with the problem size. The improvement of the fifth version is also especially evident when considering the speed-up compared to the dgemm subroutine in figure 13. With version three the maximal speed-up was close to 8 but version five the maximal speed-up is almost 30.

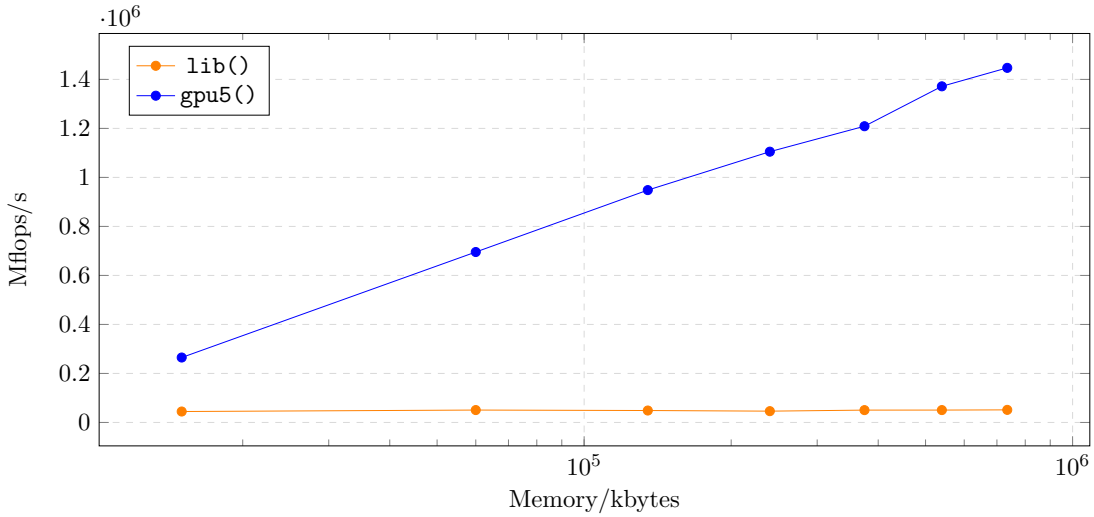


Figure 12: Comparison of `gpu5` and `lib`.

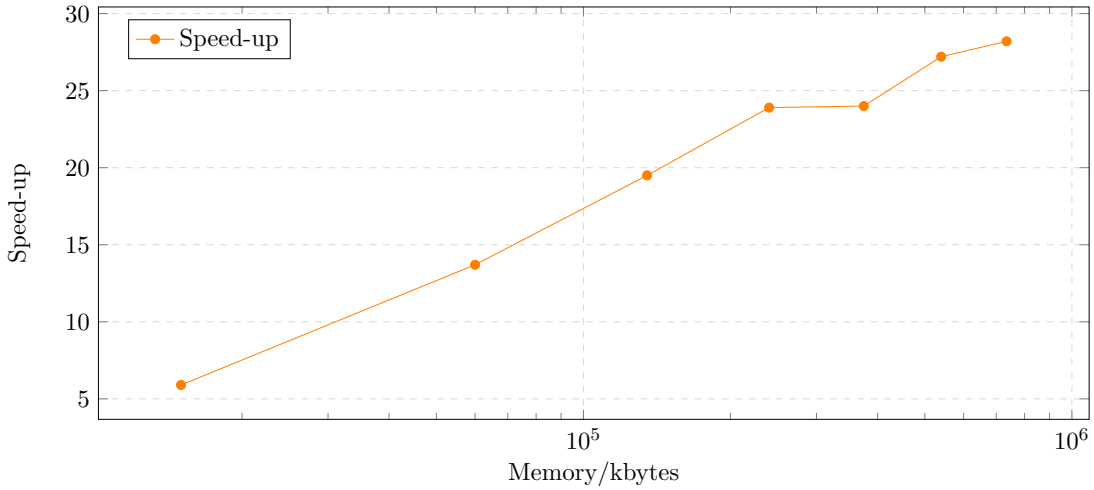


Figure 13: Speed up of `gpu5` compared to `lib`.

5.6 `gpulib`

The last matrix multiplication algorithm is the DGEMM function for GPUs that has been implemented in the function `gpulib` such that it can be run on the provided driver. The cublasDgemm function takes 14 arguments and is column-major. The matrix A and its LDA the matrix B and its LDB have been swooped in order to make cublasDgemm row-major. The implemented function is listed in algo. 10.

```
1 void matmult_gpulib(int M, int N, int K, double *A, double *B, double *C) {
2
3     // cuBLAS handle??
4     cublasHandle_t handle;
```

```
5       cublasCreate(&handle);
6
7       //Define variables on device
8       double *d_A, *d_B, *d_C;
9
10      //Get sizes of matrices
11      int size_A = M*K*sizeof(double);
12      int size_B = K*N*sizeof(double);
13      int size_C = M*N*sizeof(double);
14
15      //Allocate memory on device
16      cudaMalloc((void**)&d_A, size_A);
17      cudaMalloc((void**)&d_B, size_B);
18      cudaMalloc((void**)&d_C, size_C);
19
20      //Copy memory host -> device
21      cudaMemcpy(d_A, A, size_A, cudaMemcpyHostToDevice);
22      cudaMemcpy(d_B, B, size_B, cudaMemcpyHostToDevice);
23
24      /* */
25      int LDA = fmax(1,K); // leading dimension of A
26      int LDB = fmax(1,N); // leading dimension of B
27      int LDC = fmax(1,N); // leading dimension of C
28      double alpha = 1.0, beta = 0.0; // scaling
29      // into row-major
30      cublasDgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,N,M,K,&alpha,d_B,LDB,d_A,LDA,&beta,
        d_C,LDC);
31      /* */
32
33      //Synchronize
34      cudaDeviceSynchronize();
35
36      // copy result
37      cudaMemcpy(C, d_C, size_C, cudaMemcpyDeviceToHost);
38
39      // cleanup
40      cublasDestroy(handle);
41      cudaFree(d_A);
42      cudaFree(d_B);
43      cudaFree(d_C);
44  }
```

Algorithm 10: `gpulib` launch of kernel.

In figure 14, the number of floating point operations performed pr second is shown for different problem sizes. For small problem sizes, the CPU version of DGEMM, `lib` is faster than the GPU version, `gpulib`. This changes for matrices with dimensions a little larger than 1600, for which the GPU version becomes faster. Figure 15 shows the speed-up for the chosen problem sizes. As was the case with `gpu5`, the speed-up does not wear off for large problem sizes.

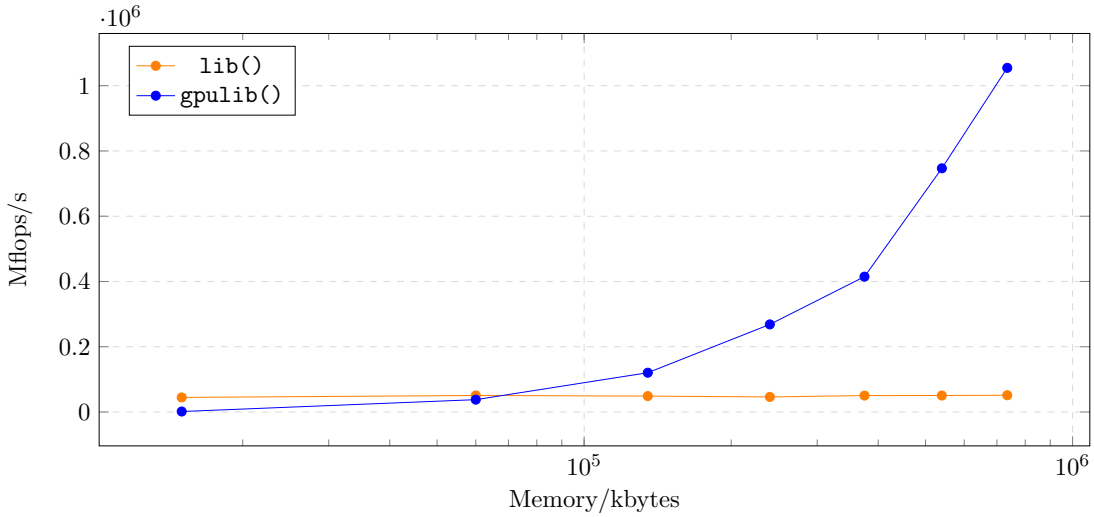


Figure 14: Comparison of `gpulib` and `lib`.

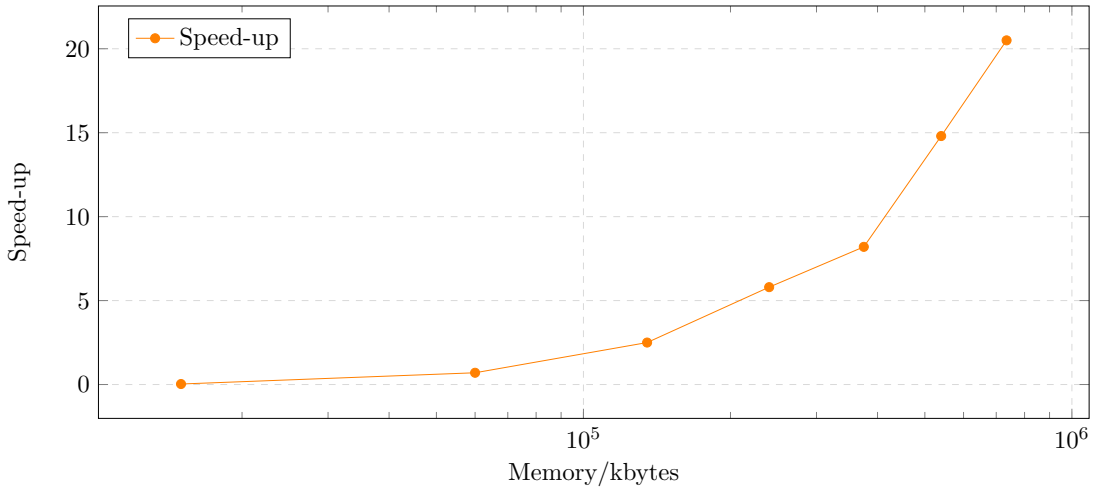


Figure 15: Speed up of `gpulib` compared to `lib`.

5.7 Comparison of the algorithms

In order to compare all implementations of matrix multiplication using GPUs, the calculated speed-ups are shown in figure 16. As the speed-up was below 1 for `gpu1`, the performance of this algorithm has been omitted from the plot. The figure clearly shows that `gpu5` has the largest speed-ups for the chosen problem sizes. It is noted that the tendency of the speed-up for `gpu5` looks linear while `gpulib` seems to have an exponential growth in this region. This might mean that `gpulib` has a larger speed-up than `gpu5` for larger problem sizes. Aside from the shared memory the implementation of `gpu5` is actually similar to `gpu2` in the sense that each thread computes 1 element of C. Hence, the advantage of using shared memory is very evident when the blue (`gpu2`) and red line (`gpu5`) in figure 16 are compared.

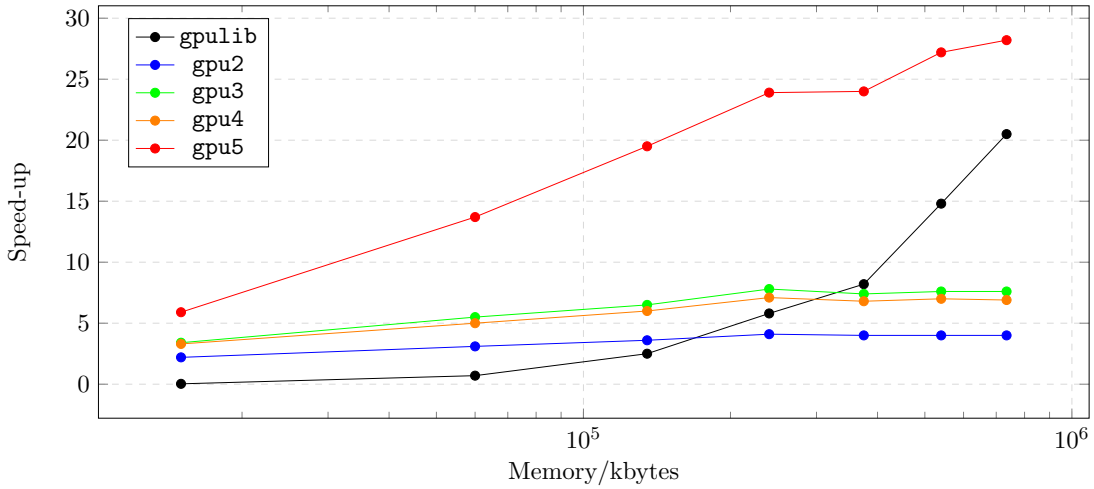


Figure 16: Speed up comparison.

The `nvvp` profiler is used to analyze the different kernel-versions above. First, the `gpu2` kernel is analyzed.

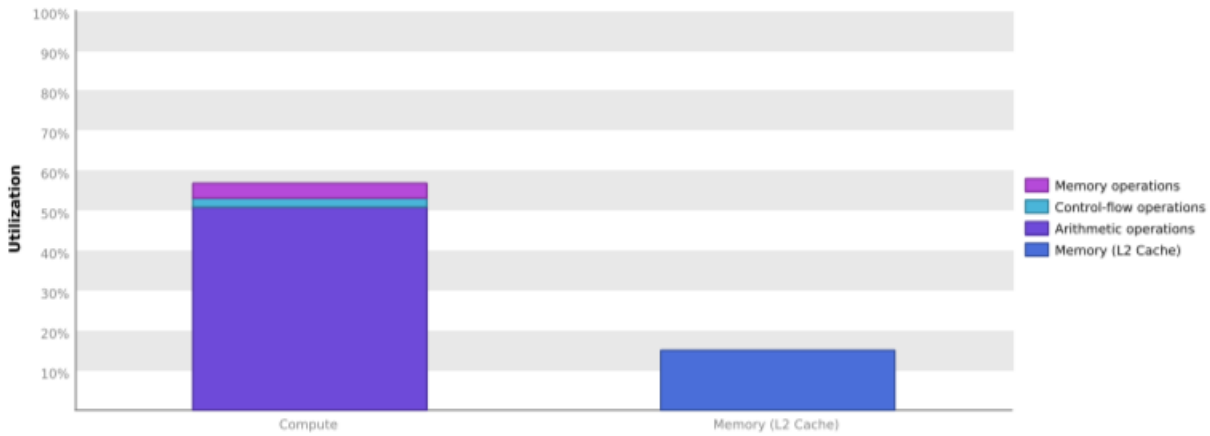


Figure 17: Kernel performance of `gpu2`.

From figure 17 above it is evident that too much time is spend on memory. This is quite surprising since matrix-matrix multiplication should be compute-bound for large problem sizes and not memory-bound. The `nvvp` profiler reveals that only 20 registers are used, while others (e.g. Hans Henrik) were able to use 32 registers. This automatically results in too many cache-misses since the 20 registers cannot store as much information, and hence memory has to be fetched more often. Therefore, time that could be spend on computing is instead spend on waiting for the memory to be loaded.

Unfortunately it was not clear what caused this low number of registers and how it could be increased, and therefore the issue could not be solved in time.

Similar problems occurred for the other versions: for `gpu3` the number of registers is 24, for `gpu4` the number is 27, and for `gpu5` the number of registers is 32. Even though the number of registers increased for each version, too much time was still spend on memory. It is possible that the new driver (which was not used due to time issues) would solve this problem. Another idea would be to run the driver on a different node.

A second suggestion for further improving the kernels applies to all of the different versions. Every single time an element of A is multiplied by an element of B, global memory is accessed (by updating the corresponding element of C) which is very time consuming. This could be solved by using a local variable inside the kernel instead. I.e. if a local variable `C_value` was used to store the intermediate value of the current element of C, it would only be necessary to access global memory once for each element of C instead of every time an element of A is multiplied by an element of B.

The implementation of version 5 actually uses a local variable inside the kernel, and this probably explains some of the extra speed-up compared to the other versions.

6 Poisson problem

This section includes several algorithms/kernels, results and analysis of those performances. The performance of the kernel setups will be compared and they have been evaluated for $N = 2048$ and `max_iter` $= 1000$. The chosen parameters gives the refrence algorithm a runtime of ≈ 1.5 secound. The main purpose of this section is to find the speedup for difference GPU implementations in reference of the best OpenMP (`jac_cpu`) version from previous assignment. The environment variable which determines the wait policy of the threads has not been set to `OMP_WAIT_POLICY=active` is in the previous assignment. The OpenMP is evaluated with `OMP_NUM_THREADS=12`. The achieved speedups are presented in table 2.
It has been chosen to validate the implementation of the difference GPU kernels by visualizing their estimates of $u(x, y)$ after the last iteration. This give a visual verfication of the kernel and source implementations. See plots in figure 19 in the appendix.
The iterative process is controlled by the host and it uses `cudaDeviceSynchronize()` to make sure the work of the threads on the devices is done before incrementing the iteration. The iterative process `while(k < max_iter)` which includes pointer switchs and new kernel calls is identical for all three GPU versions. Although the kernels are called by different kernel launch parameters: `<<<grid,block>>>`.
The initialization of the boundary conditions in u and u_old, and of the heating source given by f are done on the on the host. The and copied to the device by using the approiate cuda calls. The I/O duration for transferring the initial matrices to the device and the duration of the transferring the estimate of u back to host is included in the total compute time in order to make a fair comparison to the `jac_cpu` function.

6.1 Sequential GPU Jacobi

The kernel used in the Sequential Poisson is provided in algorithm 11.
`jac_gpu1` is called by the following launch parameters `<<<1,1>>>jac_gpu1`. This ensures it only enables one block with one thread. See algo. 16 for the complete source code.

```
1 void __global__ jac_gpu1(int N, double delta, int max_iter, double *f, double *u,
    double *u_old) {
2     int j,i;
3     for (i = 1; i < N-1; i++) {
4         for (j = 1; j < N-1; j++) {
5             // Update u
6             u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i
    *N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
7         }
8     }
9 }
```

Algorithm 11: Algo. `jac_gpu1`.

.

6.2 Naive GPU Jacobi

The Naive Poisson kernel have been implemented by using one thread per grid point which enables the high parallelism of the device. The implementation uses global memory and line 2-3 shows how the upadted element is determined. The kernel is presented in algo. 12.

```
1 void __global__ jac_gpu2(int N, double delta, int max_iter, double *f, double *u,
    double *u_old) {
2     int j = blockIdx.x * blockDim.x + threadIdx.x;
3     int i = blockIdx.y * blockDim.y + threadIdx.y;
4     if (i < (N-1) && j < (N-1) && i > 0 && j > 0) {
5         u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N +
     (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
6     }
7 }
```

Algorithm 12: Algo. `jac_gpu2`.

.

The kernel is called by following launch paramters: `<<<dim_grid,dim_block>>>jac_gpu2` which creates a 2D thread blocks. The 2D grid and block size are given by:

$$dim3 \quad dim\_grid\left(\frac{N+bs-1}{bs}, \frac{N+bs-1}{bs}\right) \tag{1}$$

$$dim3 \quad dim\_block\,(bs, bs) \tag{2}$$

where $bs = 16$ is the number of threads in each block.

6.2.1 `nvvp`

The `nvvp` analyzing tool tells that the `jac_gpu2()` has a "Low Compute Utilization" $\approx 16\%$, presented in figure 18. This is as expected, as all memory is fetched globally in the implementation, and as only few floating point operations are performed every time memory is retrieved. Due to this difficulty, the Jacobi method is memory bound. This could be optimized by splitting up the copying, such that the algorithm could copy and compute simultaneously and hereby reduce the computation time. By using more specialized analysis tools within `nvvp`, a bandwidth limitation is proposed. This limits also supports the claim that the problem is memory bound. The solution to the Poisson problem using this algorithm is presented in 19c in the appendix.

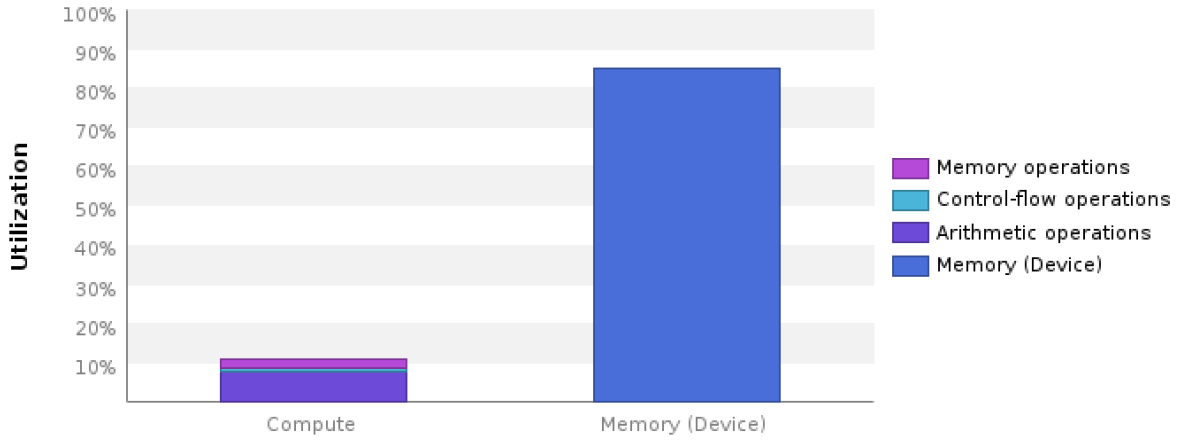


Figure 18: The `nvvp` analysis of `jac_gpu2()`. This reports that the algorithm is memory bound as the percentage of utilized memory is much bigger than the utilization of the computation.

6.3 Multiple GPU Jacobi

The third version the Jacobi version use multiply (two) GPUs. The problem is hereby split equally between the devices. It has been chosen to create a horizontal split.
The kernels, `jac_gpu3`, used to solve the Poisson problem is presented in algorithm 13.
The `cudaDeviceEnablePeerAccess()` method is used to solve the boarder issues between the top and bottom problem as the Jacobi iteration uses the adjacent grid points when updating an element. See the complete source implementation, algo. 18 in the appendix.

```
1  void __global__ jac_gpu3_d0(int N, double delta, int max_iter, double *f, double *
       u, double *u_old, double *d1_u_old) {
2      int j = blockIdx.x * blockDim.x + threadIdx.x;
3      int i = blockIdx.y * blockDim.y + threadIdx.y;
4      if (i < (N/2-1) && j < (N-1) && i > 0 && j > 0) {
5          u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N +
       (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
6      }
7      else if (i == (N/2-1) && j < (N-1) && j > 0) {
8          u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + d1_u_old[j] + u_old[i*N + (j-1)]
       + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
9      }
10 }
11
12 void __global__ jac_gpu3_d1(int N, double delta, int max_iter, double *f, double *
       u, double *u_old, double *d0_u_old) {
13     int j = blockIdx.x * blockDim.x + threadIdx.x;
14     int i = blockIdx.y * blockDim.y + threadIdx.y;
15     if (i < (N/2-1) && j < (N-1) && i > 0 && j > 0) { // i < N/2
16         u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N +
       (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
17     }
18     else if (i == 0 && j < (N-1) && j > 0) {
```

```
19          u[i*N + j] = 0.25 * (d0_u_old[(N/2-1)*N + j] + u_old[(i+1)*N+j] + u_old[i*
        N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
20      }
21  }
```

Algorithm 13: Algo. `jac_gpu3`.

The kernel is, as in `jac_gpu2`, called by the following launch paramters: `<<<dim_grid,dim_block>>>` `jac_gpu3`. Noticeable the grid is $N/2$ in the second axis in order to support the dimensions of the each subproblem. The 2D grid and block size are given by:

$$dim3 \quad dim\_grid\left(\frac{N+bs-1}{bs}, \frac{N/2+bs-1}{bs}\right) \tag{3}$$

$$dim3 \quad dim\_block\,(bs, bs) \tag{4}$$

6.4 Speedup

The expected compute speedup is 8x[1] when deploying the algorithm on the GPU compared to the CPU. Table 2 reports the achieved speedups for the three implementations.

Algo.	Speedup
`cpu`	1.0000x
`gpu1`	0.0021x
`gpu2`	10.3818x
`gpu3`	16.5419x

Table 2: This table present the speed-ups of `jac_gpu1`, `jac_gpu2` and, `jac_gpu3` in reference to the fastest CPU version from assignment 2, `cpu()`

As expected the `jac_gpu1` does not gain any improvements. The reason why is the lack of parallelism. Hence there is only launched one block with one kernel.
The speedup gained by `jac_gpu2` is $\approx 10x$ which slightly higher than the expected compute speedup. This can be caused by a version of the OpenMP implementation, which is not fully optimized. If the implementation is sub-optimal the comparison is not fair to the fully parallel implementation on the GPU.
When splitting the problem into two subproblems the expected speedup is not $2x$ between `jac_gpu2` and `jac_gpu3`. The reason is due to the nature of the Jacobi algorithm. There needs to be shared global memory access between the two devices in order to update the "middle" horizontal boarders elements. The shared global memory, accessed by peer access, is transferred on the PCIe express bus which introduce a latency and therefore not able to scale $2x$.

[1] PerformanceTuningIntro.pdf, slide 20.

7 Conclusion

In the part concerning matrix-matrix multiplication, the analysis shows that there is a significant performance gain in using GPUs. used in `lib` to four. It is especially seen how shared memory improves the performance by considering `gpu2` and `gpu5`. Both algorithms uses one thread pr element in C, but the performance of `gpu5` is much higher, as the algorithm takes advantage of shared memory. The measured speed-ups might be overestimated due to using a driver, that limits the number of threads. If the updated driver had been used, the performance of `lib` might have been much better. Regarding the implementation of the algorithms, the `nvvp` profiler analysis showed that relatively few registers are used pr. thread. This limits the performance, as a lot of time is used on copying memory back and forth. This alone was not enough to explain the amount of time used on memory on the device, why it was discovered that every time an addition was made to an element in C, this was stored and fetched from the global memory. The kernels could have been optimized by saving the element of C locally while computing it and then writing it to the global variable `d_C`.
The experimentally achieved speed-ups reported table 2 indicates a very good reason for performing this scientific computing problem on a many core multiprocessor such as a GPU or multiply GPUs compared to a traditional multi core CPU using a single treads pr. core. The enhancement by performing the Jacobi algorithm on the GPU is $\approx 10x$. Splitting the problem into two subproblems scales further to $\approx 16x$.

References

[1] NVIDIA corporation. "WHAT IS GPU-ACCELERATED COMPUTING?" In: URL: `http://www.nvidia.com/object/what-is-gpu-computing.html`.

Appendices

A lib

```
extern "C" {
void matmult_lib(int M, int N, int K, double *A, double *B, double *C) {
    int LDA = fmax(1,K); // leading dimension of A
    int LDB = fmax(1,N); // leading dimension of B
    int LDC = fmax(1,N); // leading dimension of C
    double alpha = 1.0, beta = 0.0; //
    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,M,N,K,alpha,A,LDA,B,LDB,
    beta,C,LDC);
}
}
```

Algorithm 14: Implementation of the library function `cblas_dgemm`

B jac_cpu()

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "func.h"

int main(int argc, char *argv[]) {

    int max_iter, N,i,j;

    if (argc == 3) {
        N = atoi(argv[1]) + 2;
        max_iter = atoi(argv[2]);
    }
    else {
        // use default N
        N = 128 + 2;
        max_iter = 5000;
    }
    double delta = 2.0/N;

    // allocate mem
    double *f, *u, *u_old;
    int size_f = N * N * sizeof(double);
    int size_u = N * N * sizeof(double);
    int size_u_old = N * N * sizeof(double);

    f = (double *)malloc(size_f);
    u = (double *)malloc(size_u);
    u_old = (double *)malloc(size_u_old);

    if (f == NULL || u == NULL || u_old ==NULL) {
       fprintf(stderr, "memory allocation failed!\n");
       return(1);
    }

    // initilize boarder
    #pragma omp parallel shared(f,u,u_old,N) private(i,j)
    {
    #pragma omp for
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            if (i >= N * 0.5  &&  i <= N * 2.0/3.0  &&  j >= N * 1.0/6.0  &&  j <=
     N * 1.0/3.0)
                f[i*N + j] = 200.0;
            else
                f[i*N + j] = 0.0;

            if (i == (N - 1) || i == 0 || j == (N - 1)){
                u[i*N + j] = 20.0;
                u_old[i*N + j] = 20.0;
            }
            else{
                u[i*N + j] = 0.0;
                u_old[i*N + j] = 0.0;
```

```
            }
        }
    }

    } /* end of parallel region */

    // do program
    double time_compute = omp_get_wtime();
    jac_cpu(N, delta, max_iter,f,u,u_old);
    double tot_time_compute = omp_get_wtime() - time_compute;
    // end program


    // stats
    double GB = 1.0e-09;
    double flop = max_iter * (double)(N-2) * (double)(N-2) * 10.0;
    double gflops  = (flop / tot_time_compute) * GB;
    double memory  = size_f + size_u + size_u_old;
    double memoryGBs  = memory * GB * (1 / tot_time_compute);

    printf("%g\t", memory); // footprint
    printf("%g\t", gflops); // Gflops
    printf("%g\t", memoryGBs); // bandwidth GB/s
    printf("%g\t", tot_time_compute); // total time
    printf("%g\t", 0); // I/O time
    printf("%g\t", tot_time_compute); // compute time
    printf("# cpu\n");

    //write_result(u, N, delta, "./../../analysis/pos/jac_cpu.txt");

    // free mem
    free(f);
    free(u);
    free(u_old);
    // end program
    return(0);
}
```

Algorithm 15: Algo. `jac_cpu()`.

C jac_gpu1()

```
extern "C" {
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void write_result(double *U, int N, double delta, char filename[40]) {
    double u, y, x;
    FILE *matrix=fopen(filename, "w");
    for (int i = 0; i < N; i++) {
        x = -1.0 + i * delta + delta * 0.5;
        for (int j = 0; j < N; j++) {
            y = -1.0 + j * delta + delta * 0.5;
            u = U[i*N + j];
            fprintf(matrix, "%g\t%g\t%g\n", x,y,u);
        }
    }
    fclose(matrix);
}
}

const int device0 = 0;

void __global__ jac_gpu1(int N, double delta, int max_iter, double *f, double *u,
    double *u_old) {
    int j,i;
    for (i = 1; i < N-1; i++) {
        for (j = 1; j < N-1; j++) {
            // Update u
            u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i
    *N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
        }
    }
}


int main(int argc, char *argv[]) {

    // warm up:
    double *dummy_d;
    cudaSetDevice(device0);
    cudaMalloc((void**)&dummy_d, 0);

    int i, j, N, max_iter;

    if (argc == 3) {
        N = atoi(argv[1]) + 2;
        max_iter = atoi(argv[2]);
    }
    else {
        // use default N
        N = 128 + 2;
        max_iter = 5000;
    }
    double delta = 2.0/N;
```

```
53
54      // allocate mem
55      double *h_f, *h_u, *h_u_old, *d_f, *d_u, *d_u_old;
56
57      int size_f = N * N * sizeof(double);
58      int size_u = N * N * sizeof(double);
59      int size_u_old = N * N * sizeof(double);
60
61      //Allocate memory on device
62      cudaSetDevice(device0);
63      cudaMalloc((void**)&d_f, size_f);
64      cudaMalloc((void**)&d_u, size_u);
65      cudaMalloc((void**)&d_u_old, size_u_old);
66      //Allocate memory on host
67      cudaMallocHost((void**)&h_f, size_f);
68      cudaMallocHost((void**)&h_u, size_u);
69      cudaMallocHost((void**)&h_u_old, size_u_old);
70
71      // initilize boarder
72      for (i = 0; i < N; i++){
73          for (j = 0; j < N; j++){
74              if (i >= N * 0.5  &&  i <= N * 2.0/3.0  &&  j >= N * 1.0/6.0  &&  j <=
     N * 1.0/3.0)
75                  h_f[i*N + j] = 200.0;
76              else
77                  h_f[i*N + j] = 0.0;
78
79              if (i == (N - 1) || i == 0 || j == (N - 1)){
80                  h_u[i*N + j] = 20.0;
81                  h_u_old[i*N + j] = 20.0;
82              }
83              else{
84                  h_u[i*N + j] = 0.0;
85                  h_u_old[i*N + j] = 0.0;
86              }
87          }
88      }
89
90      //Copy memory host -> device
91      double time_tmp = omp_get_wtime();
92      cudaMemcpy(d_f, h_f, size_f, cudaMemcpyHostToDevice);
93      cudaMemcpy(d_u, h_u, size_u_old, cudaMemcpyHostToDevice);
94      cudaMemcpy(d_u_old, h_u_old, size_u_old, cudaMemcpyHostToDevice);
95      double time_IO_1 = omp_get_wtime() - time_tmp;
96
97      // do program
98      int k = 0;
99      double *temp, time_compute = omp_get_wtime();
100     while (k < max_iter) {
101         // Set u_old = u
102         temp = d_u;
103         d_u = d_u_old;
104         d_u_old = temp;
105         jac_gpu1<<<1,1>>>(N, delta, max_iter, d_f, d_u, d_u_old);
106         cudaDeviceSynchronize();
107         k++;
```

```
108     }/* end while */
109     double tot_time_compute = omp_get_wtime() - time_compute;
110     // end program
111 
112     //Copy memory host -> device
113     time_tmp = omp_get_wtime();
114     cudaMemcpy(h_u, d_u, size_u, cudaMemcpyDeviceToHost);
115     double time_IO_2 = omp_get_wtime() - time_tmp;
116 
117     tot_time_compute += time_IO_1 + time_IO_2;
118 
119     // stats
120     double GB = 1.0e-09;
121     double flop = max_iter * (double)(N-2) * (double)(N-2) * 10.0;
122     double gflops  = (flop / tot_time_compute) * GB;
123     double memory  = size_f + size_u + size_u_old;
124     double memoryGBs  = memory * GB * (1 / tot_time_compute);
125 
126     printf("%g\t", memory); // footprint
127     printf("%g\t", gflops); // Gflops
128     printf("%g\t", memoryGBs); // bandwidth GB/s
129     printf("%g\t", tot_time_compute); // total time
130     printf("%g\t", time_IO_1 + time_IO_2); // I/O time
131     printf("%g\t", tot_time_compute); // compute time
132     printf("# gpu1\n");
133 
134     //write_result(h_u, N, delta, "./../../analysis/pos/jac_gpu1.txt");
135 
136     // free mem
137     cudaFree(d_f), cudaFree(d_u), cudaFree(d_u_old);
138     cudaFreeHost(h_f), cudaFreeHost(h_u), cudaFreeHost(h_u_old);
139     // end program
140     return(0);
141 }
```

Algorithm 16: Algo. `jac_gpu1()`.

D jac_gpu2()

```
extern "C" {
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void write_result(double *U, int N, double delta, char filename[40]) {
    double u, y, x;
    FILE *matrix=fopen(filename, "w");
    for (int i = 0; i < N; i++) {
        x = -1.0 + i * delta + delta * 0.5;
        for (int j = 0; j < N; j++) {
            y = -1.0 + j * delta + delta * 0.5;
            u = U[i*N + j];
            fprintf(matrix, "%g\t%g\t%g\n", x,y,u);
        }
    }
    fclose(matrix);
}
}

const int device0 = 0;
#define BLOCK_SIZE 16

void __global__ jac_gpu2(int N, double delta, int max_iter, double *f, double *u,
    double *u_old) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < (N-1) && j < (N-1) && i > 0 && j > 0) {
        u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N +
     (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
    }
}

int main(int argc, char *argv[]) {

    // warm up:
    double *dummy_d;
    cudaSetDevice(device0);
    cudaMalloc((void**)&dummy_d, 0);

    int max_iter, N,i,j;

    if (argc == 3) {
        N = atoi(argv[1]) + 2;
        max_iter = atoi(argv[2]);
    }
    else {
        // use default N
        N = 128 + 2;
        max_iter = 5000;
    }
    double delta = 2.0/N;

    // allocate mem
```

```
53      double *h_f, *h_u, *h_u_old, *d_f, *d_u, *d_u_old;
54
55      int size_f = N * N * sizeof(double);
56      int size_u = N * N * sizeof(double);
57      int size_u_old = N * N * sizeof(double);
58
59      //Allocate memory on device
60      cudaSetDevice(device0);
61      cudaMalloc((void**)&d_f, size_f);
62      cudaMalloc((void**)&d_u, size_u);
63      cudaMalloc((void**)&d_u_old, size_u_old);
64      //Allocate memory on host
65      cudaMallocHost((void**)&h_f, size_f);
66      cudaMallocHost((void**)&h_u, size_u);
67      cudaMallocHost((void**)&h_u_old, size_u_old);
68
69      // initilize boarder
70      for (i = 0; i < N; i++){
71          for (j = 0; j < N; j++){
72              if (i >= N * 0.5  &&  i <= N * 2.0/3.0  &&  j >= N * 1.0/6.0  &&  j <=
        N * 1.0/3.0)
73                  h_f[i*N + j] = 200.0;
74              else
75                  h_f[i*N + j] = 0.0;
76
77              if (i == (N - 1) || i == 0 || j == (N - 1)){
78                  h_u[i*N + j] = 20.0;
79                  h_u_old[i*N + j] = 20.0;
80              }
81              else{
82                  h_u[i*N + j] = 0.0;
83                  h_u_old[i*N + j] = 0.0;
84              }
85          }
86      }
87
88      //Copy memory host -> device
89      double time_tmp = omp_get_wtime();
90      cudaMemcpy(d_f, h_f, size_f, cudaMemcpyHostToDevice);
91      cudaMemcpy(d_u, h_u, size_u_old, cudaMemcpyHostToDevice);
92      cudaMemcpy(d_u_old, h_u_old, size_u_old, cudaMemcpyHostToDevice);
93      double time_IO_1 = omp_get_wtime() - time_tmp;
94
95      // do program
96      int k = 0;
97      dim3 dim_grid(((N+BLOCK_SIZE-1) / BLOCK_SIZE), ((N+BLOCK_SIZE-1) / BLOCK_SIZE)
        );
98      dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE);
99      double *temp, time_compute = omp_get_wtime();
100     while (k < max_iter) {
101         // Set u_old = u
102         temp = d_u;
103         d_u = d_u_old;
104         d_u_old = temp;
105         jac_gpu2<<<dim_grid,dim_block>>>(N, delta, max_iter, d_f, d_u, d_u_old);
106         cudaDeviceSynchronize();
```

```
        k++;
    }/* end while */
    double tot_time_compute = omp_get_wtime() - time_compute;
    // end program

    //Copy memory host -> device
    time_tmp = omp_get_wtime();
    cudaMemcpy(h_u, d_u, size_u, cudaMemcpyDeviceToHost);
    double time_IO_2 = omp_get_wtime() - time_tmp;

    tot_time_compute += time_IO_1 + time_IO_2;

    // stats
    double GB = 1.0e-09;
    double flop = max_iter * (double)(N-2) * (double)(N-2) * 10.0;
    double gflops  = (flop / tot_time_compute) * GB;
    double memory  = size_f + size_u + size_u_old;
    double memoryGBs  = memory * GB * (1 / tot_time_compute);

    printf("%g\t", memory); // footprint
    printf("%g\t", gflops); // Gflops
    printf("%g\t", memoryGBs); // bandwidth GB/s
    printf("%g\t", tot_time_compute); // total time
    printf("%g\t", time_IO_1 + time_IO_2); // I/O time
    printf("%g\t", tot_time_compute); // compute time
    printf("# gpu2\n");

    //write_result(h_u, N, delta, "./../../analysis/pos/jac_gpu2.txt");

    // free mem
    cudaFree(d_f), cudaFree(d_u), cudaFree(d_u_old);
    cudaFreeHost(h_f), cudaFreeHost(h_u), cudaFreeHost(h_u_old);
    // end program
    return(0);
}
```

Algorithm 17: Algo. `jac_gpu2()`.

E jac_gpu3()

```
extern "C" {
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void write_result(double *U, int N, double delta, char filename[40]) {
    double u, y, x;
    FILE *matrix=fopen(filename, "w");
    for (int i = 0; i < N; i++) {
        x = -1.0 + i * delta + delta * 0.5;
        for (int j = 0; j < N; j++) {
            y = -1.0 + j * delta + delta * 0.5;
            u = U[i*N + j];
            fprintf(matrix, "%g\t%g\t%g\n", x,y,u);
        }
    }
    fclose(matrix);
}
}

const int device0 = 0;
const int device1 = 1;
#define BLOCK_SIZE 16


void __global__ jac_gpu3_d0(int N, double delta, int max_iter, double *f, double *u, double *u_old, double *d1_u_old) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < (N/2-1) && j < (N-1) && i > 0 && j > 0) {
        u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
    }
    else if (i == (N/2-1) && j < (N-1) && j > 0) {
        u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + d1_u_old[j] + u_old[i*N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
    }
}

void __global__ jac_gpu3_d1(int N, double delta, int max_iter, double *f, double *u, double *u_old, double *d0_u_old) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < (N/2-1) && j < (N-1) && i > 0 && j > 0) { // i < N/2
        u[i*N + j] = 0.25 * (u_old[(i-1)*N + j] + u_old[(i+1)*N + j] + u_old[i*N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
    }
    else if (i == 0 && j < (N-1) && j > 0) {
        u[i*N + j] = 0.25 * (d0_u_old[(N/2-1)*N + j] + u_old[(i+1)*N+j] + u_old[i*N + (j-1)] + u_old[i*N + (j+1)] + delta*delta*f[i*N + j]);
    }
}

int main(int argc, char *argv[]) {
```

```
49
50      // warm up:
51      double *dummy_d;
52      cudaSetDevice(device0);
53      cudaMalloc((void**)&dummy_d, 0);
54      cudaSetDevice(device1);
55      cudaMalloc((void**)&dummy_d, 0);
56
57      int max_iter, N,i,j;
58
59      if (argc == 3) {
60          N = atoi(argv[1]) + 2;
61          max_iter = atoi(argv[2]);
62      }
63      else {
64          // use default N
65          N = 128 + 2;
66          max_iter = 5000;
67      }
68      double delta = 2.0/N;
69
70      // allocate mem
71      double *h_f, *h_u, *h_u_old;
72      double *d0_f, *d0_u, *d0_u_old, *d1_f, *d1_u, *d1_u_old;
73
74      int size_f = N * N * sizeof(double);
75      int size_u = N * N * sizeof(double);
76      int size_u_old = N * N * sizeof(double);
77      int size_f_p2 = N*N/2;
78      int size_u_p2 = N*N/2;
79      int size_u_old_p2 = N*N/2;
80
81      //Allocate memory on device
82      cudaSetDevice(device0);
83      cudaMalloc((void**)&d0_f, size_f/2);
84      cudaMalloc((void**)&d0_u, size_u/2);
85      cudaMalloc((void**)&d0_u_old, size_u_old/2);
86      cudaSetDevice(device1);
87      cudaMalloc((void**)&d1_f, size_f/2);
88      cudaMalloc((void**)&d1_u, size_u/2);
89      cudaMalloc((void**)&d1_u_old, size_u_old/2);
90      //Allocate memory on host
91      cudaMallocHost((void**)&h_f, size_f);
92      cudaMallocHost((void**)&h_u, size_u);
93      cudaMallocHost((void**)&h_u_old, size_u_old);
94
95      // initialize boarder
96      for (i = 0; i < N; i++){
97          for (j = 0; j < N; j++){
98              if (i >= N * 0.5  &&  i <= N * 2.0/3.0  &&  j >= N * 1.0/6.0  &&  j <=
        N * 1.0/3.0)
99                  h_f[i*N + j] = 200.0;
100             else
101                 h_f[i*N + j] = 0.0;
102
103             if (i == (N - 1) || i == 0 || j == (N - 1)){
```

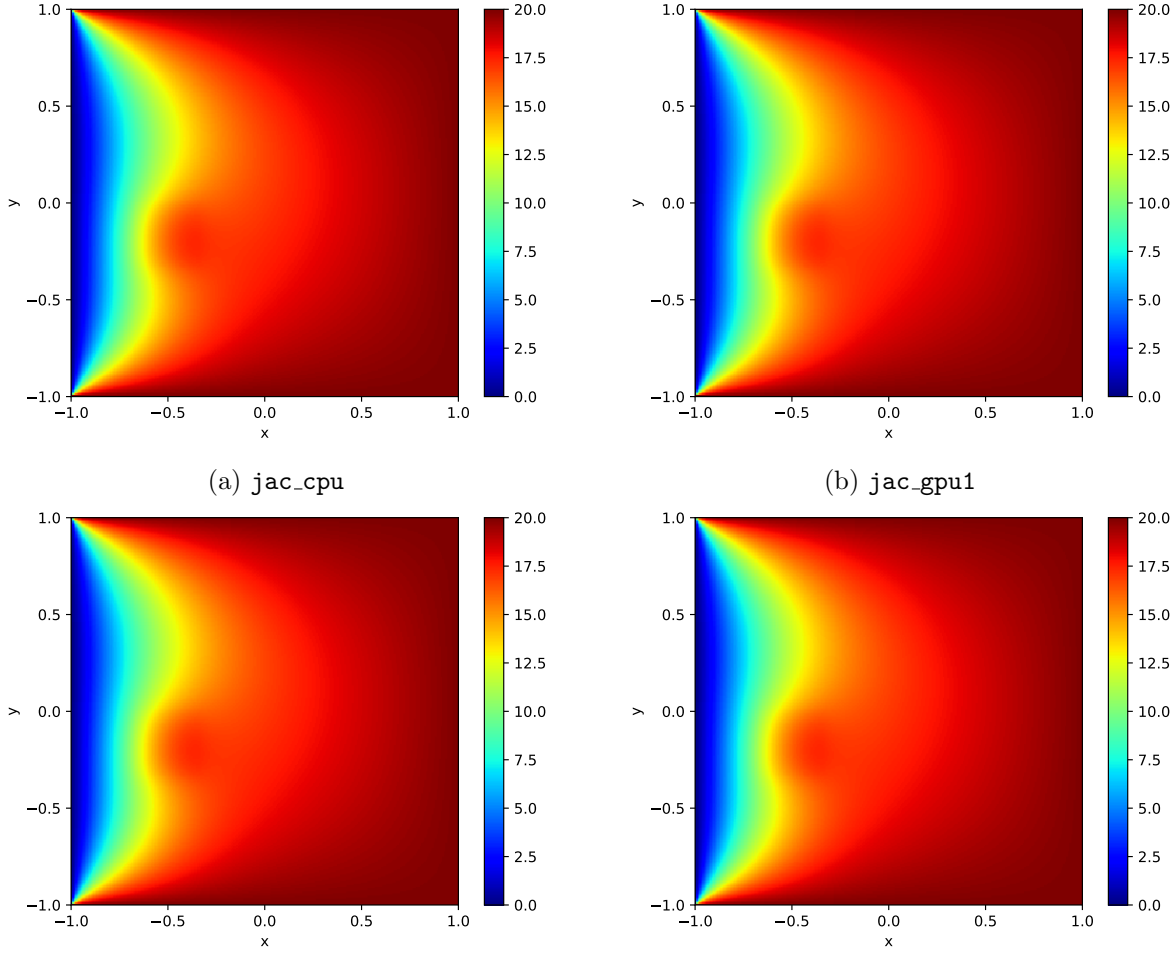
```
                h_u[i*N + j] = 20.0;
                h_u_old[i*N + j] = 20.0;
            }
            else{
                h_u[i*N + j] = 0.0;
                h_u_old[i*N + j] = 0.0;
            }
        }
    }

    //Copy memory host -> device
    double time_tmp = omp_get_wtime();
    cudaSetDevice(device0);
    cudaMemcpy(d0_f, h_f, size_f/2, cudaMemcpyHostToDevice);
    cudaMemcpy(d0_u, h_u, size_u/2, cudaMemcpyHostToDevice);
    cudaMemcpy(d0_u_old, h_u_old, size_u_old/2, cudaMemcpyHostToDevice);
    cudaSetDevice(device1);
    cudaMemcpy(d1_f, h_f + size_f_p2, size_f/2, cudaMemcpyHostToDevice);
    cudaMemcpy(d1_u, h_u + size_u_p2, size_u/2, cudaMemcpyHostToDevice);
    cudaMemcpy(d1_u_old, h_u_old + size_u_old_p2, size_u_old/2,
    cudaMemcpyHostToDevice);
    double time_IO_1 = omp_get_wtime() - time_tmp;

    // peer enable
    cudaSetDevice(device0);
    cudaDeviceEnablePeerAccess(device1,0);
    cudaSetDevice(device1);
    cudaDeviceEnablePeerAccess(device0,0);

    // do program
    int k = 0;
    dim3 dim_grid(((N +BLOCK_SIZE-1) / BLOCK_SIZE), ((N/2+BLOCK_SIZE-1) /
    BLOCK_SIZE));
    dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE);
    double *temp_p;
    double time_compute = omp_get_wtime();
    while (k < max_iter) {
        // Set u_old = u device 0
        temp_p = d0_u;
        d0_u = d0_u_old;
        d0_u_old = temp_p;
        // Set u_old = u device 0
        temp_p = d1_u;
        d1_u = d1_u_old;
        d1_u_old = temp_p;

        cudaSetDevice(device0);
        jac_gpu3_d0<<<dim_grid, dim_block>>>(N, delta, max_iter, d0_f, d0_u,
    d0_u_old, d1_u_old);
        cudaSetDevice(device1);
        jac_gpu3_d1<<<dim_grid, dim_block>>>(N, delta, max_iter, d1_f, d1_u,
    d1_u_old, d0_u_old);
        cudaDeviceSynchronize();
        cudaSetDevice(device0);
        cudaDeviceSynchronize();
        k++;
```

```
156     }/* end while */
157     double tot_time_compute = omp_get_wtime() - time_compute;
158     // end program
159 
160     //Copy memory host -> device
161     time_tmp = omp_get_wtime();
162     cudaSetDevice(device0);
163     cudaMemcpy(h_u, d0_u, size_u/2, cudaMemcpyDeviceToHost);
164     cudaSetDevice(device1);
165     cudaMemcpy(h_u + size_u_p2, d1_u, size_u/2, cudaMemcpyDeviceToHost);
166     double time_IO_2 = omp_get_wtime() - time_tmp;
167 
168     tot_time_compute += time_IO_1 + time_IO_2;
169 
170     // stats
171     double GB = 1.0e-09;
172     double flop = max_iter * (double)(N-2) * (double)(N-2) * 10.0;
173     double gflops  = (flop / tot_time_compute) * GB;
174     double memory  = size_f + size_u + size_u_old;
175     double memoryGBs  = memory * GB * (1 / tot_time_compute);
176 
177     printf("%g\t", memory); // footprint
178     printf("%g\t", gflops); // Gflops
179     printf("%g\t", memoryGBs); // bandwidth GB/s
180     printf("%g\t", tot_time_compute); // total time
181     printf("%g\t", time_IO_1 + time_IO_2); // I/O time
182     printf("%g\t", tot_time_compute); // compute time
183     printf("# gpu3\n");
184 
185     //write_result(h_u, N, delta, "./../../analysis/pos/jac_gpu3.txt");
186 
187     // peer enable
188     cudaSetDevice(device0);
189     cudaDeviceDisablePeerAccess(device1);
190     cudaSetDevice(device1);
191     cudaDeviceDisablePeerAccess(device0);
192 
193     // free mem
194     cudaFree(d0_f), cudaFree(d0_u), cudaFree(d0_u_old);
195     cudaFree(d1_f), cudaFree(d1_u), cudaFree(d1_u_old);
196     cudaFreeHost(h_f), cudaFreeHost(h_u), cudaFreeHost(h_u_old);
197     // end program
198     return(0);
199 }
```

Algorithm 18: Algo. `jac_gpu3()`.

F Visual Estimates of $u(x, y)$

The following plots in figure 19 visualizes the estimates of $u(x, y)$ for the four given approaches, `jac_cpu`, `jac_gpu1`, `jac_gpu2` and `jac_gpu3`. The algorithms have been running for 1000 iterations and for $N = 2048$.



(a) `jac_cpu` (b) `jac_gpu1`

(c) `jac_gpu2` (d) `jac_gpu3`

Figure 19: Estimate of the function $u(x, y)$.