High-Performance Computing

Student name and id: Marie Mørk {s112770} Anja Liljedahl Christensen{s162876} Andreas Vedel Jantzen {s162858} Anders Launer Bæk {s160159}

Hold name: matmult_MAAA

Hand-in: Matrix Multiplication

---

1 Summary

Different functions, performing matrix-matrix multiplication are implemented in C in order to compare performance. Cases considered are ordering of the nested loops, optimizers in the compiler and blocking. The implemented functions are compared with the CBLAS subroutine DGEMM. As memory in C is stored row-wise, it has been found that the most optimal ordering of nested loops is to have the column index in the most inner loop. The optimizers are found to enhance performance significantly. The blocked version is found to be less efficient than the optimized version of the most efficient ordering of the loops. This might be due to the implementation of the blocking in which as many columns as rows are considered at a time.

2 Statement of the problem

The goal of the assignment is to create a library of functions that perform matrix-matrix multiplications. Performance of the implementations are to be compared with the DGEMM matrix-matrix multiplication routine from the BLAS library. As input, each of the functions should take the dimensions of the matrices (M, N, K) as well as the matrices (A, B, C), according to figure 1. The problem is divided into four sub-problems;

- Wrap the call DGEMM into a function called `matmult_lib()`, that takes the same inputs as specified above.
- Implement a function that performs double precision matrix-matrix multiplication. The function should be named `matmult_nat()`.
- Functions for each of the permutations of the three nested loops needed to perform matrix multiplications. The performance of the permutations should be compared. Furthermore, the timings should be compared with and without different compiler optimizations. The functions should be named as `matmult_per()`, where `per` is the ordering of the nested loops for the given implementation, e.g. `nmk,knm`, ect.
- A blocked version of the matrix-matrix multiplication function named `matmult_blk()`. This function should take an extra input, `bs`, which specifies the blocking size.

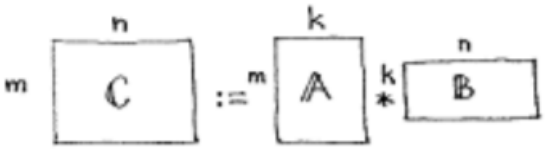


Figure 1: Visualisation of the matrix-matrix multiplications to be performed. The picture is taken from the problem description on DTU Inside.

3 Hardware and software

Specifications of the test environment are listed below:

- CPU information
  - Vendor ID: GenuineIntel
  - CPU family: 6
  - Model: 79
  - Model name: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
  - CPU MHz: 1297.656
  - L1 (d / i) cache: 32K
  - L2 cache: 256K
  - L3 cache: 30720K
- Compiler and Matrix implementations
  - The GCC compiler have been applied with and without compiler optimization using the compiler flag `-Ofast`.
  - The library function uses DGEMM from the CBLAS interface
  - Matrices are indexed in the following way: [m] [n]
  - The following line has been specified in the job script: "`#BSUB -n 1`" to make sure the batch job only uses one core.

4 Theory

Matrix multiplications are central operations in many numerical algorithms. Due to this, it is essential to make the operation fast to keep the computation time low. In order to achieve an optimal matrix-matrix multiplication algorithm, it is therefore important to take the ordering of nested loops and memory storage in the computer into account.

In C matrices are stored row-wise. Hence, when an element is fetched from memory the next couple of elements in the same row are collected as well, since they are stored in the same cache line. This is useful if the next couple of elements from the same row are needed in the near future, since

they have already been loaded into memory. If on the other hand the next element is extracted from another row, e.g. in the case of looping through a column, there will be a cache miss and a new cache line is loaded into memory. Thus, in the worst case there will be a cache miss each time a new element is needed. In the case of matrix multiplication it is necessary to loop through both rows and columns in multiple nested loops and hence there is a great chance of cache miss when the matrices are sufficiently large. If the matrices are small enough such that all three will fit in memory there should be no cache misses. For large scale matrices it is essential to optimize the implementation of the matrix-matrix multiplication in order to avoid too many cache misses.

Double precision matrix multiplication of square matrices requires space for $3 \cdot d^3$ doubles in memory, if the dimensions of the matrices is given by d. As storage of a double takes up 8 bits, the maximum dimensions for the matrices to fit in the memory of a given cache is found theoretically by calculating

$$d = \sqrt{\frac{L/8}{3}}$$

where L is the memory in the given cache. The maximum dimensions of square matrices fitting into each of the caches used are listed in table 1. In reality, the maximum allowed dimensions might be higher, as the program performs hardware prefetch.

Cache	Size [bytes]	Maximum dimension of Matrix
L_1	32K	36
L_2	256K	103
L_3	30720K	1131

Table 1: Maximum dimensions of square matrices fitting into each of the caches.

In the last sub problem of the report, blocking is implemented in order to optimize the memory usage in the algorithm. The idea behind blocking is to divide the iterations into smaller blocks, so that operations can be performed for parts of the matrices at a time, instead of considering the full size of the problem. This is especially useful when handling large problems, for which the memory needed to compute the full operation exceeds the bounds of the memory. When using an algorithm that relies on blocking it is important to use one of the respective compiler flags: `-O3` or `-Ofast`. By doing so the compiler analyses the code several lines ahead and is hereby able to optimize the code.

5 Algorithms

5.1 `matmult_lib`

The subroutine DGEMM from the library BLAS will be used to evaluate the performance of and compare with the different implementations of the matrix-matrix multiplication. For this purpose the CBLAS interface is used and wrapped into the function `matmult_lib` such that the library implementation of the matrix-matrix multiplication takes the same arguments as the other functions. The call to DGEMM using the CBLAS interface is `cblas_dgemm` which takes 14 arguments[1] as

[1] http://www.netlib.org/lapack/explore-html/dc/d18/cblas

shown in algorithm 1. From the inputs, the matrix multiplication is calculated as

$$C = \alpha \cdot A \cdot B + \beta \cdot C.$$

Here, α and β are arguments to be specified in the input. In this case α is set to 1 while β is set to 0 in order to get the wanted output. The other inputs in DGEMM (aside from M, N, K, A, B, and C) are the layout (set to rowmajor), whether the A and B matrices should be transposed (TRANSA and TRANSB) and the leading dimensions of A, B and C. As seen in the algorithm 1, the A and B matrices are not transposed, and the dimensions are set to K, N and N for A, B and C.

```
1  void matmult_lib(int M, int N, int K, double **A, double **B, double **C) {
2      int layout, TRANSA, TRANSB, LDA, LDB, LDC;
3      double alpha, beta;
4
5      layout = 101; // rowmajor
6      TRANSA = 111; // A is not transposed
7      TRANSB = 111; // B is not transposed
8
9      LDA = fmax(1,K); // leading dimension of A
10     LDB = fmax(1,N); // leading dimension of B
11     LDC = fmax(1,N); // leading dimension of C
12
13     alpha = 1.0; // no scaling
14     beta = 0.0; //
15
16     cblas_dgemm(layout, TRANSA, TRANSB, M, N, K, alpha, *A, LDA, *B, LDB, beta, *C
       , LDC);
17 }
```

Algorithm 1: `matmult_lib`

5.2 `matmult_nat`

The implementation of the `matmult_nat` function is listed in algorithm 2. For this function, it has been chosen to use the a MNK ordering of the nested loops. That is that the outer loop iterates over the dimension M, the second over N and the inner over K.

```
1  void matmult_nat(int M, int N, int K, double **A, double **B, double **C) {
2      for (int m = 0; m < M; m++) {
3          for (int n = 0; n < N; n++) {
4              C[m][n] = 0;
5              for (int k = 0; k < K; k++) {
6                  C[m][n] += A[m][k] * B[k][n];
7              }
8          }
9      }
10 }
```

Algorithm 2: `matmult_nat`

5.3 `matmult_per`

The `matmult_per` permutations are implemented as presented in algorithm 3, where the ordering of the nested loops are interchanged for each permutation of M, N and K. There exist six different

permutations of the `matmult_nat` function, namely MNK, NKM, MKN, KMN, MNK, and NMK. The `matmult_per` algorithm has been optimized in relation to the `matmult_nat` function. This has been done using stripmining, and setting all entries of C to zero using 1 for loop instead of 2 enables the compiler to perform vectorization.

```
1  void matmult_mnk(int M, int N, int K, double **A, double **B, double **C) {
2      for (int l = 0; l < M*N; l++) {
3          C[0][l] = 0;
4      }
5      for (int m = 0; m < M; m++) {
6          for (int n = 0; n < N; n++) {
7              for (int k = 0; k < K; k++) {
8                  C[m][n] += A[m][k] * B[k][n];
9              }
10         }
11     }
12 }
```

Algorithm 3: `matmult_per` for the `mnk` ordering of the loops.

5.4 `matmult_blk`

In order to implement blocking in the matrix-matrix multiplications, additional nested loops are needed. The algorithm used for blocking is based on one of the best performing versions of the `matmult_per` functions – in this case `matmult_mkn`. In algorithm 4 the function is presented. As mentioned in the problem description `matmult_blk` takes the additional input `bs`, which is the blocking size. In blocking, the matrices A, B and C are portioned into smaller matrices for which the multiplications are performed and then added to each other to get the full C matrix. In this implementation the blocks are chosen to be square matrices. This can be seen from the first three nested loops in which blocks of the matrices of size bs×bs are chosen. In the inner three nested loops the extracted matrices are multiplied and added to C. When using `matmult_blk` the compiler needs an optimizer flag, in order for the blocking to set in.

```
1  void matmult_blk(int M, int N, int K, double **A, double **B, double **C, int bs)
       {
2      // MKN
3      bs = fmax(1, fmin(bs, K));
4
5      for (int i = 0; i < M*N; i++) {
6          C[0][i] = 0.0;
7      }
8
9      for (int m0 = 0; m0 < M; m0 += bs) {
10         for (int k0 = 0; k0 < K; k0 += bs) {
11             for (int n0 = 0; n0 < N; n0 += bs) {
12                 for (int m = m0; m < fmin(m0 + bs, M); m++) {
13                     for (int k = k0; k < fmin(k0 + bs, K); k++) {
14                         for (int n = n0; n < fmin(n0 + bs, N); n++)
15                             C[m][n] += A[m][k] * B[k][n];
16                     }
17                 }
18             }
19         }
```

```
20      }
21  }
```

Algorithm 4: `matmult_blk`

6 Results

The implementation of the different functions were checked in a test-script to see if the C matrix was calculated as expected. Each of the functions `matmult_nat`, `matmult_per`, `matmult_blk`, and `matmult_lib` were tested using the square matrices:

$$A_s = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix} \qquad B_s = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

and the non-square matrices:

$$A_{ns} = \begin{pmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 6 \end{pmatrix} \qquad B_{ns} = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

which all returned

$$C_s = \begin{pmatrix} 6 & 6 \\ 12 & 12 \end{pmatrix} \qquad C_{ns} = \begin{pmatrix} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{pmatrix}$$

The calculation check sum provided by the driver has also been studied during the performance evaluation of the implemented functions, and in each of the cases the return value was 0 as expected.

Only square matrices are considered, e.g. N = M = K when the performance of the functions is evaluated.

Furthermore the performances of the above functions have been evaluated with and without the compiler flag `-Ofast` which enable compiler optimization, indicated as "opt". It has been decided to capture the hit and miss counters for the L1 cache and L2 cache in the first project.

6.1 Performance of `matmult_nat`

Figure 3 illustrates two realizations of the naive implementation of the `matmult_nat` function. The blue realization is the naive implementation without compiler optimization enabled and the black realization has compiler optimization enabled. By considering the differences between the realizations in figure 3 huge increases in performance can be achieved by using compiler optimization.

Table 2 reports the cache hits and cache miss for the `matmult_nat` function. It is observed that there are fewer hits in the optimized version for the L1 cache, while the misses are of equal size. The hypothesis would be that the number of hits increases for the optimized version, but it might be that the optimizer reduces the memory to be fetched, why the number of hits drops. Even though the number of hits reduces, we can still conclude that the optimized function performs significantly better than the non-optimized.

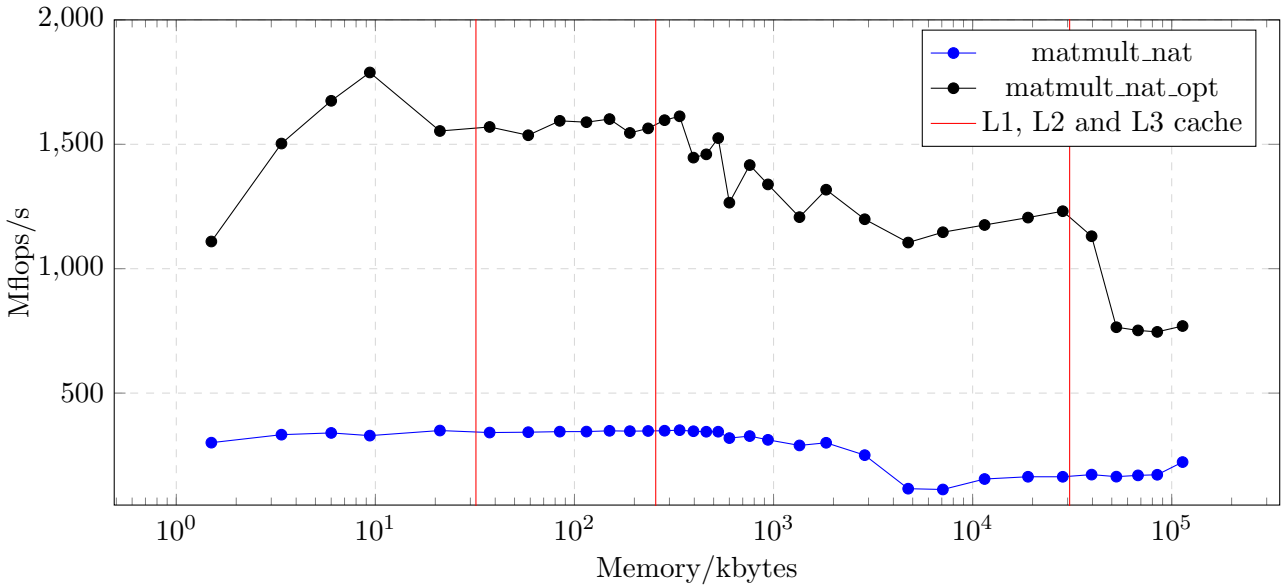


Figure 2: Performance of the `matmult_nat` function measured in Mflops/s as a function of kbytes.

	Non-optimized				Optimized			
	L1 in %		L2 in %		L1 in %		L2 in %	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_nat(50x50)`	10	0	0	0	0	0	0	0
`matmult_nat(850x850)`	129129	6072	5368	704	11651	6069	5524	544
`matmult_nat(1600x1600)`	861503	34583	134	34445	73553	31510	92	31416

Table 2: Hits and miss of the L1 and L2 cache for nat, presented in millions

The naive implementation of the matrix multiplication routine, `matmult_nat` is compared to the library subroutine DGEMM, which has been wrapped in `matmult`. In figure 3, the performance of DGEMM is shown with and without optimization in the compiler. Please note, that the axes is different from the one in 3.
It is seen from the figure that the performance of DGEMM is nearly the same whether or not an optimizer is used in the compiler. This is not surprising, as we expect the library function to be optimized in itself, and the wrapper funciton `matmult_lib` only calls the DGEMM subroutine and assigns values to some of the inputs. The performance of the library sub routine is significantly better than that of `matmult_nat`. In table 7, the hits and miss of the function are shown. All hits and misses are seen to be zero. This might be a consequence of the way the analyzer function handles the wrapper function. We would have expected the library function to have a very high number of hits and only few misses.

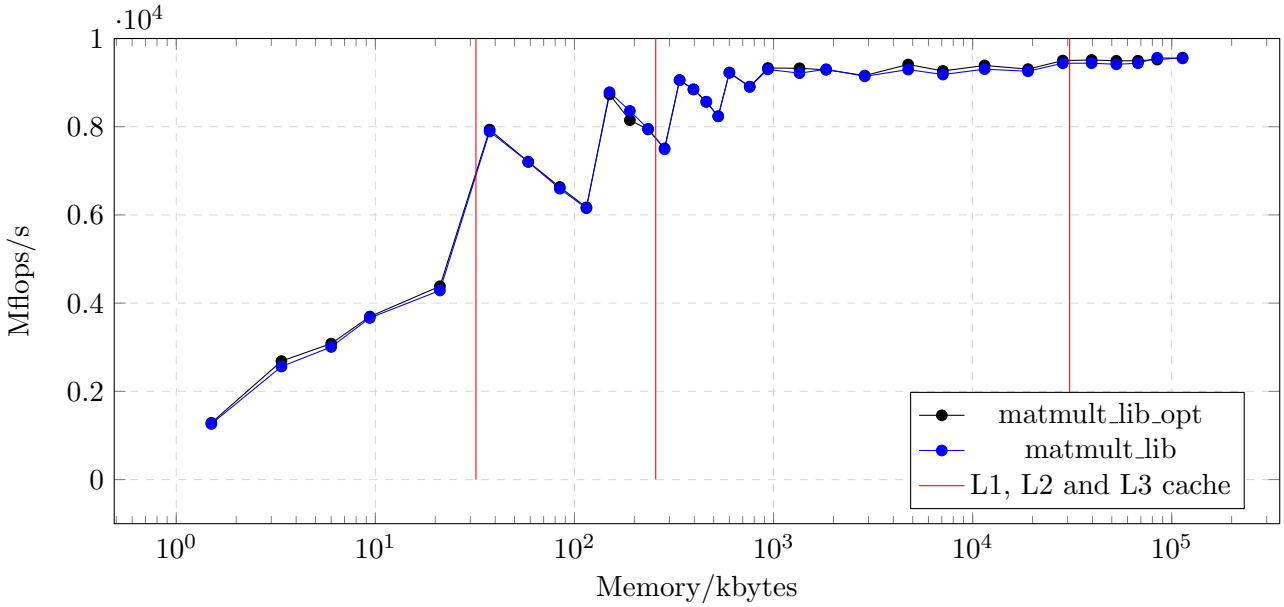


Figure 3: Performance of the `matmult_lib` function measured in Mflops/s as a function of kbytes.

	Non-optimized				Optimized			
	L1 in %		L2 in %		L1 in %		L2 in %	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_lib(50x50)`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`matmult_lib(850x850)`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`matmult_lib(1600x1600)`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3: Hits and miss of the L1 and L2 cache for lib, presented in millions

6.2 Optimal permutations of `matmult_nat`

As mentioned earlier it is possible to create several permutations of the naive `matmult_nat` function. The six possible (the naive is included) permutations are presented in figure 4 and the compiler optimized permutations in figure 5.

Figure 4 tells that `matmult_mkn` and `matmult_kmn` are the best performing permutations in the L3 cache. Hence having N as the last index, makes the matrix-matrix multiplication the fastest. This was also expected, because M and K are the row indexes of the input matrices and fits to the way memory is stored the most efficient way.
The permutations has calculated using an optimizer. This result is presented in 5. This confirms that `matmult_mkn` and `matmult_kmn` are the fastest permutations, having `matmult_mkn` a bit faster. When the optimizer is used the performance difference already occours in L1 and definately in L2. Hence, the combination of optimizer and having the right indexing makes `matmult_mkn` the fastest permutation.
Table 6, 5 and 4 presents the cache hits and misses for L1 and L2. The functions have been evaluated for 50, 850 and 1600 and the global variable has been set to perform 10 iterations in each loop in order to make comparable tests. Table 5 and 6 does again confirm that `matmult_mkn` and

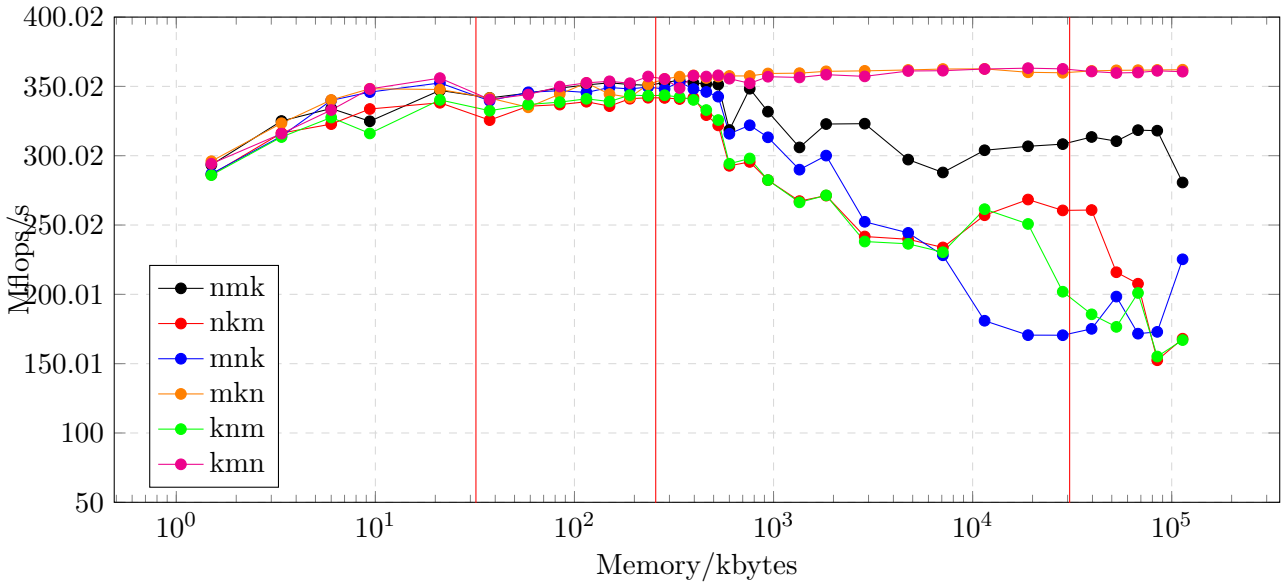


Figure 4: Performance of all permutations of the `matmult_per` functions measured in Mflops/s as a function of kbytes.

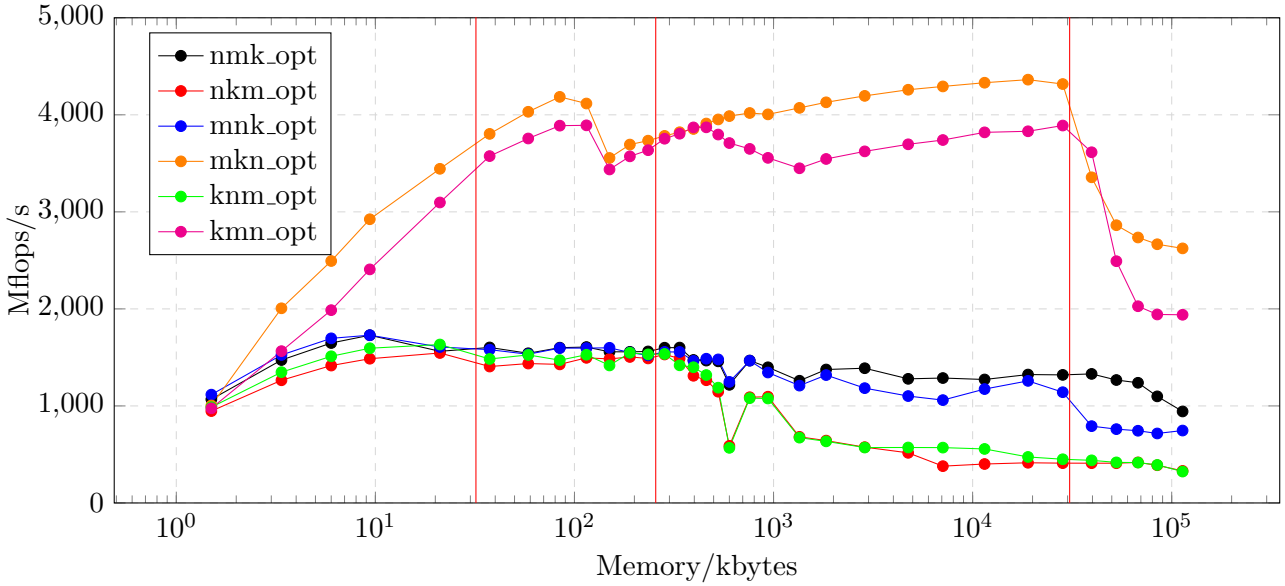


Figure 5: Performance of all permutations of the the compiler optimized `matmult_***` functions measured in Mflops/s as a function of kbytes.

`matmult_kmn` are the most optimal matrix-matrix multiplication indexing. This can be concluded because the ratio of their respective hits and misses are big, both for L1 and L2.

50x50	Non-optimized				Optimized			
	L1		L2		L1		L2	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_nmk`	10	0.0	0.0	0.0	0	0	0	0
`matmult_nkm`	10	0.0	0.0	0.0	0	0	0	0
`matmult_mnk`	10	0.0	0.0	0.0	0	0	0	0
`matmult_mkn`	10	0.0	0.0	0.0	0	0	0	0
`matmult_knm`	10	0.0	0.0	0.0	0	0	0	0
`matmult_kmn`	10	0.0	0.0	0.0	0	0	0	0

Table 4: Hits and miss of the L1 and L2 cache for 50x50

850x850	Non-optimized				Optimized			
	L1		L2		L1		L2	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_nmk`	129209	6094	6081	10	11291	6350	6338	7
`matmult_nkm`	123043	12138	11430	707	13814	13310	12714	593
`matmult_mnk`	129169	6072	5364	703	11851	6024	5448	573
`matmult_mkn`	135215	192	16	2	356	531	528	2
`matmult_knm`	123043	12144	11446	696	13693	13341	12765	576
`matmult_kmn`	135215	35	25	90		3524	525	509
18								

Table 5: Hits and miss of the L1 and L2 cache for 850x850

1600x1600	Non-optimized				Optimized			
	L1		L2		L1		L2	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_nmk`	861423	34484	86	34397	51572	33418	26	33390
`matmult_nkm`	819981	75809	121	75685	11379	64094	134	63958
`matmult_mnk`	861623	34644	124	34518	73593	31507	92	31412
`matmult_mkn`	901463	262	2464	16	22983	3790	3774	17
`matmult_knm`	820021	75806	89	75715	114034	64055	118	63937
`matmult_kmn`	901383	211	172	39	22943	3627	3521	105

Table 6: Hits and miss of the L1 and L2 cache for 1600x1600

6.3 Blocking version

As mentioned in the section on algorithms, the blocked version is based on `matmult_mkn` because it is the best performing permutation of the `matmult_nat` function, see figure 4.

Figure 6 illustrates a realization of the performance of `matmult_blk` as a function of `bs` for $M = N = K = 1500$. This problem size has been chosen for the experiment because three matrices of this size only fits in the L3 cache. Seeing as the blocking version performs operations on smaller blocks of the problem at a time, even though the full problem does not fit in the first two caches, the subproblems might fit and hence improve computation time. The red vertical lines in the figure

illustrates the theoretical maximum blocking size for the matrices to fit in the memory of cache L1, L2, and L3 respectively. The maximum sizes are calculated in the theory section and are equal to those presented in table 1.

Close to the boarder of L1, Mflops/s fluctuates a lot which can be caused by the hardware prefetching. Even though it is not visible on the logarithmic scale used for the blocking size, other experiments have shown that the drop in Mflops/s occurs just after the limit of L1. This is due to hardware prefetching and optimization of the compiler. The same behaviour is seen for L2. It is surprising that Mflops/s drops again well before the theoretical L3 limit, as we would have expected the drop to occur closer to the limit. The premature drop might be due to the implementation of blocking.

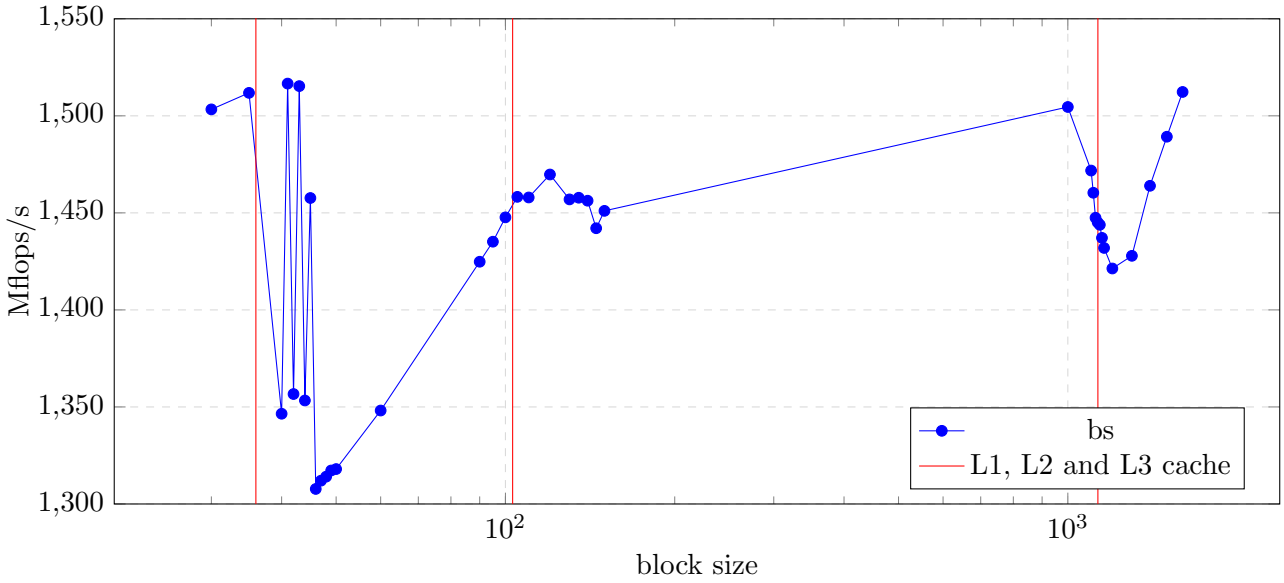


Figure 6: Performance of the `matmult_blk` function measured in Mflops/s as a function of block size.

Figure 7 illustrates the three realizations of the `matmult_blk_opt`. The three realizations illustrates that the choice of block size is depends on the problem size, e.g. the size of the matrix.

	bs = 36				bs = 103				bs = 1103			
	L1 in %		L2 in %		L1 in %		L2 in %		L1 in %		L2 in %	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
(50x50)	0	0	0	0	0	0	0	0	0	0	0	0
(850x850)	18458	26	26	1	17978	131	128	3	18418	10	10	0
(1600x1600)	123003	176	170	6	119800	931	912	20	121882	125	42	81

Table 7: Hits and misses of the L1 and L2 cache for blk, presented in millions

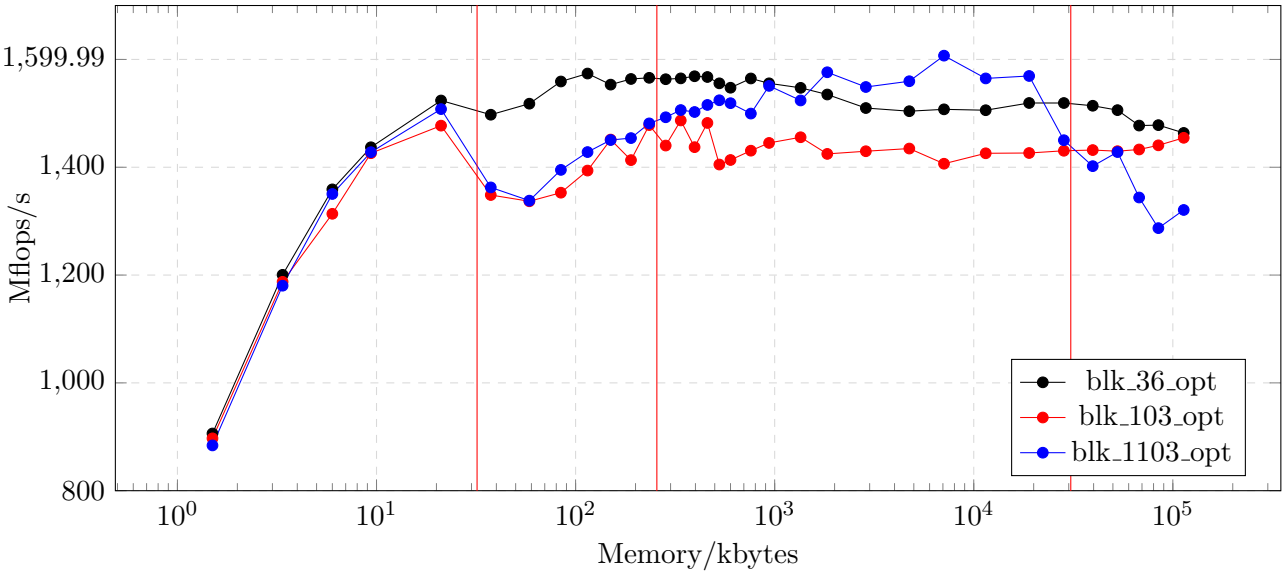


Figure 7: Performance of the `matmult_blk_opt` for different block sizes with compiler optimization.

6.4 Comparison

As can be seen in figure 8 below non of the implemented functions beat the CBLAS library function `cblas_dgemm`. The optimized permutations MKN and KMN of `matmult_nat` are the ones who come closest to `matmult_lib`. The blocked version of MKN performs worse than expected since using a blocking size of $bs = 103$ it should be almost similar to `matmult_mkn` for small problems, though of course 3 additional unnecessary outer loops are used (since the step size is the entire width of the matrices). It is possible that a different implementation of `matmult_blk` would result in better performance, e.g. if the extracted blocks were not quadratic but rectangular instead – I.e. if the number of columns were less than the number of rows. The library function increases in performance as the problem size increases as it should, since the library function is optimized for large scale problems. As discussed earlier the non-library functions experience drops in Mflops/s later than expected as a result of prefetching which is most apparent for the optimized versions of `matmult_mkn` and `matmult_kmn`.

Though the advantage of the blocked version was not visible in figure 8 it seems to pay off when considering the number of cache hits in table 8. I.e. the blocked version is the implementation with the highest number of cache hits in the L1 cache, while the naive implementation has the overall lowest number of hits and highest number of misses. The `analyzer` tool never managed to collect the number of cache misses and cache hits for any of the caches optimized or not for the library function, which is most likely caused by the implementation.

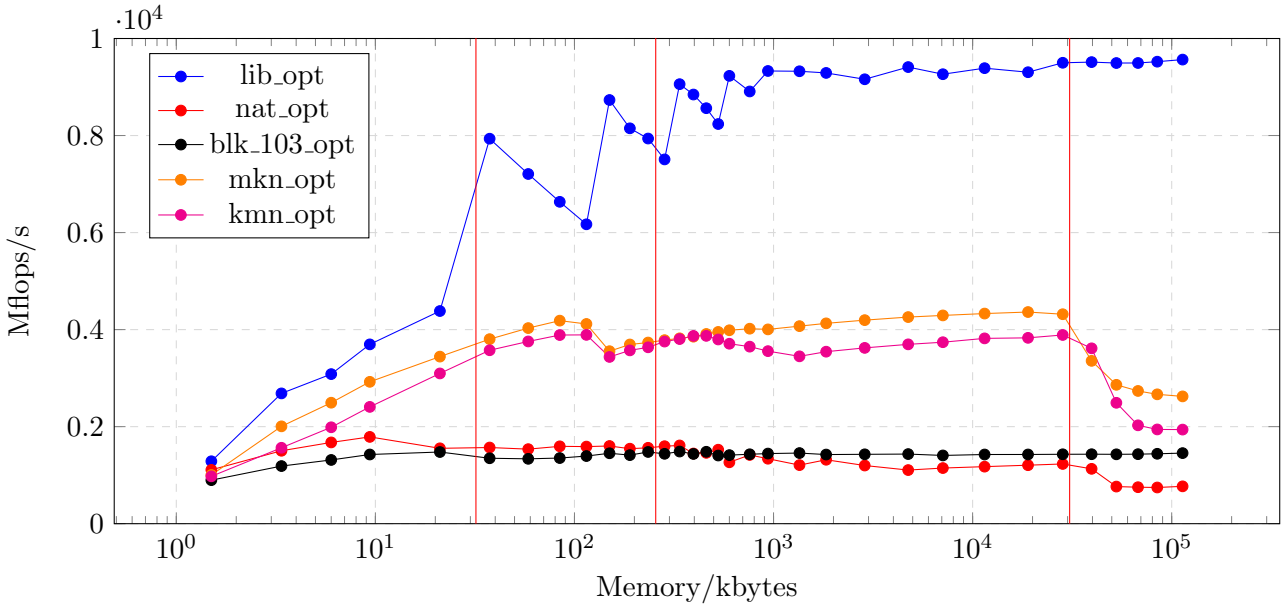


Figure 8: Performance of the compiler optimized `matmult_lib`, `matmult_nat`, `matmult_blk_1103` and `matmult_mkn` functions measured in Mflops/s as a function of kbytes.

1600x1600	Non-optimized				Optimized			
	L1		L2		L1		L2	
	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
`matmult_lib_opt`	0	0	0	0	0	0	0	0
`matmult_nat_opt`	861503	34584	134	34445	73554	31511	93	3142
`matmult_blk_103_opt`	1062545	1012	970	40	119800	931	912	20
`matmult_mkn_opt`	901464	262	246	16	22983	3790	3774	17
`matmult_kmn_opt`	901383	211	173	39	22943	3627	3521	105

Table 8: Hits and miss of the L1 and L2 cache presented in millions

7 Conclusion

The naive implementation of the `matmult_nat` has successfully been implemented. It is possible to create six permutations of the naive structure. The permutations `matmult_mkn` and `matmult_kmn` has the best performance, compared to the four other permutations.

The performances of the implementation are lower than the performance of the `matmult_lib` function which uses the DGEMM subroutine from the CBLAS library.

It is possible to achieve a huge increases in performance of the functions when using compiler optimization particularly for the `matmult_mkn` function and the `matmult_kmn` function.

Student name and id: Marie Mørk {s112770} Anja Liljedahl Christensen{s162876} Andreas Vedel Jantzen {s162858} Anders Launer Bæk {s160159}

Team name: matmult_MAAA

Hand-in: OpenMP - Poisson Problem

---

8 Summary

In this project, the Possion problem is solved numerically using two different methods; Jacobi and Gauss-Seidel. Initially, the methods are implemented in sequential code. This showed that the Gauss-Seidel method needed less iterations for the solution to converge, but at the same time that the Jacobi method was executed faster, due to a higher number of flops/s. The Jacobi method was then parallelized in different versions to investigate the scaling behaviour of the algorithms. Parallelizing not only the algorithm but also the initialization of the matrices resulted in the overall best performance. Different ways of implementing parallelization in the Gauss-Seidel method are explained but not implemented.

9 Statement of the problem

The goal of the project is to determine the steady state heat distribution in a square room with constant heat capacity, a source (radiator), and constant boundary conditions at the walls by solving the Poisson problem numerically. Two different methods are considered for solving the problem; the Jacobi method and the Gauss-Seidel method. Furthermore, the performance of the Jacobi method is to be improved by parallelizing the program using OpenMP. The scaling behaviour of the different versions of the parallelized Jacobi will be compared with that of the Mandelbrot problem. The Gauss-Seidel method cannot be parallelized in the same way as the Jacobi method, but different ways of implementing parallelization using this method will be explained and discussed.

10 Hardware and software

Specifications of the test environment are listed below:

- CPU information
  - Vendor ID: GenuineIntel
  - CPU family: 6
  - Model: 79
  - Model name: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
  - CPU MHz: 1297.656
  - L1 (d / i) cache: 32K
  - L2 cache: 256K
  - L3 cache: 30720K
- Compiler settings

- The SunCC compiler have been applied with and without compiler optimization using the compiler flags `-fast -xopenmp` and `-xopenmp=noopt` respectively. See uploaded `makefile` for further information.

- Highlights from the batch script. See `master_sh.sh` for further information.
  - `#BSUB -R "rusage[mem=2048MB] span[hosts=1]"` and `#BSUB -n 24` which dedicate the whole host.
  - `export OMP_WAIT_POLICY=active`

11 Theory

The Poisson problem is given by eq. 1. The goal is to estimate the heat distribution of a room with a radiator.

$$-f(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \qquad (x,y) \in \Omega \tag{1}$$

The heat distribution will be estimated using two methods: The Jacobi method given in eq. 2 and the Gauss-Seidel method given in eq. 3.

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k} + \Delta^2 f(i,j)) \tag{2}$$

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k} + \Delta^2 f(i,j)) \tag{3}$$

The main difference between eq. 2 and eq. 3 is that the Gauss-Seidel method uses two updated elements in the update of the center element. The Jacobi method only uses the values from the previous iteration, which makes this method simple to parallelize.

One of the most important measures of scalability is Amdahl's law. It defines the ideal speed-up $S(P)$ for an increase in number of threads when running a given algorithm. Amdahl's law is defined as:

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{f/P + 1 - f}. \tag{4}$$

where $S(P)$ is the speed-up as a function of the number of threads P. $T(\bullet)$ is the execution time for the given number of threads and f the parallel fraction of the program.

Equation 5 determines the limit of the theoretical speed-up for a given f.

$$S_{max}(f) = \frac{1}{1-f} \tag{5}$$

It is possible to achieve a parallel fraction close to 1 in the Poisson problem, hence the expectation is that a parallel implementation should scale perfectly. In reality though, there are numerous things to consider in order to ensure that the program scales properly. Amdahl's law neglects the cost of communication, bookkeeping, synchronization, initializing, closing threads etc. The costs influence the properties of the scaling, recall the plot on slide 29 in `intro_para2018_ho.pdf`, found on Inside.

There are other approaches to increase the speed-up in parallel applications executed on multiple sockets. One approach is to initialize the matrices in parallel which "places" matrix elements efficiently according to the assigned threads. The native `numactl` command can be used to control the policies for the processes and shared memory. This approach has not been implemented in the project but could be considered as future work. This could speed-up the computation even more. `NUMA` is a memory placement policy that specifies on what and how many threads the computation has to be processed on.

The number of floating point operations per second (flops/s) is used as a measure of performance when comparing the different implementations. The total number of flops needed to solve the Poisson problem is calculated as the product of the iteration in each of the nested loops times the number of floating point operations performed in the innermost loop. This is multiplied by the number of iterations needed in order for the solution to converge (or the maximum number of iterations). The total number of floating point operations is divided by the time needed to execute the code. The equation becomes:

$$flops/s = \frac{I \cdot N^2 \cdot O}{T} \tag{6}$$

where I is the number of iterations, N is the dimension of the grid (excluding the boarders), $O = 10$ is the number of floating point operations in the innermost loop and T the execution time.

12 Algorithms

This section introduces the main algorithms used to solve the Poisson problem. Additional algorithms can be found in the appendix.

12.1 Sequential code - Jacobi method

The first function is a sequential implementation of the Jacobi method, see algorithm 5. The first input is the problem size, N, excluding the borders of the grid. Other inputs are the grid spacing, Δ, the threshold of the convergence, the maximum number of iterations as well as the matrices f, u and u_{old}. In the implementation, the approximation of u at iteration number k is saved in the matrix u. As the calculation of u at timestep k is dependent on the approximation of u at time step $k-1$, the pointers are swapped between u and u_{old} just before the calculations at each iteration commence.

```
1  int jac_seq_con(int N, double delta, double threshold, int max_iter, double *f,
       double *u, double *u_old) {
2      int i,j,k=0;
3      double *temp,d=10e+10;
4
5      while (d > threshold*threshold  &&  k < max_iter) {
6          // Set u_old = u
7          temp = u;
8          u = u_old;
9          u_old = temp;
10         // Set distance = 0.0
11         d = 0.0;
12         for (i = 1; i < N-1; i++) {
13             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]);

                // calculate distance
                d += (u[i*N + j] - u_old[i*N + j]) * (u[i*N + j] - u_old[i*N + j])
    ;
            }
        }
        k++;
    }
    // return no. iterations
    return(k);
}
```

Algorithm 5: Algo. `jac_seq_con()`.

12.2 Sequential code - Gauss-Seidel method

The sequential implementation of the Gauss-Seidel method takes the same inputs as the Jacobi method except for u_{old}, which is not needed. The code is listed in algorithm 6.

```
int gauss_con(int N, double delta, double threshold, int max_iter, double *f,
    double *u) {
    int k=0;
    double u_ij,d=10e+10;
    while (d > threshold*threshold  &&  k < max_iter) {
        d = 0.0;
        for (int i = 1; i < N-1; i++) {
            for (int j = 1; j < N-1; j++) {
                // Update u
                u_ij = u[i*N + j];
                u[i*N + j] = 0.25 * (u[(i-1)*N + j] + u[(i+1)*N + j] + u[i*N + (j
    -1)] + u[i*N + (j+1)] + delta*delta*f[i*N + j]);

                // calculate distance
                d += (u[i*N + j] - u_ij) * (u[i*N + j] - u_ij);
            }
        }
        k++;
    }
    return(k);
}
```

Algorithm 6: Algo. `gauss_con()`.

12.3 OpenMP Jacobi

A simple OpenMP version of the Jacobi function is implemented in algorithm 7. In this version, the parallel region is set to be around the two inner loops, which loops over the index of the matrices. It is implemented such that the variables f, u, u_{old}, N, threads and d are shared, while i and j are private. Each time the innermost loop is reaches, an addition is made to d. This results in dataracing, because all threads needs to make an addition to the same shared variable.
In the second version can be seen in the appendix as algorithm 21. The difference between the first

and second version is that the parallel region has been moved such that it covers the whole while loop. This is an advantage, because the team of parallel threads are created once in the function instead of using fork and join for each iteration. In order to ensure that the threads does not work on different iterations, k, at the same time, k is set to be firstprivate. This means that each of the threads gets the same value of k and only one thread updates it in the end of the loop. In the second version, the dataracing of version 1 has been avoided by adding the command `#pragma omp for reduction (+ : d)` before the loops over the matrix indices.
In the third version, the function in itself is not changed. Instead, the main file is altered such that matrix initializations are parallelized.

```
1  int jac_mp(int N, double delta, double threshold, int max_iter, double *f, double
       *u, double *u_old) {
2      int i,j,threads,k=0;
3      double *temp,d=10e+10;
4      // get threads
5      #pragma omp parallel
6      {
7          #pragma omp single
8          {
9              threads = omp_get_num_threads();
10         }
11     }
12     while (k < max_iter) {
13         // Set u_old = u
14         temp = u;
15         u = u_old;
16         u_old = temp;
17         // Set distance = 0.0
18         d = 0.0;
19         #pragma omp parallel shared(f, u, u_old, N,threads,d) private(i, j)
20         {
21         #pragma omp for reduction(+ : d)
22         for (i = 1; i < N-1; i++) {
23             for (j = 1; j < N-1; j++) {
24                 // Update u
25                 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]);
26
27                 // calculate distance
28                 d += (u[i*N + j] - u_old[i*N + j]) * (u[i*N + j] - u_old[i*N + j])
       ;
29             }
30         }
31         } /* end of parallel region */
32         k++;
33     }
34     // return no. threads
35     return(threads);
36 }
```

Algorithm 7: Algo. `jac_mp()`.

The performance of the OpenMP Jacobi is to be compared with the scaling of the Mandelbrot problem. An implementation of the Mandelbrot problem has been given in the week exercise, see algorithm 22 in the appendix.

12.4 OpenMP Gauss-Seidel

The Gauss-Seidel method (eq. 3) is more difficult to parallelize than the Jacobi method. It is because of the tightly bound nature (the method is coupled to itself by recursion) of the algorithm. Due to this, an OpenMP Gauss-Seidel is not implemented as a part of this project. Instead, different ways of implementing parallelization in Gauss-Seidel will be discussed in this section.

The article "Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors" by Courtecuisse and Allard proposes several methods for solving fully coupled systems. The **Row Parallel Algorithm** is a brute force parallelization of the Gauss-Seidel method. It takes the inner loop and calculates every row by itself and combines them in the end. The **Column Parallel Algorithm** works in a similar way as the **Row Parallel Algorithm**, only that it is then the columns that are considered one at a time. Both of these methods reduces the number of syncronizations to be perfromed to N in the case of an $N \times N$ matrix [1].

The **Block-Column Parallel Algorithm** is based on the CUDA library. CUDA only works within thread blocks, hence the system matrix will be divided into g submatrices (or subproblems) each with its own constraints. The subproblems can be computed in parallel because the calculations are bounded by its own constraints. For every subproblem, the matrix is divided into two blocks; one covering the diagonal of the full original matrix and another with the off-diagonal elements. Every subgroup computes the block with the diagonal elements first and then the off-diagonal. A barrier is introduced in between the calculation of diagonal and off-diagonal blocks. The subgroups only need $2g$ synchronizations which reduces the dependence of global synchronizations that restricts scalability of the algorithm [1].

The **Atomic Update Counter Parallel Algorithm** does, contrary to the above mentioned, get rid of global synchronization. This method stores a shared integer which counts the number of diagonal blocks computed. The counter and the revision of the Gauss-Seidel method is presented in the article. Specific hardware is also part of the **Atomic Update Counter Parallel Algorithm**, because it has to produce an automated-updated counter at the same time as allocating enough memory. This eliminates all synchronizations of the Gauss-Seidel methods [1].

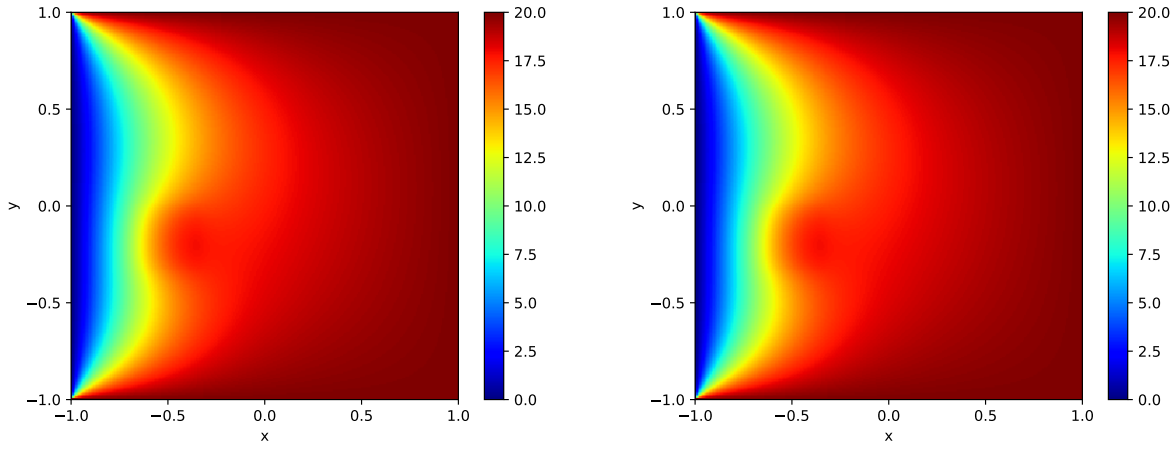
The following methods are derived from the original sequential Gauss-Seidel method from the article "Parallel methods for partial differential equations" by N.I. Lobachevsky State University of Niznhi Novgorod, [2]. The **Computation Synchronization** method recalculates the values of $u_{i,j}$ at random. The **Computation Deadlock** method restricts the access to the grid nodes and is hereby forced to parallelize which decreases the computation time. The **Interdeterminacy Elimination** method makes the storing of results and current iterations in independent matrices. The **Computation Load Balancing** method is as the name suggests a method for balancing the computation load among the used processors. The computation time of the program will hereby depend on the thread with the longest computation time [2].

13 Results

In the following sections, performance and scaling results from the implemented functions are presented. The functions are tested for different problem sizes and number of threads and optimal choices discussed.

13.1 Sequential code - Jacobi and Gauss-Seidel method

First off, the Poisson problem is solved using sequential code and two different methods; Jacobi and Gauss-Seidel. Both programs are tested using matrix sizes $N = 128$ and $N = 256$ and are set to run until they converge. The threshold of the convergence is set to $d = 10^{-10}$. In figure 9 the function estimates of $u(x, y)$ based on both methods are shown as heat maps. The estimates obtained are visually similar.



(a) Gauss-Seidel

(b) Jacobi

Figure 9: Estimate of the function $u(x, y)$. The problem size of the illustrated heats are $N = 256$. See table 9 for further evaluation details.

In table 9, details of the iterations of the two functions for problem sizes $N = 128$ and 256 are shown. As expected, the memory usage in the Gauss-Seidel method is 2/3 of the Jacobi. This is due to the fact that three matrices are used in the Jacobi method, namely `f, u` and `u_old`, whereas only `f` and `u` are needed in Gauss-Seidel. The number of iterations shows how many iterations are needed in order for $u(x, y)$ to converge. For both problem sizes, the Jacobi method needs almost twice as many iterations to reach convergence. In spite of this, the execution time is much larger in the case of Gauss-Seidel because the iterations are coupled, why the compiler optimizer fails to reduce runtime. This also means that fewer iterations and `mflops` are performed pr. second, as seen in the table. All together this information shows that if parallelization and smart memory allocation is implemented, there is a possibility of a larger performance gain for Gauss-Seidel than for Jacobi. On the other hand, Jacobi is easier parallelized as a result of the uncoupled iterations.

		Memory	Iterations	Duration	Iterations/s	mflops/s
128	`gauss_con()`	262144	51225	4.79	10704	1754
	`jac_seq_con()`	393216	101831	1.76	57717	9456
256	`gauss_con()`	1048580	194176	73.38	2646	1734
	`jac_seq_con()`	1572860	390976	25.80	15153	9931

Table 9: Comparison of the Jacobi method and Gauss-Seidel method.

13.2 OpenMP Jacobi

The Jacobi method can be parallelized and run on multiple cores with shared memory using OpenMP. In the optimal setting the speed-up of increasing the number of cores is 1, i.e. using 2 cores instead of 1 should make the program run twice as fast. As discussed in the theory section OpenMP has several constructs which can be used to optimize the parallelization and ensure correctness. In the following sections, parallelized versions of the Jacobi method are implemented and continuously improved. For this purpose the number of iterations are fixed such that k runs from 1 to `max_iter` (with `max_iter` = 5000) in order to best compare the efficiency of the different parallelized versions. Each version is run on 1, 4, 8, 16, and 24 threads for 4 different problem sizes $N \in \{256, 512, 1024, 2048\}$.

The first simple version will be used as a baseline.

13.2.1 Version 1

The first simple version has 2 parallel constructsm see algorithm 5: The first one contains a single-construct to make one of the threads extract the number of threads running the program. The second is nested within the while loop and contains a loop-construct such that the iterations from the outer for-loop (i.e. the rows) are distributed among the threads.

Figure 10 shows how increasing the number of threads for different problem sizes increases the number of flops per second. The gain is especially evident for $N = 512$ and $N = 1024$, while for $N = 2048$ there is no efficiency improvement when increasing the number of threads beyond 8. This is explained by the fact that when N is this size the problem does not fit within the 3 caches, hence memory has to be fetched from the RAM which is significantly slower and hereby slows down the process.

In figure 11 the speed-up for each problem size is compared to the theoretical speed-up. For problem sizes 256, 512, and 1024 there is a clear correlation between problem size and speed-up, i.e. as the problem size increases so does the speed-up. This, however, is not the case for $N = 2048$ which is as before most likely linked to the amount of needed memory. Since there are two sockets each running 12 threads, they are forced to communicate when the number of threads exceeds 12 which could potentially slow down the process when the problem is large enough.

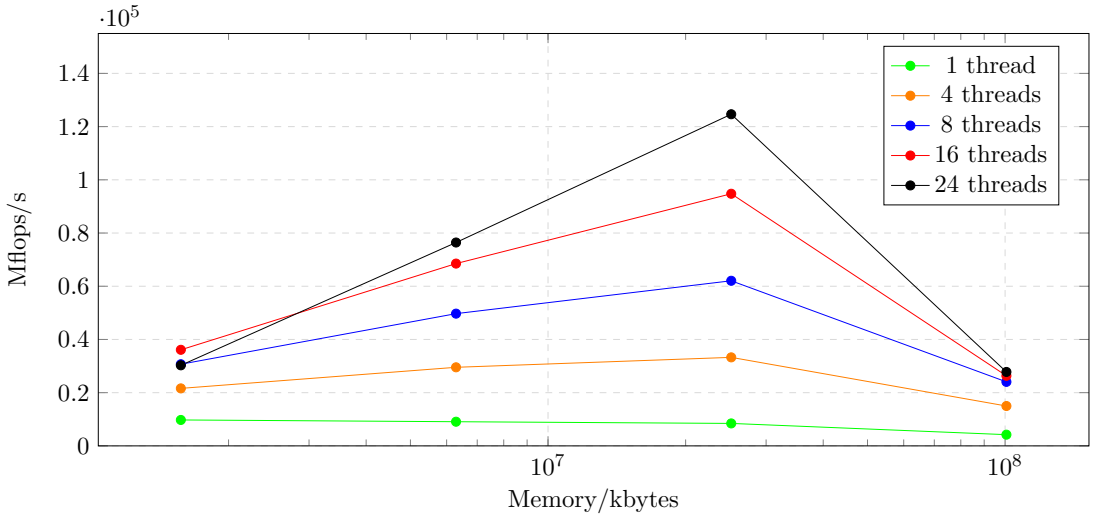


Figure 10: Jacobi version 1.

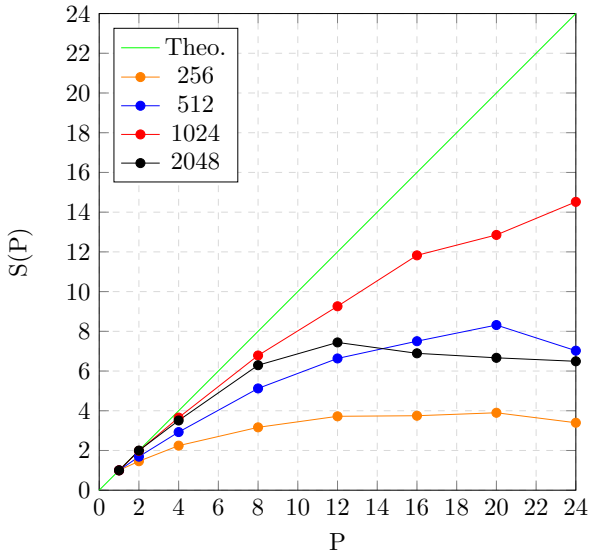


Figure 11: Scaling Jacobi version 1.

13.2.2 Version 2

For version 2 the second parallel construct is moved out to include the while-loop as well, seen in algorithm 21. In the previous version the master thread distributed the iterations among the threads each time the while-loop counter k was incremented by 1. Hence, there was a fork and a join of the threads for each k from 1 to `max_iter` which resulted in a lot of waiting time in between the actual computations. In this version the iterations are only distributed among the threads once which should decrease the waiting time. Now the pointers have to be swapped in parallel but this should only be done by one of the threads to avoid data racing. Therefore it is also crucial that the threads use the same iteration k by declaring the variable as a `firstprivate`.

Figure 12 shows how the performance is now increased for each combination of problem size and number of threads (apart from 1 thread). The improvement is the most evident when using 24 threads, while it seems there is no improvement when using 1 thread. This makes sense since there is nothing running in parallel when using only 1 thread.

In figure 13 it is hard to detect any improvement for problem sizes 256, 512, and 1024, while the speed-up for $N = 2048$ is a little better beyond 12 threads.

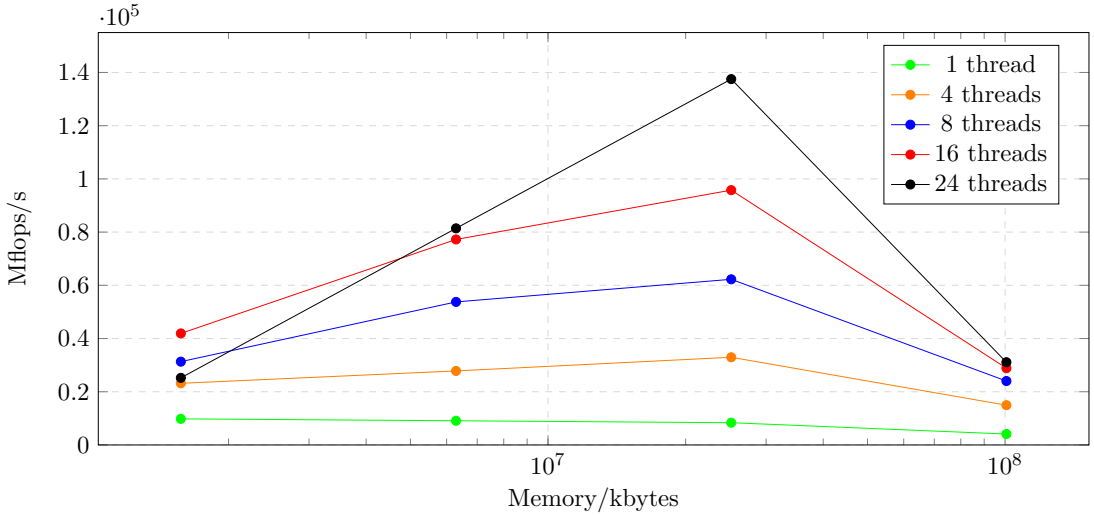


Figure 12: Jacobi version 2.

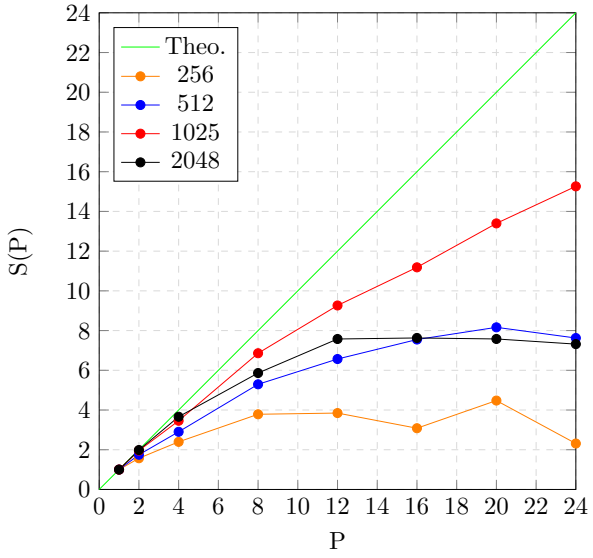


Figure 13: Scaling Jacobi version 2.

13.2.3 Version 3

For the final version the Jacobi method itself is the same as version 2, but this time the initialization of the matrices is also parallelized. Hence, the data is pre-distributed among the threads to make it more accessible when running the actual program.

In figure 14 it is once again obvious that the larger of the problem sizes are the ones to exploit the further parallelization the most. Especially $N = 2048$ utilizes the parallelized initialization of the matrices, which makes sense given the large memory consumption.

By considering figure 15 it is clear that the problem sizes 256 and 512 experience the best improvement of speed-up, while $N = 1024$ seems to be unchanged.

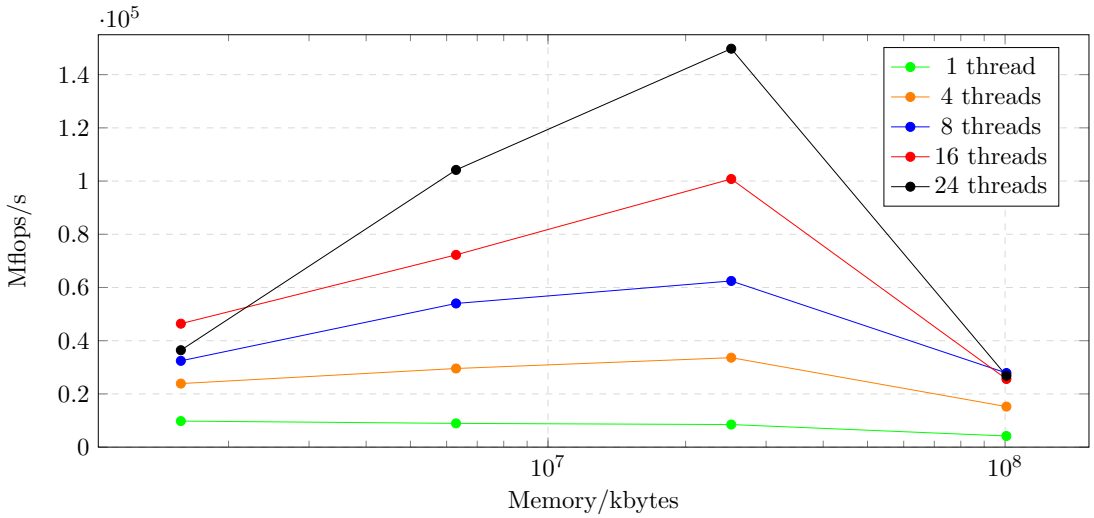


Figure 14: Jacobi version 3.

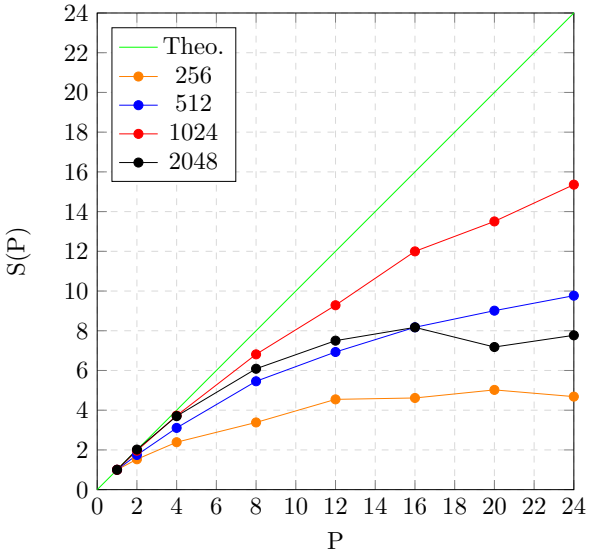


Figure 15: Scaling Jacobi version 3.

13.2.4 Comparison

In figure 16 the efficiency of the 3 different OpenMP Jacobi methods are compared for 4 different problem sizes (the same as before) using 8 and 16 cores. Version 1 is outperformed by version 2 and 3 using both 8 and 16 threads. With 8 threads version 2 and 3 are practically indistinguishable, while using 16 threads the best performing version alternates between 2 and 3 depending on the problem size.

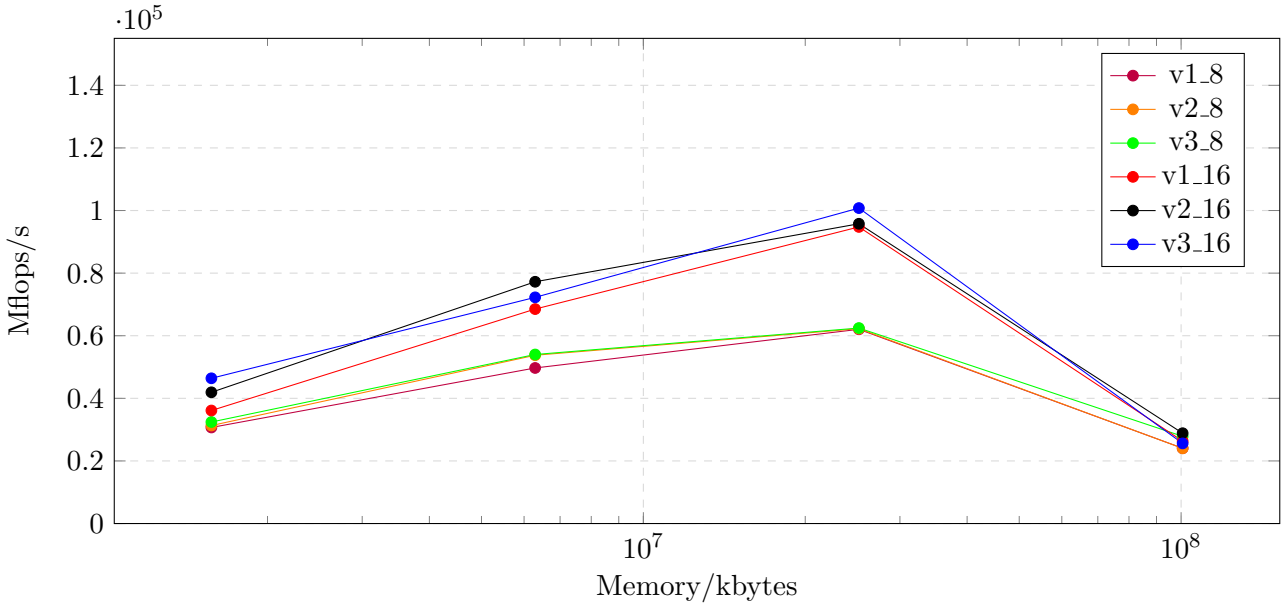
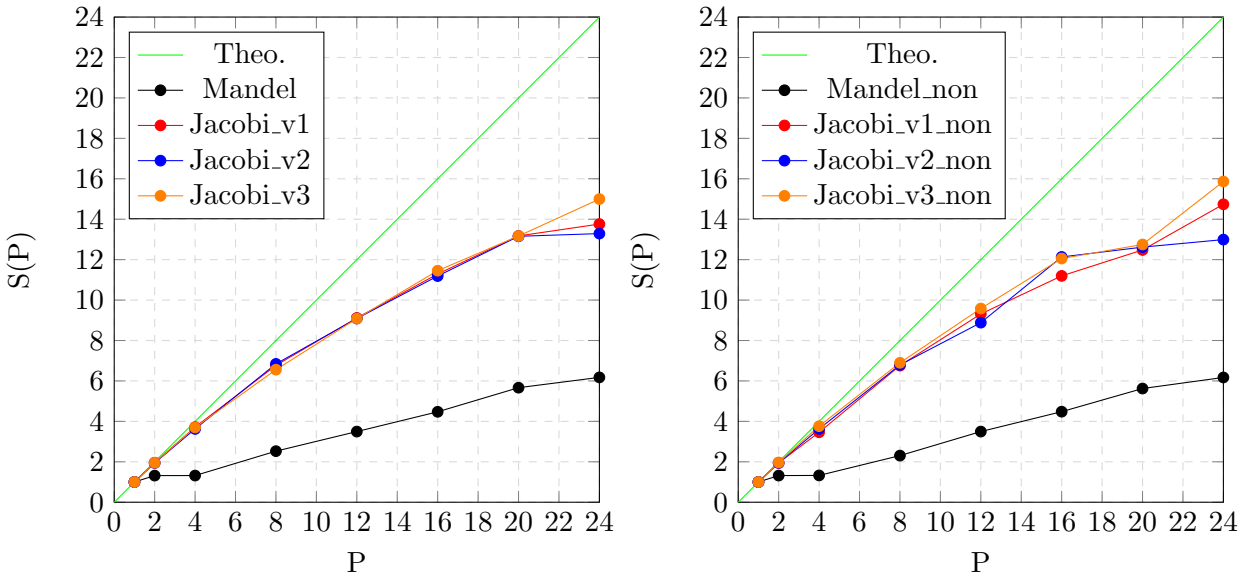


Figure 16: Jacobi version 1, 2 and 3 for 8 and 16 threads.

13.3 Mandelbrot

The scaling behaviour of the Jacobi method and of the Mandelbrot algorithm is given in figure 17a and in figure 17b with and without compiler optimization flags repectively.

The four methods have been evaluated for $N = 1024$ and with $max_{iter} = 5000$.



(a) Optimized

(b) Non optimized

Figure 17: The plots shows the scaling as function of number of threads P with following compiler flags: `-fast -xopenmp` and `-xopenmp=noopt` for 17a and 17b respectively.

There is a slight difference between the three versions of the Jacobi method when not using the mentioned compiler optimization flags, see figure 17b. The three versions follows each other for a given P except for $P = 24$.

It can be concluded that the current implementation of the Mandelbrot algorithm (algo. 22) does not scale as the implementations of the Jacobi method, see table 10 where f have been derived from eq. 4. An explanation for the scaling of the Mandelbrot is the unbalanced work load in the while loop for the assigned threads. A solution to this could be dynamic scheduling of the work balance.

Table 10 reports the average values of f for the curves in figure 17a and figure 17b. f have been derived from eq. 4.

Algo.	f_{opt}	f_{non}
`manddel()`	0.730	0.726
`jac_mp()`	0.976	0.972
`jac_mp_v2()`	0.974	0.974
`jac_mp_v3()`	0.976	0.980

Table 10: Average value of f for each algorithm. The values in the second column corresponds to figure 17a and the third column corresponds to figure 17b.

The max speed-up limit for `jac_mp_v3()` is $S_{max}(0.976) = \lfloor 41 \rfloor$. The max speed-up limit for `mandel()` is $S_{max}(0.730) = \lfloor 3 \rfloor$, see eq. 5.

14 Conclusion

The sequential codes for the Jacobi and Gauss-Seidel methods showed that while the Gauss-Seidel method needs less iterations for the solution to converge, it is still much slower than Jacobi. This is because more floating point operations are performed pr. second in Jacobi. From the analysis of the performance of the sequential programs it was found that parallelization and smarter data allocation could enhance the performance of both methods, but that the possible gain might be higher for using Gauss-Seidel, due to convergence being reached after fewer iterations and the fact that this method uses less memory.

As previously discussed the parallelized third version of the Jacobi method clearly improved the efficiency (number of flops per second) compared to the first and second version. The improvement of the speed-up is not that evident for the problem sizes 256, 512, and 1024, but when the problem doesn't fit within the first three caches and has to be stored in the RAM, there is some gain in speed-up when optimizing the parallelization – both with respect to the Jacobi method itself and the initialization of the matrices.

For future work it could be beneficial to implement the `NUMA` control policy.

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

---

15 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 $\approx \times 16$.

16 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 18. The performance of the implemented algorithms are to be compared with the cBLAS DGEMM subroutine used in Assignment 1.

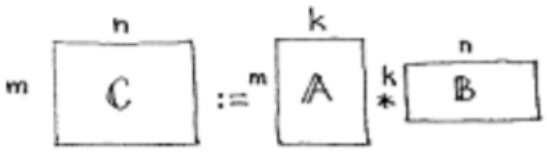


Figure 18: Visualization of the matrix orientation. The picture is borrowed from the description of Assignment 1.

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.

17 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.

18 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 [3].

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.

19 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 23.
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.

19.1 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 8.

```
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 8: `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 19 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 20, the speed-up of `gpu1` compared to `lib` is shown. As it is already apparent from figure 19, 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.

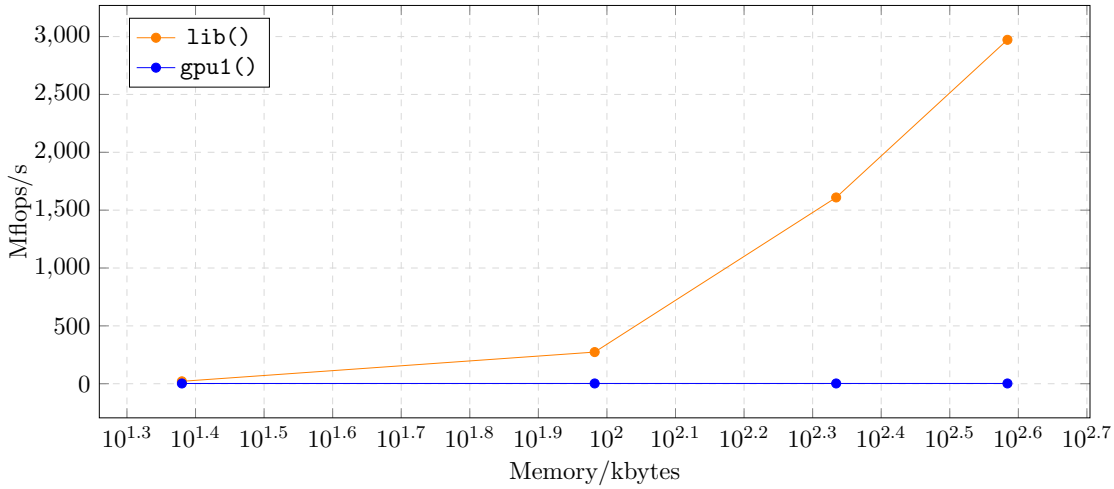


Figure 19: Comparison of `gpu1` and `lib`.

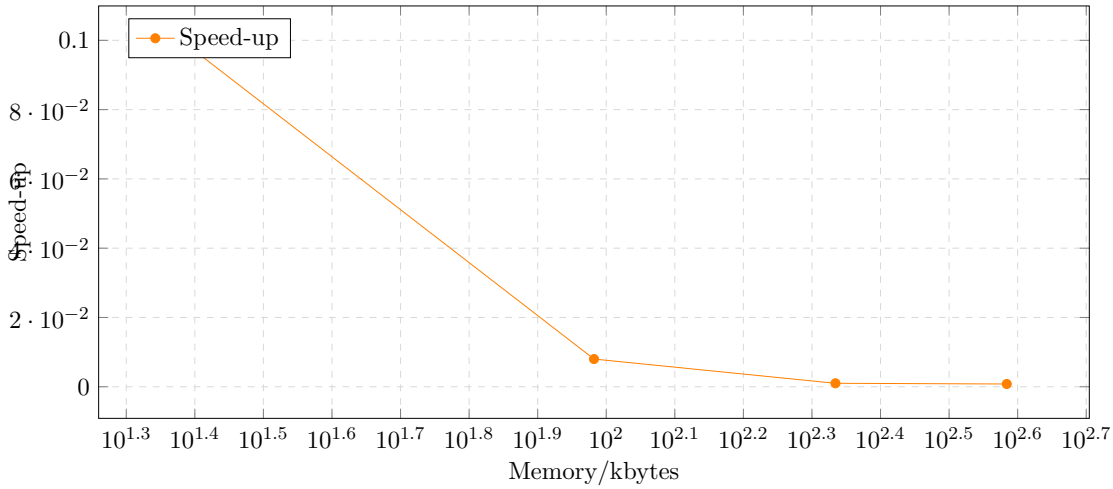


Figure 20: Speed up of `gpu1` compared to `lib`.

19.2 gpu2

The kernel for `matmult_gpu2` is listed in algorithm 9, while the changes in the launch from the CPU function is listed in 10. 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 9: `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 10: `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 21, 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.

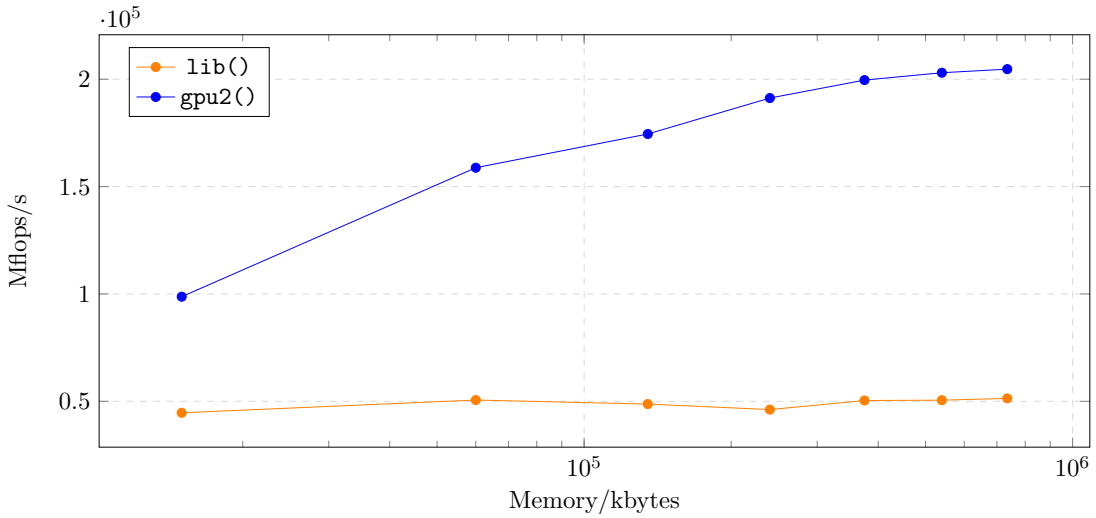


Figure 21: Comparison of `gpu2` and `lib`.

The speed-up is computed from mflops/s and reported in figure 22 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.

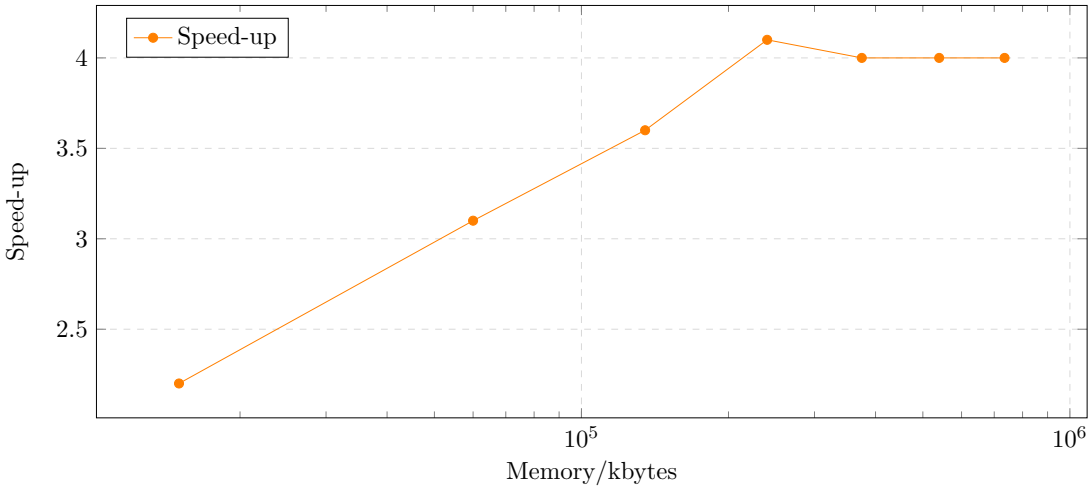


Figure 22: Speed up of `gpu2` compared to `lib`.

Using the call `nvprof --print-gpu-summary` in the shell, the different parts of the GPU function is timed. In table 11, 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 11: Time spend on Kernel, HtoD and DtoH in the kernel of `gpu2` for the chosen problem sizes.

19.3 `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 11, while the changes in the launch from the CPU function is listed in algorithm 12.
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 12. In algorithm 11 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 11: 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 12: gpu3 launch of kernel.

Again, the performance of the algorithm is compared to that of `lib`. In figure 23, 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.

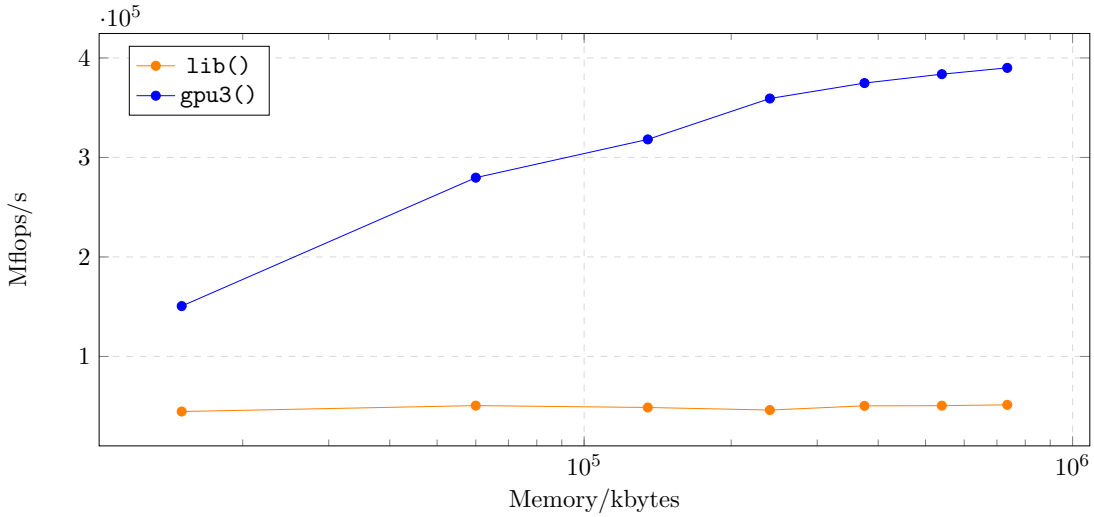


Figure 23: Comparison of `gpu3` and `lib`.

In figure 24, 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`.

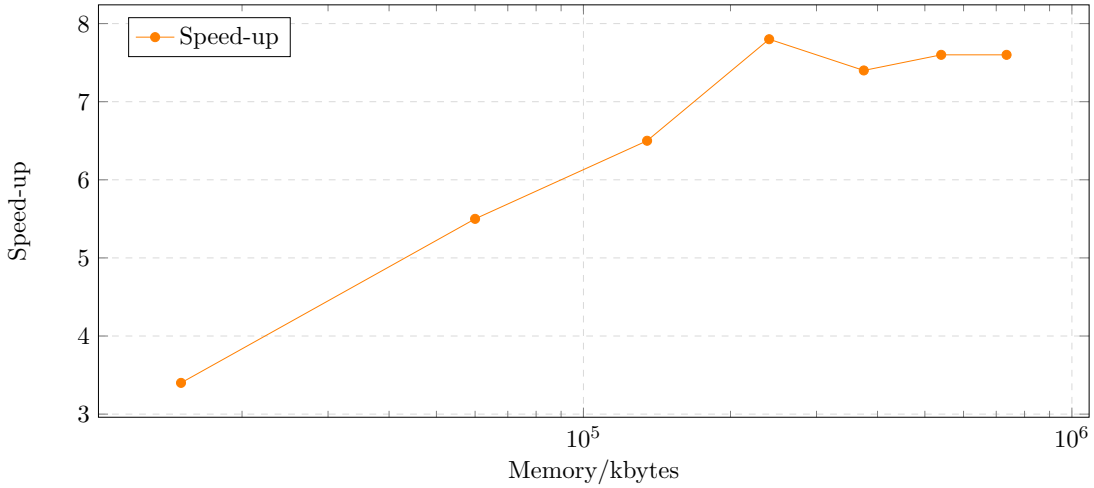


Figure 24: Speed up of `gpu3` compared to `lib`.

19.4 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 13, while the changes in the

launch from the CPU function is listed in algorithm 14.

```
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){
    int i, j, k, sn, sm, js, is;

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

    for (sn = 0; sn < stride_n; sn++){
        js = j + sn;

        for (sm = 0; sm < stride_m; sm++){
            is = i + sm;

            if (is < M && js < N){
                d_C[is*N + js] = 0.0;

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

Algorithm 13: 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 14: gpu4 launch of kernel.

In figure 25 below the GPU time is recorded for small values of `stride_m`, i.e. from 3 to 64, and in figure 26 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.

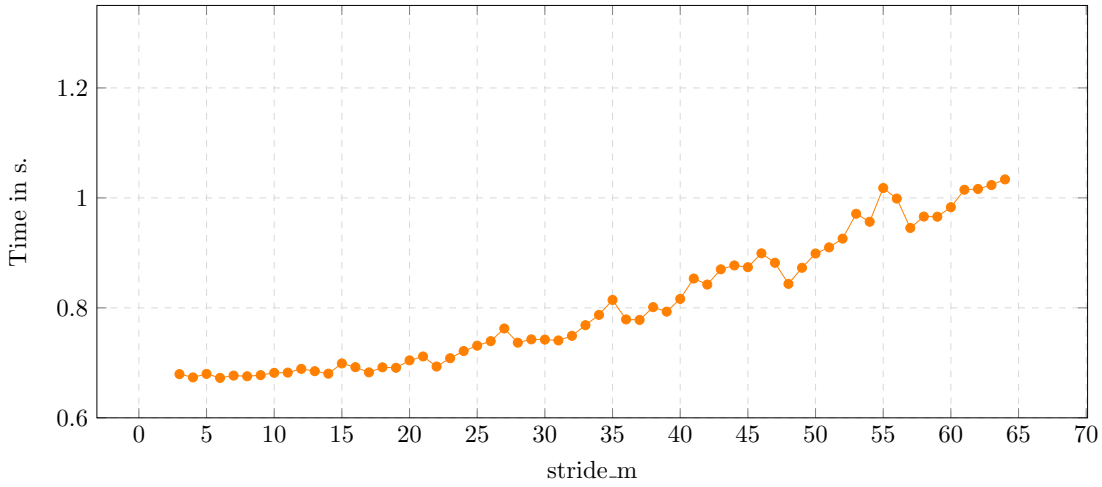


Figure 25: GPU time for `matmult_gpu4` for different small sizes of `stride_m`.

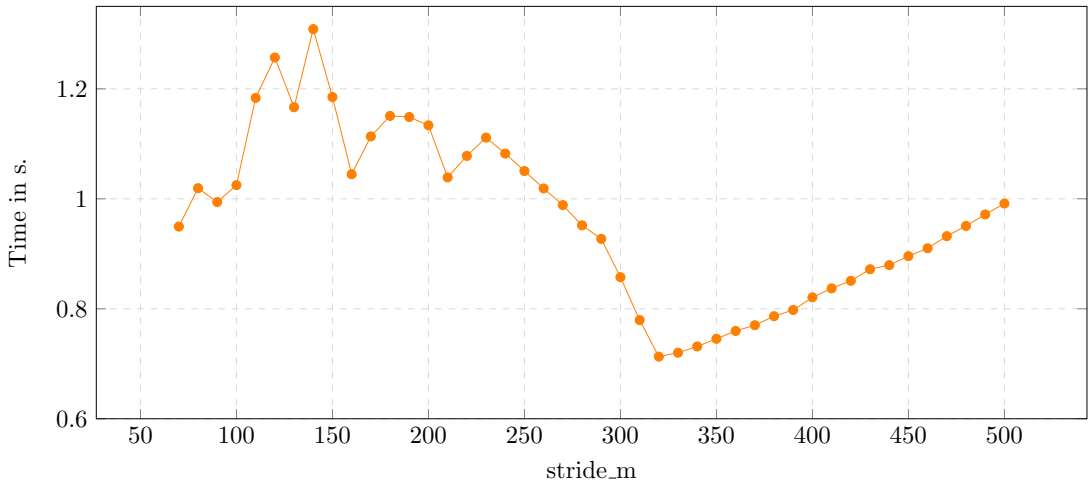


Figure 26: GPU time for `matmult_gpu4` for different large sizes of `stride_m`.

Below in algorithm 15 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 15: Optimized `gpu4` kernel.

In figure 27 below the number of floating points operations per second of `matmult_gpu4` is compared to `matmult_gpu3` and `matmult_lib`, and in figure 28 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.

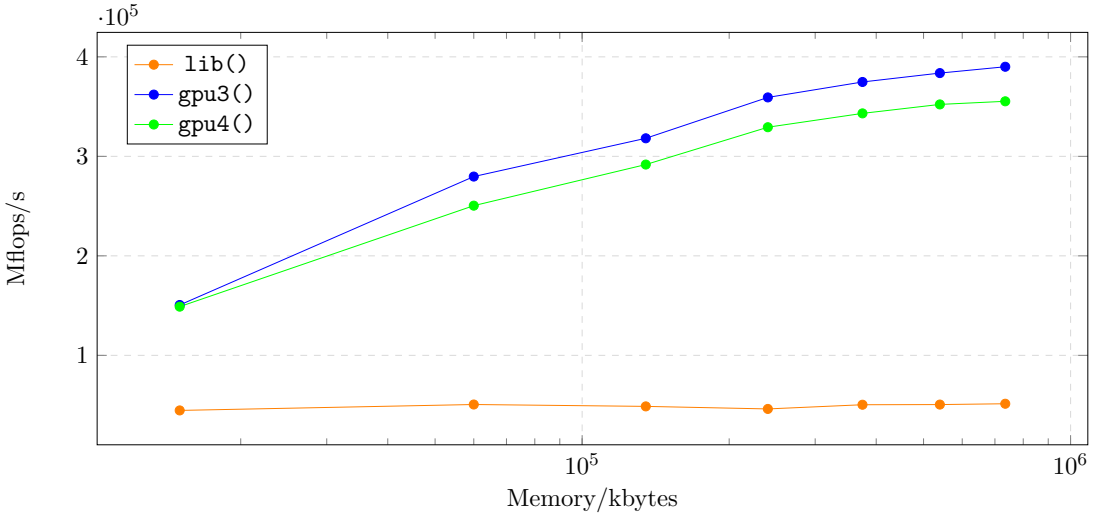


Figure 27: Comparison of `gpu4` and `lib`.

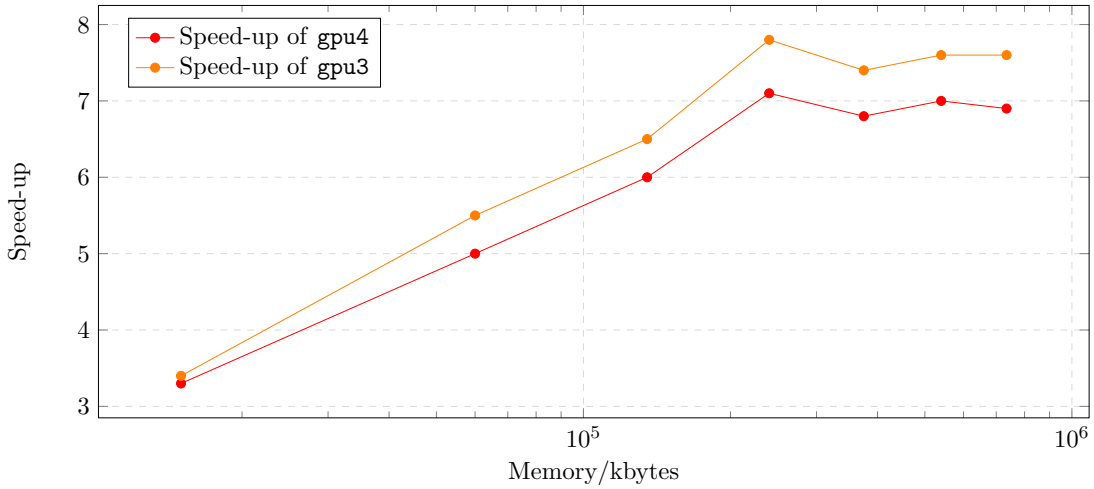


Figure 28: Speed up of `gpu4` compared to `lib`.

19.5 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 16 below.

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

```
1  void matmult_gpu5(int M, int N, int K, double *A, double *B, double *C) {
2      // Load A and B to device memory
3      Matrix d_A;
4      d_A.width = d_A.stride = K;
5      d_A.height = M;
6      size_t size = M * K * sizeof(double);
7      cudaMalloc(&d_A.elements, size);
8      cudaMemcpy(d_A.elements, A, size, cudaMemcpyHostToDevice);
9      Matrix d_B;
10     d_B.width = d_B.stride = N;
11     d_B.height = K;
12     size = K * N * sizeof(double);
13     cudaMalloc(&d_B.elements, size);
14     cudaMemcpy(d_B.elements, B, size, cudaMemcpyHostToDevice);
```

```
15
16      // Allocate C in device memory
17      Matrix d_C;
18      d_C.width = d_C.stride = N;
19      d_C.height = M;
20      size = M * N * sizeof(double);
21      cudaMalloc(&d_C.elements, size);
22
23      // Invoke kernel
24      dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
25      dim3 dimGrid(N / dimBlock.x, M / dimBlock.y);
26      gpu5_kernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
27
28      // Read C from device memory
29      cudaMemcpy(C, d_C.elements, size, cudaMemcpyDeviceToHost);
30
31      // Free device memory
32      cudaFree(d_A.elements);
33      cudaFree(d_B.elements);
34      cudaFree(d_C.elements);
35  }
36  }
```

Algorithm 16: `gpu4` launch of kernel.

In figure 29 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 30. With version three the maximal speed-up was close to 8 but version five the maximal speed-up is almost 30.

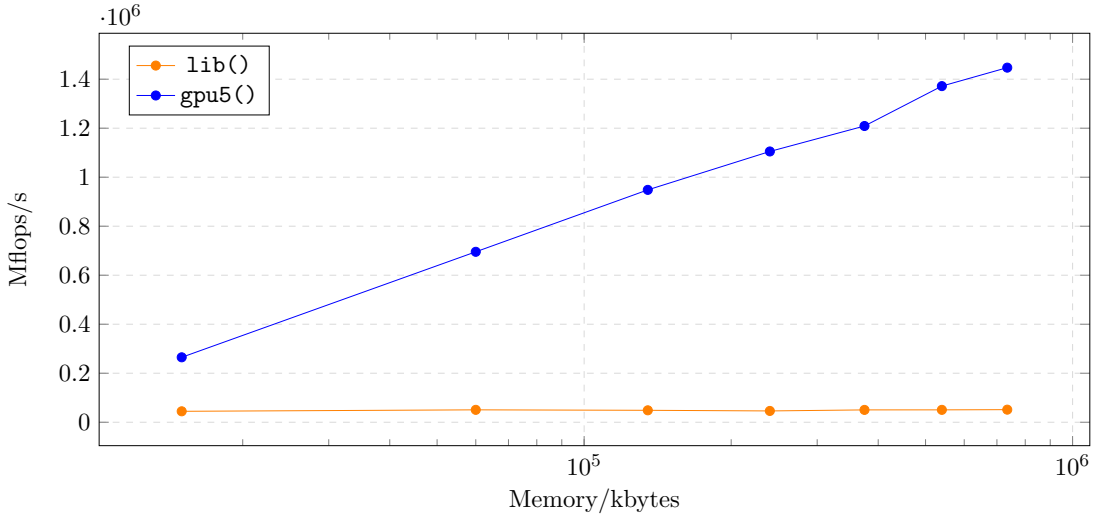


Figure 29: Comparison of `gpu5` and `lib`.

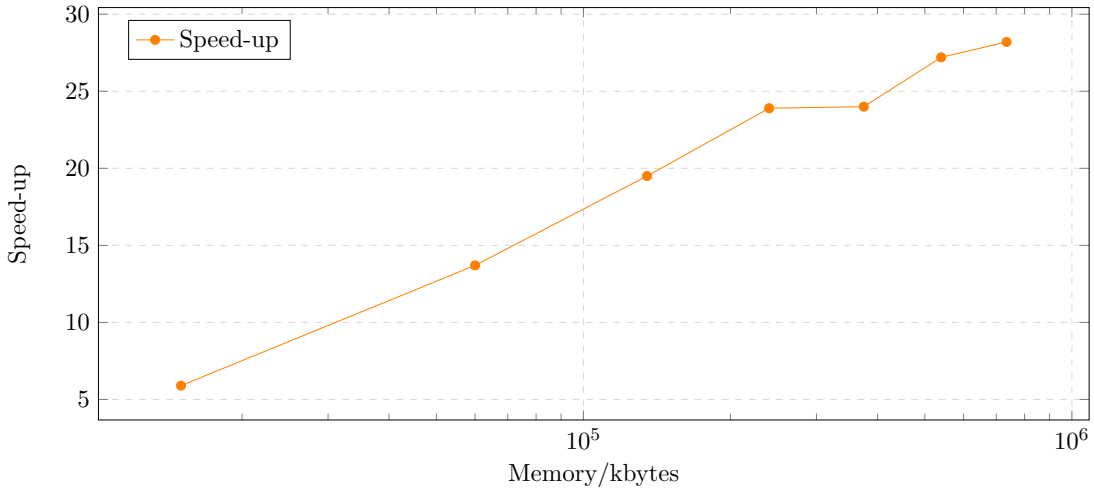


Figure 30: Speed up of `gpu5` compared to `lib`.

19.6 `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. 17.

```
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 17: `gpulib` launch of kernel.

In figure 31, 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 32 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.

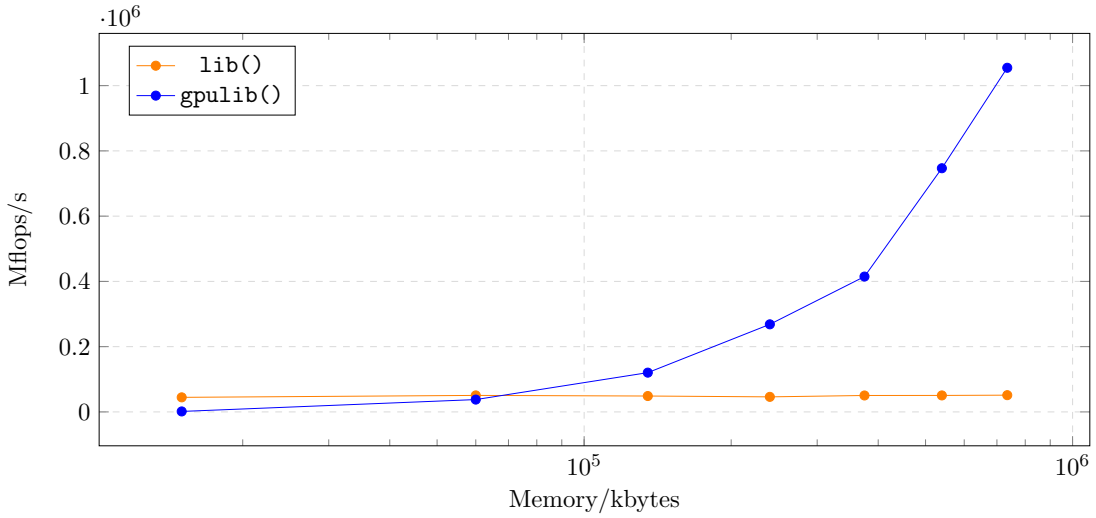


Figure 31: Comparison of `gpulib` and `lib`.

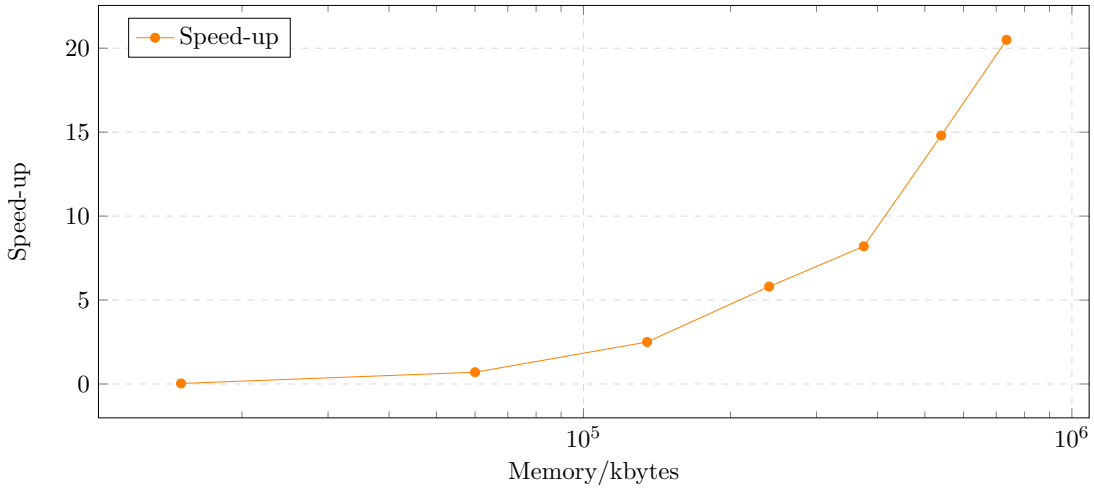


Figure 32: Speed up of `gpulib` compared to `lib`.

19.7 Comparison of the algorithms

In order to compare all implementations of matrix multiplication using GPUs, the calculated speed-ups are shown in figure 33. 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 33 are compared.

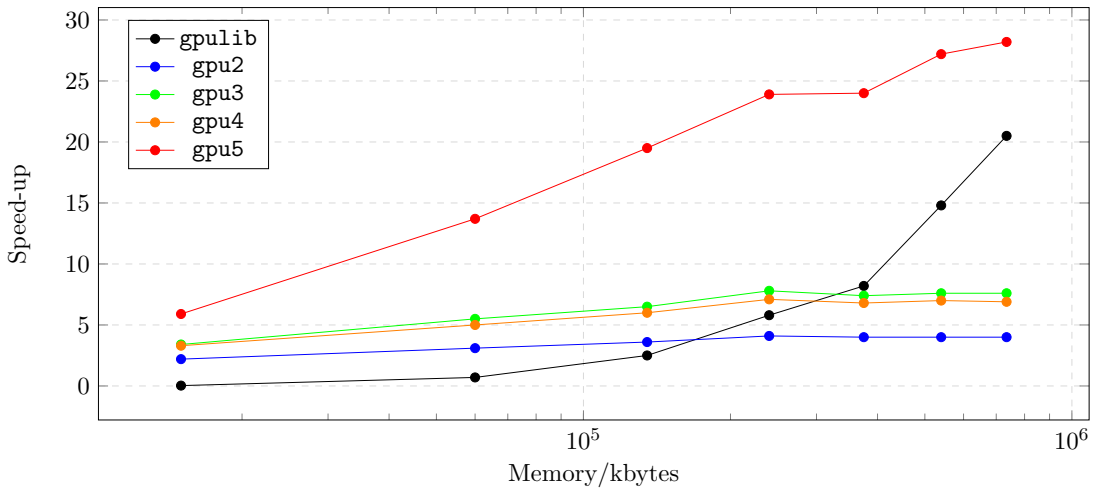


Figure 33: Speed up comparison.

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

is analyzed.

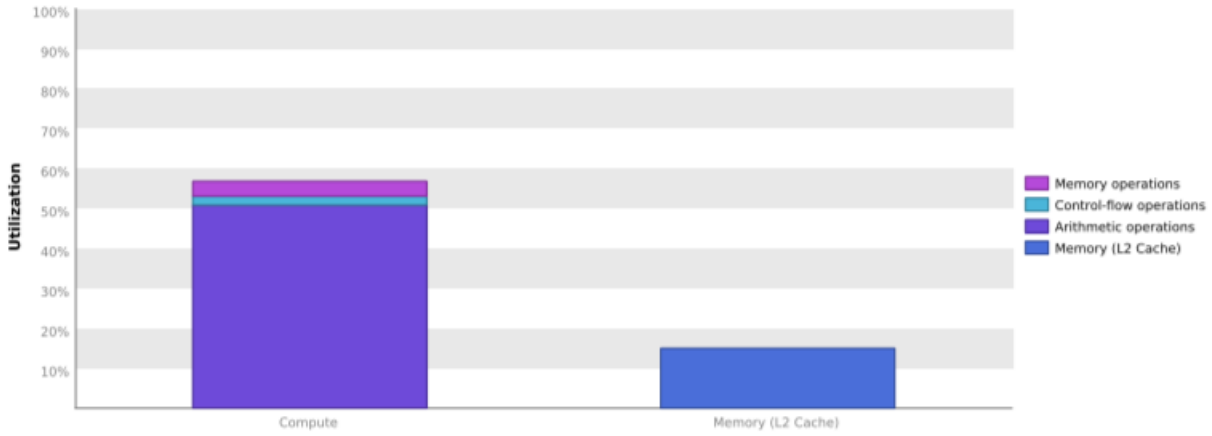


Figure 34: Kernel performance of `gpu2`.

From figure 34 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.

20 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 12.
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 36 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.

20.1 Sequential GPU Jacobi

The kernel used in the Sequential Poisson is provided in algorithm 18.
`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. 25 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 18: Algo. `jac_gpu1`.

.

20.2 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. 19.

```
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 19: 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:

$$dim3 \quad dim\_grid\left(\frac{N+bs-1}{bs}, \frac{N+bs-1}{bs}\right) \tag{7}$$

$$dim3 \quad dim\_block\,(bs, bs) \tag{8}$$

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

20.2.1 `nvvp`

The `nvvp` analyzing tool tells that the `jac_gpu2()` has a "Low Compute Utilization" $\approx 16\%$, presented in figure 35. 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 36c in the appendix.

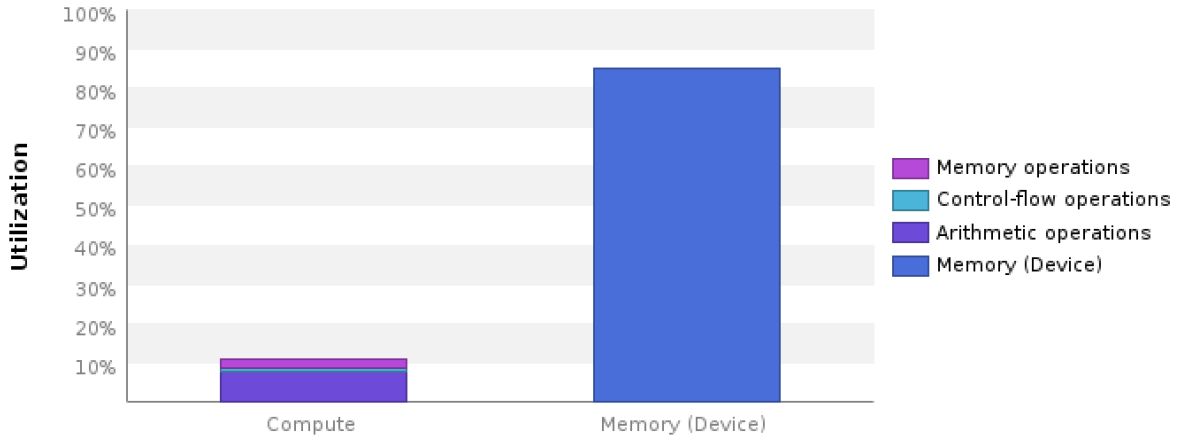


Figure 35: The `nvvp` analysis of `jac_gpu2()`. This reports that the algorithm is memory bound as the percentage of utilized memory is much bigger than the utilization of the computation.

20.3 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 20.
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. 27 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 20: 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:

$$dim3 \quad dim\_grid\left(\frac{N+bs-1}{bs}, \frac{N/2+bs-1}{bs}\right) \tag{9}$$

$$dim3 \quad dim\_block\,(bs, bs) \tag{10}$$

20.4 Speedup

The expected compute speedup is $8x^2$ when deploying the algorithm on the GPU compared to the CPU. Table 12 reports the achieved speedups for the three implementations.

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

Table 12: 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 $2x$.

[2] PerformanceTuningIntro.pdf, slide 20.

21 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 12 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] H. Courtecuisse and J. Allard. "Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors". In: *2009 11th IEEE International Conference on High Performance Computing and Communications*. 2009, pp. 139–147. DOI: 10.1109/HPCC.2009.51.

[2] N.I. Lobachevsky State University of Niznhi Novgorod. "Parallel methods for partial differential equations". In: pp. 4–14. URL: http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp12.pdf.

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

Appendices

Algo. `jac_mp_v23()`

The change between version 2 and version 3 is in its corresponding main files, see file `main_jac_mp_v2.c` and file `main_jac_mp_v3.c`.

```
1  int jac_mp_v23(int N, double delta, double threshold, int max_iter, double *f,
       double *u, double *u_old) {
2      int i,j,threads,k=0;
3      double *temp,d=10e+10;
4      // get threads
5      #pragma omp parallel
6      {
7          #pragma omp single
8          {
9              threads = omp_get_num_threads();
10         }
11     }
12     // do calculations
13     #pragma omp parallel shared(f, u, u_old, N,threads,d) private(i, j)
       firstprivate(k)
14     {
15     while (k < max_iter) {
16         #pragma omp single
17         {
18             // Set u_old = u
19             temp = u;
20             u = u_old;
21             u_old = temp;
22             // Set distance = 0.0
23             d = 0.0;
24         }
25         #pragma omp for reduction(+ : d)
26         for (i = 1; i < N-1; i++) {
27             for (j = 1; j < N-1; j++) {
28                 // Update u
29                 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]);
30                 // calculate distance
31                 d += (u[i*N + j] - u_old[i*N + j]) * (u[i*N + j] - u_old[i*N + j])
       ;
32             }
33         }
34         k++;
35     }/* end while */
36     } /* end of parallel region */
37     return(threads);
38 }
```

Algorithm 21: Algo. `jac_mp_v2()` and `jac_mp_v3()`.

Algo. `mandel()`

```
int mandel(int disp_width, int disp_height, int *array, int max_iter) {
    double  x, y, u, v, u2, v2, scale_real, scale_imag;
    int     i, j, iter, threads;

    scale_real = 3.5 / (double)disp_width;
    scale_imag = 3.5 / (double)disp_height;

    // get threads
    #pragma omp parallel
    {
        #pragma omp single
        {
            threads = omp_get_num_threads();
        }
    }

    #pragma omp parallel shared(array,disp_width,disp_height,scale_real,scale_imag
    ,max_iter) private(x, y, u, v, u2, v2, i, j, iter)
    {
    #pragma omp for
    for(i = 0; i < disp_width; i++) {
        x = ((double)i * scale_real) - 2.25;
        for(j = 0; j < disp_height; j++) {
            y = ((double)j * scale_imag) - 1.75;
            u    = 0.0;
            v    = 0.0;
            u2   = 0.0;
            v2   = 0.0;
            iter = 0;
            while ( u2 + v2 < 4.0 &&  iter < max_iter ) {
                v = 2 * v * u + y;
                u = u2 - v2 + x;
                u2 = u*u;
                v2 = v*v;
                iter = iter + 1;
            }
            // if we exceed max_iter, reset to zero
            iter = iter == max_iter ? 0 : iter;

            array[i*disp_height + j] = iter;
        }
    }
    }
    return(threads);
}
```

Algorithm 22: Algo. `mandel()`.

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
```

```
4     int LDB = fmax(1,N); // leading dimension of B
5     int LDC = fmax(1,N); // leading dimension of C
6     double alpha = 1.0, beta = 0.0; //
7     cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,M,N,K,alpha,A,LDA,B,LDB,
      beta,C,LDC);
8 }
9 }
```

Algorithm 23: 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;
```

```
54              }
55          }
56      }
57
58      } /* end of parallel region */
59
60      // do program
61      double time_compute = omp_get_wtime();
62      jac_cpu(N, delta, max_iter,f,u,u_old);
63      double tot_time_compute = omp_get_wtime() - time_compute;
64      // end program
65
66
67      // stats
68      double GB = 1.0e-09;
69      double flop = max_iter * (double)(N-2) * (double)(N-2) * 10.0;
70      double gflops  = (flop / tot_time_compute) * GB;
71      double memory  = size_f + size_u + size_u_old;
72      double memoryGBs  = memory * GB * (1 / tot_time_compute);
73
74      printf("%g\t", memory); // footprint
75      printf("%g\t", gflops); // Gflops
76      printf("%g\t", memoryGBs); // bandwidth GB/s
77      printf("%g\t", tot_time_compute); // total time
78      printf("%g\t", 0); // I/O time
79      printf("%g\t", tot_time_compute); // compute time
80      printf("# cpu\n");
81
82      //write_result(u, N, delta, "./../../analysis/pos/jac_cpu.txt");
83
84      // free mem
85      free(f);
86      free(u);
87      free(u_old);
88      // end program
89      return(0);
90  }
```

Algorithm 24: 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++;
```

```
    }/* 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("# gpu1\n");

    //write_result(h_u, N, delta, "./../../analysis/pos/jac_gpu1.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 25: 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();
```

```
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("# gpu2\n");
133 
134     //write_result(h_u, N, delta, "./../../analysis/pos/jac_gpu2.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 26: 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)){
```

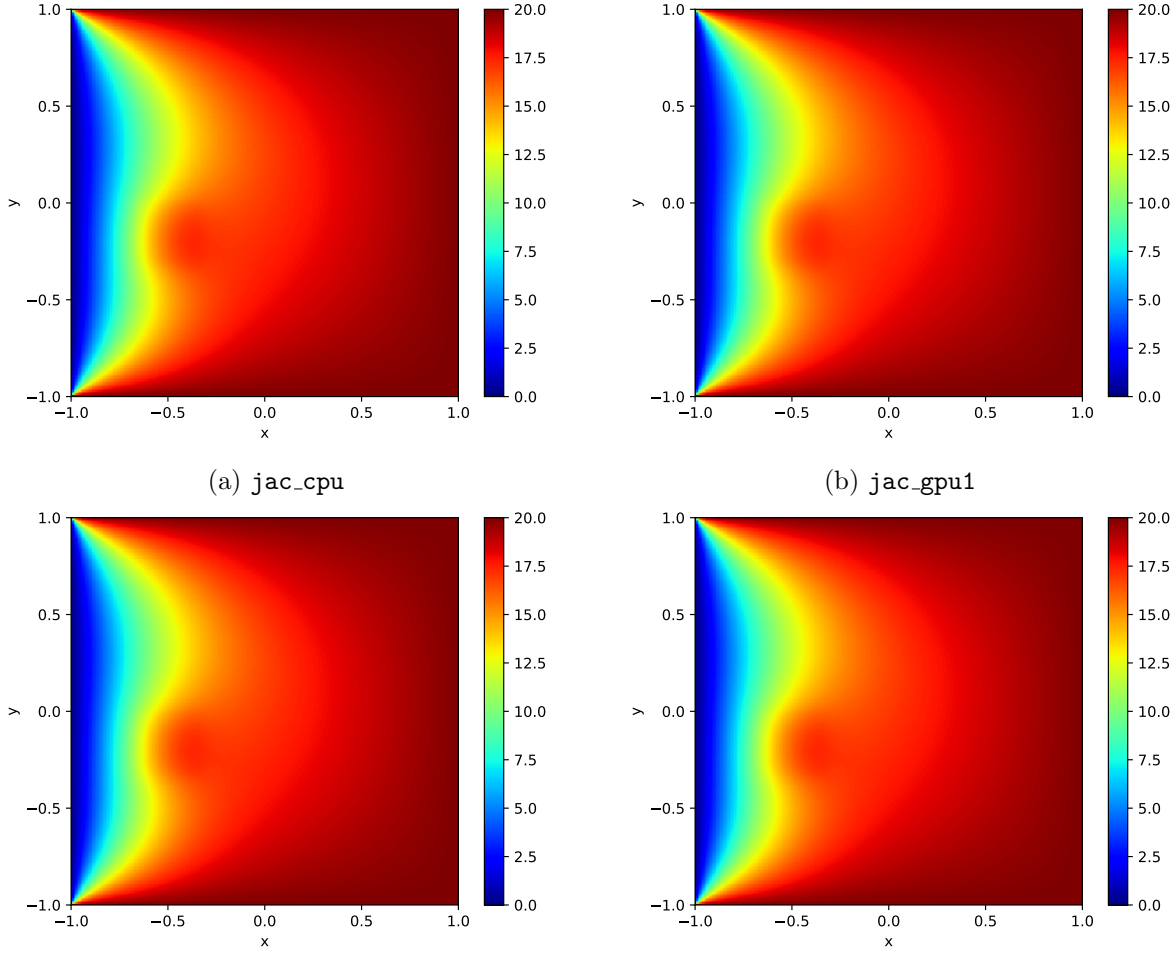
```
                h_u[i*N + j] = 20.0;
                h_u_old[i*N + j] = 20.0;
            }
            else{
                h_u[i*N + j] = 0.0;
                h_u_old[i*N + j] = 0.0;
            }
        }
    }

    //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 27: Algo. `jac_gpu3()`.

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

The following plots in figure 36 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 36: Estimate of the function $u(x, y)$.