High Performance Computing

Assignment 3: GPU Matrix Multiplication and Poisson Problem

21st January 2017

Author
Alberto Chiusole (s162501)

Collaborators
Miguel Suau de Castro (s161333)
Anders Jakobsen (s122467)

Technical University of Denmark
02614 High-Performance Computing

Contents

1 | Introduction

In this assignment we are going to adapt the code used in the two previous ones to compute matrix-matrix multiplications and solve Poisson differential equations, within graphical processing unit (GPU) cards.

The high efficiency of graphic cards performing certain computations made them attractive to be used in other areas not necessarily related to image processing. While in classical computer programming all operations were carried out by the central processing unit (CPU), general purpose GPU computing transfers the computational part to graphic cards.

The implicit parallelism of GPUs makes them especially suited for problems with large data sets, thanks to the thousands of cores available. However, this characteristic also requires great attention to memory access.

We shall compare different GPU implementations for both the matrix-matrix multiplication and the Poisson problem in order to better understand its unique architecture and thus be able to improve our code performance.

2 Matrix multiplication

2.1 CPU matrix multiplication with BLAS

In order to make a fair comparison of the GPU implementation for matrix multiplication against the CPU code, we shall make use of the DGEMM routine of the CBLAS library. This routine has been thoroughly optimized for many years and therefore represents an upper bound for the CPU version. We are going to use this function as a reference to estimate the speed up of the following GPU versions.

The only difference with respect to the code used in the first assignment is that the matrices are passed by the driver as single pointers.

```
extern "C" {
    #include <cblas.h>

    void matmult_lib(int m, int n, int k, double *A, double *B, double *C){
        double alpha = 1.0, beta = 0.0;
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);
    }
}
```

2.2 GPU 1: sequential version with one thread

As a first step, we created a basic sequential version of a CUDA kernel, and we ran it with one block of a single thread. According to the native matrix multiplication, the Kernel presents three nested loops that allow the thread to move along all rows and columns of the two matrices.

We have tested the code for different matrix sizes and, as expected, we can't see any improvement in performance when comparing with the CBLAS routine. Apart from the fact that the CPU version has been properly optimized, the need of transferring memory between the host and the device in the GPU implementation kills its performance.

Moreover, we had to stop its execution at $(m, n, k) = (1024, 1024, 1024)$, since we were running out of the maximum allowed time on the cluster.

The GPU usage for both computations and memory has also been measured by comparing with the GPU's especifications:

$$\frac{R_{code}}{R_{peak}}\% \tag{2.1}$$

where R_{code} and R_{peak} are:

$$R_{code} = \frac{\text{number floating point operations}/10^9}{\text{kernel run-time}} \tag{2.2}$$

$$R_{peak} = \frac{3584 \text{ cores} \cdot 1.53 \text{ GHz} \cdot 2 \text{ flops per core}}{3} = 3655.68 \text{ Gflops} \tag{2.3}$$

for the memory transfer:

$$\frac{Bandwidth_{code}}{Bandwidth_{peak}}\% \tag{2.4}$$

where the $Bandwidth_{code}$ and $Bandwidth_{peak}$ are:

$$R_{code} = \frac{(Bytes_{read} + Bytes_{written})/10^9}{\text{memory run-time}} \tag{2.5}$$

$$\text{bandwidth}_{peak} = 5.005 \text{ GHz} \cdot \frac{384}{8} \text{ bytes} \cdot 2 = 480.48 \text{ GB/s} \tag{2.6}$$

The results are gathered in table 2.1.

Method	**Bound**	**GPU usage**
GPU 1	Compute	0.00%
	Memory	1.79%

Table 2.1: GPU compute and memory usage for GPU 1 version.

We will be estimating these parameters in all the following implementations. The GPU specifications correspond to the TITAN X Pascal graphic card. The values corresponding to our code were measured using the nvprof command.

2.3 GPU 2: Naive version

In this second version we increase the number of threads in the Kernel launch so that every element in the result matrix C will be calculated by a single thread.

Understanding the GPU architecture is important when launching multiple threads. A GPU can run in parallel an specific number of operations at a time. When the number of threads (parallel operations), invoked in the function call is greater than the hardware constraints, some threads will be waiting for the others to finish their execution.

The hardware of a GPU is subdivided in multiprocessor units. In a multiprocessor the same sequential operation can be performed for different data at the same time. In total a multiprocessor can run 32 threads at the same time, this group of threads is called Warp.

In software, the threads are combined in a grid that contains a global space of memory (accessible for all threads). A grid is divided into blocks, where each block has its own independent space of memory that can be accessed only by threads of the same block.

Taking all the previous considerations into account, in terms of performance, it is advisable to chose a multiple of 32 when deciding the block size so that all warps are running at full capacity. However, due to hardware restrictions no more than 1024 threads are allowed in a block. Thus, we can estimate the number of blocks needed (grid size) by dividing the problem size by the block dimensions. It is important to notice that, since the number of blocks must be an integer, for some problems we will have to invoke more threads than needed. That is why the CUDA implementation includes an if clause that checks whether or not the thread falls within the matrix dimensions.

The following is the secondary code that allocates the memory in the devices and copies the matrices between the host and the device.

```
extern "C" {
    void matmult_gpu2(int m, int n, int k, double *A, double *B, double *C) {
        double* d_A, * d_B, * d_C;
        cudaSetDevice(2);
        cudaMalloc((void**)&d_A, m*k * sizeof(double));
        cudaMalloc((void**)&d_B, k*n * sizeof(double));
        cudaMalloc((void**)&d_C, m*n * sizeof(double));


        cudaMemcpy(d_A, A, m*k * sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, k*n * sizeof(double), cudaMemcpyHostToDevice);

        // Initialize the output matrix with zeroes.
        cudaMemset(d_C, 0, m*n * sizeof(double));
        dim3 BlockDim(16,16);
        dim3 NumBlocks((m-1)/16+1,(n-1)/16+1);
        m2<<<NumBlocks,BlockDim>>>(m, n, k, d_A, d_B, d_C);
        cudaDeviceSynchronize();

        cudaMemcpy(C, d_C, m*n * sizeof(double), cudaMemcpyDeviceToHost);

        cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    }
}
```

The following is the CUDA Kernel that is executed in parallel:

```
__global__ void m2(int m, int n, int k, double *A, double *B, double *C) {
  double sum = 0;
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;

  if (i < m && j < n){
    for (int h = 0; h < k; h++) {
      sum += A[i*k + h] * B[h*n + j];
    }
    C[i*n + j] = sum;
  }
}
```

The three nested loops of the sequential implementation have now turned into a single loop that moves every thread through rows and columns of A and B, and computes and stores the result in C. We have also created a new variable *sum* to keep in the registers the value of the partial summation in every iteration, and with a single access at the end of the loop, we can update the value in the C matrix.

Method	**Bound**	**GPU usage**
GPU 2	Compute	3.91%
	Memory	1.12%

Table 2.2: GPU compute and memory usage for GPU 2 version.

The code has been again tested and compared to the CPU BLAS implementation. The results of the performance measure are also collected in table 2.2. Although as indicated by the peak performance percentage our code is not using all the graphical card power, the improvement with respect to the CPU version is now obvious when using the TITAN X Pascal graphic card. This can not be seen in figure 2.1 since we submitted the batch job to the Tesla K40 GPU, in which clearly the CPU library still beats the GPU implementation.

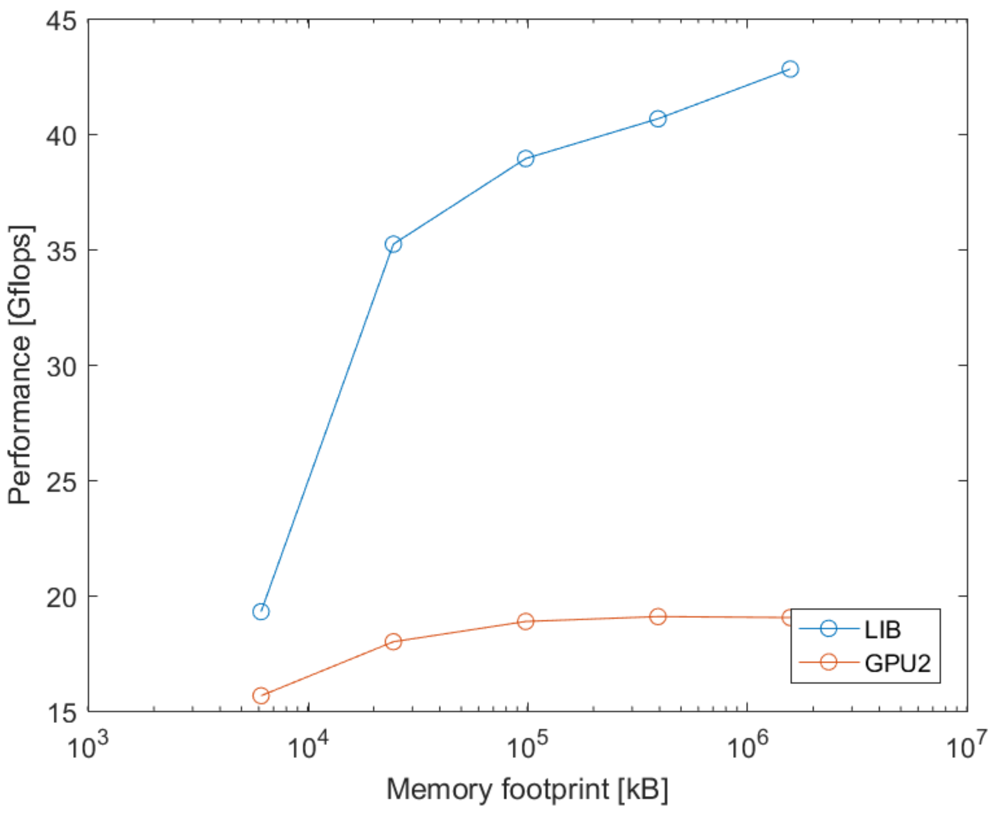

Figure 2.1: GPU2 against CPU BLAS library, there is no performance improvement when using Tesla K40 graphic cards.

2.4 GPU 3 / GPU 4: Loop unrolling

In the third version, we have been asked to apply loop unrolling to the Naive implementation. The main propose of this technique is to create spatial locality when accessing the elements of the three matrices. Now a single thread reads two rows and two columns of A and B, and computes and stores the result in C.

Although it is known that in a sequential implementation in C the access is most efficiently made in row major order, in a GPU, since all threads run in parallel at the same time, it is advisable to follow the column major order. Therefore, and according to this theory, one thread should compute its own element in the matrix C and the one below.

We have created and tested three different versions were the second element is: below the first one, to the right of the first one and one block size below the first one.

```
__global__ void m3_1(int m, int n, int k, double *A, double *B, double *C) {

  double sum1 = 0,sum2 = 0;
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;
  i *= 2;
  if (i < m && j < n){
      for (int h = 0; h < k; h++) {
        sum1 += A[i*k + h] * B[h*n + j];
        if (i+1 < m) sum2 += A[(i+1)*k + h] * B[h*n + j];
      }
  C[i*n + j] = sum1;
  if (i+1 < m) C[(i+1)*n + j] = sum2;
  }
}
```

Listing 2.1: Version with the second element below the first.

```
__global__ void m3_2(int m, int n, int k, double *A, double *B, double *C) {

  double sum1 = 0,sum2 = 0;
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;
  j *= 2;
  if (i < m && j < n){
      for (int h = 0; h < k; h++) {
        sum1 += A[i*k + h] * B[h*n + j];
        if (j+1 < n) sum2 += A[i*k + h] * B[h*n + j+1];
      }
  C[i*n + j] = sum1;
  if (j+1 < n) C[i*n + j + 1] = sum2;
  }
}
```

Listing 2.2: Version with the second element to the right of the first.

```
__global__ void m3_3(int m, int n, int k, double *A, double *B, double *C) {

  double sum1 = 0,sum2 = 0;
  int i = blockIdx.x*blockDim.x*2+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;
  if (i < m && j < n){
      for (int h = 0; h < k; h++) {
        sum1 += A[i*k + h] * B[h*n + j];
        if (i+blockDim.x < n) sum2 += A[(i+blockDim.x)*k + h] * B[h*n + j];
```

```
    }
  C[i*n + j] = sum1;
  if (i+blockDim.x < n) C[(i+blockDim.x)*n + j] = sum2;
  }
}
```

Listing 2.3: Version with the second element one block size below the first.

Surprisingly, when testing the function with the Pascal GPU for all three implementations it turned out that the fastest one was in fact the second version. We then switched to the Tesla GPU, but again the results showed the same just that in this GPU the differences were less accentuated. This is can be seen in figure 2.2.

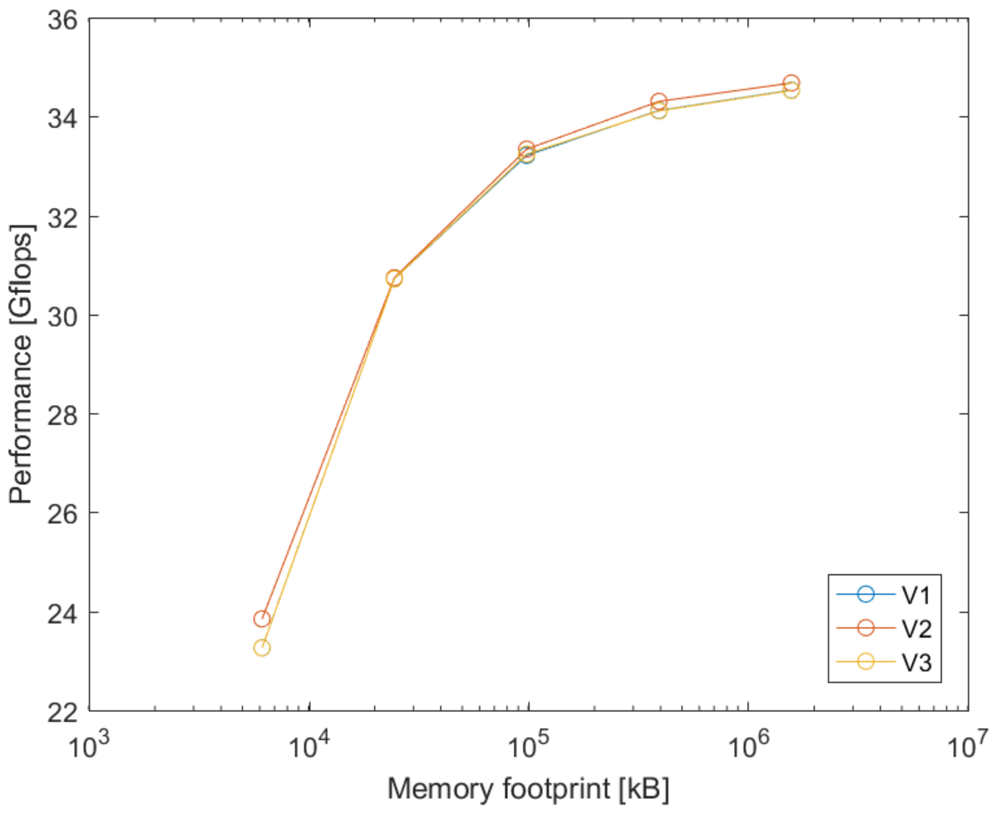

Figure 2.2: GPU3 – V1: neighbor below, V2: neighbor to the right, V3: One block dimension below.

We then tried to gain performance by reducing the number of blocks by a factor of 4. In this case two versions were created, one in which a single thread computes its element and the three ones below, and one in which it computes four elements to the right. Again, the latter turned out to be quicker.

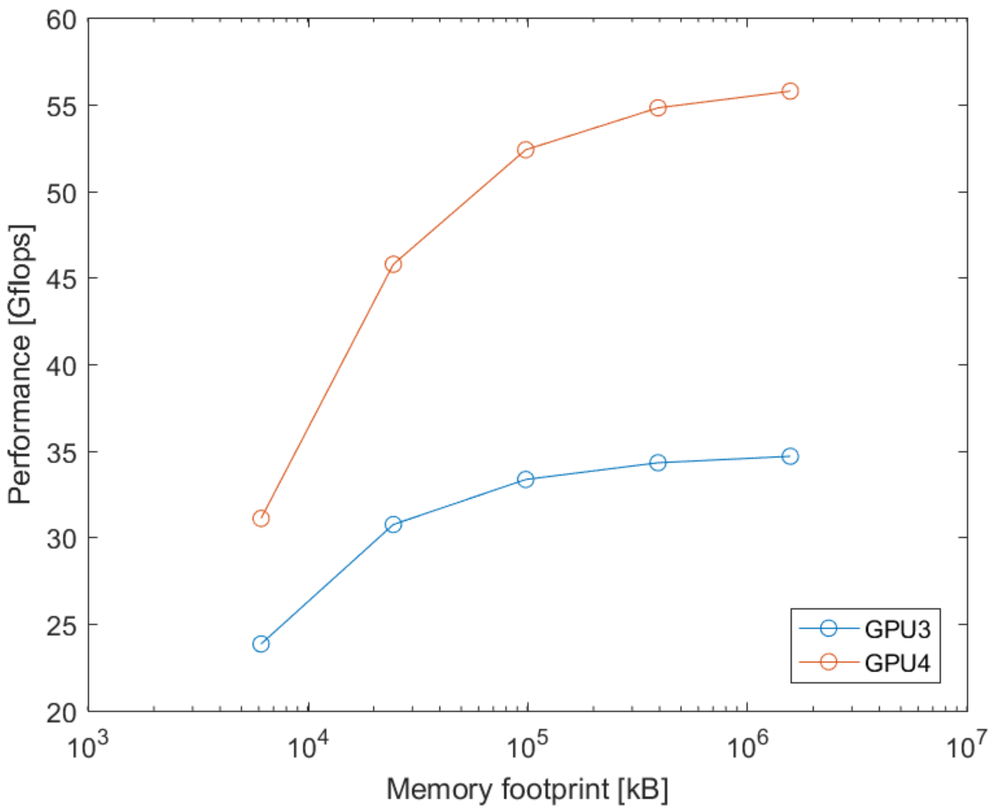

Figure 2.3: GPU3 neighbor to the right against GPU4 neighbor to the right

Comparing the performance for loop unrolling by a factor of 2 and 4, we see that unrolling by twice the amount gives an improvement by a factor of 1.5–2, for large problems, since the number of threads decreases.

Method	**Bound**	**GPU usage**
GPU 3	Compute	6.97%
	Memory	1.40%
GPU 4	Compute	10.28%
	Memory	1.29%

Table 2.3: GPU compute and memory usage

2.5 GPU 5: Shared Memory

As we mentioned when we described the GPU's architecture, each block has its own space in memory that can be used to share information between threads of the same block. The access to this memory is much faster than the access to global memory.

Knowing this, we could create a single block with one thread per element of the C matrix and fetch from the global memory to the shared space. Unfortunately for large matrices this is not possible since the size of the block is limited to 1024 threads.

Therefore the main idea is to divide the matrices A and B in small pieces so as to be able to calculate those portions separately and add them together by looping through the rows and the columns of A and B. Before doing all the computations the data from the two matrices A and B is stored in two smaller ones A_s and B_s that are located in the shared memory.

```
__global__ void m5(int m, int n, int k, double *A, double *B, double *C) {

    // This variable 'two_blocks' is passed to the kernel at its invocation
    // m5<<<gridDim,blockDim, (blockDim.x*blockDim.y * 2 * sizeof(double))>>>(parameters);
    // and we have to "split" it manually into the two variables we want to use.
    extern __shared__ double two_blocks[];
    __shared__ double* A_s;
    A_s = &two_blocks[0];
    __shared__ double* B_s;
    B_s = &two_blocks[blockDim.x*blockDim.y];

    int topleft_row_A = blockIdx.y*blockDim.y*k;
    int topleft_col_B = blockIdx.x*blockDim.x;

    // The blocks HAVE to have the same size, otherwise this matrix-matrix
    // mult on the small matrices cannot work.
    const int bl_side = blockDim.x;
    double sum;

    for (int w = 0; w < k; w += bl_side) {

        // We have to iterate over the two lines until reaching k.
        int topleft_row_A_curr_block = topleft_row_A + w;
        int topleft_col_B_curr_block = topleft_col_B + w*n;

        A_s[threadIdx.y*bl_side + threadIdx.x] = A[topleft_row_A_curr_block +
                                                   threadIdx.y*k + threadIdx.x];
        // We just need each thread to load a single cell from the huge matrix
        // A & B, no matter if they don't load the same they are going to work on.
        B_s[threadIdx.y*bl_side + threadIdx.x] = B[topleft_col_B_curr_block +
                                                   threadIdx.y*n + threadIdx.x];

        __syncthreads();

        sum = 0.0;
        for (int it=0; it < bl_side; it++) {
            sum += ( A_s[threadIdx.y*bl_side + it] * B_s[bl_side*it + threadIdx.x] );
        }

        // This second barrier syncronization is needed because there could be
        // some threads that could repeat the w_for loop and change A_s and B_s
        // while other are still reading from them.
        __syncthreads();

        C[blockIdx.y*blockDim.y*n + threadIdx.y*n +
            blockIdx.x*blockDim.x + threadIdx.x] += sum;
    }
}
```

Listing 2.4: Version with the manual loop blocking in the shared memory.

The function allocates dynamically the memory for the two small shared matrices. The memory needed for the matrices A_s and B_s is passed as a third argument in the Kernel launch and then split inside the Kernel.

The code has been again tested and compared to the CBLAS library, the results are shown in figure 2.4 and table 2.4.

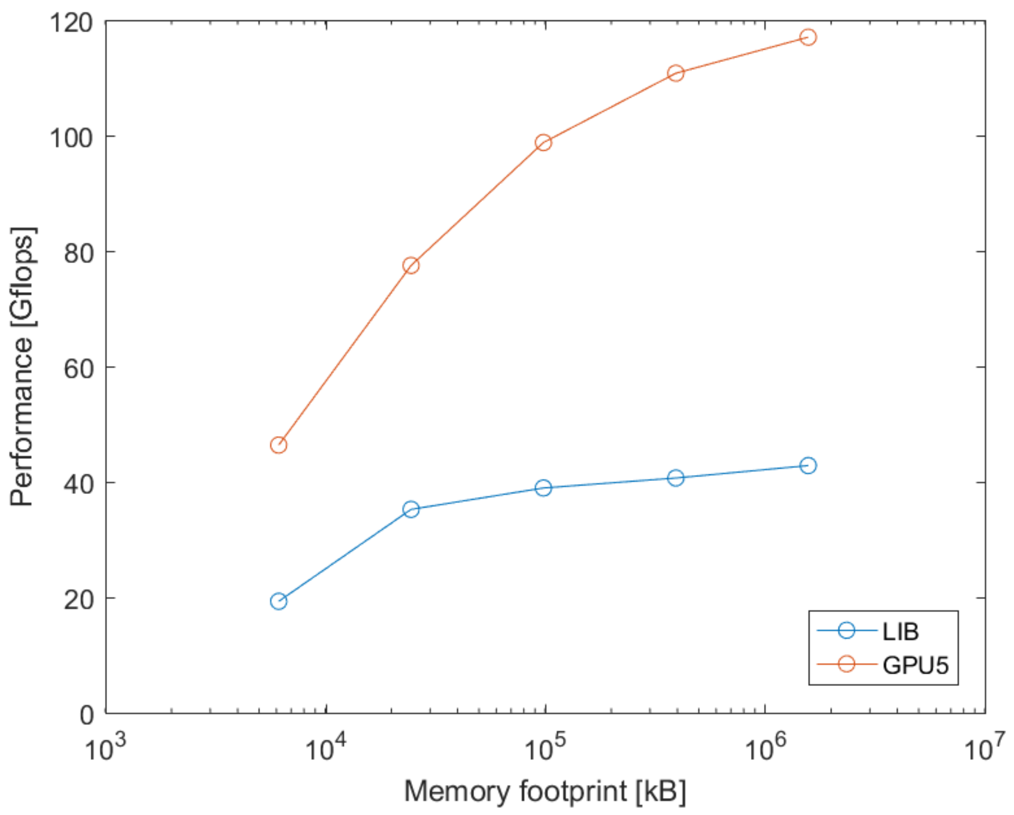

Figure 2.4: GPU5 shared memory against CPU BLAS library

Now, it is obvious that the GPU implementation is more efficient even for the Tesla graphic card. The difference becomes even larger when working with the Pascal GPU.

Method	**Bound**	**GPU usage**
GPU 5	Compute	6.64%
	Memory	1.56%
GPU Lib	Compute	9.78%
	Memory	2.35%

Table 2.4: GPU compute and memory usage.

We have also launched the kernel with different block sizes, 4, 8, 16 and 32 threads. The larger the block the more elements can fit in the small shared matrices and the more computations can be performed in each block. The performance seems to increase with the block size until it drops for 32 threads. The best choice is therefore 16 threads.

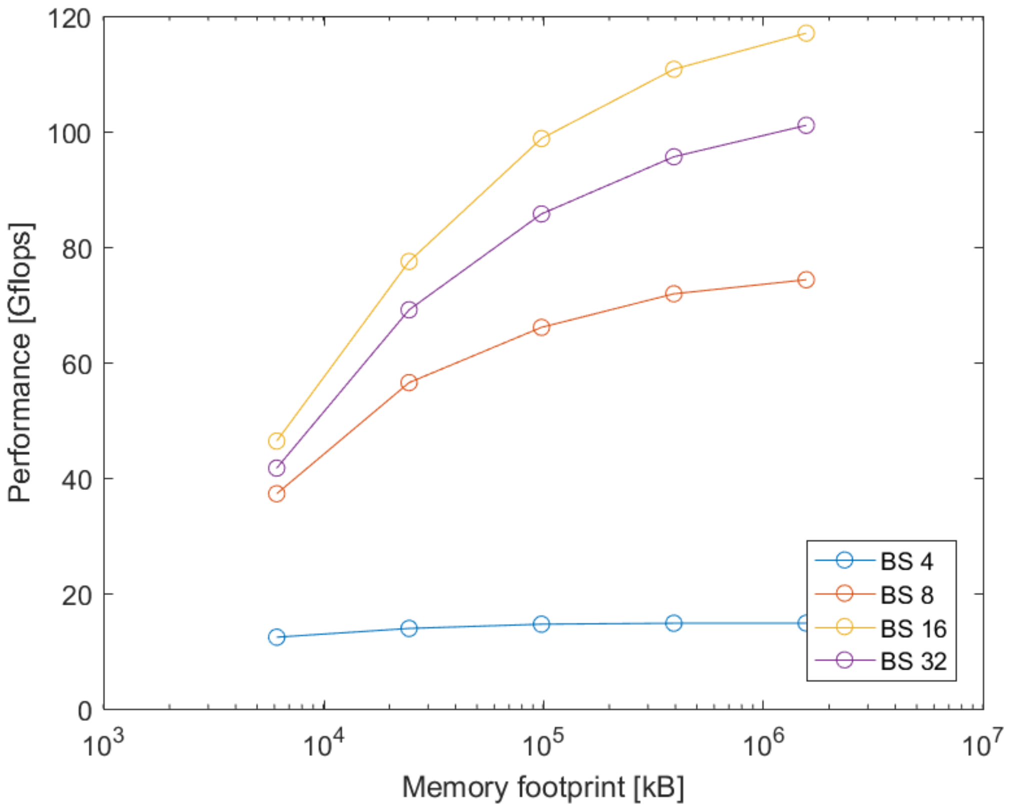

Figure 2.5: GPU5 with different block sizes

2.6 GPU CUBLAS library

In the last version of the matrix multiplication we call the DGEMM routine from the Nvidia library CUBLAS.

```
extern "C" {
  void matmult_gpulib(int m, int n, int k, double *A, double *B, double *C) {
    cudaSetDevice(2);

    cublasHandle_t handle;
    cublasCreate(&handle);

    double alpha = 1.0, beta = 0.0;

    double* d_A, * d_B, * d_C;
    cudaMalloc((void**) &d_A, m*k * sizeof(double));
    cudaMalloc((void**) &d_B, k*n * sizeof(double));
    cudaMalloc((void**) &d_C, m*n * sizeof(double));
    cudaMemcpy(d_A, A,  m*k * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B,  k*n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemset(d_C, 0,  m*n * sizeof(double));

    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &alpha, &d_A[0], k, &d_B[0], n,
     &beta, &d_C[0], n);

    cublasDestroy(handle);

    cudaMemcpy(C, d_C,  m*n * sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

  }
}
```

Listing 2.5: Version using the CUDA BLAS implementation.

As can be seen in the listed results below, we analyzed the function with the profile command and it appeared that it launches more than one CUDA Kernel, and therefore makes better use of the GPU power (see table 2.4).

```
==21014== NVPROF is profiling process 21014, command: ./matmult_f.nvcc2 gpulib 1600 1600
    1600
==21014== Profiling application: ./matmult_f.nvcc2 gpulib 1600 1600 1600
==21014== Profiling result:
==21014== Metric result:
Invocations                               Metric Name                             Metric
    Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: dgemm_sm35_ldg_nn_128x8x64x16x16
          2                             flop_count_dp   Floating Point Operations(Double
    Precision)   314867712   314867712   314867712
    Kernel: dgemm_sm35_ldg_nn_64x8x128x8x32
          2                             flop_count_dp   Floating Point Operations(Double
    Precision)   341106688   341106688   341106688
    Kernel: dgemm_sm_heavy_ldg_nn
          2                             flop_count_dp   Floating Point Operations(Double
    Precision)  7552106496  7552106496  7552106496
```

We tested the function for different problem sizes and compared to our best implementation. The results can be seen in figure 2.6. GPU lib seems to be more efficient for both computations and memory transfer. In contrast to our implementations the graph shows that the performance of the CUBLAS routine increases linearly with the problem size.

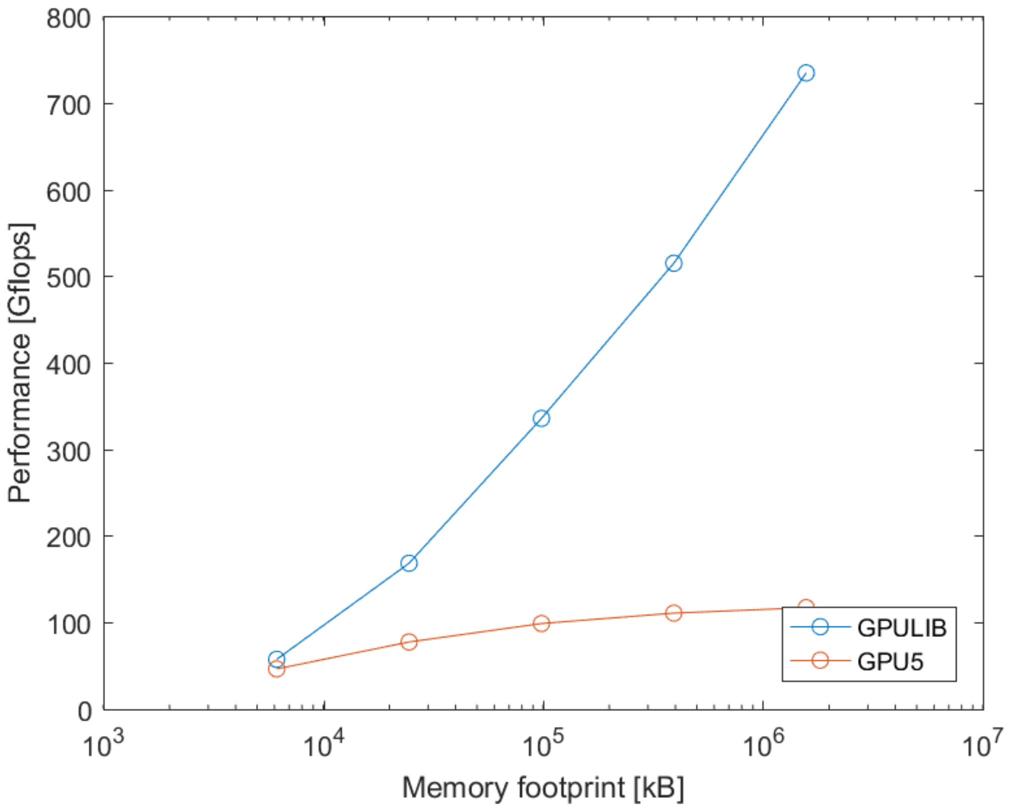

Figure 2.6: GPU5 against the CUBLAS library.

Finally in figure 2.7 we present in a single plot all the implementations we used in this report, using a base 10 logarithmic scale on both axes.
We can clearly see that the CUDA implementation performs much better than any other implementation we created or used.

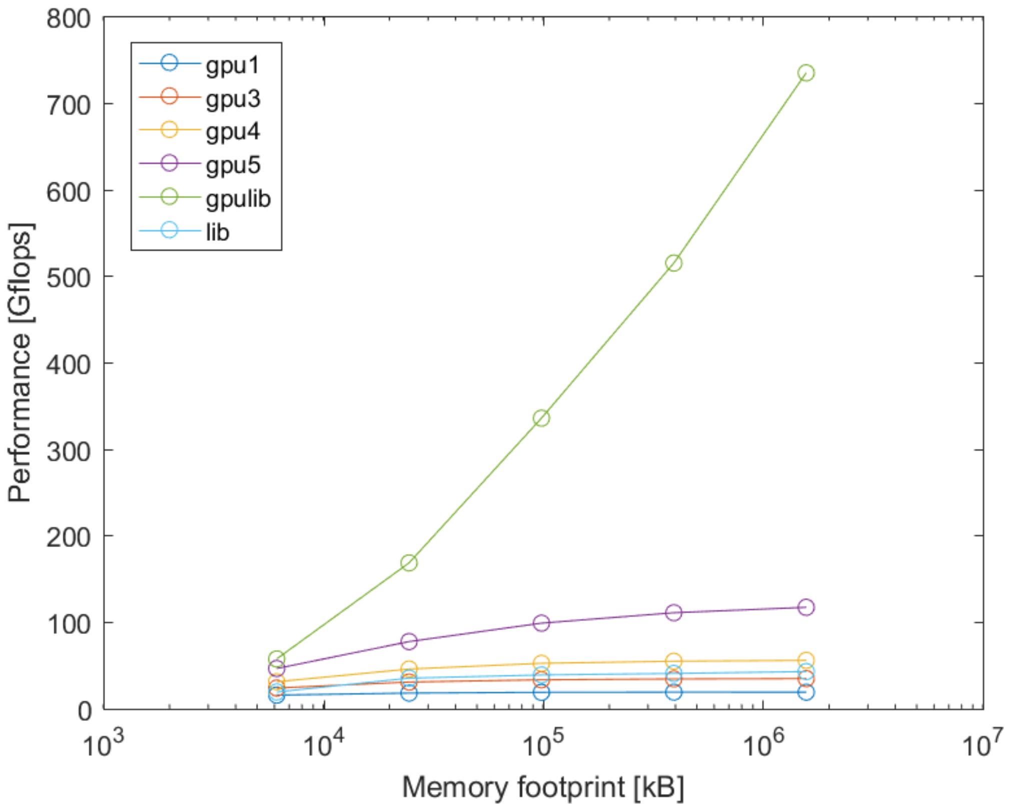

Figure 2.7: Memory footprint (kB) versus execution speed (GFLOPS) for all the matrix-matrix multiplication methods we implemented (the first version, with the single thread, is excluded).

3 The Poisson problem

Last week we implemented a solution to the Poisson problem using the Jacobi method and OpenMP to parallelize the work on multiple CPU cores. The OpenMP version was modified to only use the number of iterations as a stopping criteria (so removing the threshold calculation), and is used for comparison with the three different Cuda implementations here presented.

In this part, we will briefly explain the three implementations followed by a comparison.

3.1 Single thread version

The first GPU version was created by having the controlling loop running on the CPU and only creating a single GPU thread for updating the grid.

```
double ts, te;
ts = omp_get_wtime();
for(int k = 0; k<kmax; k++){
    jacobi<<<1,1>>>(d_u1,d_u2,d_f,N,lambda2);
    jacobi<<<1,1>>>(d_u2,d_u1,d_f,N,lambda2);
}
cudaDeviceSynchronize();
te = omp_get_wtime() - ts;
```

Note that the grid is updated twice for each iteration by switching the pointers. The same method is used for the other GPU implementations.

The Jacobi function running on the GPU then uses two for loops two update all cells of the grid.

```
__global__ void jacobi(double * uold, double * unew, double * f, int N, double lambda2){
    int M = N+2;
    for (int i = 1; i < N+1; i++){
        for (int j = 1; j < N+1; j++){
            unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
                uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
        }
    }
}
```

3.2 2D block, 2D grid version

To utilize the multiple cores on the GPU, another version was created which updated the whole grid in parallel. The main function running on the CPU was thus changed to create threads in blocks of 32 by 32 in a multiple adding up to more than the problem size. This means that more threads than needed will be created, but the extra threads will not do any floating point operations on the GPU.

```c
int blockSize = 32;
dim3 dimBlock(blockSize,blockSize,1);
int gridSize = 1 + ((N - 1) / blockSize); // N/blockSize round up.
dim3 dimGrid(gridSize,gridSize,1);

double ts, te;
ts = omp_get_wtime();
for(int k = 0; k<kmax; k++){
    jacobi<<<dimGrid,dimBlock>>>(d_u1,d_u2,d_f,N,lambda2);
    jacobi<<<dimGrid,dimBlock>>>(d_u2,d_u1,d_f,N,lambda2);
}
cudaDeviceSynchronize();
te = omp_get_wtime() - ts;
```

This removes the need for having loops inside the Jacobi function, but indexing becomes more complicated.

```c
__global__ void jacobi(double * uold, double * unew, double * f, int N, double lambda2){
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int index = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) +
    threadIdx.x;

    if(index < N*N){
        int M = N+2;
        int i = index + M + 1 + 2 * (index / N);

        unew[i] = 0.25 * (uold[i-1] + uold[i+1] + uold[i-M] + uold[i+M] + lambda2*f[i]);
    }
}
```

3.3 Multi GPU version

We wrote a version of the Poisson problem that is capable of taking advantage of multiple GPUs.

The basic idea is to split the job on multiple physical devices: the major drawback is moving the data between the two devices. By keeping half of the whole matrix on each GPU and moving only the updated line that represent the border of the other matrix, we managed to reduce the bandwidth usage to the barest minimum.
The room is simulated with a matrix made of N internal points (that represent the grid resolution) along with another layer of points all around that stand for the walls: the total size is therefore $(N + 2) \times (N + 2)$.
We divide the matrix at half of its height, so each GPU takes one half: the dimensions of each "half matrix" are now $(N + 2)$ on the x axis, and $(N/2 + 2)$ (rounded up for the second matrix) on the y axis. The important thing to remember is that now one of the four borders (for both the matrices) changes at every iteration, since it is part of the other matrix.

Therefore, we have to copy the first row from the bottom matrix to overwrite the old wall in the upper matrix, and we have to copy the last row from the upper matrix to overwrite the old wall in the bottom matrix.

As we already said in the first implementation, at every iteration of the for loop on the variable k we actually perform two iterations on the matrix, since we shift the pointers between the old and the new u matrices in both the GPUs. This means that we actually stop the execution of the program when reaching the double of the threshold set in the command line, but it does not affect the performance computation.

To avoid using too much pointers, we preferred managing the matrices using some offset values like HALF and HALF_1 (for the last line before the half).

```
int blockSize = 32;
dim3 dimBlock(blockSize,blockSize,1);
int gridSize = 1 + ( (M - 1) / (2*blockSize) ); // M/(2*blockSize) round up.
dim3 dimGrid(gridSize, gridSize, 1);

double ts, te;
ts = omp_get_wtime();
for(int k = 0; k<kmax; k++){
    cudaMemcpy(d1_u1, &d0_u1[HALF_1], M * sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(&d0_u1[HALF], &d1_u1[M], M * sizeof(double), cudaMemcpyDefault);

    // Update u2
    cudaSetDevice(0);
    jacobi<<<dimGrid, dimBlock>>>(d0_u1,d0_u2,d0_f,N,H0-2,lambda2);
    cudaSetDevice(1);
    jacobi<<<dimGrid, dimBlock>>>(d1_u1,d1_u2,d1_f,N,H1-2,lambda2);

    cudaMemcpy(d1_u2, &d0_u2[HALF_1], M * sizeof(double), cudaMemcpyDefault);
    cudaMemcpy(&d0_u2[HALF], &d1_u2[M], M * sizeof(double), cudaMemcpyDefault);

    // Update u1
    cudaSetDevice(0);
    jacobi<<<dimGrid, dimBlock>>>(d0_u2,d0_u1,d0_f,N,H0-2,lambda2);
    cudaSetDevice(1);
    jacobi<<<dimGrid, dimBlock>>>(d1_u2,d1_u1,d1_f,N,H1-2,lambda2);
}
cudaDeviceSynchronize();
te = omp_get_wtime() - ts;
```

The GPU code is basically the same as for the 2D block, 2D grid version, it is simply changed a bit to accommodate the rectangular grid of the half room, rather than the quadratic grid used for the full room in previous version.

```
__global__ void jacobi(double * uold, double * unew, double * f,
                       int width, int height, double lambda2) {
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int index = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) +
    threadIdx.x;

    if(index < width*height){
        int M = width+2;
        int i = index + M + 1 + 2 * (index / width);

        unew[i] = 0.25 * (uold[i-1] + uold[i+1] + uold[i-M] + uold[i+M] + lambda2*f[i]);
    }
}
```

3.4 Comparison

To compare the performances of different implementations, they have been run multiple times as batch jobs in the provided cluster made of NVIDIA Tesla K40 graphic cards. The OpenMP implementation has been run on a cluster with Intel Xeon E5 CPUs.
In the batch jobs we used problem sizes of N in the range from 16 to 16384, but with a relatively low number of iterations, so the run time was pretty quick for all the implementations, except for the single threaded CUDA version, which took 10 minutes for $N = 2048$, and could not end before $N = 4096$ within the 30 minutes time limit.

When we used the Jacobi method, three matrices of size $N + 2$ are needed, and thus the memory footprint is calculated as

$$Mem = 3 \cdot (N + 2)^2 \cdot \text{sizeof}(double). \tag{3.1}$$

The number of floating point operations per second, *FLOPS*, is calculated as

$$R = \frac{k_{max} \cdot N^2 \cdot 6}{t}, \tag{3.2}$$

since it takes 6 floating point operations to update every point of the $N \times N$ grid, which is repeated k_{max} times over t seconds. Note that this number is multiplied by two for the GPU implementations, as they do two updated for every iteration of the $k = 0 \ldots k_{max}$ loop. When running the program as `nvprof --metric flop_count_dp ./poisson` it comes to the same results as the numerator of the fraction in equation (3.2).

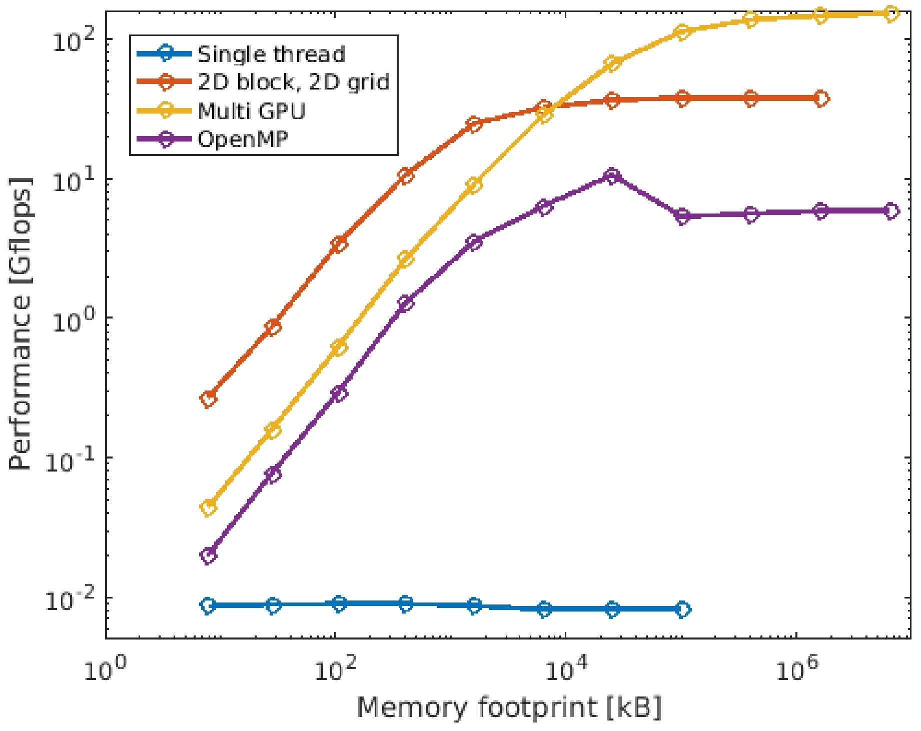

Figure 3.1: Memory footprint vs performance measured in Gflops with double logarithmic axes.

In Figure 3.1 a memory footprint vs flops plot is seen for the four versions. Note the double logarithmic axes. It is seen that single thread version is slowest, which should be expected, and that both multi threaded Cuda implementations are faster than the OpenMP. For small problem sizes, the single GPU version is quickest while the multi GPU version is quicker for larger problem sizes (it actually does the floating point operations more than twice as quick for some problem sizes, even though it is just two GPUs vs one GPU).

Another metric is the speedup, which is simply calculated as ratio of the time of the CPU implementation and the GPU implementation

$$\text{Speedup} = \frac{CPU[s]}{GPU[s]}. \tag{3.3}$$

When the speedup is above 1, it means that the execution is quicker on the GPU implementation. In Figure 3.2 the speedup can be seen for the three Cuda implementations. The information here is largely the same as in Figure 3.1, but the scaling makes more obvious the difference between running the problem on a single or on two GPUs.
Before reaching 10^4 kBs (approximately 10 MBs) the single GPU version should be preferred, while beyond that threshold using multiple physical graphic cards leads to an enormous gain in performances.

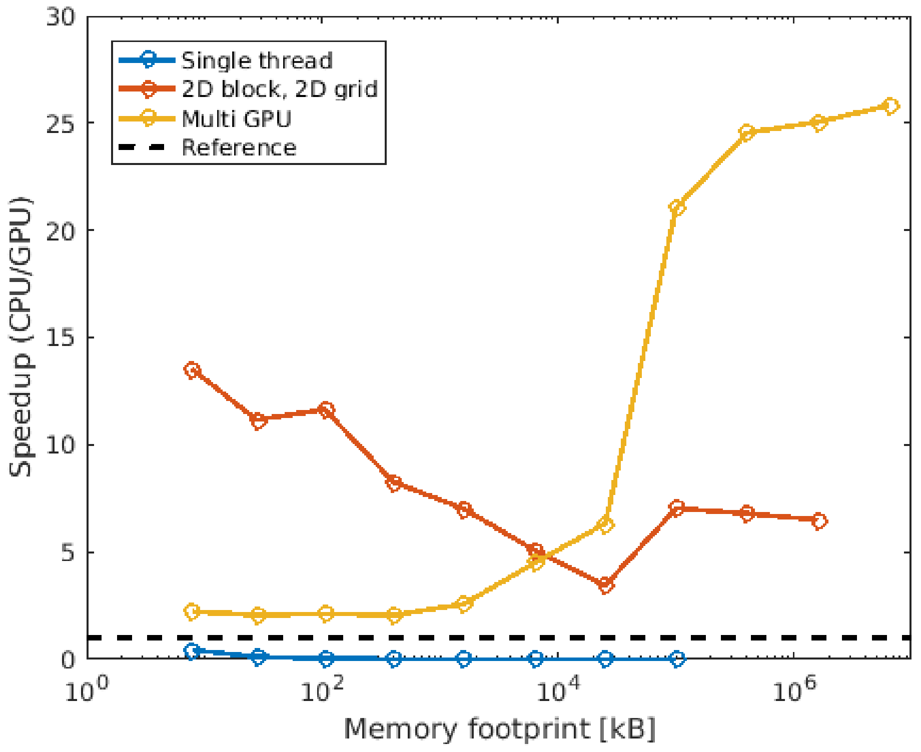

Figure 3.2: Memory footprint vs speedup (compared to OpenMP time). The reference line is a constant going through 1, marking the separation between implementations quicker and slower than the OpenMP implementation.