GPU Matrix Multiplication/ GPU Poisson Problem

Authors

Carlos Marcos Torrejón - s202464
Raúl Ortega Ochoa - s202489
Santiago Gutiérrez Orta - s200140

02614 High-Performance Computing Jan 21

January 25, 2021

Contents

1 Introduction **1**
1.1 Experimental Hardware Specification 1

2 GPU Matrix Multiplication **2**
2.1 Naive matrix multiplication 2
2.2 Improved Matrix Multiplication 3
2.3 Final Matrix Multiplication 6
2.4 Cublas Matrix Multiplication 8

3 GPU Poisson Problem **10**
3.1 Sequential Kernel Jacobi 10
3.2 Naive Kernel Jacobi 12
3.3 Dual GPU Jacobi 14
3.4 Norm GPU Jacobi 17

4 Conclusion **20**

List of Figures **I**

Part	studyno.	Name
I	s202489	Raúl Ortega Ochoa
	s202464	Carlos Marcos Torrejón
	s200140	Santiago Gutierrez Orta
II	s202489	Raúl Ortega Ochoa
III	s202464	Carlos Marcos Torrejón
IV	s200140	Santiago Gutierrez Orta

1 Introduction

The purpose of this project is to show a contrast between the performance of the CPU and the GPU for two different cases. Matrix multiplication and the poisson problem solved with the Jacobi method. These problems where performed on the CPU on previous assignments on lectures.

1.1 Experimental Hardware Specification

```
CPU

Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  24
On-line CPU(s) list:     0-23
Thread(s) per core:      1
Core(s) per socket:      12
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   79
Model name:              Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Stepping:                1
CPU MHz:                 2200.000
CPU max MHz:             2900.0000
CPU min MHz:             1200.0000
BogoMIPS:                4389.79
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                30720K
NUMA node0 CPU(s):       0-11
NUMA node1 CPU(s):       12-23
```

```
GPU

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.23.05    Driver Version: 455.23.05    CUDA Version: 11.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  A100-PCIE-40GB      On   | 00000000:37:00.0 Off |                    0 |
| N/A   55C    P0    42W / 250W |      0MiB / 40536MiB |      0%   E. Process |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
```

2 GPU Matrix Multiplication

2.1 Naive matrix multiplication

CBLAS as a reference

In order to do a fair comparison of the CPU version with the following GPU versions, we use as a reference for the speedup the DGEMM routine of the CBLAS library as in the previous assignment. It is introduced in the cuda code with an "external" linker to C.

```
/* Native CBLAS CPU implementation of matrix multiplication */
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);
}
```

Sequential version in one thread

matmult_gpu1()

This first version is carried out with 3 nested loops which move a block with a single thread through the rows/columns of each matrix.

GPU-version-1 kernel:

```
    __global__ void matmult_gpu1_kernel(int M, int N, int K, double* A, double *B, double* C
        ) {
        double temp = 0.0;

        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                temp = 0.0;
                for (int k = 0; k < K; k++) {
                    temp += A[i*K + k] * B[k*N + j];
                }
                C[i*N + j] = temp;
            }
        }
    }
```

This kernel is defined with grid-size and block-size equal to 1 to fulfill the conditions stated previously. Upon analysing the performance with a profiler we observed low occupancy with 1.86% and very low memory and SM.

Naive version one thread per element

matmult_gpu2()

On this version, we use the characteristics of the GPU for optimize the version, so that,each output element C_{ijk}, was obtained on a different thread. This is accomplished by using potency of 2 integers as block-size values. This favour the GPU architecture so that we make use of the full warps (blocks of 32 threads which run the same sentence at a time) and bypass scheduling policies. There is a limit in the total block-size of 1024 defined by the hardware.

The code of the kernel in this version is presented above.

```
1
2       /* part 2: naive implementation in GPU (one thread per element in C) */
3       __global__ void matmult_gpu2_kernel(int M, int N, int K, double* A, double* B, double* C
            ) {
4           double temp = 0.0;
5           int i = blockIdx.x * blockDim.x + threadIdx.x;
6           int j = blockIdx.y * blockDim.y + threadIdx.y;
7
8           if (i < M && j < N){ // ensure that the extra threads do not do any work
9               for (int step = 0; step < K; step++) {
10                  temp += A[j*K + step] * B[step*N + i];
11              }
12              C[j*N + i] = temp;
13          }
14      }
```

and the definition of Block number per grid and threads per block was as shown in the following piece of code.

```
1 dim3 blocksPerGrid(((N-1) / BLOCK_SIZE+1), ((M-1) / BLOCK_SIZE+1));
2 dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
```

Upon inspecting the performance with a profiler, we found a higher occupancy 77.97% and higher memory usage.

The performance of gpu1 at higher sizes was not possible to be collected, since the runtime surpassed the 1hour limit of the hpc. However, the data collected is enough to compare with the other versions. As it was expected after the long runtime, the version gpu1 was seems to performance worse than even the CPU, which was not a surprise, since CPU perform better than GPU in sequential computation. With regard to the performance of the version gpu2, the version lib perform better at small matrix sizes. However, when the size of the matrix or memory footprint increase, the CPU reach a bound and the GPU2 version 2, in contrast to the first version, parallelize process, so that outperforming the CPU version.

2.2 Improved Matrix Multiplication

matmult_gpu3()

The function *matmult_gpu3()* is a matrix multiplication function that uses the GPU as in the previous function *matmult_gpu2()*, but now each thread computes two elements

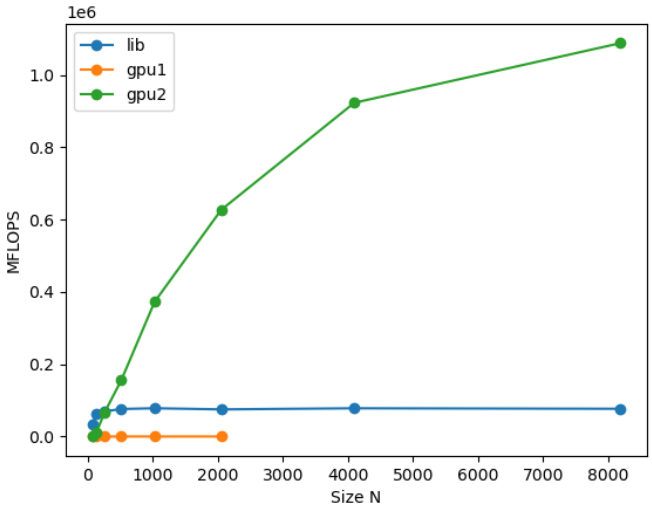

Figure 1: Mflops/s vs Memory footprint comparison.

of the resulting matrix C. This two elements can be contiguous or not, but for simplicity we will work with contiguous elements. There are then two options, we can either compute the contiguous element to the right or the one right below. In the code below the both versions are shown with commented lines.

```
__global__ void matmult_gpu3_kernel(int M, int N, int K, double* d_A, double* d_B, double*
    d_C) {
        double temp1 = 0.0;
        double temp2 = 0.0;

        int i = (blockIdx.x * blockDim.x + threadIdx.x); //*stride; // right
        int j = (blockIdx.y * blockDim.y + threadIdx.y)*stride; // below

        if (i < M && j < N) {
            for (int k = 0; k < K; k++) {
                temp1 += d_A[(i)*K + k] * d_B[k*N + j];
                /* Below neighbour */
                if (j+1 < N) { // only if not end
                    temp2 += d_A[(i)*K + k] * d_B[k*N + (j+1)];
                }
                /* Right neighbour
                if (i+1 < M) { // only if not end
                    temp2 += d_A[(i+1)*K + k] * d_B[k*N + (j)];
                } */
            }
            d_C[i*N + j] = temp1;
        /* below neighbour */
            if (j+1 < N) { // only if not end
                d_C[(i)*N + (j+1)] = temp2;
            }
            /* right neighbour
        if (i+1 < M) { // only if not end
        d_C[(i+1)*N + j] = temp2;
        }*/
        }
    }
```

As can be seen, there are now two elements being computed by the same thread, and some `if` clauses were added to avoid dimensions problem when computing the neighbour next to the limits of the matrix. Both implementations were tested for a range of different matrices to compare their performance. In the figure below the MFLOPS are plotted vs the Mmeory footprint (kBytes) for both implementations.

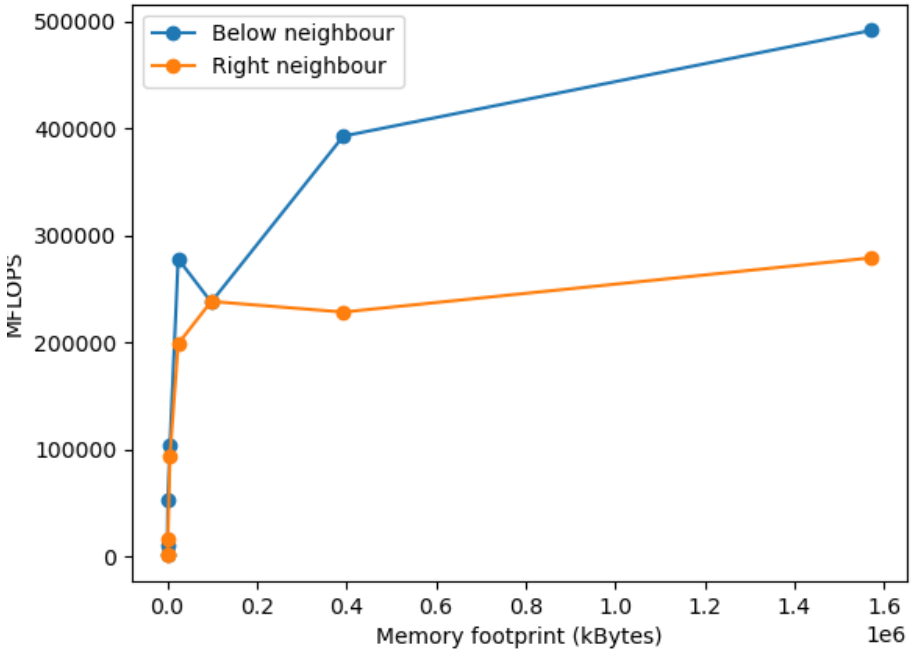

Figure 2: MFLOPS vs Memory footprint.

As can be seen from the graph in *figure 3*, the *below neighbour* implementation performs better. Since threads in the GPU run in parallel it is more efficient to use column major order. Upon inspecting the performance with a profiler, we found lower occupancy 32.27% and a lower memory usage 36.36% than in the previous implementation.

matmult_gpu4()

Function *matmult_gpu4()* is then constructed as a generalization of *matmult_gpu3()* where each thread now computes more than 2 elements of the resulting matrix C. The code below shows the device code, the kernel.

```
1 __global__ void matmult_gpu4_kernel(int M, int N, int K, double* d_A, double* d_B, double*
      d_C) {
2         double temp[stride_col][stride_row];
3         int sc, sr;
4         for (sc = 0; sc < stride_col; sc++){
```

```
5               for (sr = 0; sr < stride_row; sr++){
6                   temp[sc][sr] = 0.0;
7               }
8           }
9           int j = (blockIdx.x * blockDim.x + threadIdx.x)*stride_col;
10          int i = (blockIdx.y * blockDim.y + threadIdx.y)*stride_row;
11          if (i < M && j < N) {
12              for (int k = 0; k < K; k++) {
13                  for (sc = 0; sc < stride_col; sc++) {
14                      if (sc + j < N) {
15                          for (sr = 0; sr < stride_row; sr++) {
16                              if (sr + i < M) {
17                                  temp[sc][sr] += d_A[(i+sr)*K + k] * d_B[k*N + (j+sc)];
18                              }
19                          }
20                      }
21                  }
22              }
23              for (sc = 0; sc < stride_col; sc++) {
24                  if (sc + j < N) {
25                      for (sr = 0; sr < stride_row; sr++) {
26                          if (sr + i < M) {
27                              d_C[(i+sr)*N + (j + sc)] = temp[sc][sr];
28                          }
29                      }
30                  }
31              }
32          }
33      }
```

Next we can try different strides in columns and rows to find the optimal combination. In the next *figure 4*, for 9 different combinations (stride row x stride col) the MFLOPS are plotted for different memory footprints.

Once again we can see how the best performing stride combinations are the ones that have the large column part, namely, and in order, (2x4, 2x8), then squared strides (2x2, 4x4, 8x8) and the worst performing are (4x2, 8x4). The difference in performance is more noticeable for larger memory footprints, or larger matrices, where the best performing implementation is 2x4 (Orange in *figure 4*).

2.3 Final Matrix Multiplication

matmult_gpu5()

For implementing this version, we based our implementation on the matrix multiplication described in the shared memory section of "CUDA Programming guide"[1]. The idea of this version is split the matrix into sub-matrix so that it can be launched by the same block. Additionally the elements of the matrix A and B are allocated also in the shared memory. With this method we avoid to use the global memory for the calculation which access is much slower than in the shared memory.

The code for version gpu5 kernel is shown below.

```
1   #define BLOCKDIM 32
```

[1] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

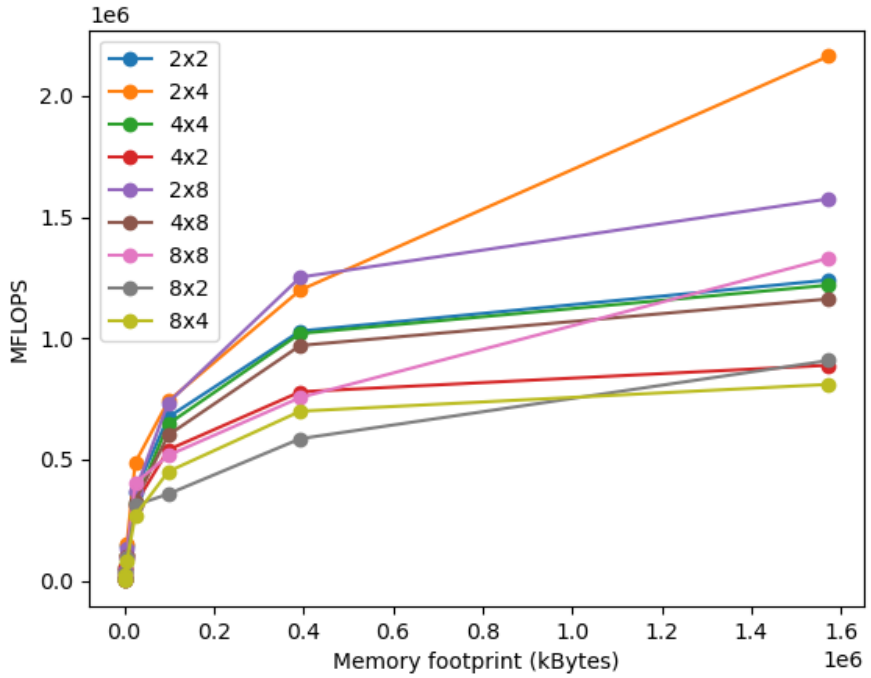

Figure 3: MFLOPS vs Memory footprint.

```
__global__ void matmult_gpu5_kernel(int m, int n, int k, double *d_a, double* d_b, double*
    d_c){
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    int row = threadIdx.y;
    int col = threadIdx.x;
    double intSum = 0.0;

    //Sub matrix C
    double *Csub = &d_c[blockRow*BLOCKDIM*n+blockCol*BLOCKDIM];

    for(int i=0; i< (k/BLOCKDIM); i++){
      //Get submatrices from A, B
      double *Asub = &d_a[blockRow*BLOCKDIM*k+BLOCKDIM*i];
      double *Bsub = &d_b[n*BLOCKDIM*i+blockCol*BLOCKDIM];

      __shared__ double As[BLOCKDIM][BLOCKDIM];
      __shared__ double Bs[BLOCKDIM][BLOCKDIM];

      As[row][col] = Asub[row*k+col];
      Bs[row][col] = Bsub[row*n+col];

      __syncthreads();

      for(int j=0; j<BLOCKDIM;j++){
        intSum += As[row][j]*Bs[j][col];
      }
      __syncthreads();
    }
    Csub[row*n+col] = intSum;
}
```

Upon inspecting the performance with a profiler, we found higher occupancy 81.60% and a higher memory usage 71.58% than in the previous implementation.

2.4 Cublas Matrix Multiplication

matmult_gpulib()

Finally, we will add a final implementation for matrix multiplication that we will use as benchmark to compare the rest of our implementations. *matmult_gpu6()* uses `cublasDgemm()`, which comes in cuBLAS library, an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA®CUDA™ runtime, allowing the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU) for faster computation. The code for this function can be seen below.

```
1  void matmult_gpulib(int m, int n, int k, double *A, double *B, double *C) {
2      cudaSetDevice(2);
3      /* declare handle */
4      cublasHandle_t handle;
5      cublasCreate(&handle);
6      double alpha = 1.0; // no prefactor
7      double beta = 0.0; // C matrix not involved
8
9      double* d_A, * d_B, * d_C;
10     cudaMalloc((void**) &d_A, m*k * sizeof(double));
11     cudaMalloc((void**) &d_B, k*n * sizeof(double));
12     cudaMalloc((void**) &d_C, m*n * sizeof(double));
13     cudaMemcpy(d_A, A,  m*k * sizeof(double), cudaMemcpyHostToDevice);
14     cudaMemcpy(d_B, B,  k*n * sizeof(double), cudaMemcpyHostToDevice);
15     cudaMemset(d_C, 0,  m*n * sizeof(double));
16
17     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);
18
19     cublasDestroy(handle);
20     cudaMemcpy(C, d_C,  m*n * sizeof(double), cudaMemcpyDeviceToHost);
21     cudaFree(d_A);
22     cudaFree(d_B);
23     cudaFree(d_C);
```

Finally we plot the MFLOPS vs Memory footprints for all the implementations described in this report, to obtain an overall picture of which versions perform the best. In the figure below this graph is shown.

As we could see there is some variation in the performance of the different version depending on the moment in which the version was run as it was warned in the lectures. As a upper bound we have the CUBLAS version which is the most optimized version of the matmult. Then it should be GPU5 version, since it is highly optimized splitting the matrix in order to use only shared memory. It is also weird that the GPU2 is above GPU3.

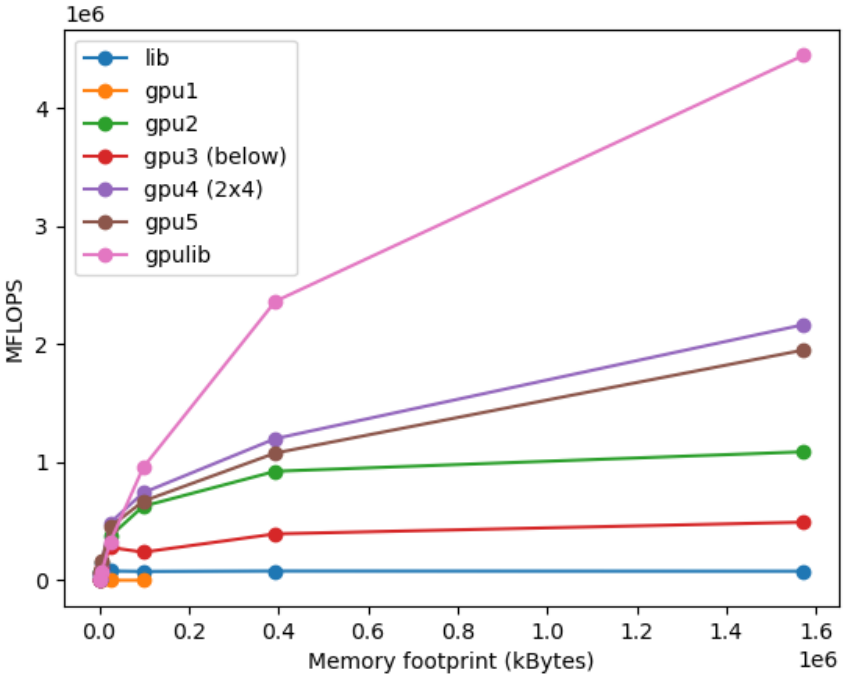

Figure 4: MFLOPS vs Memory footprint.

3 GPU Poisson Problem

3.1 Sequential Kernel Jacobi

As in the previous assignment, we want to solve the same Poisson problem using the Jacobi algorithm. However, instead of optimize a version which runs in CPU, in this project we work with GPU and implementing the code with CUDA.

Sequential version in one thread

The first GPU version of Jacobi only use 1 thread per block. The Kernel has three nested loops which moves the grid through a flattened 3d Matrix, So that, performing a sequential computation. The implemented kernel is show above.

```
__global__ void jacobi_v1(double *d_u, double *d_uOld, double *d_f, int N, int N2, int
    iter_max, double frac, double delta2){

    int i, j, k;

    for (i = 1; i < N-1; i++) {
      for (j = 1; j < N-1; j++) {
              for (k = 1; k < N-1; k++) {

                  /* Compute update of uNew */
                  d_u[i*N*N+j*N+k] = frac*(d_uOld[(i-1)*N*N+j*N+k] + d_uOld[(i+1)*N*N+j*N+k] +
                  d_uOld[i*N*N+(j-1)*N+k] + d_uOld[i*N*N+(j+1)*N+k] + d_uOld[i*N*N+j*N+(k-1)]
                      + d_uOld[i*N*N+j*N+(j+1)] + delta2*d_f[i*N*N+j*N+k]);
              //d += abs(uNew[i*N*N+j*N+k] - uOld[i*N*N+j*N+k]);
              }
        }
    }
}
```

And we defined the grid size and block size as follows.

```
jacobi_v1<<<grid, 1>>>(d_u, d_uOld, d_f, N, N2, iter_max, frac, delta2);
```

Below is presented the profile of this version which will be discussed after the table.

Section: Launch Statistics

Block Size		1
Grid Size		1
Registers Per Thread	register/thread	48
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	Kbyte/block	1.02
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	1
Waves Per SM		0.00

```
---------------------------------------- --------------- ----------------

    Section: GPU Speed Of Light
---------------------------------------- --------------- ----------------
DRAM Frequency                                cycle/nsecond             1.21
SM Frequency                                  cycle/usecond           764.98
Elapsed Cycles                                        cycle        507426112
Memory [%]                                            %               0.03
SOL DRAM                                              %               0.00
Duration                                         msecond             663.32
SOL L1/TEX Cache                                      %               3.15
SOL L2 Cache                                          %               0.02
SM Active Cycles                                    cycle        4698355.23
SM [%]                                                %               0.03
---------------------------------------- --------------- ----------------

    Section: Occupancy
---------------------------------------- --------------- ----------------
Block Limit SM                                      block                32
Block Limit Registers                               block                40
Block Limit Shared Mem                              block               164
Block Limit Warps                                   block                64
Theoretical Active Warps per SM                      warp                32
Theoretical Occupancy                                 %                  50
Achieved Occupancy                                    %                1.56
Achieved Active Warps Per SM                         warp              1.00
---------------------------------------- --------------- ----------------
```

As we can see from the different sections the memory usage is almost 0% , that is due to the fact that we are processing one element at a time. The achieved Occupancy, warps running at the same time divided by total number of warp that could be running, gives a glimpse of how optimal is the method. It is not surprising that our occupancy is 1.56%, since we are using a warp execution of 1 thread. The SM percentage is also low because we only run 1 thread (0.03%),to use the full possible power of a GPU you need much more threads per SM than number of CUDA cores in the SM.

3.2 Naive Kernel Jacobi

As we can see from the previous section, sequential operations over GPUs output worse performance than the obtained in the CPU case. In order to increase performance, it is necessary to distribute the operations over multiple threads of the GPU.

```
1
2  // KERNEL
3  __global__ void jacobi_v1(double *d_u, double *d_uOld, double *d_f, int N, int N2, int
       iter_max, double frac, double delta2){
4
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      int j = blockIdx.y * blockDim.y + threadIdx.y;
7      int k = blockIdx.z * blockDim.z + threadIdx.z;
8
9      if (i>0 && i<N-1 && j>0 && j<N-1 && k>0 && k<N-1){
10         d_u[i*N2+j*N+k] = frac*(d_uOld[(i-1)*N2+j*N+k] + d_uOld[(i+1)*N2+j*N+k] +d_uOld[i*N2
               +(j-1)*N+k] + d_uOld[i*N2+(j+1)*N+k]+d_uOld[i*N2+j*N+k-1] +d_uOld[i*N2+j*N+k+1]+
               delta2*d_f[i*N2+j*N+k]);
11     }
12 }
13
14 // MAIN LOOP
15 .
16 .
17 .
18     while(it < iter_max){
19         swap(&d_uOld, &d_u);
20         jacobi_v1<<<gridsize,blocksize>>>(d_u, d_uOld, d_f, N, N2, iter_max, frac, delta2);
21         cudaDeviceSynchronize();
22         it++;
23     }
24 .
25 .
26 .
```

For the calculations the blocksize has been set to 512 threads per block (multiple of 32) and $N^3/512$ blocks per grid as can be checked in the launch statistics since threads are executed in groups of 32 warps. Otherwise, some threads in a warp might be masked off. The following run statistics were performed for $N = 128$.

```
Section: Launch Statistics
-------------------------------------- -------------- ---------------
Block Size                                                        512
Grid Size                                                        4096
Registers Per Thread                  register/thread             22
Shared Memory Configuration Size                Kbyte          16.38
Driver Shared Memory Per Block             Kbyte/block           1.02
Dynamic Shared Memory Per Block             byte/block              0
Static Shared Memory Per Block              byte/block              0
Threads                                         thread        2097152
Waves Per SM                                                     9.48
-------------------------------------- -------------- ---------------
```

Further statistics are analysed attending to the "GPU Speed of Light" and Occupancy sections down below.

```
Section: GPU Speed Of Light
-------------------------------------- -------------- ---------------
DRAM Frequency                           cycle/nsecond            1.21
SM Frequency                             cycle/usecond          759.76
Elapsed Cycles                                   cycle          169655
Memory [%]                                           %           89.79
SOL DRAM                                             %           12.08
Duration                                        usecond          223.30
SOL L1/TEX Cache                                     %           93.04
SOL L2 Cache                                         %           46.87
SM Active Cycles                                 cycle       163732.38
SM [%]                                               %            5.35
-------------------------------------- -------------- ---------------

Section: Occupancy
-------------------------------------- -------------- ---------------
Block Limit SM                                   block              32
Block Limit Registers                            block               5
Block Limit Shared Mem                           block             164
Block Limit Warps                                block               4
Theoretical Active Warps per SM                   warp              64
Theoretical Occupancy                                %             100
Achieved Occupancy                                   %           72.35
Achieved Active Warps Per SM                      warp           46.30
-------------------------------------- -------------- ---------------
```

A very low Streaming Multiprocessors (SM) percentage (5.35%) is obtained in contrast with a high Memory percentage (89.79%) which indicates the process is highly memory bounded. We also achieve a high occupancy (72.35%) up to a 100% theoretical occupancy, indicating high instruction efficiency.

3.3 Dual GPU Jacobi

The next version makes use of two GPUs of the same model stated previously. The grid was split into two sections of dimension $N^3/2$. Each half was allocated in two separate GPUs sharing memory- enabling peer access, to compute the gradient. The events in the devices are synchronized in each iteration to block further execution of the host over the operations of each device.

```
1  // KERNELS
2  __global__ void jacobi_v3_dv0(double *d_u, double *d_uOld, double *d1_uOld, double *d_f, \
3      double frac, double delta2, int N, int N2) {
4
5    int i = blockIdx.x * blockDim.x + threadIdx.x;
6    int j = blockIdx.y * blockDim.y + threadIdx.y;
7    int k = blockIdx.z * blockDim.z + threadIdx.z;
8
9    if (0<k && k<(N-1) && 0<j && j<(N-1) && 0<i) {
10     if (i == (N/2-1)) {
11       d_u[i*N2 + j*N + k] = frac* ( \
12         d_uOld[(i-1)*N2 + j*N + k] + \
13         d1_uOld[j*N + k] + \
14         d_uOld[i*N2 + (j-1)*N + k] + \
15         d_uOld[i*N2 + (j+1)*N + k] + \
16         d_uOld[i*N2 + j*N + k-1] + \
17         d_uOld[i*N2 + j*N + k+1] + \
18         delta2 * d_f[i*N2 + j*N + k]);
19     } else if (i < (N/2-1)) {
20       d_u[i*N2 + j*N + k] =frac* ( \
21         d_uOld[(i-1)*N2+ j*N + k] + \
22         d_uOld[(i+1)*N2 + j*N + k] + \
23         d_uOld[i*N2 + (j-1)*N + k] + \
24         d_uOld[i*N2 + (j+1)*N + k] + \
25         d_uOld[i*N2 + j*N + k-1] + \
26         d_uOld[i*N2 + j*N + k+1] + \
27         delta2 * d_f[i*N2 + j*N + k]);
28     }
29   }
30 }
31
32 __global__ void jacobi_v3_dv1(double *d1_u, double *d1_uOld, double *d_uOld, double *d1_f, \
33     double frac, double delta2, int N, int N2) {
34
35   int i = blockIdx.x * blockDim.x + threadIdx.x;
36   int j = blockIdx.y * blockDim.y + threadIdx.y;
37   int k = blockIdx.z * blockDim.z + threadIdx.z;
38
39   if (0<k && k<(N-1) && 0<j && j<(N-1) && i<((N/2)-1)) {
40     if (i == 0) {
41       d1_u[i*N2 + j*N + k] = frac* ( \
42         d_uOld[(N/2-1)*N2 +j*N + k] + \
43         d1_uOld[(i+1)*N2 +j*N + k] + \
44         d1_uOld[i*N2 + (j-1)*N + k] + \
45         d1_uOld[i*N2 + (j+1)*N + k] + \
46         d1_uOld[i*N2 + j*N + k-1] + \
47         d1_uOld[i*N2 + j*N + k+1] + \
48         delta2 * d1_f[i*N2 + j*N + k]);
49     }
50         else if (i > 0) {
51       d1_u[i*N2 + j*N + k] = frac* ( \
52         d1_uOld[(i-1)*N2 + j*N + k] + \
53         d1_uOld[(i+1)*N2 + j*N + k] + \
54         d1_uOld[i*N2 + (j-1)*N + k] + \
```

```
55          d1_uOld[i*N2 + (j+1)*N + k] + \
56          d1_uOld[i*N2 + j*N + k-1] + \
57          d1_uOld[i*N2 + j*N + k+1] + \
58          delta2 * d1_f[i*N2 + j*N + k]);
59      }
60    }
61  }
62
63  // MAIN LOOP
64      while(it < iter_max){
65          swap(&d_uOld, &d_u);
66          swap(&d1_uOld, &d1_u);
67
68          cudaSetDevice(0);
69          jacobi_v3_dv0<<<gridsize,blocksize>>>(d_u, d_uOld, d1_uOld, d_f, frac, delta2, N, N2
                );
70
71          cudaSetDevice(1);
72          jacobi_v3_dv1<<<gridsize,blocksize>>>(d1_u, d1_uOld, d_uOld, d1_f, frac, delta2, N,
                N2);
73
74          cudaDeviceSynchronize();
75          cudaSetDevice(0);
76          cudaDeviceSynchronize();
77          it++;
78      }
```

The grid size and block size was set with the same values stated in the previous section. The following run statistics were performed for N=128.

```
    Section: GPU Speed Of Light
---------------------------------- --------------- ------- GPU 1 --- GPU 2
DRAM Frequency                          cycle/nsecond          1.19      1.21
SM Frequency                            cycle/usecond        750.86    759.87
Elapsed Cycles                                  cycle         93569     93963
Memory [%]                                          %         83.09     82.77
SOL DRAM                                            %          8.91      8.90
Duration                                      usecond        124.61    123.65
SOL L1/TEX Cache                                    %         93.39     93.53
SOL L2 Cache                                        %         43.22     43.85
SM Active Cycles                                cycle      83249.55  83152.31
SM [%]                                              %          7.85      7.34
---------------------------------- --------------- ----------------------
```

```
 Section: Occupancy
---------------------------------- --------------- ------- GPU 1 --- GPU 2
Block Limit SM                                 block              32        32
Block Limit Registers                          block               5         5
Block Limit Shared Mem                         block             164       164
Block Limit Warps                              block               4         4
Theoretical Active Warps per SM                 warp              64        64
Theoretical Occupancy                              %             100       100
Achieved Occupancy                                 %           76.03     76.68
Achieved Active Warps Per SM                    warp           48.66     49.08
---------------------------------- --------------- ----------------------
```

Both GPU statistics show similar results of memory bound and occupancy, but with the tasks splitted, which will translate into higher performance. A comparison of the different models discussed until now are presented in the following figures. All the runs were performed for a total of 100 iterations for different N sizes (64, 128, 254, 512).

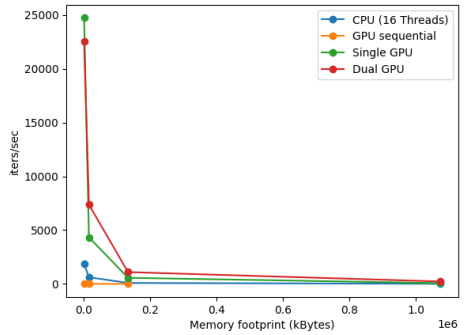

Figure 5: Iters/sec vs Memory footprint comparison.

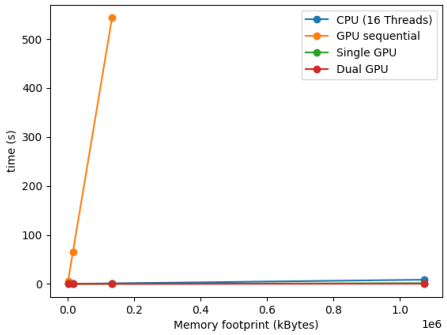

Figure 6: Runtime vs Memory footprint comparison.

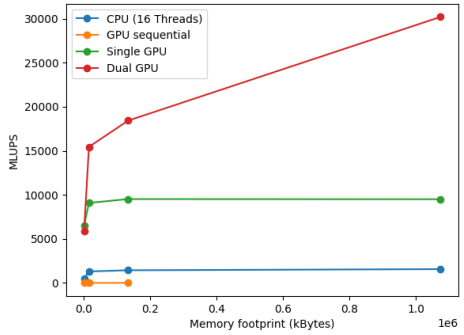

Figure 7: MLUPS vs Memory footprint comparison.

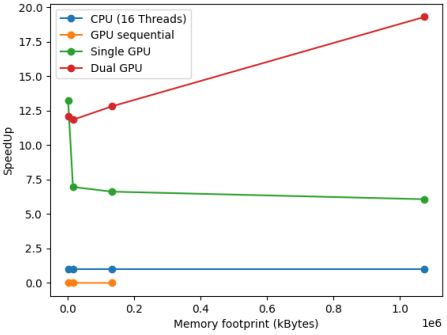

Figure 8: Speedup vs memory footprint comparison

It is clear from the above figures that sequential implementation in GPU translates into very poor performance, considerably lower than the CPU version. We observe a speedup of almost 7.5 times the fastest 16-thread CPU version for the GPU implementation presented in Section 3.2, but the most considerable speedup is presented by the dual-GPU implementation. The dual-GPU implementation also scales better with higher memory footprints, reaching a speedup of almost 20.0 times the 16-threads CPU version.

3.4 Norm GPU Jacobi

For this part of the Poisson problem we will follow the model presented in 3.2. but with the addition of a threshold conditional to check the convergence of the algorithm. Initially, a first draft was made rewriting the computed norm back to the host and forward to the device in each iteration, so both host and device would access to the updated value of the norm, following the while loop clause.

```
1  // KERNEL
2  __global__ void jacobi_v1(double *d_u, double *d_uOld, double *d_f, int N, int N2, int
       iter_max, double frac, double delta2, double *d_norm){
3
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      int k = blockIdx.z * blockDim.z + threadIdx.z;
7      double norm_value;
8      if (i>0 && i<N-1 && j>0 && j<N-1 && k>0 && k<N-1){
9          d_u[i*N2+j*N+k] = frac*(d_uOld[(i-1)*N2+j*N+k] + d_uOld[(i+1)*N2+j*N+k]+d_uOld[i*N2
               +(j-1)*N+k]+ d_uOld[i*N2+(j+1)*N+k]+d_uOld[i*N2+j*N+k-1] + d_uOld[i*N2+j*N+k+1]+
               delta2*d_f[i*N2+j*N+k]);
10
11         norm_value = ((d_u[i*N2+j*N+k]-d_uOld[i*N2+j*N+k])*(d_u[i*N2+j*N+k]-d_uOld[i*N2+j*N+
               k]));
12         atomicAdd(d_norm, norm_value);
13     }
14 }
15
16 // MAIN LOOP
17     while(it < iter_max && *h_norm>tolerance){
18         *h_norm = 0;
19
20         cudaMemcpy(d_norm, h_norm ,norm_size, cudaMemcpyHostToDevice);
21         swap(&d_uOld, &d_u);
22         jacobi_v1<<<gridsize,blocksize>>>(d_u, d_uOld, d_f, N, N2, iter_max, frac, delta2,
               d_norm);
23         cudaMemcpy(h_norm, d_norm ,norm_size, cudaMemcpyDeviceToHost);
24         cudaDeviceSynchronize();
25         it++;
26
27     }
```

This implementation performed the required tasks but presented huge huge runtimes. The line labeled CUDA API in Figure (9) contains the CUDA calls on the CPU relevant to a kernel execution. Events in red are memory transfer calls from the CPU. As can be seen in the CPU overhead, the problem in performance was caused by the iterative copy actions between host and device as shown in Figure (9).

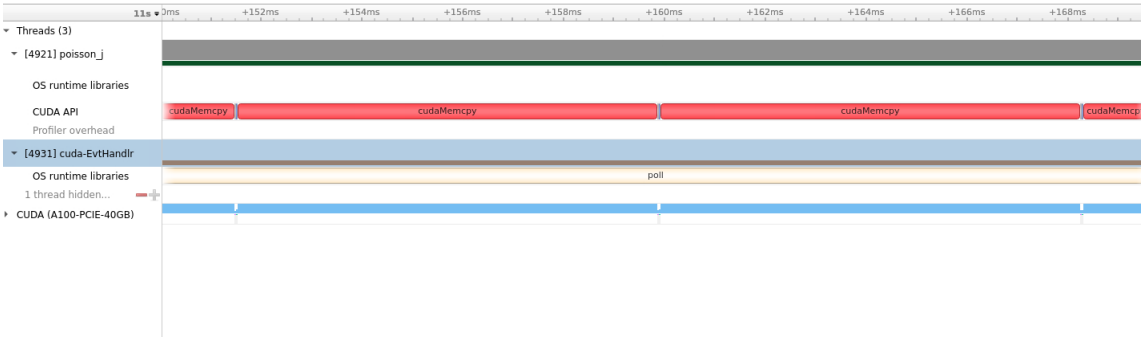

Figure 9: Profiler snapshot for N=128.

This problem in communication between host and device was solved by performing a block sum reduction statically allocating shared memory to perform the norm calculation.

```
1  // KERNEL
2  __inline__ __device__ double warpReduceSum(double value);
3  __inline__ __device__ double blockReduceSum(double value) ;
4
5  __global__ void jacobi_v1(double *d_u, double *d_uOld, double *d_f, int N, int N2, int
       iter_max, double frac, double delta2, double *d_norm){
6
7    int i = blockIdx.x * blockDim.x + threadIdx.x;
8    int j = blockIdx.y * blockDim.y + threadIdx.y;
9    int k = blockIdx.z * blockDim.z + threadIdx.z;
10   double value = 0;
11   if (i>0 && i < N-1 && j>0 && j < N-1 && k>0 && k < N-1 ){
12     d_u[i*N2+j*N+k] = frac*(d_uOld[(i-1)*N2+j*N+k] + d_uOld[(i+1)*N2+j*N+k]+
13                             d_uOld[i*N2+(j-1)*N+k] + d_uOld[i*N2+(j+1)*N+k]+d_uOld[i*N2+j*N+k-1]
                                    +
14                             d_uOld[i*N2+j*N+k+1]+delta2*d_f[i*N2+j*N+k]);
15
16       value = pow((d_u[i*N*N+j*N+k]-d_uOld[i*N*N+j*N+k]),2);
17   }
18   value = blockReduceSum(value);
19   if (threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0 )
20   {atomicAdd(d_norm, value); }
21
22 }
23
24 __inline__ __device__ double
25 blockReduceSum(double value) {
26   __shared__ double smem[32];
27   int indexThread = threadIdx.x + threadIdx.y * blockDim.x+ threadIdx.z * blockDim.y*
       blockDim.x;
28
29
30   if (indexThread < warpSize) {
31     smem[indexThread]=0;
32   }
33   __syncthreads();
34
35   value =  warpReduceSum(value);
36
37   if    (indexThread % warpSize == 0)
38     {
39     smem[indexThread / warpSize]=value;
40     }
41   __syncthreads();
```

```
42    if (indexThread < warpSize) {
43      value=smem[indexThread];
44    }
45  return warpReduceSum(value);}
46
47  __inline__ __device__ double warpReduceSum(double value)
48  {
49    for (int i = 16; i > 0; i /= 2)  {
50       value += __shfl_down_sync(-1, value, i);
51        }
52     return value;
53    }
54
55    // HOST LOOP
56      while(it < iter_max && *h_norm>tolerance){
57           *h_norm = 0.0;
58
59           cudaMemcpy(d_norm, h_norm ,norm_size, cudaMemcpyHostToDevice);
60           swap(&d_uOld, &d_u);
61           jacobi_v1<<<gridsize,blocksize>>>(d_u, d_uOld, d_f, N, N2, iter_max, frac, delta2,
                 d_norm);
62           cudaMemcpy(h_norm, d_norm ,norm_size, cudaMemcpyDeviceToHost);
63           cudaDeviceSynchronize();
64           it++;
65
66      }
```

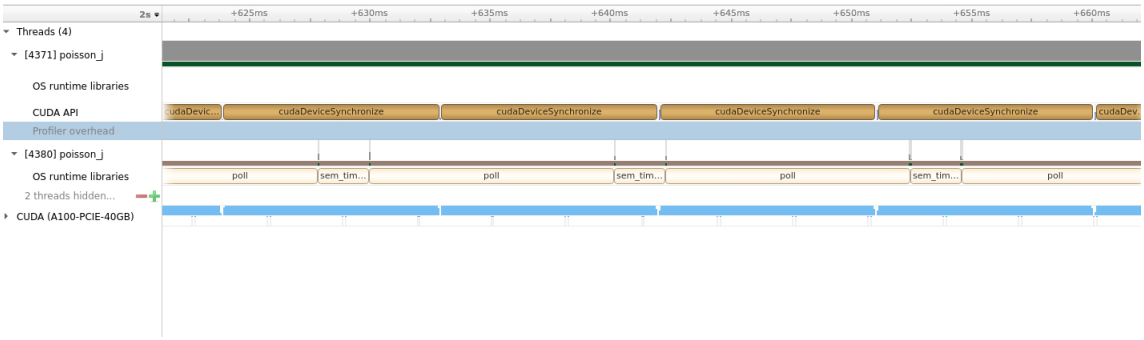

Figure 10: Profiler snapshot for N=128.

As can be seen from the profiler snapshot (Fig. 10) there is no time wasted for allocation in between iterations, highly boosting performance for this final version of the norm implementation and obtaining a performance comparable to the results observed for the naive implementation.

Size(N)	Iter.	Time(s)	Iter./s
64	973	0.067010	14520.298231
128	2925	0.981119	2981.290488
256	8361	23.701864	352.757072
512	23248	537.748876	43.232076

Table 1: Results obtained for different N sizes in the final norm implementation. The tolerance threshold was set to 1.

4 Conclusion

From the first part of the assignment we could check that CPU performs better on sequential sentences. As soon as we started to exploit the number of threads in the GPU the performance of the GPU increases considerably, giving better results for this application than the CPU due to the high level of parallelization. In contrast to what we saw computing in C/C++, with the GPU gives better results unrolling the loops into columns. Finally, by exploting the shared memory the runtime is greatly reduced, since access the global memory is much slower than accessing the shared memory.

For the second part of the assignment we compared the performance of GPU accelerated Poisson problem against the fastest CPU version treated last week. It is shown that the arquitecture of GPUs is far less efficient for sequential computation than CPUs, which translates in big performance issues for the first implementation. Consequently we implemented a parallelized version with no share memory which greatly improved the performance over the 16 multi-thread CPU implementation. The performance could be greatly increased splitting the task between two GPUs of the same model. Finally, we exposed how the use of shared memory greatly affected the performance in tasks that requires iterative communication between host and device memory.

List of Figures

1 Mflops/s vs Memory footprint comparison. 4
2 MFLOPS vs Memory footprint. 5
3 MFLOPS vs Memory footprint. 7
4 MFLOPS vs Memory footprint. 9
5 Iters/sec vs Memory footprint comparison. 16
6 Runtime vs Memory footprint comparison. 16
7 MLUPS vs Memory footprint comparison. 16
8 Speedup vs memory footprint comparison 16
9 Profiler snapshot for N=128. 18
10 Profiler snapshot for N=128. 19

List of Tables

1 Results obtained for different N sizes in the final norm implementation. The tolerance threshold was set to 1. 19