AGH University of Science and Technology

Laboratory report 6

Authors: Pedro Alexandre Simões dos Reis,
Iñaki Urrutia Sánchez
Subject: Introduction to CUDA and OpenCL
Year: 2019/2020

Contents

1 Matrix multiplication algorithm

The goal of this report is to compare different solutions (base-line version, making use of the shared memory and a version from the standard CUDA library for operating with matrices) to solve the same problem.

Since we are dealing with matrices that could not fit completely in our blocks, we will need to divide them in sub-matrices which are multiplied. Then we join the solutions of all the subproblems to get the final result.

```
j = tileNUM * BLOCK_SIZE + threadIdx.x;
i = tileNUM * BLOCK_SIZE + threadIdx.y;

// Load left[i][j] to shared mem
idx = row * dim  + tileNUM * BLOCK_SIZE + threadIdx.x;

if (idx >= dim * dim) {
    // Coalesced access
    Left_shared_t[threadIdx.y][threadIdx.x] = 0;
} else {
    // Coalesced access
    Left_shared_t[threadIdx.y][threadIdx.x] = left[row * dim + j];
}

// Load right[i][j] to shared mem
idx = (tileNUM * BLOCK_SIZE + threadIdx.y) * dim + col;

if (idx >= dim * dim) {
    Right_shared_t[threadIdx.y][threadIdx.x] = 0;
} else {
    // Coalesced access
    Right_shared_t[threadIdx.y][threadIdx.x] = right[i * dim + col];
}
```

In the code above, it is possible to observe the process of divide the matrices in smaller matrices. Since matrix size does not need necessarily a multiple of a power of two, it can happen that sometimes we ended up with more threads than we actually need. To overcome this problem it is checked before transferring the data to the sub-matrices if the *threadId* calculated in the *idx* variable is bigger than the total dimension of the the matrices. If so, then a 0 (which is neutral) is stored for calculation, else the value that it is in matrix is transferred to the respective sub-matrix.

```
for (int k = 0; k < BLOCK_SIZE; k++) {
    //no shared memory bank conflict
    temp += (Left_shared_t[threadIdx.y][k] *
       Right_shared_t[k][threadIdx.x]);
}
```

The code above is used to multiply the submatrices together and accumulate the result in temp variable. At the end of iteration the value obtained for each position in the table is added in the respective position in the result matrix. As it is possible to observe in the code bellow:

```
if ((row < dim) && (col < dim)) {
    // Store accumulated value to res
    res[row * dim + col] = temp;
}
```

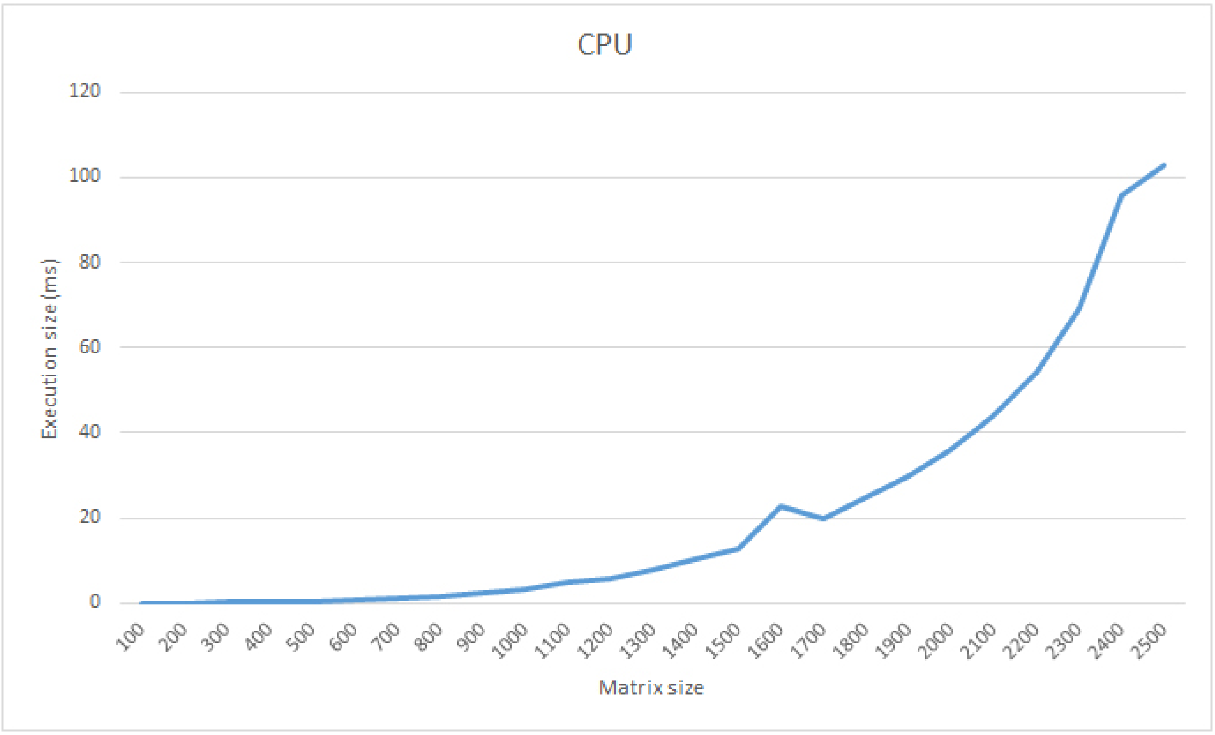
To avoid add the calculation of the "unused" threads to the final matrix, it is checked if the row and the column are in the acceptable range (i.e. they should be smaller than the dimensions of the matrix).

2 Single threaded version

This is the implementation we used to test the performance just using the CPU (with only one thread):

```
void matrixMulCPU(float* A, float* B, float* C_cpu, int dim) {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      float tmp = 0.0f;
      for (int k = 0; k < dim; k++) {
         tmp += A[i * dim + k] * B[k * dim + j];
      }
      C_cpu[i * dim + j] = tmp;
    }
  }
}
```

And we obtained the following results:



3 Base-line version

To implement the Base-line (naive) version of this algorithm, we used the GPU, allocating the data on the Unified Memory. The results are the following:

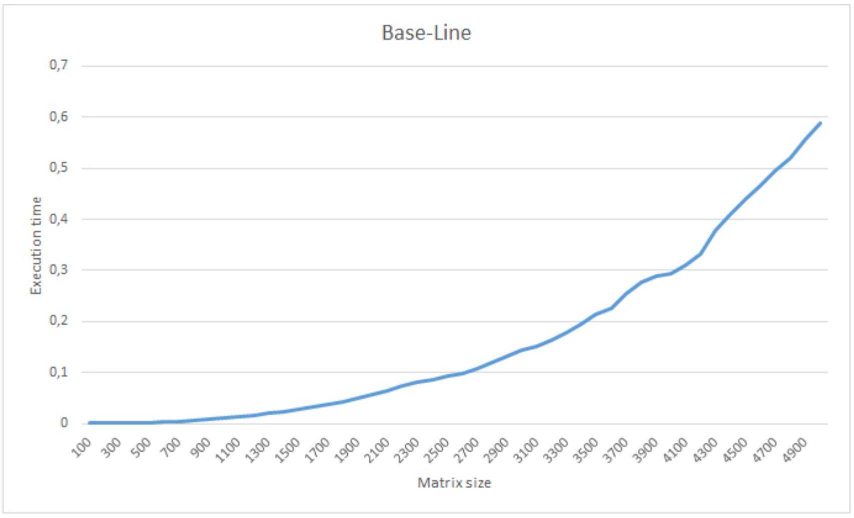


Figure 1: Base-line performance

As we could expect, the increase on performance respecting on the single-threaded CPU version is huge. The following graph shows how many times is the Base-line version faster for each matrix size.

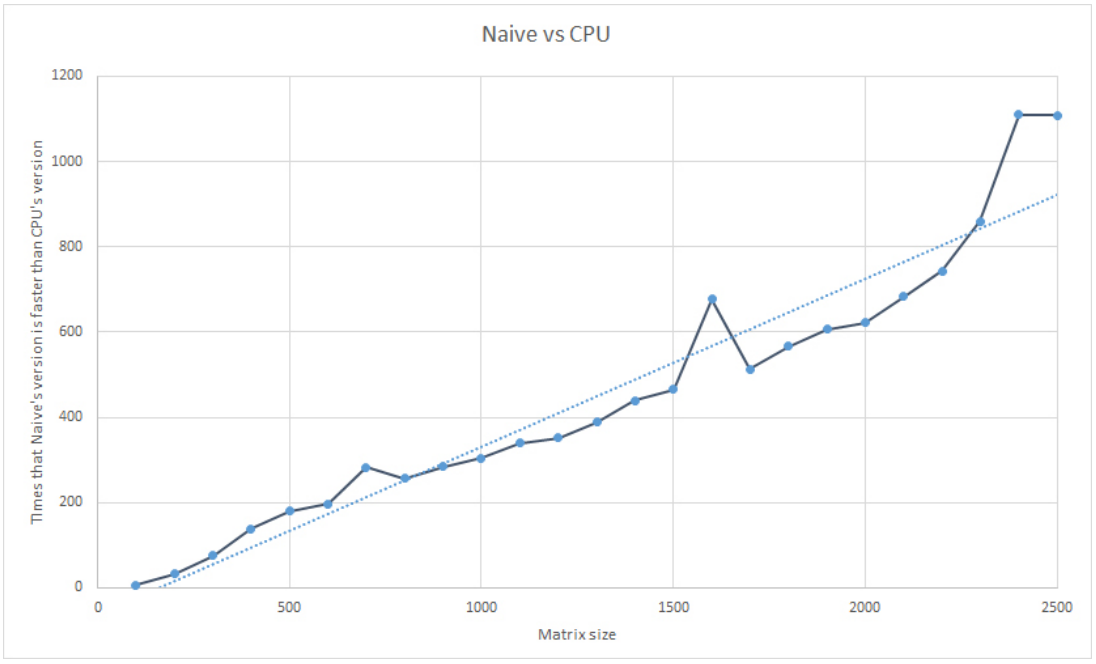


Figure 2: Naive vs CPU

4 Shared memory version

In the Shared memory version we used the GPU but taking advantage of the shared memory capabilities. The results are the following:

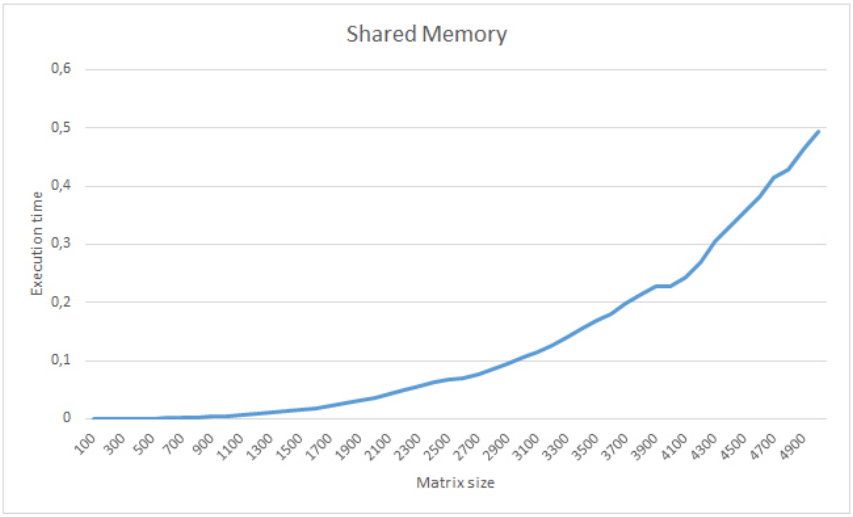


Figure 3: Shared memory version performance

Base-line and shared memory versions have a similar performance. However, if we compare both versions, we can notice that as the matrix size grows, shared memory's version gets better compared to base-line's implementation. We can see it easily in the following graph:

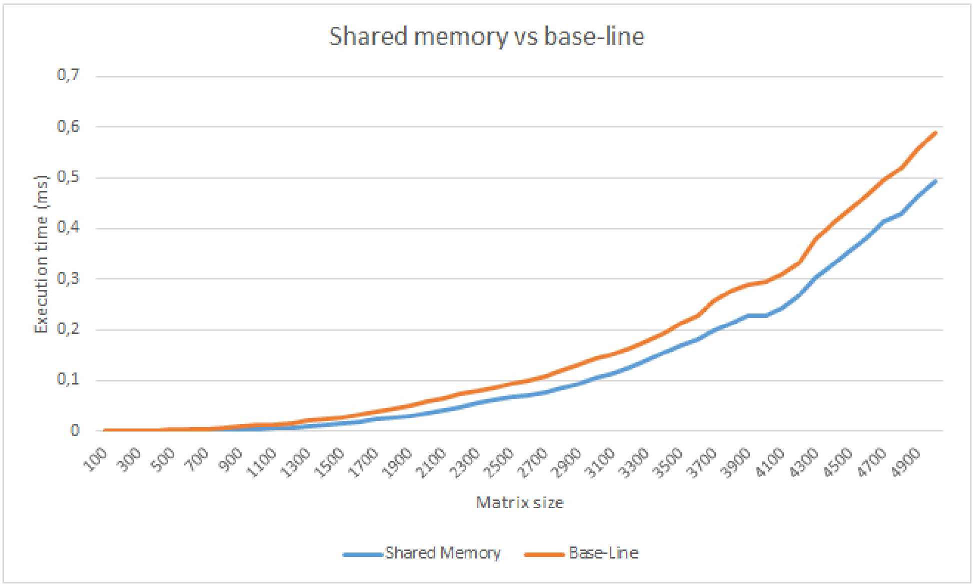


Figure 4: Shared memory version performance

5 Cublas version

The remaining implementation is the Cublas version. For this one we used the Cublas library, which is a standard CUDA library for matrix operations. This version proved to be much faster than any of our implementation, as we can see in the following graph:

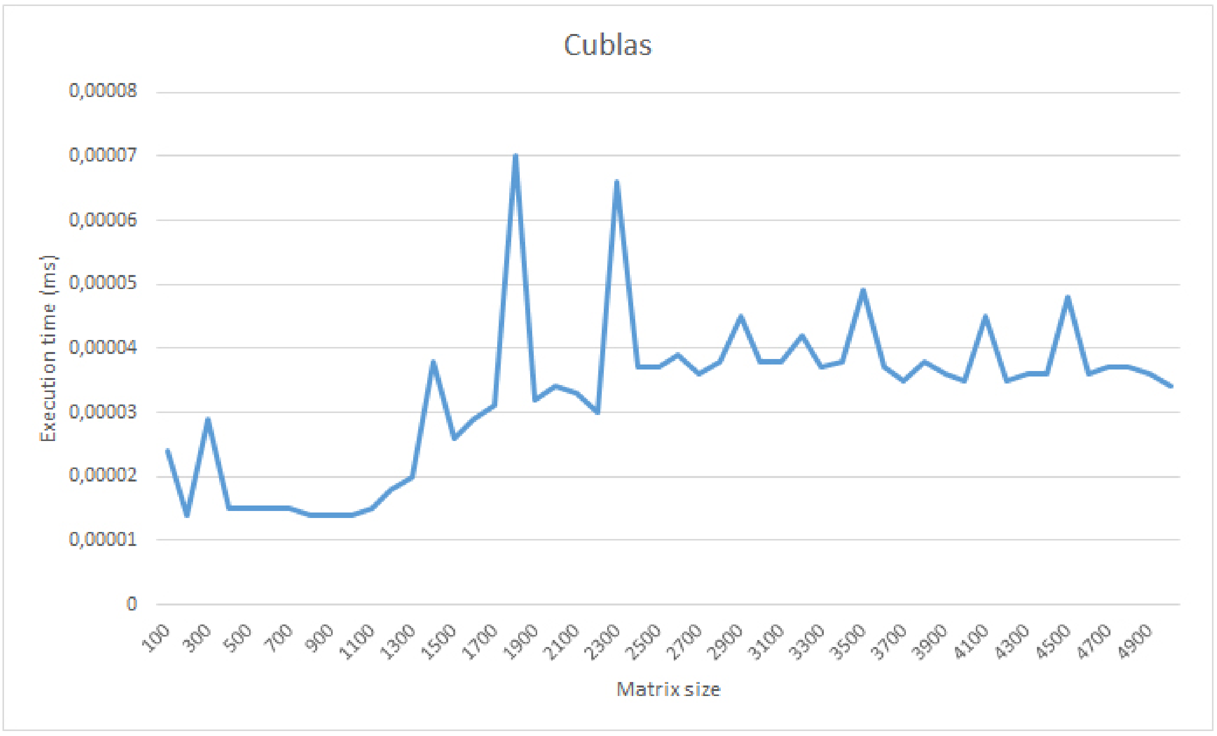


Figure 5: Cublas version performance

The chart shows that for the range of matrix sizes we worked (100-5000), the execution times are tiny and almost constant.