Report

Various matrix multiply methods.
Comparing efficiency of methods.

Paweł Lipiór, Michał Żoczek

November 30, 2019

1 Introduction

As we know very well matrix multiplication is one of the most important task in Computer Science. Unfortunately this is also very complicated and requiring a lot of computing power task. In that case for efficiency computing this task must be well optimized. CUDA offers us necessary tools to multiply matrix very quickly and efficiently. During this lab we tried to write code for multiplication non-square matrices. We decided to use two methods. First of them, what we called `Naive` is simply code using CUDA environment. Second is more difficult and more complicated. We called that `SharedMemory`, because we used `shared memory` solutions for these method. Finally we decided to compare our results to ready exists solution. We decided to use `cuBLAS` solution.

2 Preparation

Before we started to create c++ style class named `Matrix` as a way to better storage data and avoid mess in code. Every `Matrix` object contains information about number of Rows and Columns of matrix and also contains pointer to simple array of elements.

```
1  class Matrix
2  {
3  public:
4     int _width;
5     int _height;
6     float* _elements;
7
8     Matrix(int width, int height);
9     Matrix(int width, int height, MemoryType type);
10    ~Matrix();
11 private:
12    MemoryType _type;
13 };
```

Moreover we decided to code some useful functions to initialize matrices, print matrix, compare two matrices and also simple one threaded code for CPU

to be sure that our calculations are correct. We also split the code into separate files for order in whole project.

```
void MatrixPrint(Matrix &A);
void MatrixMul(Matrix &prodA, Matrix &prodB, Matrix &resC);
void MatrixCompare(Matrix &A, Matrix &B);
void InitializeMatrix(Matrix &A, int val);
void InitializeMatrix(Matrix &A);
```

```
void MatrixMul(Matrix &A, Matrix &B, Matrix &C)
{
   for(int row = 0; row < C._height; row ++)
   {
      for(int col = 0; col < C._width; col++)
      {
         for(int k = 0; k < A._width; k++)
            C._elements[(row*C._width) + col] +=
                 A._elements[(row*A._width) + k] * B._elements[col +
                 (k*B._width)];
      }
   }
}
```

In the `main` file we implemented simply functionality of setting the size of multiplied matrices. We also allocated memory for ours matrices using `cudaMalloc` function to use united memory space for host and device. At the end we defined dimensions of Block and dimensions of Grid for CUDA. We used `#define` to set `BLOCK_SIZE` as 16.

```
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid((resWidth + dimBlock.x - 1)/dimBlock.x,(resHeight +
       dimBlock.x - 1)/dimBlock.y);
```

3 'Naive' matrix multiplication solution

These solution uses quite simply kernel:

```
    __global__ void MatrixMulNaive(Matrix A, Matrix B, Matrix C)
{
   int Row = blockIdx.y * blockDim.y + threadIdx.y;

   int Col = blockIdx.x * blockDim.x + threadIdx.x;

   if((Row < C._height) && (Col < C._width))
   {
      float Pvalue = 0;
```

```
10        for(int k = 0; k < A._width; ++k)
11        {
12           Pvalue +=
                 A._elements[Row*A._width+k]*B._elements[k*B._width+Col];
13        }
14        C._elements[Row*C._width+Col] = Pvalue;
15     }
16  }
```

4 'SharedMemory' matrix multiplication solution

These solution used a little bit more complicated kernel:

```
1     __global__ void MatrixMulShared(Matrix A, Matrix B, Matrix C)
2  {
3
4
5     int Row = blockIdx.y * blockDim.y + threadIdx.y;
6     int Col = blockIdx.x * blockDim.x + threadIdx.x;
7
8      __shared__ float As[TILE_SIZE][TILE_SIZE];
9      __shared__ float Bs[TILE_SIZE][TILE_SIZE];
10
11    float Cvalue = 0;
12
13    int row = threadIdx.y;
14    int col = threadIdx.x;
15
16    for (int i = 0; i < ((A._width+TILE_SIZE-1)/ TILE_SIZE); ++i) {
17
18
19      if((Row < A._height) && ((col + i*TILE_SIZE) < A._width))
20         As[row][col] = A._elements[Row*A._width + col + i*TILE_SIZE];
21      else
22         As[row][col] = 0;
23      if((Col < B._width) && ((row + i*TILE_SIZE) < B._height))
24         Bs[row][col] = B._elements[(row + i*TILE_SIZE)*B._width + Col];
25      else
26         Bs[row][col] = 0;
27        //Synchronize threads
28        __syncthreads();
29
30        for (int j = 0; j < TILE_SIZE; ++j)
31        {
32            Cvalue += As[row][j] * Bs[j][col];
33        }
34
35        __syncthreads();
36    }
```

```
37
38      if (Row < C._height && Col < C._width)//Saving Final result into
            Matrix C
39      {
40          C._elements[Row*C._width + Col] = Cvalue;
41      }
42  }
```

In a nutshell this method slices matrix into submatrices matching the size of one block. Smaller data structure can be stored in share memory and stored data can be shared amount threads. Theoretically it should speed up calculations because shared memory is much faster but these mechanism is also complicated to menage because there is no enough space in share memory to storage all data. Last if statement in code allow us to set the right dimension of calculated matrix.

5 'cuBLAS' matrix multiplication solution

Basic Linear Algebra Subprograms (`BLAS`) is a set of low-level routines for performing linear algebra operations. `cuBLAS` is the library of CUDA implementation of BLAS. It allows users to use CUDA environment for BLAS routines. In our small project we used `cublasSgemm` function form `cublas_v2.h`. This function is quite difficult and gets many parameters like dimensions of multiplied matrices and of course pointers to elements of matrices A, B and C. For ease we decided to write another function and do things more clear.

```
1       void CublasMultiply(Matrix &matA, Matrix &matB, Matrix &matC)
2   {
3      int m = matB._width;
4      int n = matA._height;
5      int k = matA._width;
6
7      int lda = matB._width;
8      int ldb = matA._width;
9      int ldc = matB._width;
10
11     float *A = matA._elements;
12     float *B = matB._elements;
13     float *C = matC._elements;
14
15     const float a = 1;
16     const float b = 0;
17     const float *alpha = &a;
18     const float *beta = &b;
19
20     cublasHandle_t handle;
21     cublasCreate(&handle);
22
23     cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, alpha, B, lda,
           A, ldb, beta, C, ldc);
24
```

```
25
26     cublasDestroy(handle);
27 }
```

We encountered a small problem because in Fortran, in which is coded BLAS, uses column-first indexing which is not compatybile with C style array indexing. However after setting accordingly `lda, ldb, ldc` all worked correctly.

6 Performance testing

After code all necessary functionalities we were ready to test performance of these three implemented solutions. Using `nvprof` tool we were able to get time of execution for all kernels. We decided to measure time of execution for grid of points: Number of Rows = [1, 4, 16, 32, 64, 128, 256] and Number of Columns [1, 4, 16, 32, 128, 256] for matching A and B matrices. All collected date are shown in the pictures: 1 2 3

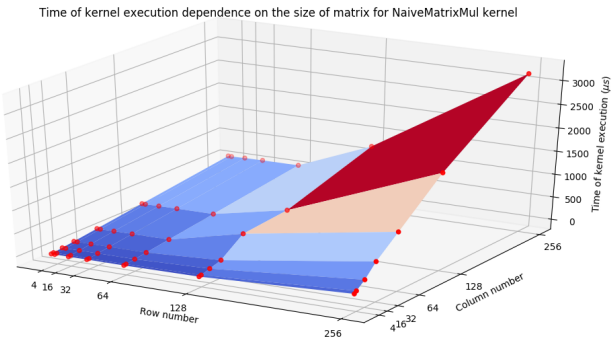

Figure 1: 'Naive' kernel matrix multiplication.

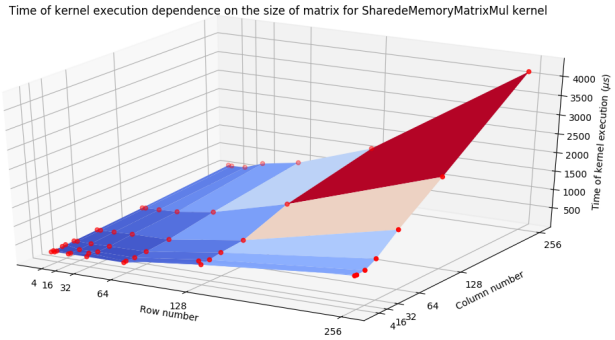

Figure 2: 'SharedMemory' kernel matrix multiplication.

Figure 3: 'cuBLAS' matrix multiplication.

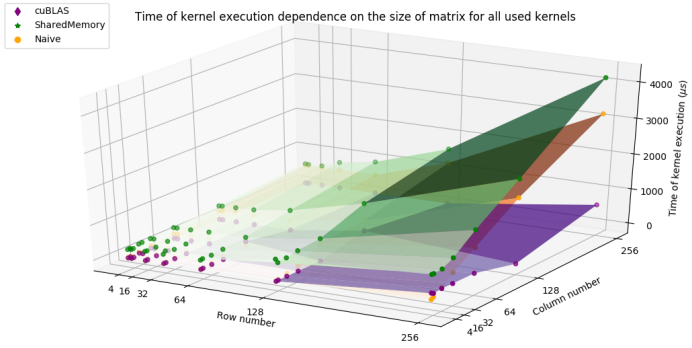

Figure 4: Comparing of all used method.

In the picture 1 and picture 2 we can see that for 'Naive' and 'SharedMemory' time of execution increases just like the size of the matrix. This is naturally the expected result. Picture 3 shows us performance results for 'cuBLAS' implementation and we can say that it is quite irregular. Picture 4 shows us comparison between those three methods. As we can see 'cuBLAS' achieved the best results, however we expected that. Alarming is that performance for 'SharedMemory' is worse than for the 'Naive' implementation. This is mean that our resource managing is not that good as we could wish, and we do things worse. We consider also that nowadays CUDA is such good optimized that automatic management of shared memory is quite efficient.