Matrix multiplication

1. Base-line

As far as the base-line version of the kernel is concerned, in order to be able to multiply matrices with non-square dimensions we only had to change the corresponding dimensions of the matrices , and add additional arguments to the kernel.

```
__global__ void MatrixMulVarKernel(float* M, float* N, float* P,
int widthAHeightB, int heightA, int widthB) {
    int Row = blockIdx.y*blockDim.y+threadIdx.y;// Calculate the
row index of the P element and M
    int Col = blockIdx.x*blockDim.x+threadIdx.x;// Calculate the
column index of P and N
    if ((Row < heightA) && (Col < widthB)) {
       float Pvalue = 0;
       for (int k = 0; k < widthAHeightB; ++k) {
            Pvalue += M[Row*widthAHeightB+k]*N[k*widthB+Col];//
each thread computes one element of the block sub-matrix
       }

       P[Row*widthB+Col] = Pvalue;
    }
}
```

2.Shared memory

First we need to accept additional arguments - dimensions, in comparison to the first Shared Memory version (for square matrices).

```
__global__ void MatrixMulVarSharedMemoryKernel(float* M, float* N,
float* P, int widthAHeightB, int heightA, int widthB)
```

The next step is to set starting points, for the next tiles in the row (M - first matrix), and column (N - second matrix). The difference make again different variables.

```
int Mstart=widthAHeightB*tileSize*blockIdx.y;
    int Mend=Mstart + widthAHeightB - 1;
    int mstep=tileSize;
    int Nstart=tileSize*blockIdx.x;
    int nstep=tileSize*widthB;
```

And here we have again our temporary variable, for storing next elements of the sum.

```
float temp=0;
```

Next, we allocate memory for our tile.

```
__shared__ float Ms[tileSize][tileSize];
__shared__ float Ns[tileSize][tileSize];
```

In this step we are multiplying the section, where are tiles fit without “cutting” them, this operation save us some time, on “ifs”, in the next part. We only had to take care of this section, which is covered by the integer amount of tiles.

```
if(Mstart < (heightA/tileSize)*tileSize*widthAHeightB &&
Nstart%widthB < (widthB/tileSize)*tileSize ){
            for(int m=Mstart,n=Nstart;m<Mend;m+=mstep,n+=nstep){
                Ms[threadIdx.y][threadIdx.x]=M[m+widthAHeightB*thr
                eadIdx.y+threadIdx.x];

                Ns[threadIdx.y][threadIdx.x]=N[n+widthB*threadIdx.
                y+threadIdx.x];
                __syncthreads();


                for (int i = 0; i < tileSize; ++i) {
                temp += Ms[threadIdx.y][i] * Ns[i][threadIdx.x];
                }
                __syncthreads();

            }
```

Here we are taking care of the rest of elements in matrices.
When we are moving out of our original matrices, the values in corresponding elements in the tile are changed to 0.0 value, in order to have no impact on whole multiplication.

```
else {//the rest of the matrix
           for(int m=Mstart,n=Nstart;m<=Mend;m+=mstep,n+=nstep){

           if(m%widthAHeightB + threadIdx.x < widthAHeightB &&
           blockIdx.y*tileSize + threadIdx.y < heightA){

                Ms[threadIdx.y][threadIdx.x]=M[m+widthAHeightB*thr
                eadIdx.y+threadIdx.x];
           }
```

```
            else{
                        Ms[threadIdx.y][threadIdx.x]=0.0;
            }

            if((n/widthB) + threadIdx.y < widthAHeightB &&
            blockIdx.x*tileSize + threadIdx.x < widthB){

                  Ns[threadIdx.y][threadIdx.x]=N[n+widthB*threadIdx.
                  y+threadIdx.x];
            }
            else{
                  Ns[threadIdx.y][threadIdx.x]=0.0;
            }
            __syncthreads();


            for (int i = 0; i < tileSize; ++i) {
                  temp += Ms[threadIdx.y][i] * Ns[i][threadIdx.x];
            }
             __syncthreads();

}
```

In the last part of the kernel, we are changing values in the result matrix, using temp variable, being careful about changing right elements.

```
if(blockIdx.y*tileSize + threadIdx.y < heightA &&
blockIdx.x*tileSize + threadIdx.x < widthB){
      P[widthB * tileSize * blockIdx.y + tileSize * blockIdx.x +
      widthB * threadIdx.y + threadIdx.x] = temp;
      }
```

Below is shown an example output of our program. Which confirms that matrices calculated with GPU and CPU are the same.

```
CPU - Matrix Multiplication: Done,
Duration: 61.067712 s

Base Line: CUDA kernel launch with 7500 blocks of 1024 threads
Base Line - Matrix Multiplication: Result is Correct
Duration: 1.382588 s

Shared Memory: CUDA kernel launch with 7500 blocks of 1024 threads
Shared Memory - Matrix Multiplication: Result is Correct
```

Duration: 1.281738 s

Everything cleared.
Good night.

3.cuBLAS

Nvidia provides us with various libraries to aid with different tasks, one of which is cuBLAS (Basic Linear Algebra Subprograms). This library is included in the `cublas_v2.h header`. We will be using one of functions that can be found there called `cublasSgemm` in order to multiply non-square matrices.

In order to use functions from culas library we must first initialize the handle to the library context. With the cublasCreate() we create a pointer to the structure containing the library context which must be then passed to further library function calls. Once everything is done the handle should be destroyed at the end using cublasDestroy(). This allows us to control the library setup in case where we are working on multiple CPU threads and devices.

It is important to remember that cublas uses column-major storage unlike c and cpp with row-major. This forces us to think about changes in indexing when calling the function.

Here is presented the function itself together with it's parameters

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const float     *alpha,
                           const float     *A, int lda,
                           const float     *B, int ldb,
                           const float     *beta,
                           float           *C, int ldc)
```

It performs the following operation: C = α*op(A)*op(B) + β*C

Where: A,B,C are matrices stored in column-major factor, m,n,k are the dimensions of multiplied matrices (A mxk; B kxn; C mxn), alpha and beta are scalars used for multiplication, and lda,ldb,ldc are the leading dimensions of matrices.(?) Transa and transb means which type of operation (op(A) and op(B)) on initial matrices is chosen (non transpose, transpose and conjugate transpose).

The final code which performs multiplication in our program is shown below:

```
cublasHandle_t handle ;

// [...]

cublasCreate (& handle );       // initialize CUBLAS context
float al =1;
float bet =0;

cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,m,n,k,&al,a,m,b,k,&bet,
P_G,m);

// [...]

cublasDestroy ( handle );
```

In the beginning we create handle pointer for the library context, which is then used in the `cublasCreate function. Then we set the values to the` α `and` β `parameters. In our case the` β = 0, as we are performing simple multiplication. When calling the function we launch it with all the parameters which were discussed before. `CUBLAS_OP_N` parameter means that the factor matrices are not transposed in any way. After the multiplication is finished we destroy the handle used in the process.

Below is shown an example output of our program.

Cublas - Matrix Multiplication: Done,

Duration: 0.027595 s