1.Writing the matrix multiplication code.

Our first task was to create a blank project in Nsight and start writing code from scratch. Despite the possibility of using general template for CUDA projects, we have decided to create the code ourselves basing on the knowledge acquired in previous classes and lectures.

It was assumed that used matrices are square in order to simplify the problem.
We began with creating CPU version of algorithm to understand the main idea and also to be able to check the results of computing done by the GPU. This was especially useful when sizes of matrices were too large to simply analyze the output visually. The created sample of code is shown below:

```
for(int i=0;i<matdim;i++){                  //rows
      for(int j=0;j<matdim;j++){            //columns
            float temp=0;
            for(int k=0;k<matdim;k++){
                  temp+=M[i*matdim+k]*N[matdim*k+j];
            }
            P_C[i*matdim+j]=temp;
      }
}
```

All matrices: M,N, and the result of multiplication P_C are stored in one dimensional table with the size of $matdim^2$. All of them were allocated using the Unified memory, where M and N were filled with values in simple loop. In order to perform the multiplication we have to iterate through all elements in nth column of N and nth row of M. This is why there is need of the `temp` variable to store the multiplication result of each element, which is then assigned to proper cell in the P_C matrix. After confirming that the program performs as planned, we proceeded to implement Kernel provided during the 5th lecture:

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int
Width) {
      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 < Width) && (Col < Width)) {
            float Pvalue = 0;
            for (int k = 0; k < Width; ++k) {
                  Pvalue += M[Row*Width+k]*N[k*Width+Col];
            }

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

```
}
```

The sample presented above was prepared to be copy pasted, so the only action needed to make it work was to prepare the grid layout, which in this case has to be two dimensional. Using `dim3` vector we set the layout remembering that maximal amount of threads in block is 1024, and the amount of blocks in grid has to be large enough to cover whole matrix with threads. Algorithm that calculates the grid layout is shown below:

```
int tpb=32
dim3 threadsPerBlock(tpb, tpb);
dim3 blocksPerGrid((matdim + tpb - 1)/tpb,(matdim + tpb - 1)/tpb);
```

As expected this code worked properly and successfully went through verification process.

2. Implementing shared memory

In the previous attempt all threads accessed global memory from the GPU, which is not optimal in cases where we operate on large structures of data. This is why we tried to implement shared memory - a special type, which is located directly on the streaming multiprocessor (SM). Thanks to that it has much higher bandwidth and significantly lower latency - roughly 100 times. This is what we are trying to achieve: to overcome the bottleneck connected with migrating data which is then processed by threads.

Each SM has one memory chip where SharedMem is located, and it is allocated per block of threads. All of them have access to the same memory, and cannot access SharedMem from other blocks. What is different from CPU and serial code - here in CUDA we actually can programm SharedMem and use it to our advantage. In order to do that we have to understand how it works and know it’s limits.

Thread synchronization is one of the problems which sparks when using SharedMem. Despite the fact that threads work in parallel, it does not mean that they all execute their tasks at the same time. This may lead to events where threads may try to access data which is not yet stored in the SharedMem or hasn’t been processed in the previous phase. This is called a race condition. In order to avoid this situation after each phase we have to set up a barrier synchronisation where we start next stage after all threads have finished their job. CUDA provides a `__synchthreads()` synchronisation primitive, which allows the next stage to be performed, only when all threads called `__synchthreads()`.

Unfortunately SharedMem cannot be as large as global memory. Although it has a significantly higher speed. it’s size is about 48KB. It is obvious then that we cannot simply load whole data structure into it, but instead - split it and use only what we need at a particular moment. In our program we do it by dividing the matrices into tiles.

The final kernel which utilizes SharedMem is shown below:

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int
Width) {
//We need to iterate with tiles - starting point and end needed
for tiles
      int Mstart=Width*BLOCK_SIZE*blockIdx.y;    //rows of matrix M
      int Mend=Mstart+Width-1;
      int mstep=BLOCK_SIZE;
      int Nstart=BLOCK_SIZE*blockIdx.x;     //columns of matrix N
      int nstep=BLOCK_SIZE*Width;
      float temp=0;

      //loop through tiles

      for(int m=Mstart,n=Nstart;m<Mend;m+=mstep,n+=nstep){
            __shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
            __shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

      Ms[threadIdx.y][threadIdx.x]=M[m+Width*threadIdx.y+threadIdx.
      x];
      Ns[threadIdx.y][threadIdx.x]=N[n+Width*threadIdx.y+threadIdx.
      x];
      __syncthreads();


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

      __syncthreads();

      }
P[Width * BLOCK_SIZE * blockIdx.y + BLOCK_SIZE * blockIdx.x +
Width * threadIdx.y + threadIdx.x] = temp;
}
```

At the beginning we create variables in order to set boundaries of future iterations (*Mstart*, *Mend*, *Nstart*), because we are dealing with square matrix, setting the boundary in the first (M) matrix is enough.
This boundaries are set accordingly to the direction of adding operation - M matric: horizontally, N matrix: vertically. The variables with point at the start of operations are initialized with *blockIdx* properties, what enables multiplication of whole matrices.
Moreover we create *nstep* and *mstep* variables, which enables operations on consecutive tiles of our matrices,

In the loop we are performing operations on particular row (M) and column (N) of tiles. First, we are transferring data from global to shared memory in order to increase performance. We

create two matrices (*Ms* and *Ns*) of the same size as our tiles, and copying data of corresponding coordinates.
Then we synchronize threads.
Afterwards the data in *Ms* and *Ns* matrices are multiplied and added to *temp* variable, this process is repeated throughout the whole loop, thus data from whole row (M) and column (N) of tiles, are used.
Then we again synchronize threads.

At the end we are copying the data stored in *temp* variable to our P - matrix, which is a result of multiplication of M and N matrices.
Of course we specify the cells according to the respective tiles (*blockIdx* property) and the cells within a tile (*threadIdx* property).

3.Performance test

Accordingly to our initial assumption, matrix multiplication program which uses Shared Memory are significantly faster.
At the time, when our matrices have the size of 6400x6400, the time consumed on matrix multiplication is over three times longer in the program where the Shared Memory isn't in use, the difference is equal to 2.77 s.
Moreover, when we have a look at the matrices of the size 14400x14400, the execution of the kernel which doesn't use Shared Memory takes almost a minute, where the kernel optimized with Shared Memory needs almost 4.5 times less time with the duration of 12.75 seconds.