ACCELERATOR BASED PROGRAMMING UPPSALA UNIVERSITYFALL
ASSIGNMENT 2: Programming with CUDA

I have performed runs on the HSE univeristy HPC-cluster (https://hpc.hse.ru/en/hardware/hpc-cluster/). The nodes are NVIDIA Tesla v100 (https://www.nvidia.com/en-us/data-center/v100/) nodes. Memory bandwidth is 900 GB/s and the performance is 15.7 teraFLOPS for Single-Precision.

Our general goal is to compute matrix-matrix product. In case of two matrices with M*N and N*K elements the result matrix has M*K elements.

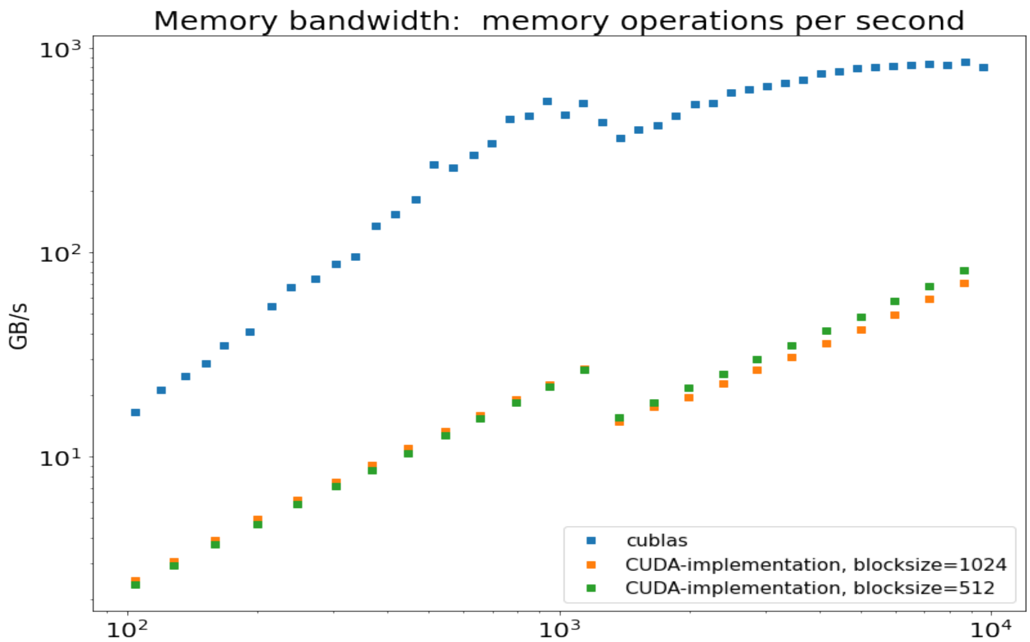
To measure memory bandwidth for matrix-vector product (when K=1), we use (M*N+N*K+M*K) factor.
To measure computational performance for matrix-matrix product, we use 2*M*N*K factor.

Simple parallel matrix-vector implementation.

In this task, we implement the simplest matrix-vector function:

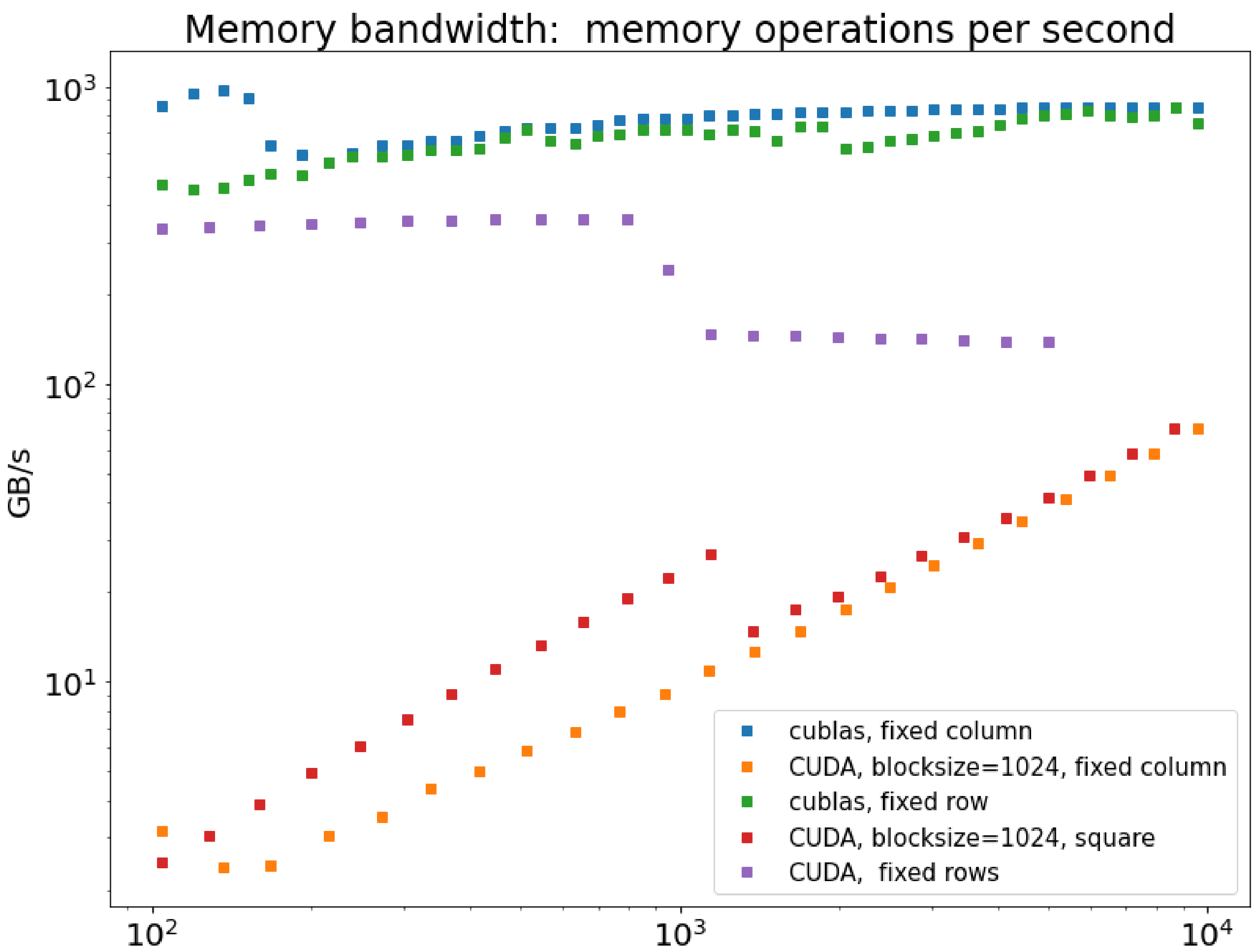
```
__global__ void compute_matrix_vector_product(const int M, const int N, const float *x, const float *y, float *z)
{
 const int idx = threadIdx.x+blockIdx.x*blockDim.x;
     if (idx < M)
     {
         z[idx] = 0;
         for (unsigned int j = 0; j < N; j++)
           z[idx]+=(x[M*j+idx]+ y[j]);
     }
}
```

First, we perform computing for square matrices. We compare our performance with CUBLAS library functions.



The implementation of product has much lower performance in comparison to Cublas. I suppose, to achieve more performance, the loop in kernel function also can be paralleled. The maximum achieved memory bandwidth by Cublas is 852.994 GB/s while my is 81.6256GB/s.

Next, we perform calculation for cases when either number of rows or number of columns is fixed. First, we fix N = 10000 and vary M. After, we fix N = 16384 and vary M.



For large fixed number of rows, we have quite good performance (because in our implementation we paralleled over rows).

<u>Simple Transponation</u>

Here we implement the naive transponation:

```
__global__ void transposeSimple(const int N, const int M, const float *x, const float *y)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  if (xIndex < N && yIndex < M)
   {
      y[xIndex+N*yIndex] = idata[yIndex+M*xIndex];
   }
}
```

This implementation lost a lot of performance because does not use localization of data in cache memory. But it is really easy to implement.

We compute product $A^t x$ for M=N=5000 using this implementation and Cublas.
Cublas implementation showed 783.068 GB/s and mine 42.3618 GB/s.

Matrix-Matrix product impleentation.
Here, we Implement matrix-matrix product for case M=N=K.

```
__global__ void compute_matrix_matrix_product(const int N, const float *x, const float *y, float *z)
{
int i=threadIdx.x+blockIdx.x*blockDim.x;
int j= threadIdx.y+blockIdx.y*blockDim.y;
float value = 0;
if (i<N && j<N)
   {
for ( int k=0;k<N; k++)
            value +=x[i*N+k]*y[k*N+j];

            z[i*N+j] = value;
   }
}
```

We measure computing performance.