Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 3

«The degree of objects similarity»

Student

Miroshnikova A.D. **j4133c**

(Surname, initials) Group

Saint-Petersburg, 2020

1. GOAL OF LABORATORY WORK

The goal of the work is to become acquainted with parallel computations on GPU using CUDA.

2. TASK DEFINITION

The problem to be solved is calculation of distance matrix for matrix A:

Input:

$$A(m \times n)_{m \neq n} = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n} \\ \dots & \dots & \dots & \dots \\ a_{m,0} & a_{m,1} & \dots & a_{m,n} \end{pmatrix}$$

The distance matrix is matrix calculated as distance between every pair of rows in matrix A. In this task, squared Euclidean distance was selected:

$$d_{i,j} = \sum_{k=0}^{n} (a_{i,k} - a_{j,k})^2$$

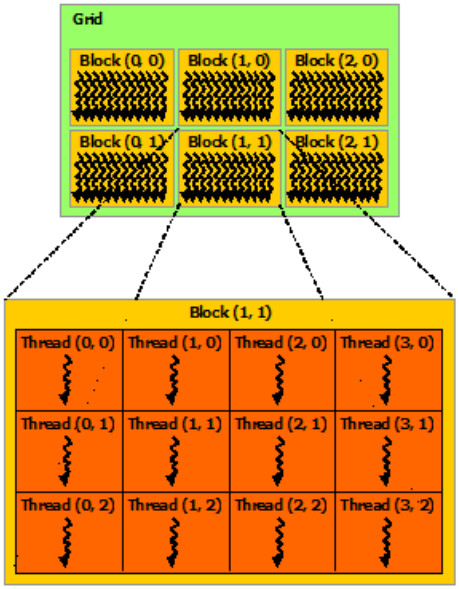
The problem should be solved in sequential way, and with use of CUDA with global and shared memory.

3. BRIEF THEORY

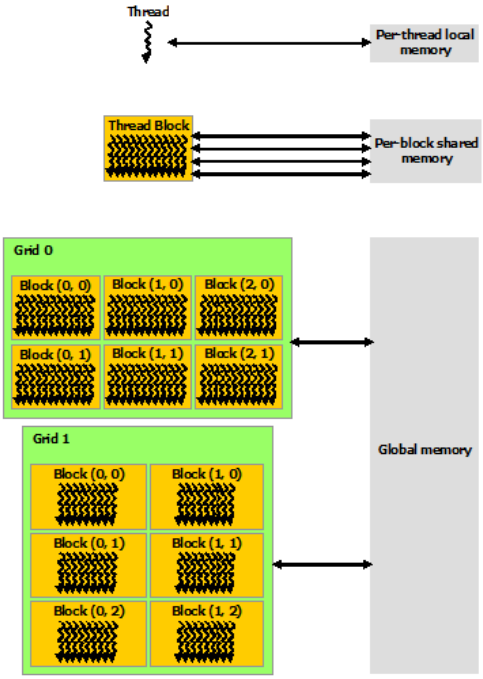
CUDA is a parallel computing platform and programming model developed by Nvidia for general computing on its own GPUs (graphics processing units). CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.

The CUDA Toolkit includes libraries, debugging and optimization tools, a compiler, documentation, and a runtime library to deploy applications. It has components that support deep learning, linear algebra, signal processing, and parallel algorithms. In general, CUDA libraries support all families of Nvidia GPUs, but perform best on the latest generation.

Parallelized approach to CUDA programming involves dividing the problem into grids and solving the problem for each grid with set of CUDA threads.



CUDA threads may access data from multiple memory spaces during their execution . Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.



The code which is executed by each thread on GPU is called a kernel.

Numba is an open-source JIT compiler that translates a subset of Python and NumPy into fast machine code using LLVM, via the llvmlite Python package. It offers a range of options for parallelising Python code for CPUs and GPUs, often with only minor code changes. Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. Kernels written in

Numba appear to have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically.

4. ALGORITHM (METHOD) OF IMPLEMENTATION

The code for computations was implemented using Python 3.6.9 and *numba* library (version 0.52.0). The GPU used during measurements was GeForce GTX 1060 MaxQ, version of CUDA 10.2. The data was analyzed using Python 3.6.9. Complete source code and data gathered are available on github https://github.com/Miffka/parallel_algo_itmo/tree/master/task3.

The sequential approach is straightforward — we use two loops to iterate twice over rows of matrix and calculate similarity within the third loop (here we used vectorized numpy approach to save time, but under the hood in the C++ code there is still a third loop).

```
def similarity_sequential(objs, sim_matrix):
    """
    Arguments
    ---------
        objs        (np.ndarray) : np.ndarray of shape (num_objects, dimensions)
        sim_matrix  (np.ndarray) : empty (or zero) array of shape (num_objects, num_objects)

    Returns
    -------
        sim_matrix  (np.ndarray) : full array of shape (num_objects, num_objects)

    Calculates l2 distance between objects in sequential manner.
    """
    num_objects = objs.shape[0]
    for row in range(num_objects):
        for col in range(num_objects):
            sim_matrix[row, col] = np.sum((objs[row] - objs[col]) ** 2)

    return sim_matrix
```

The CUDA algorithms involve two parts — the kernel function which is executed on a GPU, and the host function which is responsible for data exchange between host and device, and for division of problem to subproblems.

```
@cuda.jit()
def kernel_similarity_global(objs, sim_matrix):
    """
    Calculates l2 distance between row and column in numba global.
    """
    row, col = cuda.grid(2)
    if row < sim_matrix.shape[0] and col < sim_matrix.shape[1]:
        current_sum = 0
        for j in range(objs.shape[1]):
            current_sum += (objs[row, j] - objs[col, j]) ** 2
        sim_matrix[row, col] = current_sum


def similarity_global(objs, sim_matrix):
    """
    Calculates l2 distance between objects in numba using global memory.
    """
    num_objects = objs.shape[0]
    objs_cuda_global = cuda.to_device(objs)
    sim_matrix_global = cuda.to_device(sim_matrix)

    kernel_similarity_global[(BLOCKS_PER_GRID, BLOCKS_PER_GRID), (NTHREADS, NTHREADS)](
        objs_cuda_global, sim_matrix_global
    )
    sim_matrix_global.copy_to_host(sim_matrix)

    return sim_matrix
```

The initial matrix was divided to subproblems by two dimensions. In the case of global memory algorithm each block of threads had to copy the data for subproblem to the shared memory multiple times.

In the case of shared memory algorithm each block of threads had to copy the data for subproblem to the shared memory only one time, thus optimizing communication operations.

```
@cuda.jit()
def kernel_similarity_shared(objs, sim_matrix):
    """
    Calculates l2 distance between row and column in numba with shared memory.
    """
    shared_objs_row = cuda.shared.array((NTHREADS, NTHREADS), dtype=float32)
    shared_objs_col = cuda.shared.array((NTHREADS, NTHREADS), dtype=float32)
    row, col = cuda.grid(2)
    thread_row = cuda.threadIdx.x  # from 0 to NTHREADS - 1
    thread_col = cuda.threadIdx.y  # from 0 to NTHREADS - 1

    if row < sim_matrix.shape[0] and col < sim_matrix.shape[1]:
        current_sum = 0.0
        for i in range(BLOCKS_PER_GRID):
            shared_objs_row[thread_row, thread_col] = objs[row, thread_col + i * NTHREADS]
            shared_objs_col[thread_row, thread_col] = objs[col, thread_row + i * NTHREADS]
            cuda.syncthreads()
            for j in range(NTHREADS):
                current_sum += (shared_objs_row[thread_row, j] - shared_objs_col[thread_col, j]) ** 2
            cuda.syncthreads()
        sim_matrix[row, col] = current_sum


def similarity_shared(objs, sim_matrix):
    """
    Calculates l2 distance between objects in numba using shared memory.
    """
    num_objects = objs.shape[0]
    objs_cuda_global = cuda.to_device(objs)
    sim_matrix_global = cuda.to_device(sim_matrix)

    kernel_similarity_shared[(BLOCKS_PER_GRID, BLOCKS_PER_GRID), (NTHREADS, NTHREADS)](
        objs_cuda_global, sim_matrix_global
    )
    sim_matrix = sim_matrix_global.copy_to_host()

    return sim_matrix
```

The experiments were designed as follows. For each computation method — «sequential», «CUDA with global memory», «CUDA with shared memory» the following parameters were selected:

- size of the problem (i.e. size of matrix A or *objs* in the code) — {128x128, 256x256, 512x512, 1024x1024, 2048x2048};
- number of threads (i.e. size of subproblems) — {8, 16,32}.

The time of execution was measured for each combination of parameters. The first run of GPU algorithm was thrown away since the numba code is compiled during this run and the first run was always slower than the consequent runs.

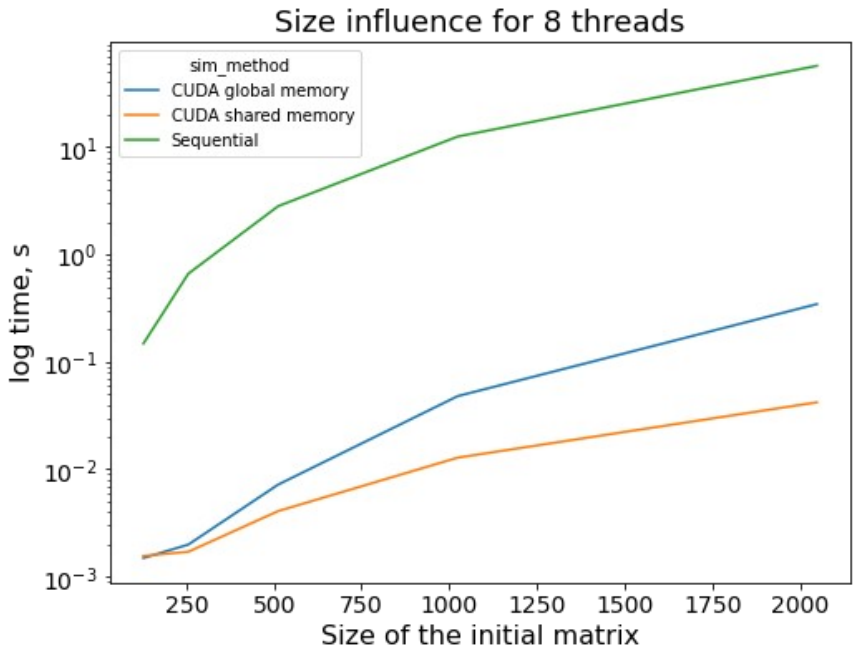
5. RESULT AND EXPERIMENTS

The results were analyzed in Python 3.6.9.

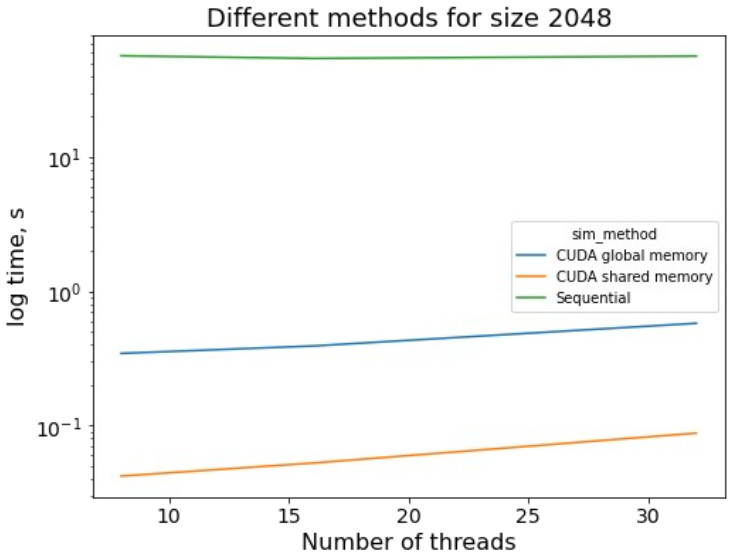
The expected results were the following:

- the time needed for execution should be increasing with size of the problem increasing;
- the time needed for execution should be decreasing with number of threads increasing;
- CUDA with shared memory should be faster than CUDA with global memory, which in turn should be faster than sequential algorithm.

The time needed for computations increased with size of the problem. In the first experiments the first call of CUDA algorithms was always slower than consequent runs because of JIT compilation and GPU warming up, so the first run was thrown away. It can be seen that for the large problem sizes (256 and higher) shared memory algorithm takes less time than global memory algorithm, but for small-size problem both CUDA algorithms have comparable running time.



The influence of the number of threads is unexpected – the time increases with increasing the number of threads. This is unexpected but the documentation says that the optimal number of



threads should be selected for each problem individually, so probably for our size of the problems and our implementation 8 threads is actually perfect.

6. CONCLUSION

Different algorithms for computation of similarity of the objects was implemented using numba API for CUDA computations. The parallel implementations were faster than sequential implementation for all sizes of problem, but for small problems global-memory and shared-memory algorithms show compatible times. The number of threads needed for computations depends not only on the GPU used, but also on the problem.