FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	K-means clustering algorithm on parallel platforms
Student:	Emil Eyvazov
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

1) Study basic K-means clustering algorithm [1] and its improved variants with a triangle inequality [2, 3].
2) Design and implement multithreaded versions of algorithms using OpenMP technology.
3) Design and Implement parallelized versions of algorithms on CUDA.
4) Compare the performance of algorithms from 2) and 3) with python sklearn implementation.
5) Evaluate the quality of clusters obtained from presented algorithms using different clustering scores.

[1] S. Lloyd, "Least squares quantization in PCM," in IEEE Transactions on Information Theory, vol. 28, no. 2, pp. 129-137, March 1982, doi: 10.1109/TIT.1982.1056489.
[2] Elkan, C.: Using the triangle inequality to accelerate k-means. In: Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML 2003, pp. 147–153. AAAI Press (2003)
[3] Hamerly, Greg. (2010). Making k-means Even Faster.. Proceedings of the 2010 SIAM International Conference on Data Mining. 130-140. 10.1137/1.9781611972801.12.
[4] sklearn library, https://scikit-learn.org/

Bachelor's thesis

K-MEANS CLUSTERING ALGORITHM ON PARALLEL PLATFORMS

Emil Eyvazov

Faculty of Information Technology
Department of computer science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.
January 3, 2023

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor doc. Ing. Ivan Šimeček, Ph.D. for helping me with thesis and especially with the subject NIE-GPU that I took, which helped me a lot with understanding NVIDIA CUDA technology and efficient ways of using it.

Declaration

In Prague on January 3, 2023 ....

Abstract

Implementation of K-means clustering algorithm on multithreaded platform using OpenMP and on GPU using CUDA technology. Comparison of time of execution of CUDA implementation with multithreaded and sequential implementations on CPU.

Keywords K-means clustering, OpenMP, GPU, CUDA shared memory, Elkan's and Hamerly's triangle inequality

Abstrakt

Implementace shlukovacího algoritmu K-means na vícevláknové platformě pomocí OpenMP a na GPU pomocí technologie CUDA. Porovnání doby provádění implementace CUDA s vícevláknovými a sekvenčními implementacemi na CPU.

Klíčová slova K-means shlukování, OpenMP, GPU, sdílená paměť CUDA, Elkanova a Hamerlyho trojúhelníková nerovnost

List of abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecure
GPGPU	General-Purpose Graphics Processing Units
GPU	Graphics Processing Unit
OpenMP	Open Multi-Processing
SM	Streaming Multiprocessor

Introduction

The goal of thesis is to implement K-means clustering algorithm using parallelization technologies like OpenMP (Open Multi-Processing) [1] on CPU (Central Processing Unit) and CUDA (Compute Unified Device Architecture) [2] on GPU (Graphics Processing Unit), and compare their performance with sequential algorithm on CPU.

K-means clustering [3] is one of the most important algorithms in Machine Learning. For that reason, it is essential to have an efficient algorithm that would be able to assign a cluster to thousands of points.

The basic algorithm consists of finding the distance between each point and each cluster, assigning each point to the closest cluster, and finding an average of coordinates of all points in a given cluster, further assigning the value of that average of coordinates as a centroid of that cluster.

The implementations of K-means clustering will also include triangle inequality heuristics to reduce the number of needless distance calculations. As K-means clustering algorithm consists of multiple iterations and each iteration consists of finding the closest cluster to each point, which means that Euclidean distances [4] would have to be calculated, some of those distance calculations could be avoided, as in most cases, points don't tend to often change the cluster to which they are assigned to.

Two variations of triangle inequality heuristics will be covered: Charles Elkan's [5] and Greg Hamerly's [6]. Charles Elkan being the first to suggest the usage of triangle inequality between point and two clusters to determine if the current point should change the cluster that it is assigned to. Greg Hamerly improved on Charles Elkan's idea to use triangle inequality on points and clusters and proposed an algorithm that will usually be faster and use much less memory than original Elkan's implementation.

Besides having an efficient sequential algorithm, proper parallelization of given algorithm is as much important. For parallelization purposes, multithreading using OpenMP and NVIDIA CUDA GPGPU (General-Purpose Graphics Processing Units) [7] technology will be used.

To obtain better clusters, proper initialization of centroids must be achieved. The basic algorithms will consist of selecting random points as initial centroids. As randomly selected initial centroids could potentially lead to poor quality of final clusters, a better way of initializing those clusters must found. One of the most famous and widely used centroids initialization algorithm is K-means++ [8], that uses the previous centroid initializations to create a new centroid.

The rest of this thesis is organized as follows:

- Chapter 1 describes the K-means algorithm and possible heuristics to speed up the algorithm.
- Chapter 2 describes parallel platforms on which different solvers could be implemented.
- Chapter 3 describes implementation of solvers different platform and parallelization of them.

- Chapter 4 describes testing and test results of solvers. The results include time of execution of different solvers and Silhouette scores of K-means++ and random centroids initialization methods.
- Chapter 5 summarizes obtained testing results to discuss the solvers and their efficiency.

Chapter 1

Clustering and K-means algorithm in general

1.1 Overview of K-means algorithm

K-means clustering is one of the essential Machine Learning algorithms that originates from signal processing. K-means clustering algorithm separates points into K clusters. Basic algorithm consists of following steps:

1. Initialize centers of clusters, also referred as **centroids**. Points are chosen as initial centroids. Choice of those points as initial centroids is done either randomly or by particular algorithm. Initialization of centroids is covered more in Chapter 1.4.

2. For each point, calculate distance between point's coordinates and each cluster's coordinates. Assign this point to the closest cluster.

3. In case if no point has changed cluster that it is assigned to, stop the algorithm, as each point is correctly assigned to the closest cluster.

4. For each cluster, calculate average of coordinates of all points in this cluster and assign that obtained average as new coordinates of that cluster.

5. Repeat from step 2.

1.2 Triangle inequality heuristics

In most of the cases, clusters don't tend to drastically change their coordinates in each iteration, which means that points in those clusters tend to stay in those clusters. This makes most of distance calculations for finding closest cluster to each point redundant.

1.2.1 Elkan's triangle inequality

Charles Elkan was first to introduce triangle inequality as a way of drastically decrease the number of distance calculations [5]. In order to decrease distance calculations, previous values of those calculations should be used. Elkan's method consists of finding so called **upper bound** and **lower bounds** for each point.

Upper bound of each point is a maximum possible value of distance from this point to the cluster that it is currently assigned to. Lower bound is calculated between each point and each cluster as a minimum possible distance to that cluster. Each point has $K-1$ lower bounds (lower bound to the cluster that this point is currently assigned to is not needed). Thus, by comparing the value of upper bound of this point to all lower bounds of it, it is possible to establish if there is a cluster that is potentially closer to this point or not. As, if maximum possible distance between a given point p and a cluster that it is currently assigned to c_i is lower than or equal to minimum distances to all other clusters $c_j : 0 \leq j < K \wedge j \neq i$, then cluster c_i is the closest cluster to the point p and there is no need to calculate distances between point p and each cluster c_j.

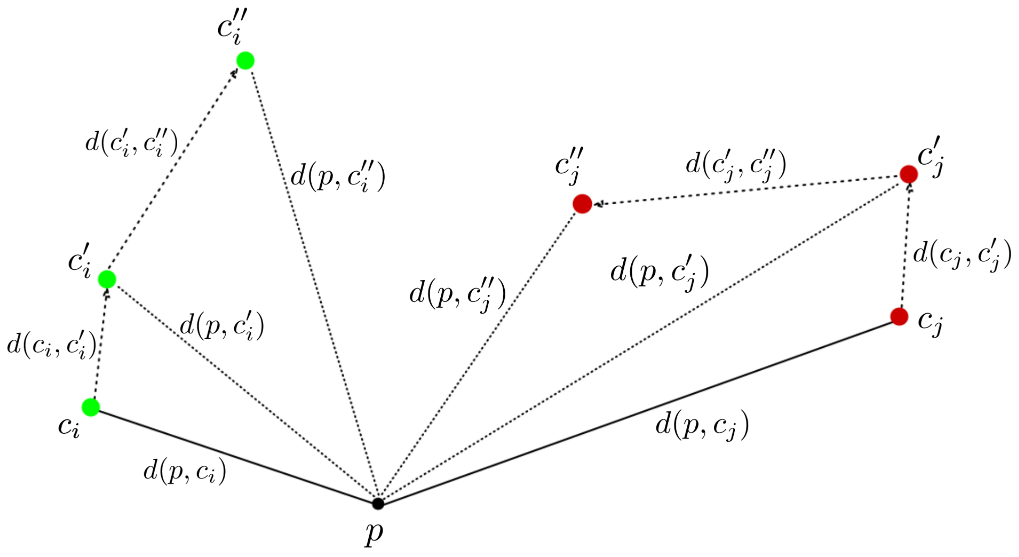

Figure 1 Distances between point and clusters

Figure 1 illustrates an example of distances between a point p and two clusters c_i and c_j in 3 iterations of the algorithm:

1. In the first iteration, point p is assigned to cluster c_i, as a distance between p and c_i, i.e. $d(p, c_i)$, is less than a distance between p and c_j, i.e. $d(p, c_j)$. As p is assigned to c_i, the upper bound of point p is $d(p, c_i)$. Lower bound of point p to cluster c_j is $d(p, c_j)$.

2. In the second iteration, centroid of cluster c_i moved to c'_i and a centroid of cluster c_j moved to c'_j. As in this iteration, the distance between p and a cluster it is assigned to c'_i is equal to $d(p, c'_i)$, in order to not calculate values of distances, an approximation with an upper bound to c'_i and a lower bound to c'_j could be used. The upper bound of p is an approximated value of an upper bound of $d(p, c'_i)$, which is $d(p, c_i) + d(c_i, c'_i)$. Effectively, upper bound is found via triangle inequality, as, if values $d(p, c_i)$, $d(c_i, c'_i)$, and $d(p, c_i)$ would be sides of a triangle, triangle inequality for $d(p, c'_i)$ would look like:

$$|d(p, c_i) - d(c_i, c'_i)| < d(p, c'_i) < d(p, c_i) + d(c_i, c'_i)$$

   In the same fashion the lower bound to c'_j of p, i.e. lower bound of $d(p, c'_j)$, is found via triangle inequality:

$$|d(p, c_j) - d(c_j, c'_j)| < d(p, c'_j) < d(p, c_j) + d(c_j, c'_j)$$

thus lower bound of $d(p, c'_j)$ is equal to $d(p, c_j) - d(c_j, c'_j)$. In case if value of $d(p, c_j) - d(c_j, c'_j)$ is less than 0, the lower bound of $d(p, c'_j)$ would be set to 0.

Finally, the values of upper bound and lower bound for p are compared:

- If the value of upper bound is less than or equal to the value of lower bound, then there is no need to calculate precise distances $d(p, c'_i)$ and $d(p, c'_j)$, as the maximum possible value of $d(p, c'_i)$ is less than or equal to the minimum possible value of $d(p, c'_j)$ and this would mean that c'_i is still the closest cluster to p.
- In case the upper bound of p, i.e. maximum possible value of $d(p, c'_i)$, is greater than the lower bound to c'_j , i.e. the minimum possible value of $d(p, c'_j)$, the precise distances would have to be calculated.

In this iteration, upper bound turns to be less than a lower bound, so calculations of distances $d(p, c'_i)$ and $d(p, c'_j)$ are not needed.

3. In the third iteration, centroids move again: $c'_i \to c''_i$ and $c'_j \to c''_j$. This time, the centroids moved a lot, which resulted in the upper bound of $d(p, c''_i) = d(p, c'_i) + d(c'_i, c''_i)$, being greater than a lower bound of $d(p, c''_j) = d(p, c'_j) - d(c'_j, c''_j)$.

   So, in this iteration, the precise distances $d(p, c''_i)$ and $d(p, c''_j)$ must be calculated.

Although such method removes a lot of distance calculations between points and clusters, it adds distance calculations between clusters' previous and current centroids. But, as there are going to be fewer clusters than points, those additional distance calculations between centroids do not lead to significant performance drawback.

1.2.2 Hamerly's improvement

Although Charles Elkan's triangle inequality served its purpose of reducing redundant distance calculations between points and clusters, it brought additional upper and lower bounds calculations. The most significant impact being made by lower bound calculations. As lower bound must be calculated between each point and each cluster in each iteration of algorithm, memory bandwidth becomes an issue in platforms that are sensitive to memory addressing, e.g. CUDA.

In order to get rid of multiple lower bound calculations between each point and each cluster, Greg Hamerly introduced his variant of triangle inequality heuristics [6]. Hamerly's method consists in only one upper bound and only one lower bound per point. Upper bound is calculated as it is in Elkan's method, but a lower bound for each point is calculated as a minimum possible value of distance to the second closest cluster.

In Elkan's method, the lower bound to cluster c'_j(from Figure 1) is calculated as

$$d(p, c'_j) = d(p, c_j) - d(c_j, c'_j)$$

but in Hamerly's method that same equation would be

$$d(p, c'_j) = d(p, c_j) - \max(d(c_k, c'_k) : 0 \leq k < K)$$

So, in Hamerly's method the lower bound of the point is found as a previous value of the lower bound to the second closest cluster minus the value of the distance of the cluster that moved the most.

This way we ensure that even if the second closest cluster moved the most(the calculated value of the lower bound to that cluster is close to 0), in case if upper bound to the closest cluster was less than or equal to the lower bound to the second closest cluster, the closest cluster to the given point hasn't changed and there is no need to calculate all distances between that point and all clusters.

1.2.3 Drawbacks of triangle inequality heuristics

The possible drawback of both Elkan's and Hamerly's triangle inequality heuristics consists in extra calculations of lower and upper bounds. In the memory sensitive platforms, e.g. CUDA, loading lower and upper bounds, and further calculations of them could take more time than distance calculations between points and clusters. This drawback could be more significant in Elkan's heuristic, as Elkan's heuristic consists in calculating lower bounds between each point and each cluster, more data should be loaded from global memory, thus calculations of lower bounds in each iteration takes more time.

1.3 Metrics for measuring quality of clusters

In order to calculate the quality of obtained clusters, i.e. how good clusters are separated from each other or how dense they are, different metrics are used:

- Silhouette score: the most famous method used for measuring the quality of obtained clusters is a Silhouette score, see [9]. The Silhouette score is between -1 and 1:

Silhouette score	Quality of clusters
-1	Clusters did not properly form, better clustering of given points can be achieved
0	Some clusters overlap
1	The clusters are well separated from one another

  So, the closer the silhouette score to 1, the better are clusters that were formed.

- Calinski-Harabasz score: also used to determine the quality of clusters [10]. The higher Calinski-Harabasz score indicates better clustering.

- Davies-Bouldin score: the last method that is used for determining the quality of clusters is Davies-Bouldin score [11]. The lower Davies-Bouldin score indicates better clustering.

1.4 Initialization of centroids with K-means++

K-means++ is an algorithm for initial centroid initializations [8]. This algorithm is used, as it creates sophisticated initial centroids that will eventually result in higher quality clusters.

The initial centroids obtained by K-means++ algorithm are mutually distant from each other, thus there are no situations, when clusters are very close to each other and the points close to them are divided in two clusters, whereas they could represent only one cluster.

To obtain mutual distance between initial centroids, K-means++ adds clusters one by one, where each new clusters is chosen as a point with maximum distance from all the existing clusters. The algorithm is

1. Randomly pick one point as the first centroid.
2. Find closest cluster to each point.
3. Find the farthest point from its cluster and assign it as a new centroids.
4. Repeat from step 2 as long as the required number of centroids is initialized.

Figure 2 illustrates an example K-means++ step by step with 10 points and 4 centroids that should be initialized:

1. The first centroid is picked by random and it is point 0.

2. The distances are calculated between the centroid 0 and all points and point 1 is the farthest, thus it becomes the second centroid.
3. The closest cluster is calculated for each point and each point is assigned to either cluster 0 or 1. The farthest point from its cluster is chosen and it is point 2.
4. The closest cluster is again calculated for each point and now point 3 is the farthest point from its cluster, thus it becomes the last centroid.

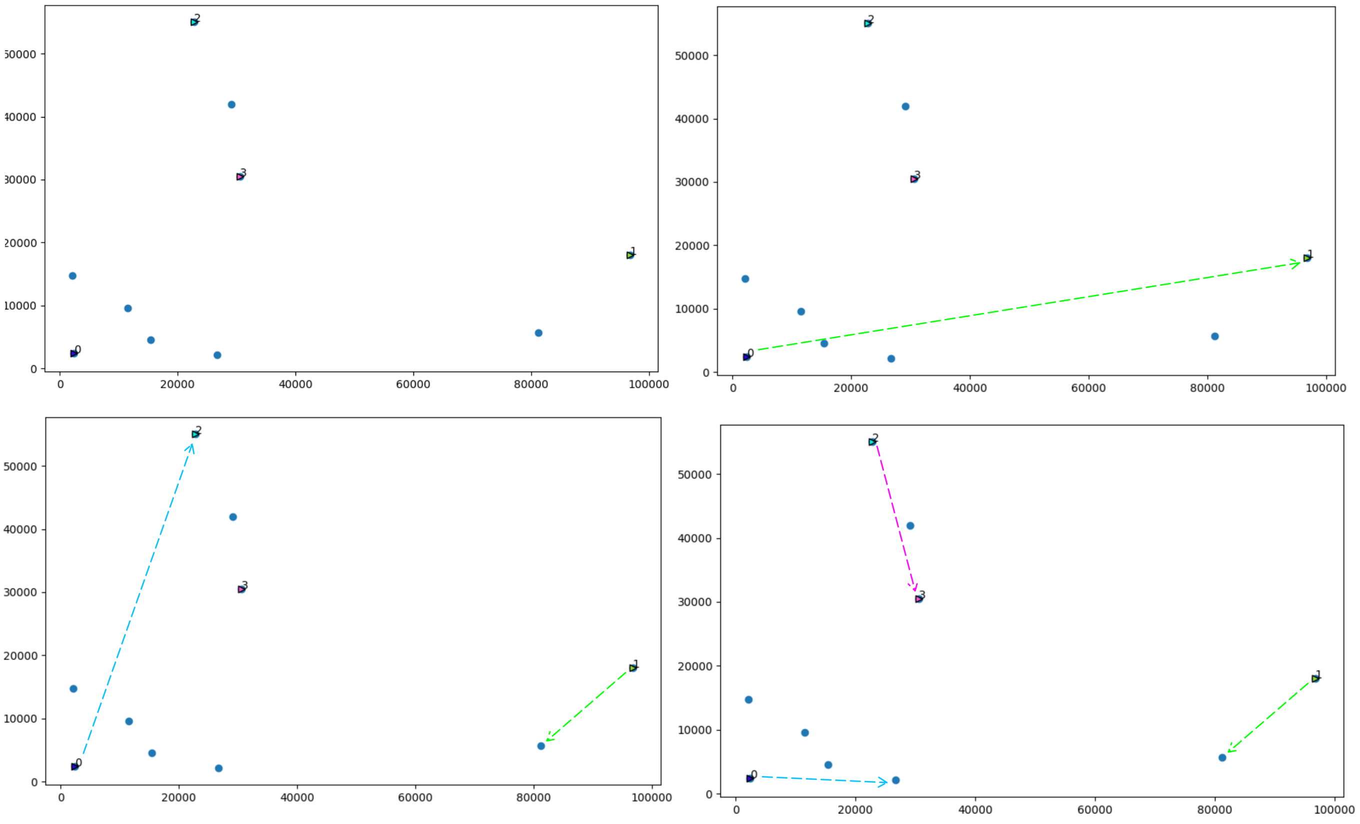

Figure 2 K-means++ by example

All the implementations of K-means++, i.e. sequential, OpenMP, and CUDA, are very similar to the original K-means clustering algorithm, as K-means++ algorithm consists of as many iteration as many centroids are needed. Each iteration, simply computes closest cluster to each point, finds the farthest point from all clusters, and makes that point a new centroid.

Chapter 2

Parallel platforms

2.1 OpenMP technology

OpenMP (Open Multi-Processing) [1] is an API for programming of multithreaded applications on shared memory. OpenMP is used for parallelism within a node of multiple cores that share (virtual) memory. OpenMP is a huge library, but only the parts important to this thesis will be covered.

OpenMP programming model is based upon **fork-join** mechanism, when a **master thread** creates a **parallel region** for **slave threads** to execute that region of code concurrently, as illustrated on Figure 3.

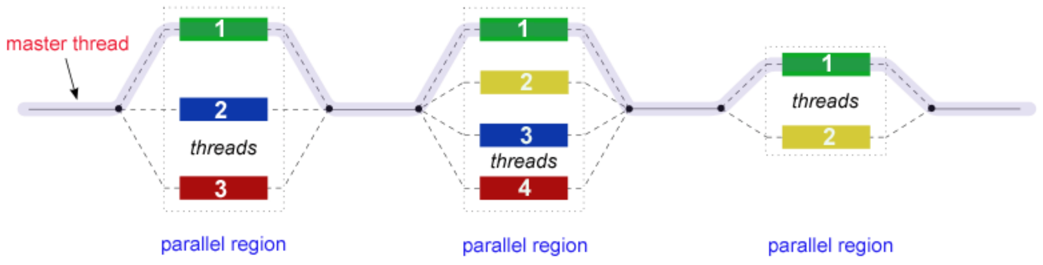

Figure 3 OpenMP fork-join model (source: LLNL HPC-tutorials)

Getting into parallel region is achieved via OpenMP directives, mainly being

Code listing 2.1 Parallel region

```
int i, j;
omp_set_num_threads(num_threads);
#pragma omp parallel private(i) shared(j)
{
    // parallel region
}
```

Such a directive would create a *num_threads* number of threads in a thread pool that would be used in future parallel regions. Each thread will get its own copy of variable i and the changes made to that variable won't affect the copy of it in master thread, whereas, variable j will be shared among threads, thus modifications done to it via one thread would be visible to other threads including the master thread.

2.1.1 Multithreaded parallelization methods

There are two main parallelization methods in OpenMP: task and data parallelisms. Task parallelism is achieved via `task` directive, but it won't be further covered, as OpenMP implementations in this thesis don't use task parallelism.

Data parallelism is a parallelization method that is achived via splitting the data, e.g. array, into separate disjoint regions that would be processed by concurrent threads. It is achived via parallel `for` directive:

Code listing 2.2 Data parallelism

```
int * array;
int arr_size, chunk_size;
// initialization of the array

omp_set_num_threads(num_threads);
#pragma omp parallel
{
    #pragma omp for schedule(static, chunk_size)
    for(int i = 0; i < arr_size; ++i) {
        // some operations on array
    }
}
```

Each thread gets its iterations that are separated from other threads' iterations by chunks. Array of size *arr_size* is divided into chunks of size *chunk_size* and each chunk is statically assigned to threads, e.g. with 3 slave threads, *arr_size* = 14, and *chunk_size* = 3, the chunks would be assigned to threads accordingly:

1. Iterations 0–2 will be processed by thread 0.
2. Iterations 3–5 will be processed by thread 1.
3. Iterations 6–8 will be processed by thread 2.
4. Iterations 9–11 will be processed by thread 0.
5. Iterations 12–13 will be processed by thread 1.

There are other scheduling mechanisms like `dynamic` and `guided`, where chunks could be assigned to threads according to the threads being free or busy at that moment, but those scheduling mechanisms won't be covered further.

It is possible to omit *chunk_size*, in that case, the iterations are going to be equally distributed into chunks for each thread.

2.1.2 Reduction

There are several mechanisms to accumulate results from all threads into one result in the master thread. One of the widely used mechanisms is **parallel reduction**, which is implemented in OpenMP to be convenient for usage. Reduction is achieved via `reduction(op:variable)`, where *op* is an operation that should be done accumulatively on each thread's copy of *variable*. There are different possible reduction operations like: +, -, * and so on.

Listing 2.3 illustrates an example of a reduction with 2 slave threads and *chunk_size* = 2. As there are 5 iterations, they will be split into chunks:

1. Iterations 0–1 will be processed by thread 0.
2. Iterations 2–3 will be processed by thread 1.

3. Iteration 4 will be processed by thread 0.

Code listing 2.3 OpenMP reduction

```
int sum = 0, chunk_size = 2;
omp_set_num_threads(2);
#pragma omp parallel
{
    #pragma omp for schedule(static, chunk_size) reduction(+:sum)
    for(int i = 0; i < 5; ++i)
        sum += 10;
}
```

Which means, that thread 0 will get 3 iterations, so variable sum will be increased 3 times, so at the end $sum = 30$. The thread 1 will get 2 iterations, so the value will be $sum = 20$. Eventually, the values of sum, from both slave threads, will be accumulated to obtain the value 50. So, master thread will get the value of $sum = 50$.

2.2 CUDA technology

CUDA [2] is a parallel computing platform and an API for GPGPU that is developed by NVIDIA. As CPU and GPU work together to efficiently compute different tasks in a process called **co-processing**, an efficient and easy to use technology is needed to enable such a paradigm. Such a technology is CUDA that makes co-processing easier between CPU that is referred to as **host** and GPU that is referred to as **device**.

2.2.1 GPU architecture

GPU is a special device with its architecture being very different from CPU's architecture. CPU has multiple powerful cores with very high clock speed and a rich instruction set architecture. On the other hand, GPU's cores are much simpler and slower, but this allows the manufacturers to utilize many more cores on GPU rather than on CPU, which means that many more threads will run concurrently on GPU compared to CPU. This gives a GPU a capability to compute many simple computations concurrently, which is very useful for an application with abundance of mathematical computations. GPU consists of multiple SMs (Streaming Multiprocessors that) have a common L2 cache and one global memory.

2.2.1.1 Thread groupings

Figure 4 illustrates an architecture of one Streaming Multiprocessor of Kepler GPU. Each Streaming Multiprocessor consists of multiple Streaming Processors, which are essentially cores where threads are going to run on. Threads are running concurrently in groups which are called **warps**. Each warp consists of 32 threads. GPU follows SIMT (Single Instruction Multiple Threads) model, which means that each thread in a warp executes the same instruction at a time. This could bring to **branch divergence**, when a branch, based on its condition, could cause threads, that don't satisfy the branch condition, wait until other threads, that satisfy the branch condition, execute the body of the branch. Such divergence could bring to serialization of the code that brings to the decrease of performance. Execution of warps and their scheduling is done via warp scheduler.

Multiple warps form a **block**, blocks together form a **grid**. The number of threads in one block is limited to 1024.

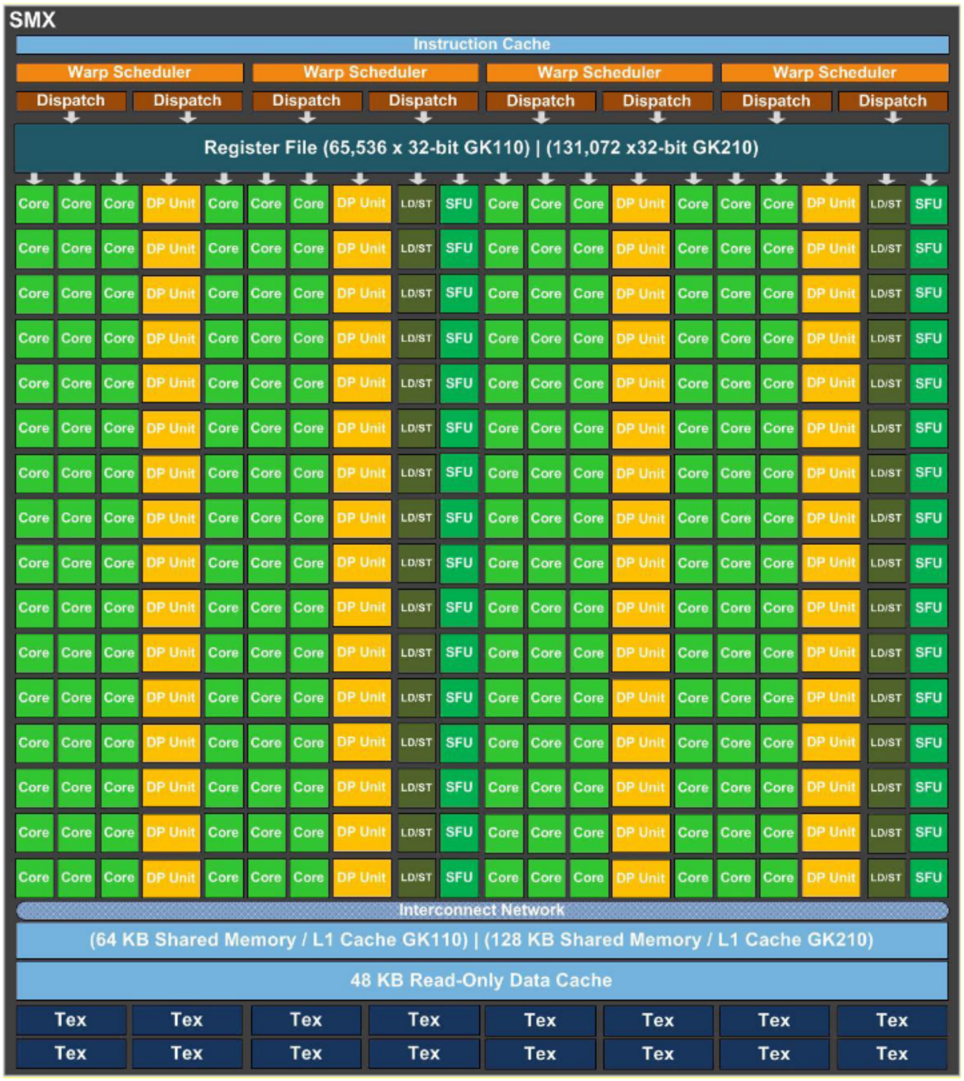
SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit GK110) | (131,072 x32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconnect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

Figure 4 Streaming Multiprocessor Architecture (source: Nvidia)

2.2.1.2 Coalesced memory accesses

One of the main performance obstacles of parallel and distributed systems is memory bandwidth, as a lot of consecutive memory accesses could bring up to decrease in performance. CUDA enables coalescing of consecutive memory accesses into one, which could prevent the drastic drop in performance due to memory bandwidth. As the accessed data is transferred from global memory into cache, it is crucial for consecutive threads to access consecutive memory addresses.

2.2.1.3 Shared memory

Each Streaming Multiprocessor has its own L1 cache which is shared among all the cores of that SM. L1 cache can be partially dedicated to a programmer as a so called **shared memory**. Shared memory is limited and, if allocated, each block gets its own memory address from shared memory. Shared memory can be upto 100 times faster than global memory, so it can serve as a significant performance boost if used appropriately.

2.2.1.4 Double precision units

Each core consists of ALU (Arithmetic Logic Unit) and FPU (Floating Point Unit), so 32-bit operations can be quickly executed via each core. The situation is different with double precision numbers, as GPUs were initially created for graphics and gaming, 64-bit calculations weren't a necessity, so old versions of GPUs didn't support 64-bit double precision calculations. With occurence of GPGPU in 2008, a need for better precision became a necessity, thus a double precision units were added to each Streaming Multiprocessor, but as there were fewer double precision units compared to single precision units, the performance of calculations with double precision is significantly lower than of single precision calculations. Figure 4 illustrates double precision units as *DP Unit*.

2.2.2 CUDA programming model

To efficiently utilize NVIDIA GPUs, CUDA toolkit [12] is used, which serves as an extension of C++. The main part of CUDA toolkit is a CUDA **kernel**, which represents an extension of standard C++ function that is recognized by NVIDIA NVCC compiler, that will eventually run on a device.

In order to run a kernel, a configuration should be given. Configuration of the kernel is:

1. Block configuration: number of threads in a block. Could be given up to 3 dimensions: x, y, z.
2. Grid configuration: number of blocks per grid. Also could be given up to 3 dimensions.
3. Size of shared memory per block(in bytes).
4. Stream that will run this kernel. Won't be further covered, as all kernels in this thesis are running on a default stream.

To define number of threads per block, and number of blocks per grid, `dim3` structure is used. This structure holds the values of x, y, and z coordinates respectively.

Kernel is called in the same way as any C++ function would be called with only addition of $<<< \cdots >>>$ after a function name and before the parantheses with parameters. Kernel configuration with block dimension, grid dimension, and the size of allocated shared memory per block are defined between $<<< \cdots >>>$.

Any allocated memory from host can't be sent to device directly, as device has its own global memory. In order to transfer data from host to device, the memory should be allocated in device memory. Allocation and deallocation in a global memory of the device is done via special CUDA functions: `cudaMalloc` and `cudaFree` . Copying data from host to device is done via `cudaMemcpy` function.

Code listing 2.4 Example of kernel configuration

```
dim3 block_dim(5, 3, 1);
dim3 grid_dim(2, 2);
int shared_mem_size = 20;

my_kernel<<<block_dim, grid_dim, shared_mem_size>>>(cuda_array_in, cuda_array_out, arr_size);
```

Listing 2.4 illustrates a host code that serves as an example of block and grid configurations for kernel `my_kernel` . Blocks are of dimensions $5 \cdot 3 \cdot 1$, so 15 threads are configured per block. Grid dimensions are $2 \cdot 2$, resulting in 4 blocks in total.

CUDA kernels are non-blocking, which means that host, after calling kernel, won't wait for the kernel to finish and will start execution of the next instruction. In order to wait for an execution device kernel, host must call a function `cudaDeviceSynchronize` right after the kernel execution of which host is waiting for. Kernels and special CUDA functions for memory allocation, deallocation, and copy, however, execute on the device in such an order in which they were called. It is possible to run kernels concurrently on a device and it is achieved via CUDA streams, but streams won't be covered any further, as all the kernels in this thesis will run on a default stream.

In order to declare a kernel, special keyword `__global__` must be used right before the function declaration and definition.

Code listing 2.5 Example kernel

```
1 __global__ void example_kernel(int * arr, int arr_size) {
2     int thread_pos = blockDim.x * blockIdx.x + threadIdx.x;
3 
4     extern __shared__ int shared_arr[];
5     shared_arr[threadIdx.x] = arr[thread_pos];
6     __syncthreads();
7 }
```

Listing 2.5 illustrates an example of a kernel that only loads data into shared memory:

1. Line 2: absolute position of a thread is found via CUDA specific variables:
   - `blockDim` — dimension of the block, i.e. number of threads in the block.
   - `blockIdx` — index of the block in the grid.
   - `threadIdx` — index of the thread in its block.

   All of these variables are objects of `dim3` structure, so each variable has 3 members: x, y, z.
2. Line 4: shared memory is declared with the size that was given to kernel as its configuration.
3. Line 5: data is written form global memory to shared memory.
4. Line 6: threads in a block are synchronized via `__syncthreads` function, so that all threads in a block load the data into shared memory.

This serves just as an example of how kernel is defined and how data is loaded into shared memory, in real applications, kernel would have done some calculations on the data from shared memory, then would have stored the results in global memory.

Chapter 3

Implementation and parallelization

3.1 Sequential implementations

Listing 3.1 illustrates the main loop of the algorithm:

1. Line 3: flag that indicates if algorithm has converged is set to `true` , as by default it is considered that no point has changed the cluster it is assigned to.
2. Line 4: function is called to find the closest cluster to each point.
3. Lines 5–6: if no point has changed the cluster it is assigned to, the algorithm converges.
4. Line 7: function is called to find the new centroid of each cluster by finding average of all points in that cluster.

Code listing 3.1 Main algorithm loop

```
1  bool flag_finished;
2  while(true) {
3      flag_finished = true;
4      find_closest_cluster(&flag_finished);
5      if(flag_finished)
6          break;
7      find_cluster_average();
8  }
```

Listing 3.2 illustrates a `find_closest_cluster` templated function with template parameter `T` that receives

- Pointer to a flag parameter that indicates if algorithm has converged.

and find the closest cluster to each point:

1. Lines 2–22: iterate over all points:
    a. Lines 3–8: if a point is assigned to some cluster, find the distance between that point and the cluster it is assigned to. This distance is considered as minimum distance.
    b. Lines 10–21: iterate over all clusters:
        i. Lines 11–14: find the distance between a point and a cluster.

ii. Lines 16–20: if point is not assigned to any cluster or the distance to the current cluster is less than the minimum distance, consider this distance as a new minimum distance and assign the point to the current cluster. As the cluster to which this point is assigned to has changed, set the flag to `false` .

Code listing 3.2 Function for finding the closest cluster to each point

```
1  T min_cluster_dist, current_cluster_dist;
2  for(auto & point : m_points->m_vec_points) {
3      if(point.m_cluster) {
4          min_cluster_dist = calculate_distance(
5              (const Coordinate<T> *) (&(point.m_coordinate)),
6              (const Coordinate<T> *) (&(point.m_cluster->m_coordinate))
7          );
8      }
9
10     for(auto & cluster : m_exec_results->m_clusters) {
11         current_cluster_dist = calculate_distance(
12             (const Coordinate<T> *) (&(point.m_coordinate)),
13             (const Coordinate<T> *) (&(cluster.m_coordinate))
14         );
15
16         if(!(point.m_cluster) || current_cluster_dist < min_cluster_dist) {
17             min_cluster_dist = current_cluster_dist;
18             point.m_cluster = &cluster;
19             *flag_finished = false;
20         }
21     }
22 }
```

Listing 3.3 illustrates a `find_cluster_average` templated function with template parameter `T` that finds average of X and Y coordinates of all points in that cluster and assigns those values to that cluster.

Code listing 3.3 Function for finding the closest cluster to each point

```
for(int cluster_id = 0; cluster_id < m_exec_results->m_clusters.size(); ++cluster_id) {
    Cluster<T> * cluster = &(m_exec_results->m_clusters[cluster_id]);
    T sum_x = 0, sum_y = 0;
    int cluster_size = 0;

    for(const auto & point : m_points->m_vec_points) {
        if(point.m_cluster->m_cluster_id == cluster->m_cluster_id) {
            sum_x += point.m_coordinate.m_x;
            sum_y += point.m_coordinate.m_y;
            ++cluster_size;
        }
    }

    if(cluster_size > 0) {
        cluster->m_coordinate.m_x = sum_x / cluster_size;
        cluster->m_coordinate.m_y = sum_y / cluster_size;
    }
}
```

3.1.1 Elkan's heuristic

For finding upper and lower bounds, `find_closest_cluster` function should be modified:

- In each iteration over vector of points, upper bound and lower bounds should be updated as illustrated on Listing 3.4.

1. Line 2: the distance that the cluster's centroid, that this point is assigned to, has moved should be added to the current value of the upper bound of this point.
2. Lines 4–18: iterate over all clusters:
   a. Lines 5–6: if the iterated cluster is the currently assigned cluster to this point, just continue, as no comparison of upper and lower bounds should be made.
   b. Lines 8–12: calculate the lower bound between this point and currently iterated cluster. The lower bound is a difference between current value of it and the distance that this cluter's centroid moved. In case if the obtained value of lower bound is negative, assign 0 to it.
   c. Lines 14–17: if calculated upper bound is greater than the value of a lower bound for the currently iterated cluster, then this cluster could be closer to this point than a cluster that this point is currently assigned to. In such case, actual distances between this point and all clusters should be calculated and bounds should be reset, so the flag `was_updated` is set to `true` .
   d. Lines 20–21: if the bounds shouldn't be updated(upper bound of this point was less than or equal to lower bounds to all clusters), then no distance calculations should be made, so the outermost loop that iterates over points in `find_closest_cluster` function should continue to the next point.
   e. Lines 23–27: distance to the cluster, that this point is currently assigned to, is calculated and the value of this distance is assigned to the upper bound of this point.

- In the inner loop of `find_closest_cluster` function that iterates over all clusters, the calculated `current_distance` should be assigned to the lower bound between this point and currently iterated cluster. This is done to reset the value of a lower bound, in case if `was_updated` flag is `true` and a reset of bounds is needed.

Code listing 3.4 Updating upper and lower bounds for Elkan's heuristic

```
1  if(point->m_cluster) {
2      *upper_bound += m_cluster_prev_dist[point->m_cluster->m_cluster_id];
3
4      for(auto & cluster : m_exec_results->m_clusters) {
5          if(point->m_cluster->m_cluster_id == cluster.m_cluster_id)
6              continue;
7
8          lower_bound = &(m_points_bounds[point_id].m_lower_bounds[cluster.m_cluster_id]);
9          *lower_bound -= m_cluster_prev_dist[cluster.m_cluster_id];
10
11         if(*lower_bound < 0)
12             *lower_bound = 0;
13
14         if(*upper_bound > *lower_bound) {
15             was_updated = true;
16             break;
17         }
18     }
19
20     if(!was_updated)
21         continue;
22
23     min_cluster_dist = calculate_distance(
24         (const Coordinate<T> *) (&(point->m_coordinate)),
25         (const Coordinate<T> *) (&(point->m_cluster->m_coordinate))
26     );
27     *upper_bound = min_cluster_dist;
28 }
```

Function `find_cluster_average` to find the new centroid for each cluster should also be modified to find the distance between current cluster centroid and new cluster centroid. The last `if` branch should be modified as illustrated in Listing 3.5:

1. Lines 2–4: find the new coordinates of the centroid.
2. Lines 5–9: calculate the distance between current and new centroids.
3. Lines 10–11: assign coordinates of the new centroid to this cluster.

Code listing 3.5 Finding distances between clusters' centroids

```
1  if(cluster_size > 0) {
2      Coordinate<T> cluster_new_coordinates{
3          sum_x / cluster_size, sum_y / cluster_size
4      };
5      m_cluster_prev_dist[cluster_id] =
6          KMeansClusteringSequential<T>::calculate_distance(
7              (const Coordinate<T> *) (&(cluster->m_coordinate)),
8              (const Coordinate<T> *) (&(cluster_new_coordinates))
9          );
10     cluster->m_coordinate.m_x = cluster_new_coordinates.m_x;
11     cluster->m_coordinate.m_y = cluster_new_coordinates.m_y;
12 }
```

3.1.2 Hamerly's heuristic

The function `find_closest_cluster` is modified:

- Update of bounds is much simpler than in Elkan's heuristic, as only one lower bound is needed for each point. Listing 3.6 illustrates updating of bounds:
  1. Line 2: upper bound is calculated the same as it was calculated for Elkan's heuristic.
  2. Lines 3–6: lower bound is calculated. The updated value of a lower bound is a difference of current value of a lower bound with the maximum distance between clusters' previous and current centroids, i.e `m_max_cluster_prev_dist` . The value of `m_max_cluster_prev_dist` is calculated in `find_cluster_average` function.
  3. Lines 8–9: if the value of upper bound for this point is less than or equal to the value of lower bound of this point, there is no need to calculate distances between this point and all clusters, as the cluster, this point is currently assigned to, is definitely the closest one.

Code listing 3.6 Updating upper and lower bounds for Hamerly's heuristics

```
1  if(point->m_cluster) {
2      *upper_bound += m_cluster_prev_dist[point->m_cluster->m_cluster_id];
3      *lower_bound -= m_max_cluster_prev_dist;
4
5      if(*lower_bound < 0)
6          *lower_bound = 0;
7
8      if(*upper_bound <= *lower_bound)
9          continue;
10 }
```

- In case if upper bound of this point is greater than the lower bound, the bounds must be reset and it is done inside an inner loop of the `find_closest_cluster` that iterates over all clusters to find distance between each cluster and this point. As the new value of the lower bound is the distance to the second closest cluster to this point, the two values of minimum distances should be held: `min` and `prev_min`. Listing 3.7 illustrates the reset of bounds for this point:

1. Lines 2–5: calculate the distance from this point to currently iterated cluster.
2. Lines 7–11: if the value of minimum distance is not set or the distance to the currently iterated cluster is less than the minimum distance, assign the value of the current minimum distance to the second minimum distance, as previous minimum is the second minimum, and assign the value of the distance to the current cluster to the minimum distance.
3. Lines 11–12: if the value of the second minimum is not set or the value of the distance to the currently iterated cluster is less than the current value of second minimum, assign the value of the distance to the currently iterated cluster to the second minimum distance. This branch would execute only in case if the distance to the currently iterated cluster is greater than or equal to the distance to the closest cluster, that is the value of `min`, but it is less than the distance to the second closest cluster, so the value of the distance to the second closest cluster should be updated.
4. Lines 15–16: the minimum distance is assigned to the upper bound and the second minimum distance is assigned to the lower bound.
5. Lines 18–21: if this point is not assigned to any cluster or the obtained closest cluster is different than an already assigned one, assign this point to the obtained closest cluster.

Code listing 3.7 Finding closest and second closest clusters for Hamerly's heuristics

```
1  for(auto & cluster : m_exec_results->m_clusters) {
2      current_cluster_dist = calculate_distance(
3          (const Coordinate<T> *) (&(point->m_coordinate)),
4          (const Coordinate<T> *) (&(cluster.m_coordinate))
5      );
6
7      if(min == -1 || current_cluster_dist < min) {
8          prev_min = min;
9          min = current_cluster_dist;
10         point_cluster = &cluster;
11     } else if(prev_min == -1 || current_cluster_dist < prev_min)
12         prev_min = current_cluster_dist;
13 }
14
15 *upper_bound = min;
16 *lower_bound = prev_min;
17
18 if(!point->m_cluster || point_cluster->m_cluster_id != point->m_cluster->m_cluster_id) {
19     point->m_cluster = point_cluster;
20     *flag_finished = false;
21 }
```

The only difference between Hamerly's version of `find_cluster_average` and Elkan's version of that function is that Hamerly's version also finds the maximum value out of all distances between clusters' current and new centroids, i.e. the greatest value in `m_cluster_prev_dist` array.

3.2 OpenMP implementations

OpenMP implementations are the same as sequential implementations with the only difference: data parallelism with parallel `for` loop. Parallel `for` is added for the outermost loop of both functions:

- `find_closest_cluster`: the outmost loop that traverses through all points is parallelized as illustrated in Listing 3.8.

Code listing 3.8 Addition of parallel loop for find_closest_cluster function

```
#pragma omp parallel
{
    #pragma omp for schedule(static) reduction(+:flag_finished)
    for(int point_id = 0; point_id < m_num_points; ++point_id) {
        // find the closest cluster to this point
    }
}
```

One more important addition is a reduction of the flag that indicates if no point has changed the cluster, that this point is assigned to, in this iteration and the algorithm should converge. The reduction is done with an addition operator, so after `find_closest_cluster` was called, `flag_finished` should be compared with the number of points and if the equality holds true, the algorithm converges.

- `find_cluster_average` : the outmost loop that traverses through all clusters is parallelized as illustrated in Listing 3.9.

Code listing 3.9 Addition of parallel loop for find_cluster_average function

```
#pragma omp parallel
{
    #pragma omp for schedule(static)
    for(int cluster_id = 0; cluster_id < m_num_clusters; ++cluster_id) {
        // find the average of coordinates of all points in this cluster
    }
}
```

3.3 CUDA implementations

3.3.1 Implementation in global memory

3.3.1.1 Finding the closest cluster to each point

In order to find the closest cluster to each point, the `cuda_find_closest_cluster` is used with configuration:

- Block dimension of `dim3 blockDim(block_dim)` .
- Grid dimension of `dim3 gridDim(ceilf((float) num_points / block_dim))` .

Thus, each point is processed via one CUDA thread.

Listing 3.10 illustrates a `cuda_find_closest_cluster` templated kernel with template parameter `T` :

1. Lines 1–3: calculate the absolute position of the thread that will represent a point and check if it is out of bounds of total number of points.
2. Lines 5–6: if any heuristics were used, the upper bound of this point is fetched from the global array.
3. Lines 8–10: initialize local variables that will be used in this kernel.
4. Lines 12–16: if this point has already been assigned to any cluster, any heuristics were used, and bounds of this point shouldn't be changed, return, as there is no potential closer cluster to this point rather than the one that has already been assigned to it.
5. Lines 18–34: iterate over all clusters:

   a. Lines 19–21: calculate the distance from this point to the currently traversed cluster.
   b. Lines 23–26: if the Elkan's heuristic is used, assign the value of the calculated distance to the lower bound between this point and currently traversed cluster.
   c. Lines 28–33: update shortest and second shortest distances.

6. Lines 35–36: if any heuristics were used, assign the value of the shortest distance to the upper bound of this point.
7. Lines 37–38: if Hamerly's heuristic was used, assign the value of the second shortest distance to the lower bound of this point.
8. Lines 40–43: if this point hasn't been previously assigned to any cluster or the obtained closest cluster to this point is not the cluster that has already been assigned to this point, update the cluster that this point is assigned to and set the flag that indicates that this point hasn't changed its cluster to 0.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.10 Kernel for finding the closest cluster to each point in global memory

```
 1  int point_id = blockDim.x * blockIdx.x + threadIdx.x;
 2  if(point_id >= num_points)
 3      return;
 4
 5  if(heuristics_type == Elkan || heuristics_type == Hamerly)
 6      upper_bound = &(upper_bounds[point_id]);
 7
 8  T point_x = points_x[point_id], point_y = points_y[point_id];
 9  int point_cluster_id_val = point_cluster_id[point_id];
10  arr_flag_finished[point_id] = 1;
11
12  if(point_cluster_id_val > -1 && (heuristics_type == Elkan || heuristics_type == Hamerly) &&
13      !was_updated[point_id]
14  ) {
15      return;
16  }
17
18  for(int i = 0; i < num_clusters; ++i) {
19      current_distance = cuda_calculate_distance(
20          (T) point_x, (T) point_y, (T) clusters_x[i], (T) clusters_y[i]
21      );
22
23      if(heuristics_type == Elkan) {
24          lower_bound = &(arr_lower_bounds[point_id * num_clusters + i]);
25          *lower_bound = current_distance;
26      }
27
28      if(shortest_distance < 0 || current_distance < shortest_distance) {
29          second_shortest_distance = shortest_distance;
30          shortest_distance = current_distance;
31          id_closest = i;
32      } else if(second_shortest_distance < 0 || current_distance < second_shortest_distance)
33          second_shortest_distance = current_distance;
34  }
35  if(heuristics_type == Elkan || heuristics_type == Hamerly)
36      *upper_bound = shortest_distance;
37  if(heuristics_type == Hamerly)
38      arr_lower_bounds[point_id] = second_shortest_distance;
39
40  if(point_cluster_id_val == -1 || id_closest != point_cluster_id_val) {
41      point_cluster_id[point_id] = id_closest;
42      arr_flag_finished[point_id] = 0;
43  }
```

In order to use heuristics, upper and lower bounds should be calculated for each point in advance, i.e. before calling `cuda_find_closest_cluster` kernel.

For Elkan's heuristic, `cuda_update_bounds` kernel is used with the same configuration that kernel `cuda_find_closest_cluster` is called with. Listing 3.11 illustrates `cuda_update_bounds` templated kernel with template parameter `T` :

1. Lines 1–3: calculate the absolute position of the thread that will represent a point and check if it is out of bounds of total number of points.
2. Line 5: fetch the index of the cluster to which this point has already been assigned to.
3. Lines 6–7: if this point is not assigned to any cluster, then no bounds have been set before for this point and the point's bounds must be calculated.
4. Lines 8–27: if this point has already been assigned to any cluster:
   a. Line 10: set the value `was_updated` flag for this point to 0, as initially it is considered, that no bounds should be recalculated.
   b. Line 11: increment the upper bound of this point by the value of the distance that the cluster, that this point is assigned to, has moved.
   c. Lines 13–26: iterate over all clusters:
      i. Lines 14–16: if the cluster that is currently iterated is the same that this point has already been assigned to, continue, as there is no need to compare the upper and lower bounds to the same cluster.
      ii. Lines 17–20: update lower bound of this point to currently iterated cluster.
      iii. Lines 22–25: if updated upper bound is greater than updated lower bound to this cluster, then the bounds must be recalculated, thus value of `was_updated` for this point is set to 1, and there is no need to update and compare bounds for this point any further.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.11 Kernel for updating the bounds of each point for Elkan's heuristic in global memory

```
1  int point_id = blockDim.x * blockIdx.x + threadIdx.x;
2  if(point_id >= num_points)
3      return;
4
5  int point_cluster_id_val = point_cluster_id[point_id];
6  if(point_cluster_id_val == -1)
7      was_updated[point_id] = 1;
8  else {
9      T * lower_bound;
10     was_updated[point_id] = 0;
11     upper_bounds[point_id] += cluster_prev_dist[point_cluster_id_val];
12
13     for(int i = 0; i < num_clusters; ++i) {
14         if(i == point_cluster_id_val)
15             continue;
16
17         lower_bound = &(arr_lower_bounds[point_id * num_clusters + i]);
18         *lower_bound -= cluster_prev_dist[i];
19         if(*lower_bound < 0)
20             *lower_bound = 0;
21
22         if(upper_bounds[point_id] > *lower_bound) {
23             was_updated[point_id] = 1;
24             break;
25         }
26     }
27 }
```

For Hamerly's heuristic, the `cuda_update_bounds` kernel is very similar to Elkan's heuristic version of this kernel, with the only difference being in the body of the `else` statement of that kernel, as there is only one lower bound for each point. Listing 3.12 illustrates the body if the `else` statement of `cuda_update_bounds` kernel for Hamerly's heuristic that has the same configuration as `cuda_update_bounds` kernel for Elkan's heuristic:

1. Lines 1-2: are the same as they were for Elkan's heuristic, i.e. set the value `was_updated` flag for this point to 0 and increment the value of the upper bound.
2. Lines 4-10: decrement the value of the lower bound of this point by the value of the distance of the cluster the centroid of which has moved the most in one iteration of the algorithm.
3. Lines 12-13: if the value of the upper bound of this point is greater than the value of the lower bound for this point, set the `was_updated` flag for this point to 1, as the bounds must be recalculated.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.12 Kernel for updating the bounds of each point for Hamerly's heuristic in global memory

```
1  was_updated[point_id] = 0;
2  upper_bounds[point_id] += cluster_prev_dist[point_cluster_id_val];
3
4  T lower_bound = arr_lower_bounds[point_id];
5  lower_bound -= *max_cluster_prev_dist;
6
7  if(lower_bound < 0)
8      lower_bound = 0;
9
10 arr_lower_bounds[point_id] = lower_bound;
11
12 if(upper_bounds[point_id] > lower_bound)
13     was_updated[point_id] = 1;
```

3.3.1.2 Checking flags for algorithm convergence

After finding the closest cluster to each point, the array of flags, that indicates if points have changed their clusters or not, should be reduced to find out if no point has changed its cluster or not. The default value of the flag is 1 that indicates that a point hasn't changed its cluster, otherwise the flag is 0. So, the array should be reduced to find the minimum value and in case if the minimum value is 1, the algorithm should converge, as no point has changed its cluster from the previous iteration.

Parallel reduction is used in order to speed up the search in an array, as the complexity of simple linear search is $O(n)$, parallel reduction is used to speed up the search to $O(log(n))$, where n is the number of elements in the array. The array of flags is stored in `arr_flag_finished` .

Figure 5 illustrates parallel reduction of `arr_flag_finished` with 7 elements: there are 3 iterations in total, as each iteration requires only half as many threads as there are elements. As total number of elements is 7, $\lceil O(log(7)) \rceil = 3$ iterations are needed to reduce this array. Each iteration:

- In the case of odd number of elements:
  - $\dfrac{num\_elements}{2} + 1$ threads are needed.
  - For the next iteration, the number of elements will be $\dfrac{num\_elements + 1}{2}$.
- In the case of even number of elements:

- $\frac{num\_elements}{2}$ threads are needed.
- For the next iteration, the number of elements will be $\frac{num\_elements}{2}$.

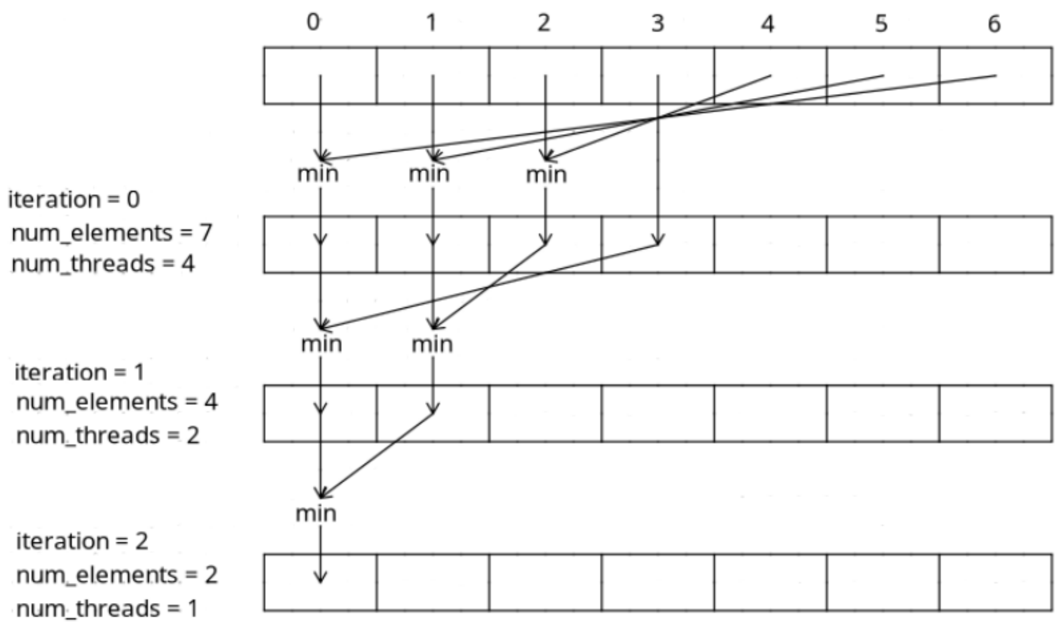

Figure 5 Parallel reduction to find minimum

Listing 3.13 illustrates the host code of parallel reduction:

1. Line 2: call the kernel that should be reduced in global memory.
2. Lines 4–6: calculate the number of elements for the next iteration of parallel reduction. It is calculated as it was explained for Figure 5.

Code listing 3.13 Parallel reduction to find minimum in global memory(host code)

```
1 while(num_elements > 1) {
2     // call the kernel that should be reduced
3 
4     if(num_elements % 2 == 1)
5         ++num_elements;
6     num_elements /= 2;
7 }
```

The kernel `cuda_find_min_reduction` is used to find the minimum value in a given array. The configuration of the kernel is

- Block dimension of `dim3 blockDim(block_dim)`.
- Grid dimension depends on the number of elements:
  - If the number of elements is odd, then `dim3 gridDim(ceilf((float) (num_elements / 2 + 1) / block_dim))`.
  - If the number of elements is even, then `dim3 gridDim(ceilf((float) (num_elements / 2) / block_dim))`.

Listing 3.14 illustrates `cuda_find_min_reduction` templated kernel with template parameter `T` :

1. Line 1: calculate the absolute position of thread.
2. Lines 2–3: If thread's position is greater than the half of the number of elements in the array, that should be reduced, or is equal to it, no need to proceed any further. The element in the middle is not reduced either as it doesn't have a mirror element with which it will be compared.
3. Lines 5–13: calculate the position of mirror entry. The mirror entry's value is compared with value of the current thread and a minimum is written into `min` variable.
4. Lines 15–16: if the number of elements is 2, i.e. it is the last iteration of parallel reduction and there is only one thread running, the obtained minimum is written into the `flag_finished` .

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.14 Parallel reduction to find minimum in global memory(device code)

```
 1  int thread_pos = blockDim.x * blockIdx.x + threadIdx.x;
 2  if(thread_pos >= num_elements / 2)
 3      return;
 4
 5  int thread_mirror_pos = num_elements - thread_pos - 1;
 6  int min;
 7
 8  if(arr_flag_finished[thread_pos] < arr_flag_finished[thread_mirror_pos])
 9      min = arr_flag_finished[thread_pos];
10  else
11      min = arr_flag_finished[thread_mirror_pos];
12
13  arr_flag_finished[thread_pos] = min;
14
15  if(num_elements == 2)
16      *flag_finished = min;
```

After obtaining the minimum, the value of `flag_finished` is copied to the host to check if the algorithm should converge or not: in case if the value is 1, i.e. no point has changed its respective cluster, the algorithm converges, otherwise, the algorithm continues.

3.3.1.3 Finding the average of each cluster

To find a new centroid for each cluster, the average coordinates of all points in each cluster is be calculated. Average can be found as a sum of all points in that cluster divided by the number of those points. To efficiently find the sum of all points, parallel reduction is used for each cluster.

To find an average of all points in a cluster via parallel reduction, `cuda_find_average_reduction` kernel is used. Kernel is called in the same reduction loop, as `cuda_find_min_reduction` is called, as illustrated on Listing 3.13. The only difference, though, is a grid dimension: as parallel reduction is called for each kernel, grid will also have a Y dimension, each entry of which will represent one cluster.

Listing 3.15 illustrates `cuda_find_average_reduction` templated kernel with template parameter `T` :

1. Lines 1-2: calculate the absolute position of thread in X axis and the index of the cluster that is represented as a Y axis.
2. Lines 4–5: if the number absolute position of thread is greater than or equal to the number of elements in current iteration of parallel reduction, don't go any further, as the number of threads must be equal to the half of the number of elements.

3. Line 7: calculate the position of mirror entry.
4. Lines 8–13: increment the X and Y values by the mirror entry values of the corresponding arrays as those arrays represent the sum of the coordinates of all points in that cluster.
5. Lines 15–30: if there are only 2 elements left to reduce, i.e. this is the last iteration of parallel reduction, and the number of points in the cluster is greater than 0:
   a. Lines 16–19: calculate the average of coordinates of that cluster via dividing the obtained sum of all points with the number of points in that cluster.
   b. Lines 21–26: if any type of heuristic is used, the distance between previous and current centroids of this cluster is calculated.
   c. Lines 28–29: coordinates of this cluster are written to the corresponding arrays.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.15 Parallel reduction to find average in global memory

```
1  int thread_pos = blockDim.x * blockIdx.x + threadIdx.x;
2  int cluster_id = blockIdx.y;
3
4  if(thread_pos >= num_elements / 2)
5      return;
6
7  int thread_mirror_pos = num_elements - thread_pos - 1;
8  array_average_x[cluster_id * num_points + thread_pos] +=
9      array_average_x[cluster_id * num_points + thread_mirror_pos];
10 array_average_x[cluster_id * num_points + thread_pos] +=
11     array_average_x[cluster_id * num_points + thread_mirror_pos];
12 array_num_points_in_cluster[cluster_id * num_points + thread_pos] +=
13     array_num_points_in_cluster[cluster_id * num_points + thread_mirror_pos];
14
15 if(num_elements == 2 && array_num_points_in_cluster[cluster_id * num_points + thread_pos] > 0) {
16     T updated_cluster_x = array_x[cluster_id * num_points + thread_pos] /
17         array_num_points_in_cluster[cluster_id * num_points + thread_pos];
18     T updated_cluster_y = array_y[cluster_id * num_points + thread_pos] /
19         array_num_points_in_cluster[cluster_id * num_points + thread_pos];
20
21     if(heuristics_type == Elkan || heuristics_type == Hamerly) {
22         cluster_prev_dist[cluster_id] = cuda_calculate_distance_average(
23             (T) clusters_x[cluster_id], (T) clusters_y[cluster_id],
24             (T) updated_cluster_x, (T) updated_cluster_y
25         );
26     }
27
28     clusters_x[cluster_id] = updated_cluster_x;
29     clusters_y[cluster_id] = updated_cluster_y;
30 }
```

3.3.2 Implementation in shared memory

3.3.2.1 Finding the closest cluster to each point

The main idea behind algorithms in shared memory lies in chunking. When finding the closest cluster for each point, every point calculates a distance to every cluster (unless Elkan's or Hamerly's heuristic is used, which could prevent calculation of some distances), so the cluster coordinates that are fetched from the global memory could be reused for multiple points.

As an example, assume that 256 threads are used in a block and each thread represents a point. Each point could fetch the coordinates of one cluster and write those coordinates into shared memory that will be reused by all threads in that block, thus threads could calculate distances between a point that they represent and each cluster in the shared memory.

The only caveat of calculations in chunks is that parallel reduction should be used to accumulate results from different chunks. In the case of finding the closest cluster to each point, the closest cluster will be found in each chunk, thus the results from different chunks should be reduced to find the cluster with the shortest distance to the given point.

So, the shared memory version of finding the closest cluster algorithm consists of two parts:

- Find the closest cluster to each point in each chunk of clusters.
- Reduce the results from all chunks of clusters to obtain the closest cluster to each point.

To find the closest cluster to each point in chunks, `cuda_find_closest_cluster_chunks` kernel is used with configuration:

- Block dimension of `dim3 blockDim(block_dim)` .
- Grid dimension of `dim3 gridDim(ceilf((float) num_points / block_dim), num_clusters_chunks)` .
- Shared memory size of `block_dim * sizeof(CudaCoordinate<T>)` .

Listing 3.16 illustrates the part of `cuda_find_closest_cluster_chunks` kernel that is different from `cuda_find_closest_cluster` :

1. Line 1: get the index of this chunk of clusters as an entry of this block in a Y dimension of the grid.
2. Line 2: calculate the offset of clusters in this chunk, i.e. an index of the first cluster in this chunk of clusters.
3. Lines 4–11: initialize the shared memory and write the cluster coordinates in this chunk from global array to shared memory.
4. Lines 13–14: if the index of this point is greater than or equal to the total number of points, don't continue any further. The checking for the out of bounds of this thread is done only at this stage and not earlier, as it could happen that the number of points in this block is less than the number of clusters in this chunk (could happen in the case if it is the last block in this row of the grid and not all threads in this block would be used), thus every thread, even if it wouldn't be used, should first fetch the cluster's coordinates from the global array, so that other threads could use this cluster's coordinates to calculate distances.
5. Lines 16–20: if any type of heuristic is used and bounds of this point shouldn't be recalculated:
   a. Lines 17–18: If this block belongs to the first row of the grid, write the flag that indicates that this point hasn't changed its cluster to `arr_flag_finished` . This checking is done so that the same point in different cluster chunks, i.e. in different grid rows, doesn't try to write to the same memory location to create a race condition.
   b. Line 19: return, as there is no need to calculate distances.
6. Lines 22–24: iterate over all clusters in this chunk and find the cluster with shortest distance to this point and cluster with second shortest distance to this point.
7. Lines 26–29: write the obtained results into the temporary array `tmp_points_clusters` . This array will be further reduced to obtain the closest cluster to this point and the second closest cluster to this point.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.16 Kernel for finding the closest cluster to each point in shared memory

```
int clusters_chunk_id = blockIdx.y;
int clusters_offset = clusters_chunk_size * clusters_chunk_id;

extern __shared__ uint8_t shared_array_clusters_1[];
CudaCoordinate<T> * shared_array_clusters = (CudaCoordinate<T> *) shared_array_clusters_1;

if(threadIdx.x < clusters_chunk_size && threadIdx.x + clusters_offset < num_clusters) {
    shared_array_clusters[threadIdx.x].m_x = clusters_x[clusters_offset + threadIdx.x];
    shared_array_clusters[threadIdx.x].m_y = clusters_y[clusters_offset + threadIdx.x];
}
__syncthreads();

if(point_id >= num_points)
    return;

if((heuristics_type == Elkan || heuristics_type == Hamerly) && !was_updated[point_id]) {
    if(!blockIdx.y)
        arr_flag_finished[point_id] = 1;
    return;
}

for(int i = 0; i < clusters_chunk_size && i + clusters_offset < num_clusters; ++i) {
    // calculate the shortest and second shortest distances
}

int tmp_arr_id = point_id * num_clusters_chunks + clusters_chunk_id;
tmp_points_clusters[tmp_arr_id].m_cluster_id = id_closest;
tmp_points_clusters[tmp_arr_id].m_cluster_distance = shortest_distance;
tmp_points_clusters[tmp_arr_id].m_second_cluster_distance = second_shortest_distance;
```

Figure 6 illustrates a parallel reduction of an array with 11 elements in shared memory:

- At the beginning of each iteration of the reduction, all threads in a given block fetch data from array in global memory and write it to shared memory, as shared memory represents a portion of an array that will be reduced by threads in this block.
- Size of each portion is equal to the double of number of threads in the block, as each thread reduces 2 entries in an array. Thus the number of blocks is $\frac{num\_elements}{block\_dim \cdot 2}$.
- In each iteration, the first entry in the array that will be processed by this block and the last entry in the array that will be processed by this block is calculated, i.e. `block_offset` and `block_last_pos` correspondingly.
- After the threads in each block have reduced their corresponding portion of the array, the results are written back to the array to the first entry in this block, i.e. `block_offset` . This is done in this fashion, so that blocks don't write to the same region and no racing condition occurs.
- In order to prepare this array for the next iteration of parallel reduction, the results should be shifted, so that the elements, that will be reduced in the future, are consecutively located in the array. This is done via shifting the values from each block's first entry to the temporary array and further copying the values from temporary array to the original array. Shifting occurs at the end of each iteration of parallel reduction.
- After the shifting occured, the parallel reduction may continue with its next iteration.

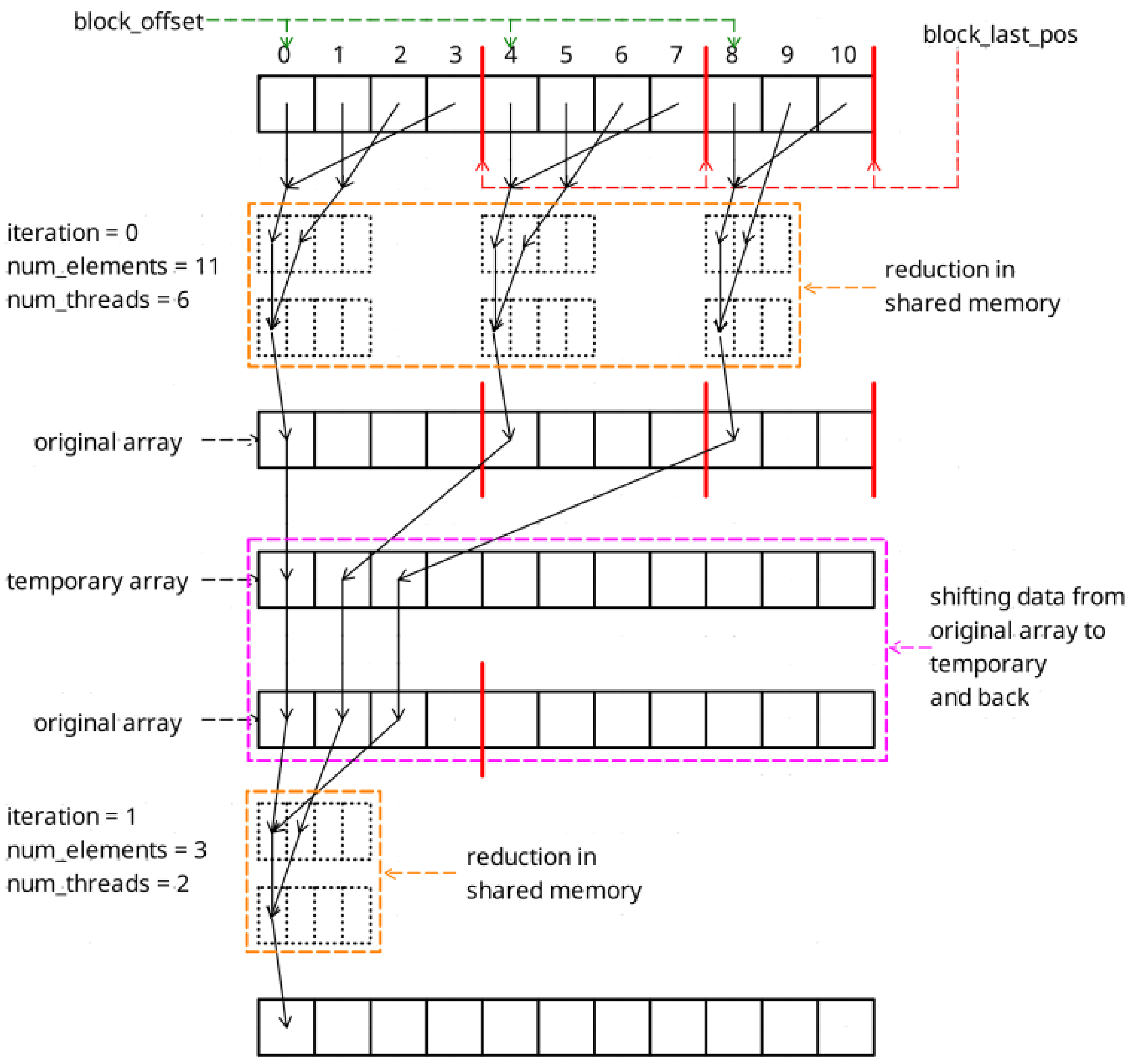

Figure 6 Reduction in shared memory

The benefit of reduction in shared memory lies in decreasing number of accesses to global memory via decreasing the number of parallel reduction iterations and making most of the reduction in the kernel itself using shared memory. The number of iterations of parallel reduction in shared memory implementation is equal to

$$\lceil \log_{block\_dim \cdot 2} num\_elements \rceil$$

where the base of logarithm is equal to the double of number of threads in a block.

Listing 3.17 illustrates the host code of parallel reduction in shared memory:

1. Line 2: call the kernel that should be reduced in shared memory.
2. Lines 4–8: calculate the number of elements for the next iteration of parallel reduction.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.17 Parallel reduction to find the closest cluster in shared memory(host code)

```
while(num_elements > 1) {
    // call kernels that should be reduced in shared memory

    if(num_elements % (block_dim * 2)) {
        num_elements /= (block_dim * 2);
        ++num_elements;
    } else
        num_elements /= (block_dim * 2);
}
```

Kernel `cuda_find_closest_cluster_reduction_shared` is used to reduce the obtained closest clusters to each point from each chunk. Each point is processed via one kernel grid row. The kernel is called from the parallel reduction loop, illustrated on Listing 3.17 with configuration

- Block dimension of `dim3 blockDim(block_dim)` .
- Grid dimension of `dim3 gridDim(ceilf((float) num_elements / (block_dim * 2)), num_points)` .
- Share memory size of `2 * block_dim * sizeof(CudaTmpPointCluster<T>)` .

The kernel consists of 3 parts:

1. Initialization of shared memory.
2. Blockwise parallel reduction in shared memory.
3. Writing back the results to the global memory.

Listing 3.18 illustrates the first part of the kernel:

1. Line 1: calculate the index of this point by the index of the block in the Y dimension of the grid, as each row of blocks of the grid processes one point.
2. Lines 2–3: if any heuristic is used and the bounds of the point shouldn't be updated, return, as no distance calculations are needed.
3. Lines 5–7: calculate the offset of this block, i.e. the entry in the array that this block of threads will start reduction with, absolute position of this thread in the array, and the last entry in the array that this block of threads will reduce.
4. Line 12: calculate the number of elements that this block of threads will reduce.
5. Lines 13–18: if the index of this thread in the block exceeds the half of the number of elements that should be reduced or there is only one thread in this block with only one element (so, no reduction is needed), return.
6. Lines 20–23: calculate the index of the mirror entry that will be reduced with this thread's entry. The index of the mirror entry is calculated both in the array that is in the global memory and in the shared memory.
7. Lines 25–43: initialize shared memory and copy the portion of the array in the global memory, that will be reduced by threads in this block, to shared memory.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.18 Initialization of shared memory in kernel for reducing the obtained closest clusters to each point in each chunk

```
int point_id = blockIdx.y;
if((heuristics_type == Elkan || heuristics_type == Hamerly) && !was_updated[point_id])
    return;

int block_offset = blockDim.x * blockIdx.x * 2;
int thread_pos = block_offset + threadIdx.x;
int block_last_pos = block_offset + blockDim.x * 2 - 1;

if(block_last_pos >= num_elements)
    block_last_pos = num_elements - 1;

int num_block_elements = block_last_pos - block_offset + 1;
if((num_block_elements % 2 == 0 && threadIdx.x >= num_block_elements / 2) ||
    (num_block_elements % 2 == 1 && threadIdx.x > num_block_elements / 2) ||
    (threadIdx.x == 0 && thread_pos == block_last_pos)
) {
    return;
}

int thread_mirror_pos = block_last_pos - threadIdx.x;
int shared_thread_mirror_pos = block_last_pos - block_offset - threadIdx.x;
int tmp_arr_id = point_id * num_clusters_chunks + thread_pos;
int tmp_arr_mirror_id = point_id * num_clusters_chunks + thread_mirror_pos;

extern __shared__ uint8_t shared_array;
CudaTmpPointCluster<T> * tmp_shared_arr = (CudaTmpPointCluster<T> *) shared_array;

tmp_shared_arr[threadIdx.x].m_cluster_distance =
    tmp_points_clusters[tmp_arr_id].m_cluster_distance;
tmp_shared_arr[threadIdx.x].m_second_cluster_distance =
    tmp_points_clusters[tmp_arr_id].m_second_cluster_distance;
tmp_shared_arr[threadIdx.x].m_cluster_id =
    tmp_points_clusters[tmp_arr_id].m_cluster_id;

if(threadIdx.x != shared_thread_mirror_pos) {
    tmp_shared_arr[shared_thread_mirror_pos].m_cluster_distance =
        tmp_points_clusters[tmp_arr_mirror_id].m_cluster_distance;
    tmp_shared_arr[shared_thread_mirror_pos].m_second_cluster_distance =
        tmp_points_clusters[tmp_arr_mirror_id].m_second_cluster_distance;
    tmp_shared_arr[shared_thread_mirror_pos].m_cluster_id =
        tmp_points_clusters[tmp_arr_mirror_id].m_cluster_id;
}
__syncthreads();
```

After the portion of the array, that should be reduced by this block of threads, was copied from global array to shared array, that portion should now be reduced. Listing 3.19 illustrates the actual reduction inside the `cuda_find_closest_cluster_reduction_shared` kernel:

1. Lines 3–7: if the index of this thread in the block exceeds the half of the number of elements that is left to be reduced, return, as each thread reduces 2 entries in an array.
2. Line 9: calculate the position of the mirror entry in the shared memory.
3. Lines 10–34: if the index of this thread in the block is not equal to the mirror entry's index in the shared memory (this checking is done, so that no reduction occurs between elements in the same entry):
    a. Lines 11–28: if the distance to the cluster in the mirror entry is less than the distance to the cluster in this entry, update the current entry's closest cluster, i.e. assign the cluster of the mirror entry to the current entry, as illustrated in lines 24–27.
    One more important step is to update the second closest cluster, which comprises of 2 different scenarios:

- Lines 14–19: in this scenario, the mirror entry's distance to the second closest cluster is less than this entry's distance to the closest cluster, the mirror entry's second closest distance is assigned to this entry's second closest distance. Table below is given as an example of such a scenario.

	this entry	mirror entry
distance to the closest cluster	10	6
distance to the second closest cluster	12	8

- Lines 19–22: in this scenario, this entry's distance to the closest cluster is less than a distance to the mirror entry's second closest cluster, thus the value of this entry's closest cluster is assigned to this entry's second closest cluster. Table below is given as an example of such a scenario.

	this entry	mirror entry
distance to the closest cluster	10	6
distance to the second closest cluster	12	15

b. Lines 28–33: if this entry's distance to the closest cluster is less than or equal to mirror entry's distance to the closest cluster and the mirror entry's distance to the closest cluster is less than this entry's distance to the second closest cluster, there is no need to update this entry's closest cluster, but the mirror entry's distance to the closest cluster should be assigned to this entry's second closest cluster. Table below is given as an example of such a scenario.

	this entry	mirror entry
distance to the closest cluster	10	16
distance to the second closest cluster	18	20

4. Lines 35–42: if there were only 2 elements left to reduce, thus only one thread to reduce them, the result is written back from shared memory to global memory.

5. Lines 43–45: the number of elements is updated for the next iteration of reduction inside the block.

6. Line 47: threads are synchronized before the next iteration of reduction inside the block.

After the array was reduced inside the block, the results are written back to the array in global memory. Each block writes the obtained result into the first of the array that was processed by that block, i.e. `block_offset` from Figure 6. Listing 3.20 illustrates the third part of the kernel:

1. Lines 1–11: if there is only one block in this row of the grid and there is only one thread operating in this block, i.e. this is the last iteration of parallel reduction:

   a. Lines 2–3: if any heuristic is used, assign the distance to the closest cluster to the upper bound of this point.

   b. Lines 4–5: if Hamerly's heuristic is used, assign the distance to the second closest cluster to the lower bound of this point.

   c. Lines 6–10: if an index of obtained closest cluster is not equal to the index of the cluster that this point is currently assigned to, assign this point to the closest cluster and assign the flag that indicates that this point hasn't changed its cluster to 0. Otherwise, set the flag to 1, as point hasn't changed its cluster.

For all definitions of variables, arrays, and structures in the listing, see Appendix A.

Code listing 3.19 Blockwise parallel reduction in kernel for reducing the obtained closest clusters to each point in each chunk

```
int tmp_num_elements = num_block_elements;
while(tmp_num_elements > 1) {
    if((tmp_num_elements % 2 == 0 && threadIdx.x >= tmp_num_elements / 2) ||
        (tmp_num_elements % 2 == 1 && threadIdx.x > tmp_num_elements / 2)
    ) {
        return;
    }

    shared_thread_mirror_pos = tmp_num_elements - threadIdx.x - 1;
    if(threadIdx.x != shared_thread_mirror_pos) {
        if(tmp_shared_arr[shared_thread_mirror_pos].m_cluster_distance <
            tmp_shared_arr[threadIdx.x].m_cluster_distance
        ) {
            if(tmp_shared_arr[shared_thread_mirror_pos].m_second_cluster_distance <
                tmp_shared_arr[threadIdx.x].m_cluster_distance
            ) {
                tmp_shared_arr[threadIdx.x].m_second_cluster_distance =
                    tmp_shared_arr[shared_thread_mirror_pos].m_second_cluster_distance;
            } else {
                tmp_shared_arr[threadIdx.x].m_second_cluster_distance =
                    tmp_shared_arr[threadIdx.x].m_cluster_distance;
            }

            tmp_shared_arr[threadIdx.x].m_cluster_distance =
                tmp_shared_arr[shared_thread_mirror_pos].m_cluster_distance;
            tmp_shared_arr[threadIdx.x].m_cluster_id =
                tmp_shared_arr[shared_thread_mirror_pos].m_cluster_id;
        } else if(tmp_shared_arr[shared_thread_mirror_pos].m_cluster_distance <
                tmp_shared_arr[threadIdx.x].m_second_cluster_distance
        ) {
            tmp_shared_arr[threadIdx.x].m_second_cluster_distance =
                tmp_shared_arr[shared_thread_mirror_pos].m_cluster_distance;
        }
    }
    if(tmp_num_elements == 2 && threadIdx.x == 0) {
        tmp_points_clusters[tmp_arr_id].m_cluster_distance =
            tmp_shared_arr[threadIdx.x].m_cluster_distance;
        tmp_points_clusters[tmp_arr_id].m_second_cluster_distance =
            tmp_shared_arr[threadIdx.x].m_second_cluster_distance;
        tmp_points_clusters[tmp_arr_id].m_cluster_id =
            tmp_shared_arr[threadIdx.x].m_cluster_id;
    }
    if(tmp_num_elements % 2 == 1)
        ++tmp_num_elements;
    tmp_num_elements /= 2;

    __syncthreads();
}
```

Code listing 3.20 Writing back the results to the global memory in kernel for reducing the obtained closest clusters to each point in each chunk

```
if(gridDim.x == 1 && threadIdx.x == 0) {
    if(heuristics_type == Elkan || heuristics_type == Hamerly)
        upper_bounds[point_id] = tmp_shared_arr[0].m_cluster_distance;
    if(heuristics_type == Hamerly)
        arr_lower_bounds[point_id] = tmp_shared_arr[0].m_second_cluster_distance;
    if(point_cluster_id[point_id] != tmp_shared_arr[0].m_cluster_id) {
        point_cluster_id[point_id] = tmp_shared_arr[0].m_cluster_id;
        arr_flag_finished[point_id] = 0;
    } else
        arr_flag_finished[point_id] = 1;
}
```

As the results from individual blocks are located at the beginning entry of each block, those results must be shifted to the beginning of the array, such that the next iteration of parallel reduction gets the array with data in consecutive entries. The shift is done via two kernels:

- `cuda_shift_results_to_tmp_closest_cluster` : shifts entries from the beginning of each block to the temporary array in consecutive order.
- `cuda_copy_shifted_results_from_tmp_closest_cluster` : copies entries from temporary array back to original array.

The shift in details is illustrated on Figure 6. The listings of these kernels can be found in Appendix A.3.

The heuristics kernels can also be implemented in shared memory via chunking:

- For Elkan's heuristic:
  1. The upper bound is updated, in the same fashion as it was updated for global memory implementation. It should be done in a separate kernel before updating lower bounds.
  2. The lower bounds of each point are calculated in chunks, i.e. between a point and clusters in a given chunk. The lower bounds in this chunk are compared with the previously computed upper bound for this point. In case if upper bound is greater than any lower bound in this chunk, 1 is written to `was_updated` array for this particular point and this particular chunk.
  3. To obtain accumulated results, `was_updated` array is reduced in shared memory to obtain the maximum value in the array. In case if the final value is 0, there is no need to recalculate bounds, as for each chunk of clusters, the corresponding entry's value in `was_updated` array is 0, thus for every chunk of clusters, the cluster, to which this point is currently is assigned to, is the closest cluster to that point.
- For Hamerly's heuristic everything is much easier, as there is only one lower bound per point and shared memory is used to store variable `max_cluster_prev_dist` .

3.3.2.2 The remaining kernels

Kernel to find minimum value in an array of flags that indicates if points have changed their cluster or not is modified to use reduction in shared memory. The implementation of the kernel is quite similar to the implementation of `cuda_find_closest_cluster_reduction_shared` with the difference that instead of the closest and second closest cluster, the minimum value is found.

Kernel to find the average coordinates is also modified to support calculations in shared memory in chunks. The chunks are the chunks of points that are copied from global memory to shared memory, so that this chunks of points could be used by other clusters in this block of threads. The finding of average in shared memory consists of two parts:

- Finding the sum of points in a given chunk that are in a given cluster.
- Reduction of the sums from all chunks of points to obtain a new centroid coordinates for a given cluster.

The kernel to find the sum in chunks is very similar to `cuda_find_closest_cluster_chunks` kernel.

3.4 Sklearn implementation

Python's Sklearn library contains implementation of K-means clustering [13]. The implementation is `sklearn.cluster.KMeans` with parameters:

- Number of clusters.
- Initialization method of clusters: could be random or K-means++. The clusters could be generated in advance and sent as this parameter to KMeans constructor.
- Number of initializations: used for random initialization of centroids to denote the number of times centroids will be initialized with different seeds to obtain the best centroid initialization.
- Number of maximum iterations: the maximum number of times an algorithm should run.
- Tolerance level, i.e. floating point precision.
- Random state: used for random number generations for random generation of cluster centroids.
- Algorithm type: standard Lloyd's algorithm or an algorithm with Elkan's triangle inequality.

Listing 3.21 illustrates configuration of KMeans used, where

- k_value is a number of clusters.
- np_arr_clusters is an array with initial centroid coordinates.
- algorithm_type is a type of algorithm that is used: Lloyd's or Elkan's.

Code listing 3.21 Configuration of Sklearn implementation

```
KMeans(n_clusters = k_value, max_iter = 100000, random_state = 0, init = np_arr_clusters,
    tol = 0, n_init = 1, copy_x = True, algorithm = algorithm_type)
```

In order to cluster the points, fit method is used on the KMeans object illustrated on Listing 3.21. Two dimensional list is sent as a parameter of fit method as point coordinates.

Chapter 4

Testing and evaluation of results

4.1 Solvers

All the solvers decribed in previous chapter will be included in testing, i.e. sequential solvers, OpenMP solvers, CUDA solvers in global and shared memory, Sklearn solver. All solvers except for Sklearn solver is tested has 3 versions:

- Basic, which is a basic Lloyd's algorithm.
- Elkan, which is Elkan's triangle heuristic.
- Hamerly, which is Hamerly's improvement of Elkan's triangle heuristic.

The results of the solvers, i.e. clusters to which the points are assigned to, are compared to the results of sequential basic solver. To make sure that sequential basic solver gives produces correct results, the results of it are compared to the Sklearn Python library implementation of K-means algorithm.

OpenMP solvers are tested in 3 different configurations (number of threads in parallel region): 4, 8, 12. CUDA solvers are tested in global and shared memory in 3 different block dimensions (number of threads per block): 64, 256, 1024.

The centroid initialization method of choice is K-means++ for all solvers. Random initialization will be used only for some tests only to compare the produced Silhouette of clusters [9] with random centroids initialization and K-means++ centroids initialization.

4.2 Types of tests

Tests will be done with single and double precision values. This is done as some systems, especially GPU, has less double precision modules than single precision modules, it is clearly illustrated on Figure 4, thus the time of execution is different for single and double precision values.

The precision of calculations and precision with which results of the solvers will be compared to sequential basic solver varies:

- 10^{-4} for single precision values.
- 10^{-12} for double precision values.

These precisions are chosen as precisions less than assigned could result to imprecisions, thus incorrect results for some calculations. To prevent the imprecision and to get correct results, the precisions were chosen accordingly.

4.3 Testing environment

The testing environment is

- GPU: NVIDIA GeForce RTX 2080 Ti.
- CPU: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz.
- CUDA Version: 11.4.
- Compiler command with options: nvcc -std=c++11 -Xcompiler -fopenmp.

4.4 Testing sample and time measurement

This sample was used for testing [14]. Values in the sample file are parsed consecutively as X and Y coordinates of the points.

For sequential, OpenMP, and CUDA solvers, time is measured via C++ `std::chrono` library, simply by taking the time before calculations (after all host and device memory allocations and memory copying from host to device) and time after the algorithm has converged (before all memory copying from device to host and device and host memory freeing).

For Sklearn solver, time is measured by `time()` function in `time` library, that returns the time in seconds since epoch.

4.5 Testing with double precision values

Figure 7 illustrates a time of executon of sequential and OpenMP solvers:

- Sequential solvers were slower than Sklearn implementation. sequential solver with Hamerly's heuristic is the fastest among sequential solvers with Elkan's heuristic being the second fastest among sequential solvers.
- OpenMP solvers behave similar to sequential solvers, i.e. Hamerly's heuristic is the fastest and Elkan's heuristic is the second fastest among OpenMP solvers right after the Hamerly's heuristic. All OpenMP solvers are faster than basic sequential solver, but only with 8 and 12 number threads, OpenMP solvers get faster than Sklearn implementation.

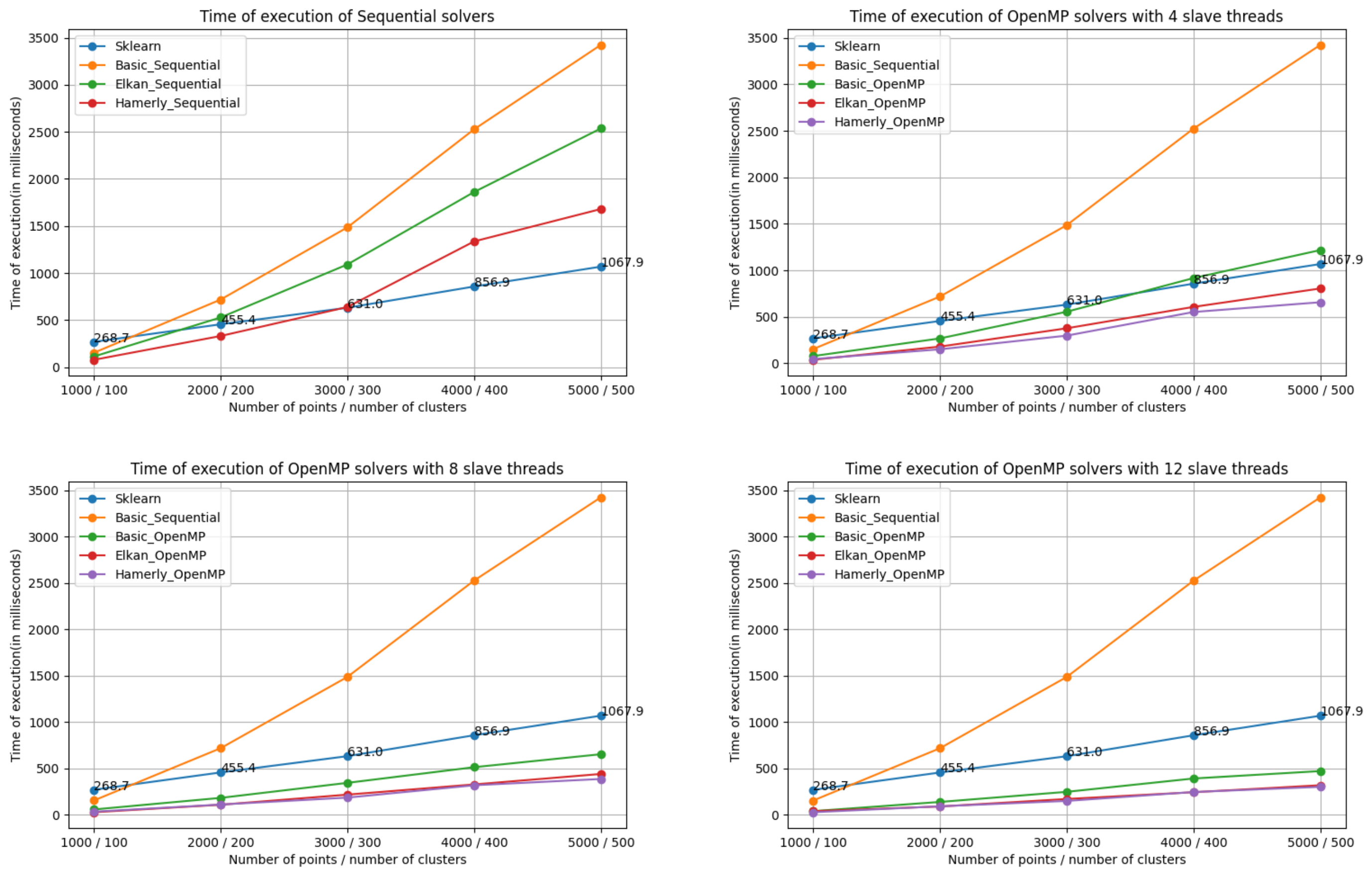

Figure 7 Time of execution of sequential and OpenMP solvers with double precision values

Figure 8 illustrates a time of executon of CUDA solvers in global and shared memory:

- The block dimension of 64 threads is the best block dimension for all solvers in both global and shared memory, as solvers with 64 threads in block are the fastest, which can be explained with high SM occupancy, as there are more blocks that are executed in parallel with lower number of threads per block.
- Solvers in shared memory are faster than solvers in the global memory with 64 and 256 threads per block.
- In most of the cases, Hamerly's heuristic is the fastest with Elkan's heuristic competing in the second place. Only in configuration with 1024 threads, Elkan's heuristic is faster than Hamerly's heuristic.
- All CUDA solvers in configurations with 64 and 256 threads were faster than Sklearn solver from Figure 7 and, as SKlearn is faster than sequential and OpenMP solvers, transitively, CUDA solvers are faster than sequential and OpenMP solvers too.

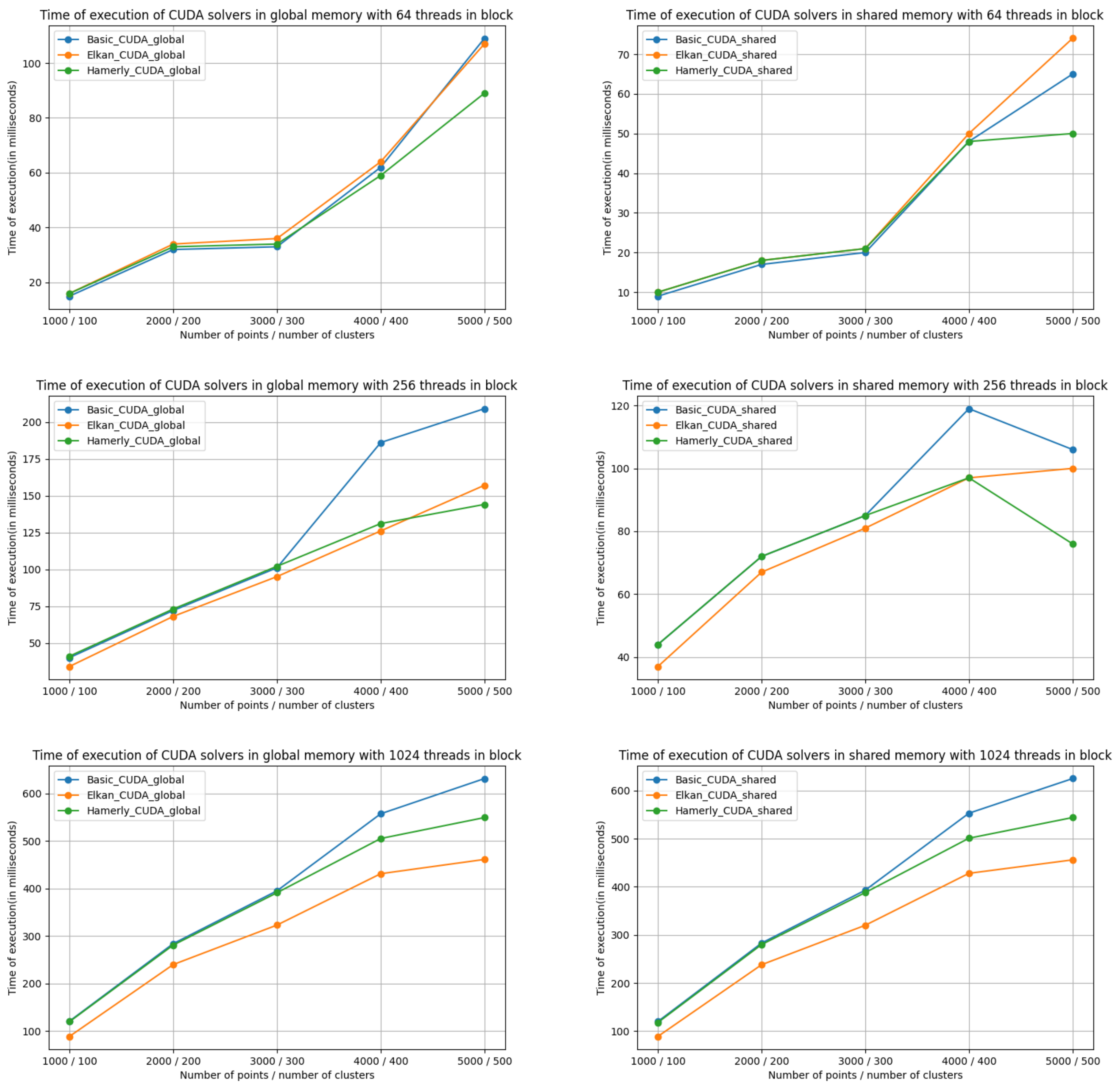

Figure 8 Time of execution of CUDA solvers in global memory with double precision values

4.6 Testing with single precision values

Figure 9 illustrates a time of executon of sequential and OpenMP solvers:

- The situation is quite similar to the one with double precision, i.e. the sequential solvers are slower than Sklearn and Hamerly's heuristic solver is faster than Elkan's heuristic solver.

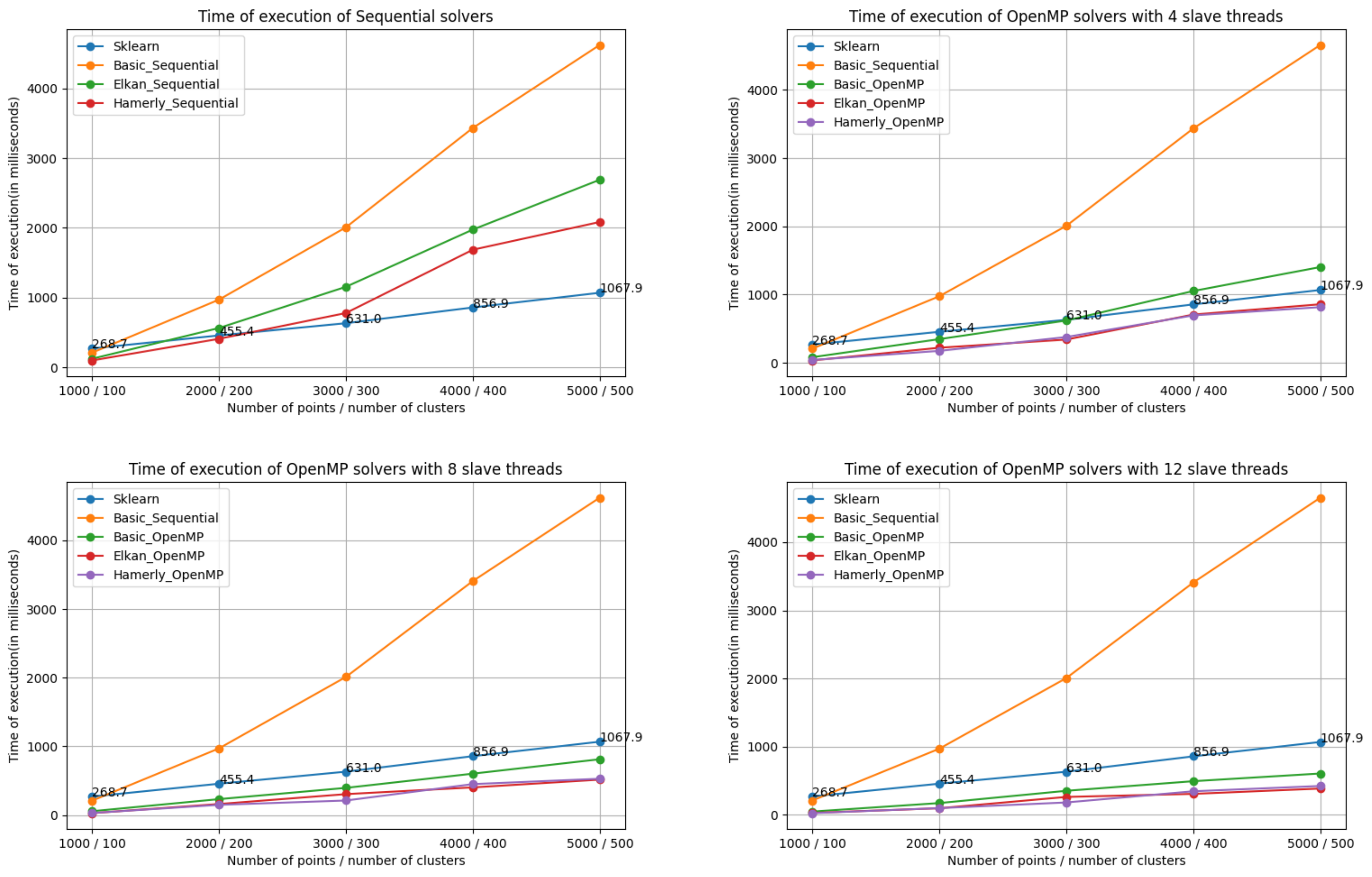

Figure 9 Time of execution of sequential and OpenMP solvers with single precision values

Figure 10 illustrates a time of execution of CUDA solvers in global and shared memory:

- The configuration with 64 threads in block is still the best configuration, with only exception of Elkan's heuristic solver in shared memory, which is faster with configuration of 256 threads in block, rather than 64 threads.
- The main difference with double precision values is that the basic solver in both global and shared memory is faster than Hamerly's and Elkan's solvers. It can be explained by the speed of calculations with single precision. Calculations with double precision are slow, as there aren't as many double precision units in SM, as single precision units for calculations, thus prevention of those calculations lead to significant performance boost. But with single precision values, calculations are very fast and accessing the global memory for calculating upper and lower bounds in Hamerly's and Elkan's heuristics becomes a hurdle, which impacts performance in a much more negative way, rather than prevention of already fast calculations with single precision values.

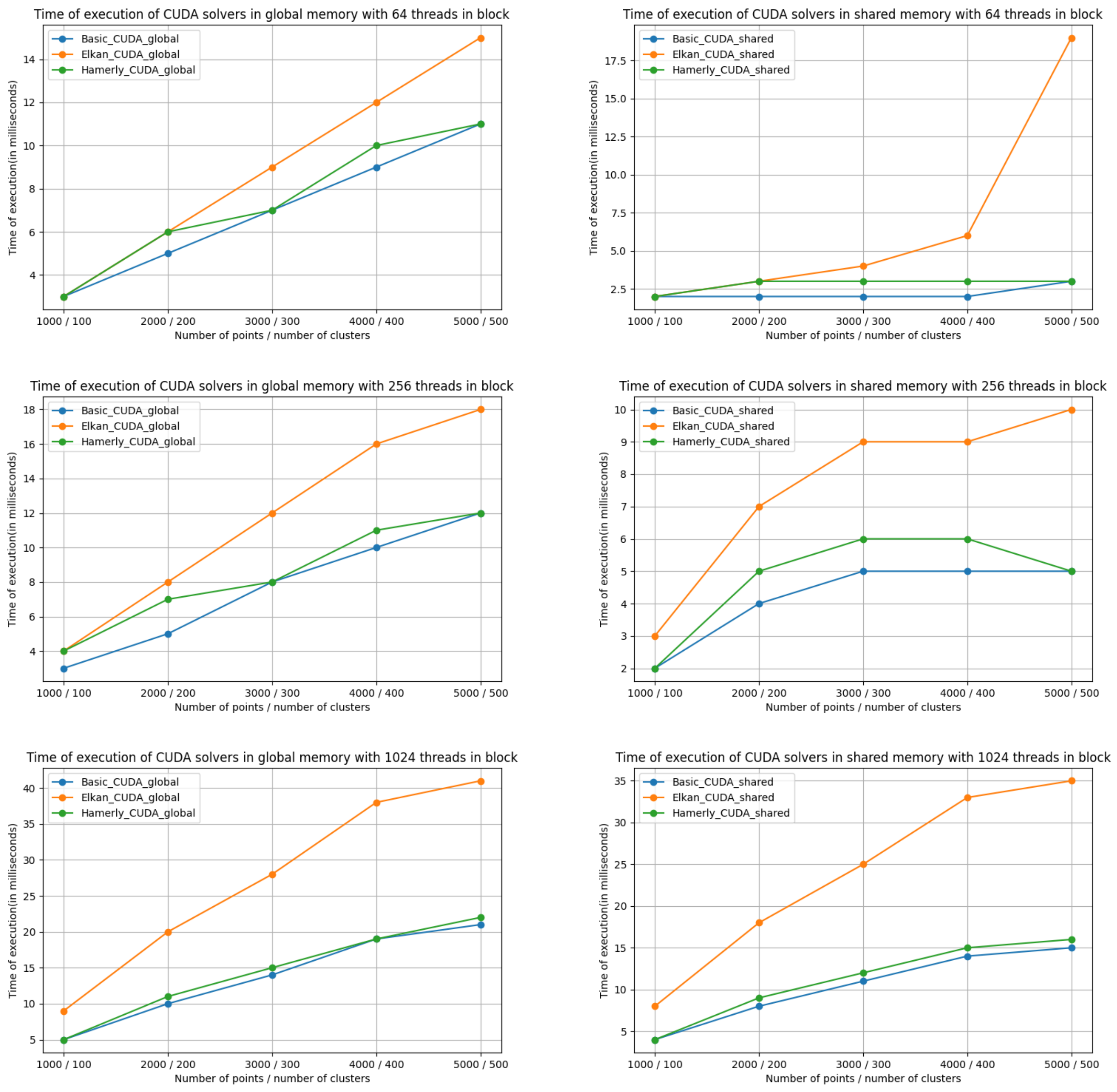

Figure 10 Time of execution of CUDA solvers in global memory with single precision values

4.7 The scores

- Silhouette score:

  The score is obtained via Sklearn implementation of Silhouette score, see [9]. Figure 11 illustrates a histogram of Silhouette scores of random and K-means++ centroids initialization methods. The Silhouette scores are computed for different number of points and clusters:

Number of clusters	Number of points
100	1000
200	2000
300	3000
400	4000
500	5000

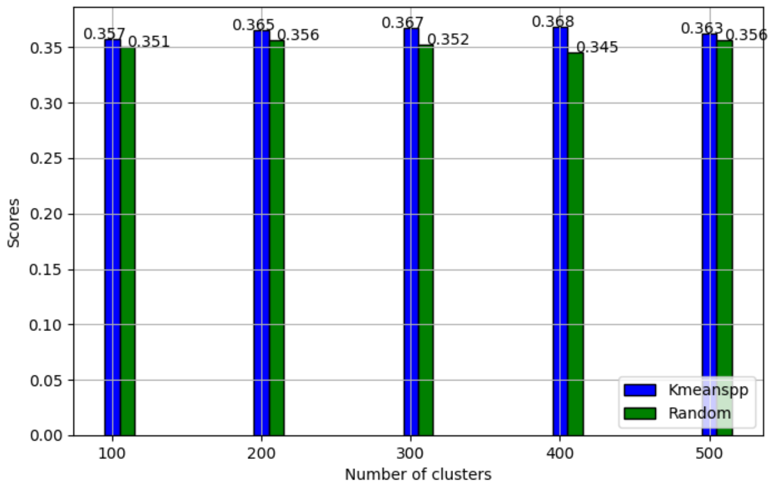

Figure 11 Histogram of Silhouette scores

In all instances the Silhouette scores of clusters obtained by K-means++ centroids initialization method are higher than random centroids initialization method. Thus, clusters obtained by K-means++ are more well separated from one another and are more dense than clusters obtained by random initialization method, which means that quality of clusters with K-means++ initialization is higher.

- Calinski-Harabasz score:

  The score is obtained via Sklearn implementation of Calinski-Harabasz score, see [10]. Figure 12 illustrates a histogram of Calinski-Harabasz scores of clusters obtained by random centroids initialization and K-means++ centroids initialization.

  For all instances, the score of K-means++ was higher than of random initialization, which means that K-means++ centroids initialization results in higher quality of clusters than random centroids initialization.

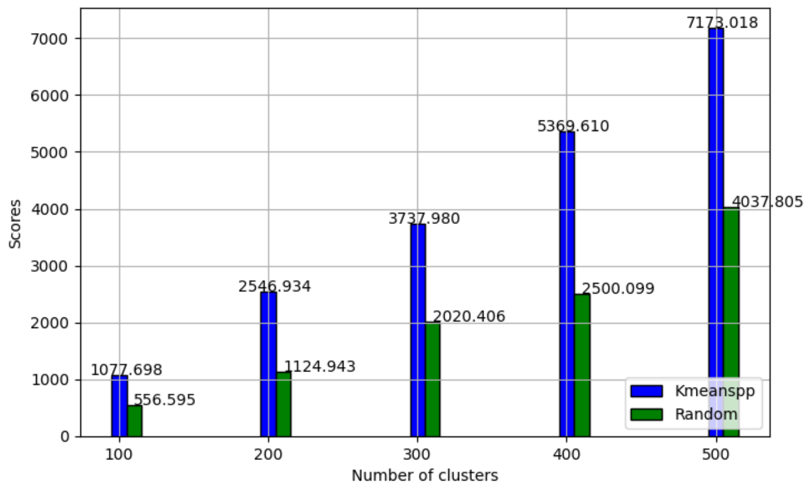

Figure 12 Histogram of Calinski-Harabasz scores

- Davies-Bouldin score:

  The score is obtained via Sklearn implementation of Davies-Bouldin score, see [11]. Figure 13 illustrates a histogram of Davies-Bouldin scores of clusters obtained by random centroids initialization and K-means++ centroids initialization.

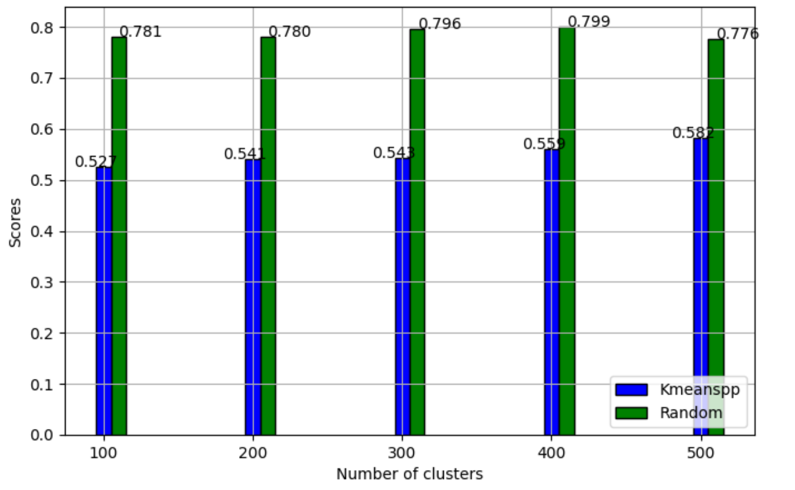

Figure 13 Histogram of Davies-Bouldin scores

For all instances, the score of K-means++ was lower than of random initialization, which means that K-means++ centroids initialization results in higher quality of clusters than random centroids initialization.

Chapter 5
Conclusion

The goal of this work was to study K-means clustering algorithm, triangle inequality heuristics, and implement solvers on different platforms.

Solvers were implemented on single thread on CPU (sequential solvers), multiple threads on CPU (OpenMP solvers), and on GPU (CUDA solvers). The solvers were eventually compared with Python's Sklearn library implementation of K-means clustering, as it is the most widely used K-means implementation in Data Mining field.

As it turned out, sequential solvers were slower than Sklearn implementation. OpenMP solvers with 8 and 12 threads were faster than Sklearn implementation, as with increase of number of threads, more points could be assigned to more threads, thus finishing the overall job faster. Throughout testing sequential and OpenMP solvers with and without triangle inequality heuristics, it became obvious that Elkan's triangle inequality heuristic is faster than basic implementation of K-means clustering and that Hamerly's heuristic is faster than Elkan's. The reason that Hamerly's heuristic was faster than Elkan's lied in the fact that, while in Elkan's heuristic we had to calculate lower bound between each point and each cluster, in Hamerly's heuristic we only need to calculate one lower bound for each point.

All CUDA solvers were faster than Sklearn implementation, thus faster than sequential and OpenMP solvers. The best configuration for CUDA solvers turned out to be 64 threads in the block, because more blocks can be scheduled concurrently and more SMs are occupied, thus increasing the performance. CUDA solvers in shared memory were faster than solvers in global memory, as reduction in block was used, thus utilizing shared memory that is allocated for each block in SM, rather than repeatedly referring to global memory. In addition to reduction in block, shared memory algorithms use chunking of the array from the global memory, thus the chunk of an array that will be used by all threads in the block is loaded only once and number of accesses to the global memory is reduced, which leads to significant performance boost. With double precision values and configurations of 64 and 256 threads per block, CUDA solvers in shared memory were approximately 1.4–1.6 times faster than CUDA solvers in global memory. With single precision values and configurations of 64 and 256 threads per block, CUDA solvers in shared memory were approximately 1.8–2 times faster than CUDA solvers in global memory.

For CUDA solvers, Elkan's heuristic was usually faster than basic solver and Hamerly's heuristic was faster than Elkan's heuristic. But it holds true only when using double precision values, as double precision calculations are very expensive, due to fewer double precision units in GPU compared to single precision units. For single precision values though, basic solver was faster than Elkan's and Hamerly's, as to calculate upper and lower bounds, global memory should be referenced, and due to the fact that single precision calculations are very fast, it was more efficient to calculate all the distances, rather than access global memory in order to calculate bounds.

Silhouette, Calinski-Harabasz, and Davies-Bouldin scores of random and K-means++ centroids initializations showed that K-means++ centroids initialization results in clusters of higher quality, i.e. more dense and more separated from other clusters.

Appendix A

Definition of variables, arrays, and structures used in CUDA kernels

A.1 Variables and arrays

- `T * points_x` : array with X coordinates of the points.
- `T * points_y` : array with Y coordinates of the points.
- `T * clusters_x` : array with X coordinates of the points.
- `T * clusters_y` : array with Y coordinates of the points.
- `int * point_cluster_id` : array with indexes of clusters to which each point is assigned to.
- `int * arr_flag_finished` : array with flags for each point to denote if the point has changed its cluster or not.
- `int * arr_flag_finished_shifted` : array with shifted values of `arr_flag_finished` .
- `int * flag_finished` : flag that denotes if all points have changed their clusters or not.
- `int num_points` : number of points.
- `int num_clusters` : number of clusters.
- `T * arr_lower_bounds` : array with lower bounds.
- `T * upper_bounds` : array with upper bounds.
- `T * cluster_prev_dist` : array with distances of each cluster's centroid coordinates to its previous centroid coordinates.
- `int * was_updated` : array with flags that for each point to denote if upper or lower bounds of that point should be updated.
- `int * was_updated_shifted`
- `CudaTmpPointCluster<T> * tmp_points_clusters` ;

- `CudaTmpPointCluster<T> * tmp_points_clusters_shifted`
- `T * array_average_x` : array with X values of averages of all points in a given cluster that will serve as an X coordinate of the new centroid of that cluster.
- `T * array_average_y` : array with Y values of averages of all points in a given cluster that will serve as an Y coordinate of the new centroid of that cluster.
- `T * array_average_x_shifted` : array with shifted values of `array_average_x` .
- `T * array_average_y_shifted` : array with shifted values of `array_average_y` .
- `int * array_num_points_in_cluster` : array for each cluster that denotes the number of points in that cluster.
- `int * array_num_points_in_cluster_shifted` : array with shifted values of `array_num_points_in_cluster` .
- `T * tmp_cluster_prev_dist` : array that has values of `cluster_prev_dist` that will be reduced to find maximum.
- `T * tmp_cluster_prev_dist_shifted` : array with shifted values of `tmp_cluster_prev_dist` .
- `T * max_cluster_prev_dist` : maximum value of the distance that any cluster has between its current centroid and previous centroid.
- `HeuristicsType heuristics_type` : heuristic that is used for elimination of distance calculations.
- `block_dim` : number of threads in one CUDA block.

A.2 Structures

- `HeuristicsType` is an enumerator with members:
  - None: no special heuristic is used to eliminate distance calculations.
  - Elkan: Elkan's triangle heuristic is used to eliminate distance calculations.
  - Hamerly: Hamerly's triangle heuristic is used to eliminate distance calculations.
- `CudaCoordinate` is a templated structure with template parameter `T` and members:
  - `T m_x` : X coordinate.
  - `T m_y` : Y coordinate.
- `CudaTmpPointCluster` is a templated structure with template parameter `T` and members:
  - `T m_cluster_distance` : distance between a given point and the closest cluster in a given chunk of clusters.
  - `T m_second_cluster_distance` : distance between a given point and the second closest cluster in a given chunk of clusters.
  - `int m_cluster_id` : index of the cluster in a given chunk of clusters with distance `m_cluster_distance` to a given point.
- `CudaSharedSumChunks` is a templated structure with template parameter `T` and members:

  - `T m_point_x` : X coordinate of the point.
  - `T m_point_y` : Y coordinate of the point.
  - `int m_point_cluster_id` : index of the cluster to which this point is assigned to.
- `CudaSharedFindAverage` is a templated structure with template parameter `T` and members:
  - `T m_sum_cluster_x` : sum of all X coordinates of clusters in this chunk of points.
  - `T m_sum_cluster_y` : sum of all Y coordinates of clusters in this chunk of points.
  - `int m_num_points_in_cluster` : number of all points in this chunk of points that are assigned to this cluster.

A.3 Kernels

Listing A.1 illustrates `cuda_shift_results_to_tmp_closest_cluster` kernel. Configuration is

- Block dimension of `dim3 blockDim(block_dim` .
- Grid dimension of `dim3 gridDim(ceilf(ceilf((float) num_elements / block_dim * 2) / block_dim), num_points)` .

Code listing A.1 Kernel for shifting data from original array to temporary array

```
int thread_pos = blockDim.x * blockIdx.x + threadIdx.x;
int point_id = blockIdx.y;

if((heuristics_type == Elkan || heuristics_type == Hamerly) && !was_updated[point_id])
    return;

if(thread_pos >= num_blocks)
    return;

int tmp_arr_id = point_id * num_clusters_chunks + thread_pos;
int tmp_arr_block_id = point_id * num_clusters_chunks + thread_pos * block_num_elements;
tmp_points_clusters_shifted[tmp_arr_id].m_cluster_distance =
    tmp_points_clusters[tmp_arr_block_id].m_cluster_distance;
tmp_points_clusters_shifted[tmp_arr_id].m_second_cluster_distance =
    tmp_points_clusters[tmp_arr_block_id].m_second_cluster_distance;
tmp_points_clusters_shifted[tmp_arr_id].m_cluster_id =
    tmp_points_clusters[tmp_arr_block_id].m_cluster_id;
```

Listing A.2 illustrates `cuda_copy_shifted_results_from_tmp_closest_cluster` kernel. Configuration is

- Block dimension of `dim3 blockDim(block_dim` .
- Grid dimension of `dim3 gridDim(ceilf(ceilf((float) num_elements / block_dim * 2) / block_dim), num_points)` .

Code listing A.2 Kernel for shifting data from temporary array to original array

```
int thread_pos = blockDim.x * blockIdx.x + threadIdx.x;
int point_id = blockIdx.y;

if((heuristics_type == Elkan || heuristics_type == Hamerly) && !was_updated[point_id])
    return;

if(thread_pos >= num_blocks)
    return;

int tmp_arr_id = point_id * num_clusters_chunks + thread_pos;
tmp_points_clusters[tmp_arr_id].m_cluster_distance =
    tmp_points_clusters_shifted[tmp_arr_id].m_cluster_distance;
tmp_points_clusters[tmp_arr_id].m_second_cluster_distance =
    tmp_points_clusters_shifted[tmp_arr_id].m_second_cluster_distance;
tmp_points_clusters[tmp_arr_id].m_cluster_id =
    tmp_points_clusters_shifted[tmp_arr_id].m_cluster_id;
```

Bibliography

1. BLAISE BARNEY, LAWRENCE LIVERMORE NATIONAL LABORATORY. *OpenMP*. 2022. Available also from: `https://hpc-tutorials.llnl.gov/openmp/`.
2. NVIDIA. *CUDA*. 2022. Available also from: `https://www.nvidia.com/en-gb/geforce/technologies/cuda/`.
3. LLOYD, S. Least squares quantization in PCM. *IEEE Transactions on Information Theory*. 1982, vol. 28, no. 2, pp. 129–137. Available from DOI: `10.1109/TIT.1982.1056489`.
4. O'NEILL, Barrett (ed.). Front Matter. In: *Elementary Differential Geometry (Second Edition)*. Second Edition. Boston: Academic Press, 2006, p. iii. ISBN 978-0-12-088735-4. Available from DOI: `https://doi.org/10.1016/B978-0-12-088735-4.50001-8`.
5. ELKAN, Charles. Using the Triangle Inequality to Accelerate K-Means. In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. Washington, DC, USA: AAAI Press, 2003, pp. 147–153. ICML'03. ISBN 1577351894. Available from DOI: `10.5555/3041838.3041857`.
6. HAMERLY, Greg. Making k-means Even Faster. In: *SDM*. 2010. Available also from: `https://www.semanticscholar.org/paper/Making-k-means-Even-Faster-Hamerly/103e3167b2308987a809d5eed679dff213861664`.
7. GHORPADE-AHER, Jayshree; PARANDE, Jitendra; KULKARNI, Madhura; BAWASKAR, Amit. GPGPU processing in CUDA architecture. *CoRR*. 2012, vol. abs/1202.4347. Available also from: `https://www.researchgate.net/publication/220484926_GPGPU_processing_in_CUDA_architecture`.
8. ARTHUR, David; VASSILVITSKII, Sergei. K-Means++: The Advantages of Careful Seeding. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. SODA '07. ISBN 9780898716245. Available from DOI: `10.5555/1283383.1283494`.
9. SCIKIT LEARN. *Silhouette score*. 2022. Available also from: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html`.
10. SCIKIT LEARN. *Calinski-Harabasz Score*. 2022. Available also from: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.calinski_harabasz_score.html`.
11. SCIKIT LEARN. *Davies-Bouldin Score*. 2022. Available also from: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html`.
12. NVIDIA. *CUDA Toolkit*. 2022. Available also from: `https://docs.nvidia.com/cuda/`.

13. SCIKIT LEARN. *Sklearn K-means*. 2022. Available also from: `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`.

14. GÖRLITZ, Olaf; SIZOV, Sergej; STAAB, Steffen. PINTS: peer-to-peer infrastructure for tagging systems. In: *IPTPS*. 2008, p. 19.

Contents of enclosed CD

`Thesis`	the directory with latex source file and thesis pdf
`src`	the directory of source codes
`CSV_Files`	generated CSV files
`Common_Files`	files used by every solvers
`Cuda`	CUDA solvers
`Graps`	generated graphs
`OpenMP`	OpenMP solvers
`Scripts`	Python scripts for graph generations
`Sklearn`	Sklearn implementation
`Tests`	tests
`Samples`	Sample files used for testing