# 3. Algorithm Parallelization and Optimization on GPUs

Team : parlab03

Ioannis Palaios - el18184

## I. Introduction

We were assigned to develop three different parallel versions of the K-means clustering algorithm using the CUDA framework and the C language and then benchmark and compare them. These 3 version were benchmarked on a Nvidia GPU and specifically the model of the GPU is Tesla V100-SXM2-32GB. At first, we have to benchmark all the versions for the vim K-means clustering of the configuration : Size, Coords, Clusters, Loops = 256, 2, 16, 10 and these CUDA block sizes : 32, 64, 128, 256, 512, 1024. Then, we were asked to benchmark and comment on the specific operations happening, like CPU to GPU data transfer, time spent on the GPU or CPU etc. and for the configuration which instead of 2 Coordinates, it has 16.

## II. K-Means Naive Version

In the first version, called *Naive*, we assign to the GPU the most compute intensive part of the algorithm, the calculation of the nearest clusters in each iteration. This part of the algorithm consists of a for loop to calculate the distance of each object to each cluster, and this distance calculation needs to take into account all the dimensions so another for loop for the number of coordinates is needed. So it effectively is a triple nested for loop with a total loop size of *number_of_objects*number_of_clusters*number_of_coordinates*. As it tries to find the cluster with the minimum distance from each object, once this distance is calculated it compares it with the current minimum distance (and if it is less it sets it equals). After, this type of triple nested for loop is finished, it checks to see if this object changed it's membership to the previous cluster it belonged and if it does, it increments *devdelta*, a variable common for all threads and then changes the cluster that this object belongs to. Of course, the incrementation of the *devdelta* variable needs to happen in a thread safe as multiple multiple threads incrementing it will create a wrong final value.

First of all, we have to decide the pattern with which we assign work to threads. We essentially will assign each thread to find the distance of one object to each cluster, as this is the easiest and most obvious way to parallelize this triple for loop without much changes in the given code. So, in each k-means loop (or each kernel execution) we would require the number of threads spawned to be at least equal to the number of objects (which will be 256 for all of our benchmarks). As a result, the number of blocks will be equal to the number of objects divided by the block size and +1 because the integer division will round down the resulting float to an integer (and thus producing less blocks than required). The variable assignment is shown on Listing 1 below.

```
1 const unsigned int numClusterBlocks = numObjs/blockSize + 1;
```

Listing 1: Grid Size calculation on all of the K-means versions.

We could also split the workload of the distance calculations for one object to two or more threads, with each of these threads finding the distances from this one object to *some* of the clusters. But this would require changes in the way we define and compare the current *dist* variable and the current *min_dist* variable, namely each thread would have to keep it's own *min_dist* variable and after the end of the for loop, we should take and compare the minimum of all these variables (and the index of the corresponding cluster).

In each iteration, after the GPU distance calculation, some data, namely the *devdelta* variable and the *membership[]* array, must be transferred from the GPU memory to the main memory (CPU memory) in order for the CPU to perform some calculations. In the start of each iteration, before the GPU calculations, only the *devdelta* variable and the *clusters[]* array need to be transferred to the GPU's memory as the delta variable needs to be reinitialized to zero and the clusters array needs to be refreshed as after the CPU part the cluster coordinates have changed. The *membership[]* array does not need to be transferred as it has not been changed since the last iteration and it already is onto the GPU's memory. The transfers of each iteration are shown in Listing 2 below.

```
    ...
// Initial CPU to GPU transfers : objects[][] and membership[]
checkCuda(cudaMemcpy(deviceObjects, objects,
              numObjs*numCoords*sizeof(double), cudaMemcpyHostToDevice));
checkCuda(cudaMemcpy(deviceMembership, membership,
              numObjs*sizeof(int), cudaMemcpyHostToDevice));
do {
    // CPU to GPU : clusters[][] and dev_delta_ptr
    checkCuda(cudaMemcpy(deviceClusters, clusters,
            numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));
    checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
        ...
    // CUDA Kernel Invocation
        ...
    // GPU to CPU : membership[][] and dev_delta_ptr
    checkCuda(cudaMemcpy(membership, deviceMembership,
            numObjs*sizeof(int), cudaMemcpyDeviceToHost));
    checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double),
    cudaMemcpyDeviceToHost));
} while(...)
    ...
```

Listing 2: Data transfers of each iteration of the K-means algorithm.

The CUDA Kernel of our implementation will be the function *find_nearest_cluster()* and as a result will be declared with the *__global__* keyword. As with every CUDA kernel, we have to differentiate the work done from each thread by getting it's (global or local) ID. For this reason, we will declare the *get_tid()* function in order to return the global ID of each thread (in the one dimensional blocks and grid) and as this function can only be called from the GPU we use the *__device__* keyword. Its source code is shown on Listing 3 below.

```
__device__ int get_tid() {
    return threadIdx.x + blockIdx.x * blockDim.x;
}
```

Listing 3: The source code of the get_id() function which returns the global thread ID in one-dimensional blocks and grid.

Then, we have to define the function responsible for returning the euclidean distance between a specific object and cluster : *euclid_dist_2()*. This function will take as parameters the arrays of objects and clusters and the total numbers of coordinates, objects and clusters in order to traverse the arrays efficiently. Also, we will take the IDs of the pair of the given object and cluster we want

to find the distance of. As this function is also supposed to run on the GPU we will declare it with the *__device__* keyword (not sure why the *__host__* keyword is added on the given implementation). The source code of the *euclid_dist_2()* function is shown on Listing 4 below.

```
__host__ __device__ inline static
double euclid_dist_2(int    numCoords,
                     int    numObjs,
                     int    numClusters,
                     double *objects,     // [numObjs][numCoords]
                     double *clusters,    // [numClusters][numCoords]
                     int    objectId,
                     int    clusterId)
{
    int i;
    double ans=0.0;
    for(i = 0; i < numCoords; i++) {
        ans += (objects[objectId*numCoords+i] - clusters[clusterId*numCoords+i]) * (
    objects[objectId*numCoords+i] - clusters[clusterId*numCoords+i]);
    }
    return(ans);
}
```

Listing 4: The source code of the euclid_dist_2() function which returns the euclidean distance of a cluster with an object for a given number of coordinates.

The CUDA kernel function (or the *__global__* function) is the function *find_nearest_cluster()*, as mentioned above. This function makes use of the the two functions mentioned above and is essentially the function that will be run from each thread on the grid. At some Kernel invocations, we might have more threads than we actually need and that's why we have to make sure we only assign work to the thread we actually need. This will be done with an if conditional statement which will check if the threads global ID is less than the number of objects. As we are creating the exact number of threads we need (or a little more), we are not dealing with the fact that one thread must be assigned work for more than one object. The source code of this function is shown on Listing 5 below.

```
__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters
,                         double *objects,           //  [numObjs][numCoords]
                          double *deviceClusters,    //  [numClusters][numCoords]
                          int *deviceMembership,          //  [numObjs]
                          double *devdelta)
{

    /* Get the global ID of the thread. */
    int tid = get_tid();

    if (tid < numObjs) {
        int   index, i;
        double dist, min_dist;
        index = 0;
        /* Find the distance of the object with id equal to tid and the cluster with
     id equal to 0 and assign it to the min_dist variable */
        min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, 0);
        for (i=1; i<numClusters; i++) {
            /* Find the distance of the object with id equal to tid and the cluster
    with id equal to i and assign it to the dist variable */
```

```
            dist = euclid_dist_2(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, i);
            /* Compare the dist variable with the min_dist variable */
            if (dist < min_dist) {
                min_dist = dist;
                index    = i;
            }
        }
        if (deviceMembership[tid] != index) {
            atomicAdd(devdelta, 1.0);
        }
        deviceMembership[tid] = index;
    }
}
```

Listing 5: The source code of the find_nearest_cluster() function which is the CUDA kernel of the naive implementation of the K-means algorithm.

We now benchmark this version for the configuration Size, Coords, Clusters, Loops = 256, 2, 16, 10 and block_size = 32, 64, 128, 256, 512, 1024. The execution time bar plot and the speedup plot are shown on Figure 1 and 2 below, respectively.

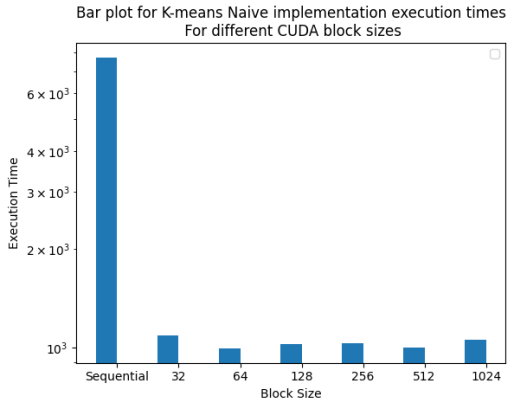


Figure 1: Bar Plot of the execution times of the Naive version of the K-Means Clustering Algorithm CUDA implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} and different block sizes.

We observe an 7x to 8x speedup from the sequential version for all the different block sizes. Of course, this happens due to the efficient work distribution to the multiple CUDA threads utilizing efficiently the given GPU resources, efficiently parallelizing the algorithm's most compute intensive part. Obviously, the overhead from the data transfer between the main memory and the GPU's memory is not significant in comparison to the parallel execution of the *find_nearest_cluster()* function. This performance gain is due to the problem's nature, namely, a lot of uncomplicated and simple operation with no major dependencies between them. But, essentially, in each iteration we spawn 256 (or more threads) to achieve a speedup of 7.5x. If we were thinking with CPU terms this parallelization would not be worth energy wise, but because now we use a GPU, thread spawning and execution is not the same as the CPUs so it actually be worth it energy-wise. As we see more clear on the speedup plot, the performance differs for the different block sizes (not a lot). This might be due to better Streaming Multiprocessor resource utilization and occupancy on the given workload, the Kernel launch overhead and L1 cache and Register usage. The best

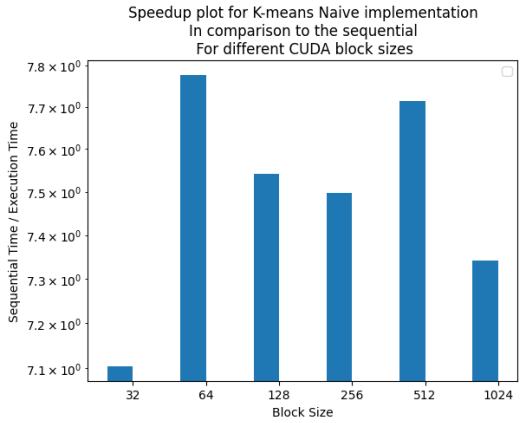


Figure 2: Bar Plot of the speedup of the execution time of the Naive version of the K-Means Clustering Algorithm CUDA implementations divided by the execution time of the sequential, for the Configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} and different block sizes.

performance is observed for the block size of 64 threads which is likely due to optimizing all the above parameters.

## III. K-Means Transpose version

In order to transpose the objects & clusters arrays from row-based to column-based indexing we will have to initialize and access them differently. The initialization is shown on Listing 6 below.

```
    ...
double  **dimObjects = (double**) calloc_2d(numCoords, numObjs, sizeof(double));
double  **dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double))
    ;
    ...
for (i = 0; i < numCoords; i++) {
    for (j = 0; j < numObjs; j++) {
        dimObjects[i][j] = objects[j*numCoords+i];
    }
}
for (i = 0; i < numCoords; i++) {
    for (j = 0; j < numClusters; j++) {
        dimClusters[i][j] = dimObjects[i][j];
    }
}
    ...
```

Listing 6: The initialization of the transposed arrays : objects and clusters - For the K-means Transpose version.

We also modify the *euclid_dist_2()* function in order to calculate the distance for the transposed clusters[] and objects[] arrays. We also change the function's name from *euclid_dist_2()* to *euclid_dist_2_transpose()*. The source code of this function is shown on Listing 7 below.

```
__host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
                    int    numObjs,
                    int    numClusters,
```

```
                    double *objects,      // [numCoords][numObjs]
                    double *clusters,     // [numCoords][numClusters]
                    int    objectId,
                    int    clusterId)
{
    int i;
    double ans=0.0;

    for(i = 0; i < numCoords; i++) {
        ans += (objects[objectId+(numObjs*i)] - clusters[clusterId+(numClusters*i)])
     * (objects[objectId+(numObjs*i)] - clusters[clusterId+(numClusters*i)]);
    }
    return(ans);
}
```

Listing 7: The initialization of the transposed arrays : objects and clusters - For the K-means Transpose version.

The Bar plot of the execution times for the Transpose version is shown below on Figure 3. The Speedup plot of the execution times for the Transpose and Naive version is also shown on Figure 4.

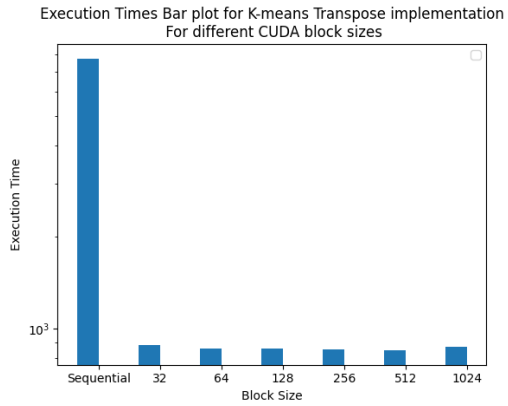


Figure 3: Bar Plot of the execution times of the Transpose version of the K-Means Clustering Algorithm CUDA implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} and different block sizes.

As it is observed on both plots, now as block size increases, performance increases except from the transition from block size of 512 to 1024 which actually gets worse. From this observation, we understand that each Streaming Multiprocessor can be fully utilized at the size of 512 and will actually be better than more blocks of smaller size, for the Transpose version.

In the Naive version, the accesses happen in this way : *objects[thread_id*2+i]* where i is either 0 or 1 depending on which stage of the for loop we are in (but each warp has the same value of i). We observe that the threads belonging to the same warp, with 32 consecutive thread ids, perform misaligned memory accesses which means that they do not access continuous memory addresses. Thus, as the *objects[]* array resides in global memory, the transactions being performed to the global memory are not coalesced and instead are performed separately decreasing the possible bandwidth and introducing extra latency to our kernel.

In the Transpose version, the accesses happen in this way : *objects[thread_id+(256*i)]*, where i is also either 0 or 1. Contrary to the previous version, the threads with the 32 consecutive thread

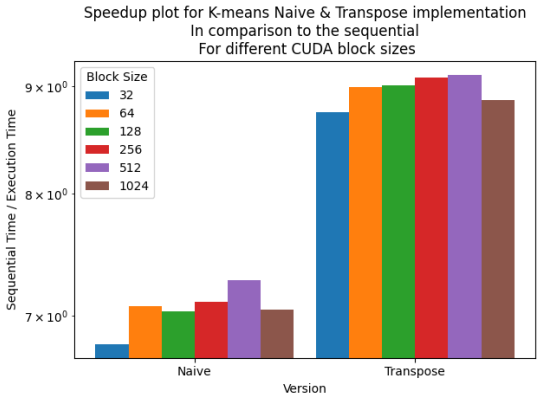


Figure 4: Bar Plot of the speedup of the execution time of the Naive and Transpose versions of the K-Means Clustering Algorithm CUDA implementations divided by the execution time of the sequential, for the Configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} and different block sizes.

ids are making aligned accesses and thus the memory transactions can be coalesced and less transactions are performed, greatly increasing bandwidth. This is the reason why we achieve this performance gain when executing the Transpose version, in comparison to the Naive version.

## IV. K-Means Shared Version

For this version, we were assigned to place the *clusters[]* array on the Shared Memory. Because this type of memory is located on-chip, it is much faster than the global memory we used on the two previous versions. Shared memory is allocated per thread block, so only threads from the same block share this memory. As a result, it makes sense, to use this memory for each block separately, and in order to do this, we have to use each thread's local id (id for the threads belonging in the same block) which can be accessed simple through the *threadIdx.x* variable. In CUDA, shared memory size can either be defined statically or dynamically. As the *clusters[]* array size is not known on compile time but instead is dependent on the variables *numCoords* and *numClusters*, we will dynamically allocate the shared memory we need. It's size can be passed through an optional third execution configuration kernel parameter. Then, the declaration and initialization of the shared memory array happens inside the corresponding kernel. The initialization is being done in a parallel fashion, where we increment each local thread id by the block size and when it surpasses the size of the clusters array, each thread finished it's corresponding initialization. It obviously is guaranteed that that the shared memory clusters array will be properly initialized this way. Both the dynamic allocation and declaration of the shared memory are shown in Listing 8 below.

```
__global__ static
void find_nearest_cluster(int numCoords,
                          int numClusters,
                             ... ,
                          double *deviceClusters,
                             ...
                          ) {
    extern __shared__ double shmemClusters;
    int local_tid = threadIdx.x;
```

```
    int clusters_size = numCoords*numClusters;
    int block_threads = blockDim.x;
    int idx = local_tid;
    while(idx < clusters_size) {
        shmemClusters[idx] = deviceClusters[idx];
        idx += block_threads;
    }
    __syncthreads();
        ...
}

kmeans_gpu (...) {
        ...
    const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(
    double);
        ...
    do {
            ...
        find_nearest_cluster
            <<< block_size, grid_size, clusterBlockSharedDataSize >>>
            (...);
    } while (...)
        ...
}
```

Listing 8: Dynamic Allocation - Declaration and initialization of the Shared Memory on the K-means Shared version.

Again, we benchmark this Shared version for the same configuration and block sizes as above. The speedup plot of all the 3 versions (including Shared) is being shown on Figure 5.

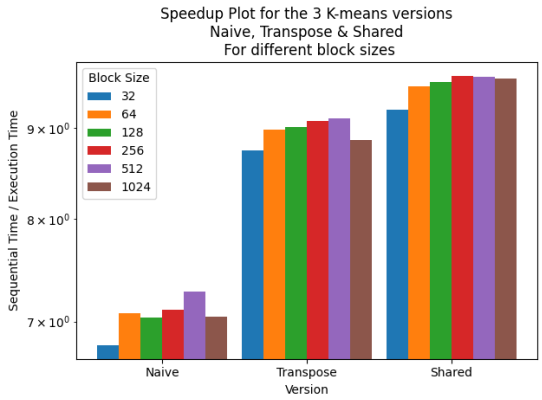


Figure 5: Speedup Plot of the 3 K-means algorithm versions - Naive, Transpose and Shared versions - for the Configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} and different block sizes.

As we expected, a small speedup is observed in comparison to the Transpose version (and to the Naive version of course) due to the reduced latency of the memory accesses due to the change from storage of the *clusters[]* array in the global memory to the shared memory. Shared memory bank conflicts are being avoided because all the threads in a warp access the exact same element of this array (and thus the same memory banks) and as a result, the corresponding memory transactions are being coalesced. But possibly, in the Transpose version, the contents of

the *clusters[]* array were being cached in the L1 cache, thus reducing the total memory latency of this version. As a result, placing the clusters[] array on the shared memory does not make such a big performance improvement. A bigger performance improvement could be achieved by also placing the *objects[]* array on the shared memory, while carefully avoiding memory bank conflicts.

## V. Further Benchmarking

In order to fully understand the way our CUDA program is run we have to benchmark our application in different phases (or parts) of our program. The phases exist in each iteration (inside the dowhile()) and can be categorized as below :

- CPU to GPU phase : The part where the data, namely the *clusters[]* array and the *dev_delta_ptr* pointer, are transferred from the CPU's main memory (Host Memory) to the GPU's global memory (Device Memory)
- GPU phase : The part where the program is run entirely on the GPU's Streaming Multiprocessors. It is essentially the time of the kernel execution.
- GPU to CPU phase : The part where the data, namely the *membership[]* array and the *dev_delta_ptr* pointer, are transferred from the GPU's device memory to the Host memory.
- CPU phase : The part in each iteration where the data are processed on the CPU and the cluster centers are updated and replaced etc. Essentially, this part consists of two double nested for loop of size numObjs*numCoords and numClusters*numCoords respectively.

As the only thing we expect to differ between the 3 versions is the time spent on the GPU phase, we will benchmark only our best performing version (the Shared version) and observe the percentage of time that is spent on each phase. This fact is also confirmed by our benchmarks. Also, we will not compare the Shared version for the different block sizes as similar performance is observed for all of them.

In Figure 6 below, a pie chart of the time of each phase is shown. We observe that over 50% of the total time is spent on the CPU phase due to serially executing this double nested kernel. Of course this part can also be parallelized with a little bit of modifications. The next most time consuming phase is the GPU phase which consumes 25% of our total time. This is the part we tried to parallelize in all of this assignment and improving performance on that part is non-trivial.

Lastly, the 3rd most time consuming phase is the data transfer from the GPU to CPU which consumes around 18.8% of the time while the phase of data transfer from the CPU to the GPU consumes 0%. But they also transfer a different amount of data (ignoring the *dev_delta_ptr* as it is transferred on both phases). The total size of data transferred on each iteration and total time spent (in ms) for each phase is :

- CPU to GPU phase : The size of the *clusters[]* array which is *numCoords*numClusters*sizeof(double) = 256 bytes* and 0.22 ms total time.
- GPU to CPU phase : The size of the *membership[]* array which is *numObjs*sizeof(int) = 256 * 4 bytes = 1024 bytes* and 151.17 ms total time.

Although the second phase transfers only 4x more data, it consumes around 700x more time. This imbalance is maybe due to the different loading and storing bandwidths of each memory (device and host), various bottlenecks encountered or different memory alignment etc. Further

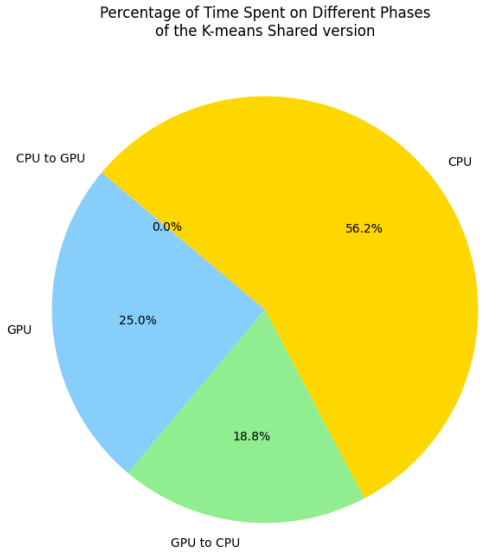


Figure 6: Pie chart of the percentage of total time spent on the 4 different phases of the K-means Shared implementation for block_size equal to 1024.

benchmarks is needed to actually find the actual reason of this imbalance but this is out of the scope of this assignment.

We are also conducting benchmarks for the configuration which instead of 2 Coordinates, has 16, namely : **{Size, Coords, Clusters, Loops} = {256, 16, 16, 10}**. Also, we will benchmark them for the same block sizes as above. The benchmarks are shown on Figure 7 below.

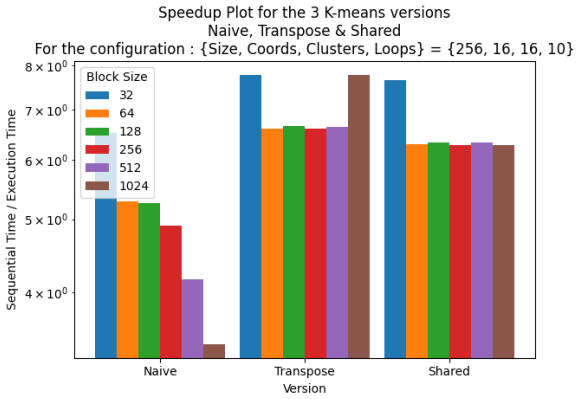


Figure 7: Speedup plot for the 3 versions of the K-means algorithm for the configuration Size, Coords, Clusters, Loops = 256, 16, 16, 10.

First, we observe that in all of these versions, the best performing block size is the 32. This maybe happens due to better utilization on the resources of the corresponding Streaming Multiprocessor, mainly registers but also the L1 cache. Secondly, the Shared version ceases to be the best performing one, because the Transpose version performs better (although not by a big margin). In order to understand the reason of this performance change we have to understand what changed

with the change of these configurations.

Actually, the only thing that changed is the number of coordinates on our K-means algorithm. For the GPU part, this means bigger *objects[][]* and *clusters[][]* arrays and also a bigger for-loop for the distance calculations. But why these performance differentiation between these 3 versions? The answer is that we are not sure, but here is our theories :

- Likely, the shared memory initialization is not being done efficiently for the threads belonging to the block. This might be due to thread divergence, the call of *__syncthreads()* which creates a barrier in the execution or just because we are not doing efficient parallel initialization. As a result, the shared memory allocation and initialization is not worth the trade off between taking the *clusters[]* array from the global memory and storing it in the L1 cache.
- Possibly, the increased size of the shared memory we have allocated does not leave a lot of space for the L1 cache, in order to store the *objects[]* array etc. Our shared memory array goes from the size of *256 bytes* to *2048 bytes* (8x). As the *objects[]* array has the same size as above *256 MB*, then we assume that there is not enough space in the L1 cache in order to store both of these two arrays at the same time (not even one) so at one time a part of the *objects[]* array is stored onto the L1 which likely introduces cache misses etc.
- Shared Memory bank conflicts still do not happen. This is due to that each warp is accessing the same memory addresses (belonging in the same bank of course) so these memory transactions are coalesced into one.

As a result, we should only prefer the Shared version instead of the Naive when the size of the two arrays is small enough, having not many more than 2 coordinates, in order for them to be able to fit in the L1 cache of the GPU we are using.

## VI. K-means All-GPU version

As we observed above, the majority of the execution time (namely 56.2%) of our program (for all versions) is spent on the CPU, in order to calculate and update the new cluster centers. Also, another major phase of our program is the data transfer from the GPU to the CPU (needed for the CPU to perform the calculations mentioned above). In order to improve our program's performance, we are going to implement this *All-GPU* version, where we do not execute these two phases and we instead execute everything on the GPU. Essentially, by only using the GPU for all the calculations in each iteration, we both skip the data transfer needed and parallelize the calculation and update of the new cluster centers.

In order to achieve this, we will split the total calculations of the iteration in two kernels, namely : *find_nearest_cluster()* and *update_centroids()*. The first kernel will calculate the same things as all the previous versions but,now, will also update the new cluster centers and their size (the *newClusters[][]* and the *newClusterSize[]*). The *update_centroids()* will average the sum and replace the old cluster centers with the new clusters. The data of both of these two kernel will stay in the GPU's global memory (device memory) and so their transfer will not be needed for the execution of these kernels. We will use the same pointers to reference them for these two kernels, namely : *devicenewClusterSize*, *devicenewClusters* and *deviceCluster*. Of course, the *update_centroids()* Kernel is spawned with different Grid and Block dimensions, spawning at least *numCoords*numClusters* threads. The source code of each iteration is shown on Listing 9 below.

```
1 do {
2         timing_internal = wtime();
3         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
```

```
        find_nearest_cluster
            <<< numClusterBlocks, numThreadsPerClusterBlock,
    clusterBlockSharedDataSize >>>
            (numCoords, numObjs, numClusters,
             deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters,
     deviceMembership, dev_delta_ptr);
        cudaDeviceSynchronize(); checkLastCudaError();
        checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double),
    cudaMemcpyDeviceToHost));

        const unsigned int update_centroids_block_sz = (numCoords* numClusters >
    blockSize) ? blockSize: numCoords* numClusters;
        const unsigned int update_centroids_dim_sz =  numCoords*numClusters/
    blockSize + 1;
        update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0
    >>>
            (numCoords, numClusters, devicenewClusterSize, devicenewClusters,
    deviceClusters);
        cudaDeviceSynchronize(); checkLastCudaError();

        delta /= numObjs;
        loop++;
        timing_internal = wtime() - timing_internal;
        if ( timing_internal < timer_min) timer_min = timing_internal;
        if ( timing_internal > timer_max) timer_max = timing_internal;
    } while (delta > threshold && loop < loop_threshold);
```

Listing 9: The source code of the All-GPU version's iterations.

The renewed source code of the *find_nearest_cluster()* kernel is shown on Listing 10 below. We observe that now that both additions of *deviceNewClusterSize[index]* and *deviceNewClusters[j*numClusters+index]* is now being done using the *atomicAdd()* atomic function in CUDA, as if we don't use this atomic operation, multiple threads can access/read and increment the same parts of the array concurrently and thus produce wrong results. Of course, as explored in the second exercise, atomic operation can slow down our execution and thus it might be beneficial to somehow increment a seperate buffer for each thread and later reduce them into the arrays we have above (but it might not be worth the effort).

```
__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          double *objects,           //  [numCoords][numObjs]
                          int *deviceNewClusterSize,    // [numClusters]
                          double *deviceNewClusters,       //  [numCoords][
    numClusters]
                          double *deviceClusters,    //  [numCoords][numClusters]
                          int *deviceMembership,          //  [numObjs]
                          double *devdelta)
{
     extern __shared__ double shmemClusters[];
    int local_tid = threadIdx.x;
    int block_threads = blockDim.x;
    int clusters_size = numCoords*numClusters;

    int idx = local_tid;
    while(idx < clusters_size) {
        shmemClusters[idx] = deviceClusters[idx];
        idx += block_threads;
    }
```

```
    __syncthreads();

    int tid = get_tid();

    if (tid < numObjs) {
        int   index, i, j;
        double dist, min_dist;
        index = 0;
        min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
     shmemClusters, tid, 0);
        for (i=1; i<numClusters; i++) {
            dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
     shmemClusters, tid, i);
            if (dist < min_dist) {
                min_dist = dist;
                index    = i;
            }
        }
        if (deviceMembership[tid] != index) {
            atomicAdd(devdelta, 1.0);
        }

        deviceMembership[tid] = index;

        atomicAdd(&deviceNewClusterSize[index], 1);

        for(j=0; j<numCoords; j++) {
            atomicAdd(&deviceNewClusters[j*numClusters+index], objects[j*numObjs+tid
    ]);
        }
    }
}
```

Listing 10: The source code of the All-GPU version's find_nearest_cluster() kernel.

The source code of the *update_centroids()* kernel is shown on Listing 11 below. We are essentially trying to mimic the CPU serial execution of averaging the sum and replacing the old cluster centers, but now in parallel fashion. We assign work only at the first *numClusters*numCoords* threads of the grid and each thread is responsible for calculating the average cluster position for each coordinate. The *i* and *j* we define are also trying to mimic the serial execution, and they stand for the index of the cluster and coordinate respectively.

```
__global__ static
void update_centroids(int numCoords,
                      int numClusters,
                      int *devicenewClusterSize,           //  [numClusters]
                      double *devicenewClusters,    //  [numCoords][numClusters]
                      double *deviceClusters)    //  [numCoords][numClusters])
{
    int tid = get_tid();
    int i = tid % numClusters;
    int j = tid / numClusters;
    if(tid < numClusters*numCoords) {
        if(devicenewClusterSize[i] > 0)
            deviceClusters[tid] = devicenewClusters[tid] / devicenewClusterSize[i];
        devicenewClusters[tid] = 0.0;
    }
    __syncthreads();
    if(tid < numClusters*numCoords && j == 0)
        devicenewClusterSize[i] = 0;
```

```
19 }
```

Listing 11: The source code of the All-GPU version's find_nearest_cluster() kernel.

The rest of the source code of this version is identical to the previous versions - except for some array copies from or to the GPU before and after the iterations.

We benchmarked this version for the configurations Size, Coords, Clusters, Loops = 256, 2, 16, 10 and 256, 16, 16, 10. The results are shown in the Speedup plots of Figure 8 and 9 respectively, comparing them for the 3 previous versions. In both of these configurations, as we expected, the All-GPU version performs more than 2x better than all the others. As we previously mentioned, this due to the efficient parallelization of a compute intensive part of the program (inside each iteration) and avoiding the data transfer between the CPU and GPU. As a result, in this version, the performance is not hindered by the bandwidth of the CPU and GPU memory link and the serial execution of a compute intensive part.

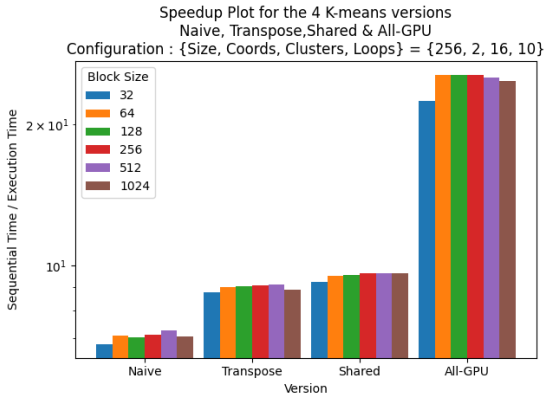


Figure 8: Speedup plot for the 4 versions of the K-means algorithm for the configuration Size, Coords, Clusters, Loops = 256, 2, 16, 10.

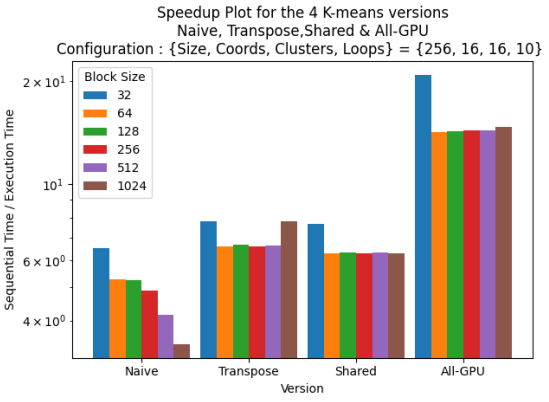


Figure 9: Speedup plot for the 4 versions of the K-means algorithm for the configuration Size, Coords, Clusters, Loops = 256, 16, 16, 10.

The block size plays a similar role as in the Shared version, with minimal differences between the different sizes - except the case for the block size of 32, which in the configuration with 2

coordinates performs worse than the other sizes and for the configuration with 16 coordinates, performs 1.3x better than the other sizes. This is similar performance difference as the block sizes in the shared version and the causes for these difference are likely the same.

Now we have to ask ourselves if the *update_centroids()* function is suitable for GPU execution. The answer is : it depends. First, it depends from where the three parameter arrays are located and where are they going to be used next. If they are located on the Host memory and they are going to be need there after the GPU execution, it might be beneficial to process them on the CPU as a two way data transfer will be avoided. But, for the benchmarks we performed for the configuration with 2 or 16 Coordinates and 16 Clusters and for the specific phases of the execution, it was shown that it is beneficial to perform this function on the GPU as in each iteration of our program we always process the data on the GPU, so the 3 arrays are located on the GPU before and after so it does not make any sense to transfer the data to the CPU incurring both the data transfer overhead and the serial execution overhead. As a result, the only time it might make sense to process this function on the CPU is when the data are stored on Host memory before and must be located there after it's execution. But, for each other case, executing this function on the GPU is can improve performance significantly.

The performance of the two configurations we benchmarked is on par with the other versions' performance difference and is likely for the same reasons (which we mentioned above).

## VII. Conclusions

In this assignment, we efficiently developed 3 different versions of the K-means algorithm benefiting from GPU parallelization of the algorithm's most compute intensive part, using Nvidia's CUDA API. The 3 different versions where the Naive, where we *naively* parallelized the part without taking into account misaligned memory accessing, the Transpose, where by transposing two arrays the parallelization became more efficient and the Shared version, where we took advantage of the Shared Memory of each SM. We benchmarked them for different block sizes and 2 different configurations, exploring their differences and advantages compared to one another while also visualizing the results of the benchmarks performed. We explored the time spent on the specific parts of the program like CPU or GPU computing and data transfer, while also we tested another configuration for 16 coordinates instead of 2. Lastly, we implemented a version of the K-means algorithm called All-GPU which avoids the data transfer between the CPU and the GPU (which happened on all previous versions) and instead processes every compute intensive task of each iteration on the GPU. We observed a significant performance increase from this version in comparison to the other version, which we commented on and explained the reasons why this happens.

## References

[1] How to Access Global Memory Efficiently in CUDA C/C++ Kernels - Mark Harris | link

[2] Using Shared Memory in CUDA C/C++ - Mark Harris | link

[3] Programming Massively Parallel Processors - A Hands-on Approach - David B. Kirk and Wen-mei W. Hwu | link

[4] CUDA Toolkit Documentation v8.0 | link