UNIVERSITÀ DEGLI STUDI FIRENZE

DIEF Dipartimento di Ingegneria Industriale

K-means

Parallel Computing Course

Antonio Castellucci
Michela Crulli

UNIVERSITÀ DEGLI STUDI FIRENZE

Overview

1. Introduction
   - What is K-means
   - What we have done
2. Implementation
   - ImagesHandler
   - C++ implementation
   - OpenMP implementation
   - CUDA implementations
3. Dataset
4. Results
   - Test platform
   - Tests
5. Conclusions

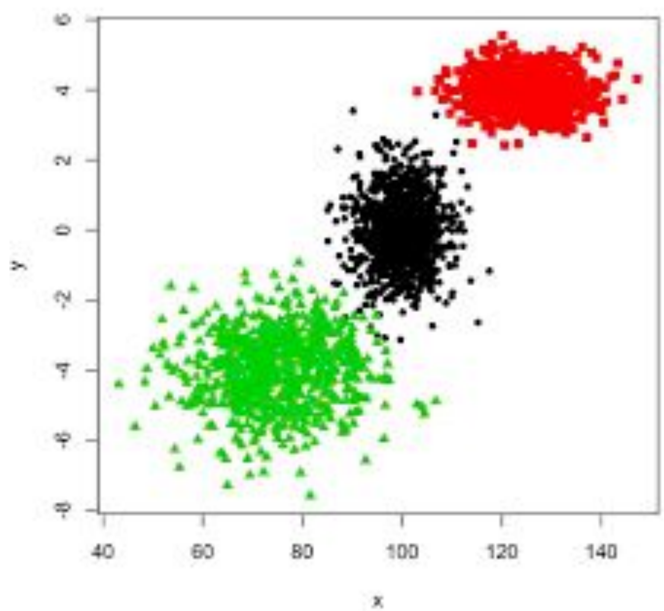
Introduction

In this project, we realize a program which do **k-means** clustering with three different approaches:

- a **sequential** implementation
  - C++
- three **parallel** implementation
  - OpenMP
  - CUDA naive
  - CUDA with reduction

- ## What is K-means?

K-means clustering aims to **partition** input data into **k cluster** in which each data belongs to the cluster with the **nearest mean**.

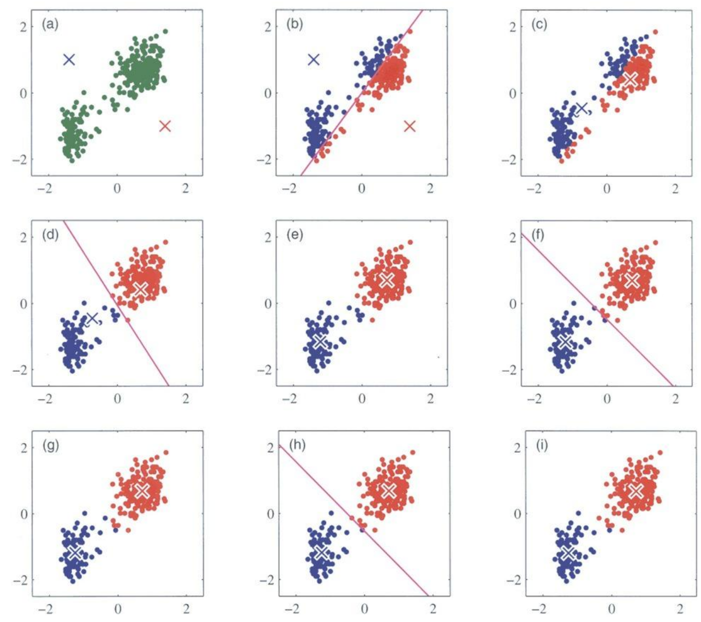


Introduction

- Steps:
  1. Decide number of clusters
  2. Distance from each point to each centroids
  3. Re-calculate of each centroid with the minimization of:

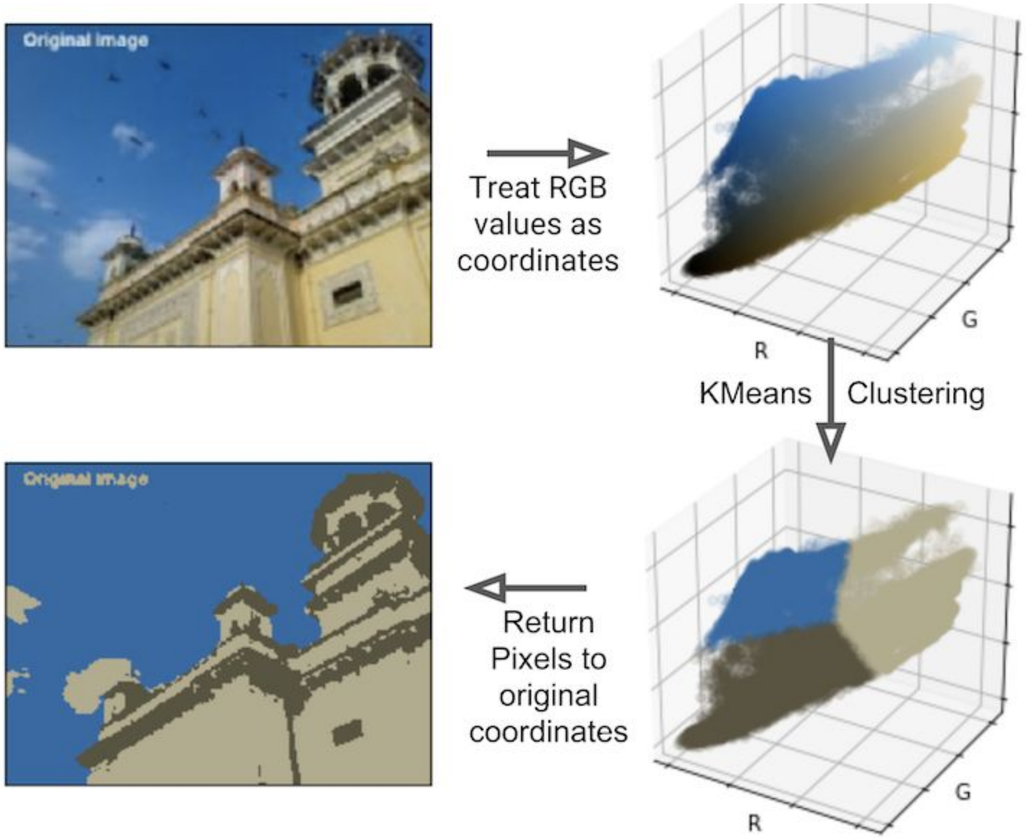
$$J = \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$$

  4. Iterate from step two until in two iteration clusters don't change



- What we have done

K-means applied to **images** to have a better and immediate view



Introduction

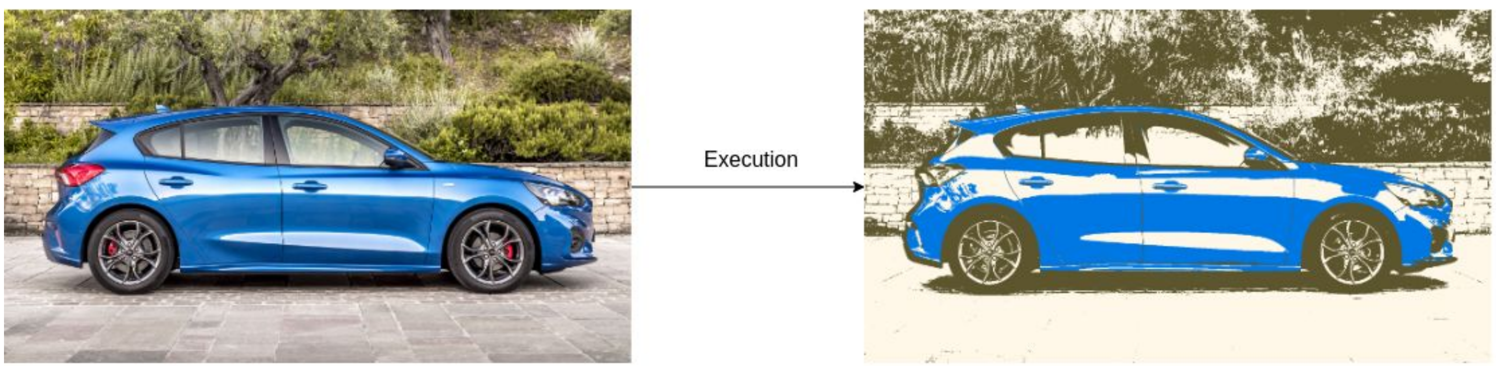
- Input output structure
  - scriptable programs with unique interface between three versions

```
./fastCudaKmeans ./testImages/car.jpg RGB 3 500
```

```
selected file: ./testImages/car.jpg
mode: RGB
number of clusters: 3 clusters
iterations: 500 iterations

 image processing...

PROCESSING TIME: 0.811982 s
```

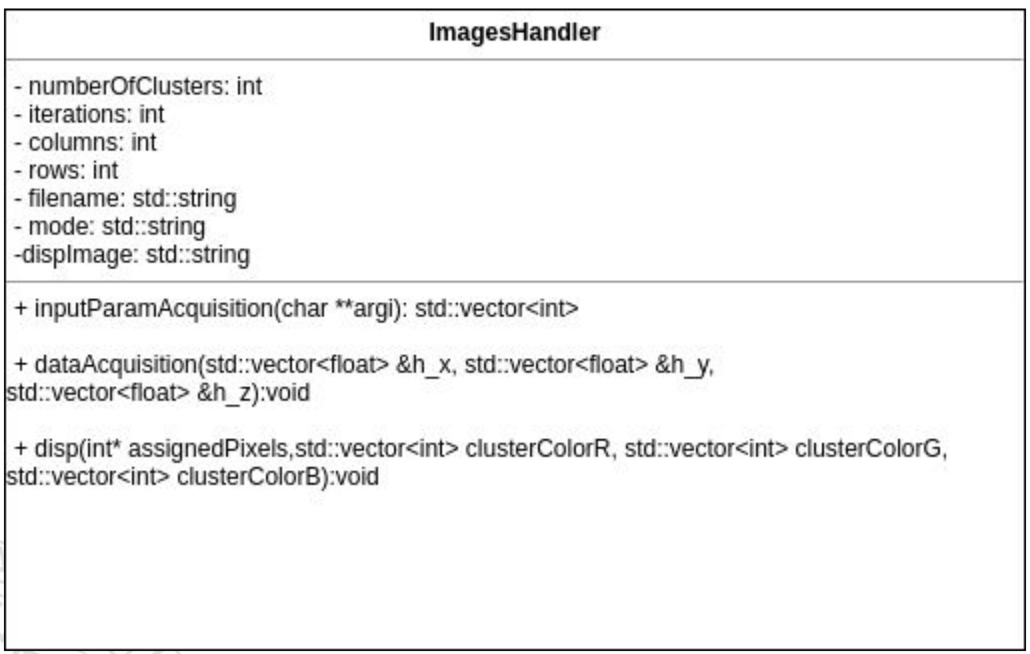


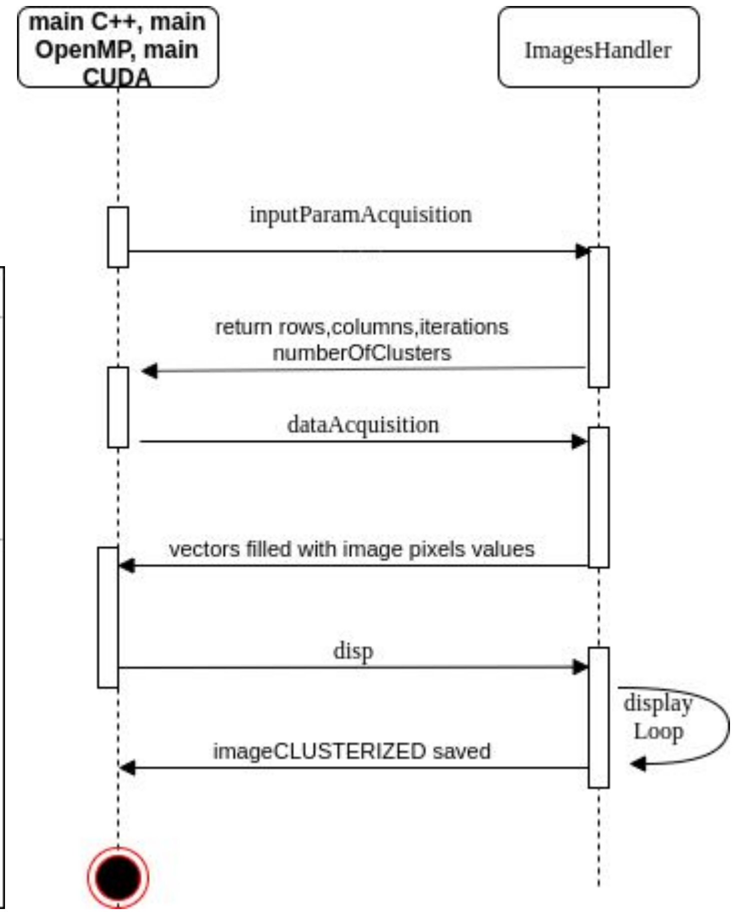
car.jpg

carCLUSTERIZED.jpg

Implementation

- ImagesHandler
  - What is? It’s a **common class** that define interfaces for pixels values acquisition and processed data handling
  - Why? For the separation of concerns and code reusability





Implementation

- C++ implementation

Algorithm 1 c++

```
Input: data, number_of_cluster, number_of_iterations
Output: means, data.assignments
for  i=0 to number_of_cluster
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    for point = 0 to data.size()
        best_distance = d(data[point], means[0])
        best_cluster = 0
        for cluster = 1 to number_of_cluster
            dist = d(data[point], means[cluster])
            if dist < best_distance
                best_distance = dist
                best_cluster = cluster
            end if
        end for
        data.assignments[point] = best_cluster
    end for
    for point = 0 to data.size()
        cluster = data.assignments[point]
        new[cluster].x+ = data[point].x
        new[cluster].y+ = data[point].y
        new[cluster].z+ = data[point].z
        counts[cluster]+ = 1
    end for
    for cluster = 0 to number_of_cluster
        count = counts[cluster]
        means[cluster].x = new[cluster].x/count
        means[cluster].y = new[cluster].y/count
        means[cluster].z = new[cluster].z/count
    end for
end for
```

Implementation

- OpenMP implementation

Algorithm 2 openMP

```
Input: data, number_of_cluster, number_of_iterations
Output: means, data.assignments
for  i=0 to number_of_cluster
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    #pragma omp parallel for
    for point = 0 to data.size()
        best_distance = d(data[point], means[0])
        best_cluster = 0
        for cluster = 0 to number_of_cluster
            dist = d(data[point], means[cluster])
            if dist < best_distance
                best_distance = dist
                best_cluster = cluster
            end if
        end for
        data.assignments[point] = best_cluster
    end for
    for point = 0 to data.size()
        cluster = data.assignments[point]
        new[cluster].x+ = data[point].x
        new[cluster].y+ = data[point].y
        new[cluster].z+ = data[point].z
        counts[cluster]+ = 1
    end for
    #pragma omp parallel for
    for cluster = 0 to number_of_cluster
        count = counts[cluster]
        means[cluster].x = new[cluster].x/count
        means[cluster].y = new[cluster].y/count
        means[cluster].z = new[cluster].z/count
    end for
end for
```

Implementation

- CUDA Naive implementation

Algorithm 3 cudaKmeans

```
Input: data, k=number_of_cluster,number_of_iterations
Output: means, data.assignments
for i=0 to k
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    assign_cluster(data, means, k, new_sums)
    cudaDeviceSynchronize()
    compute_new_means(means, new_sums)
    cudaDeviceSynchronize()
end for
```

Algorithm 5 assign_cluster

```
Input: data, means, new_sums, k
Output: data
index = blockIdx.x * blockDim.x + threadIdx.x
if index ≥ data_size return
end if
best_distance = d(data[index],means[0])
best_cluster = 0
for cluster = 1 to k
    distance = d(data[index], means[cluster])
    if distance < best_distance
        best_distance = distance
        best_cluster = cluster
    end if
end for
data.assignments[index]=best_cluster
atomicAdd(new_sums.x[best_cluster],data.x[point])
atomicAdd(new_sums.y[best_cluster], data.y[point])
atomicAdd(new_sums.z[best_cluster], data.z[point])
atomicAdd(counts[best_cluster], 1)
```

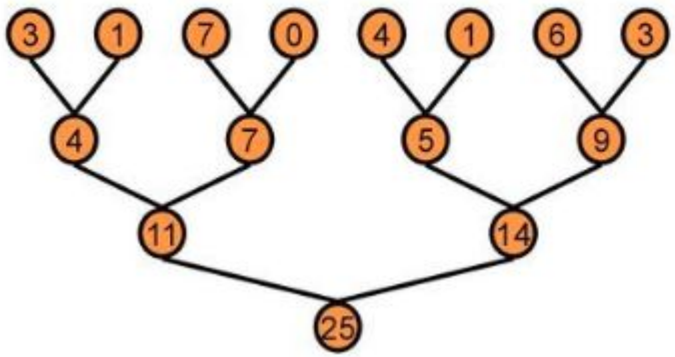
Algorithm 4 compute_new_means

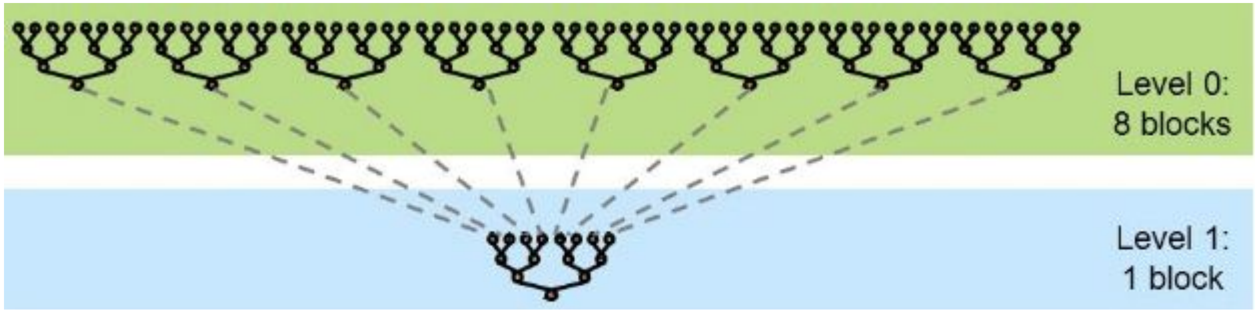
```
Input: means, new_sums
Output: means
cluster = threadIdx.x
means.x[cluster] = new_sum.x[cluster] / counts[cluster]
means.y[cluster] = new_sum.y[cluster] / counts[cluster]
means.z[cluster] = new_sum.z[cluster] / counts[cluster]
```

- Reduction

Mark Harris tricks:

- Unrolling loops
- Multiple adds/threads



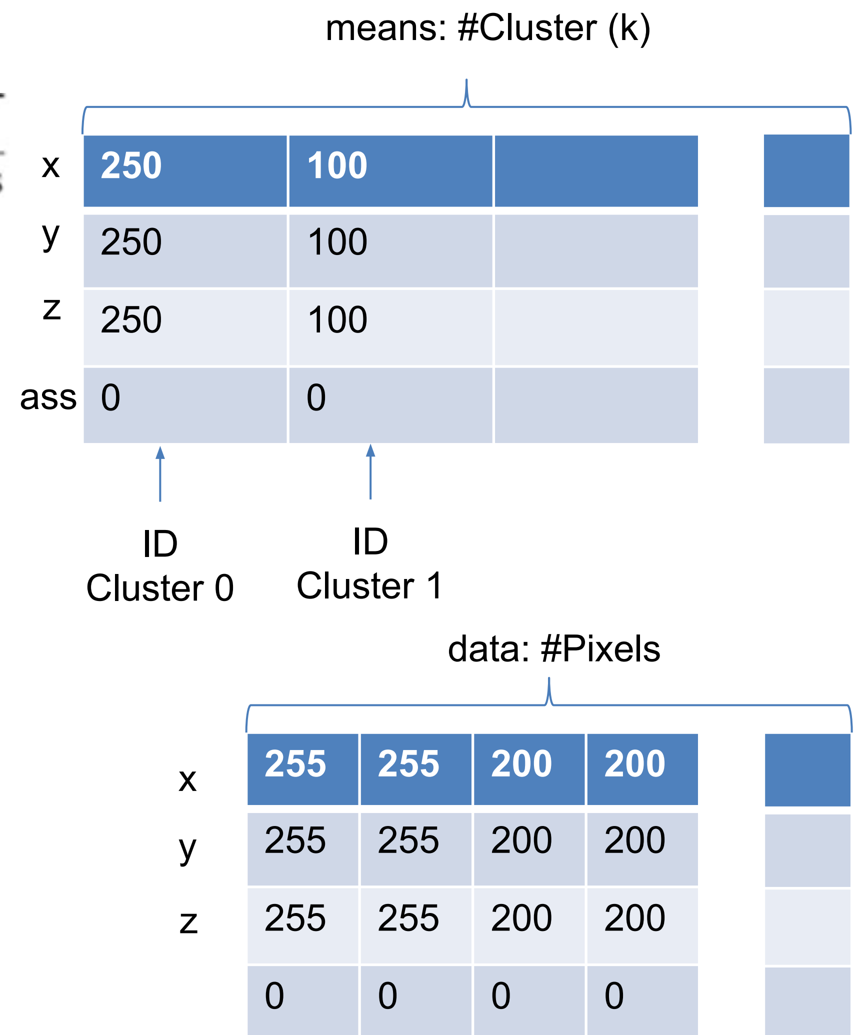


Implementation

- fastCuda 1.0

Algorithm 6 fastCudaKmeans 1.0

```
Input: data, k=number_of_cluster,number_of_iterations
Output: processed data
for i=0 to k
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    assign_cluster(data, means, k)
    cudaDeviceSynchronize()
    for cluster = 0 to k
        populate(data, clusterData, cluster)
        cudaDeviceSynchronize()
```



Implementation

- assign_cluster(data, means, k)

Data: #Pixels

x	**255**	**255**	**200**	**200**	
y	255	255	200	200	
z	255	255	200	200	
	0	0	0	0	

assign_cluster

Data: #Pixels

255	**255**	**200**	**200**	
255	255	200	200	
255	255	200	200	
0	0	1	1	

ID Cluster 0

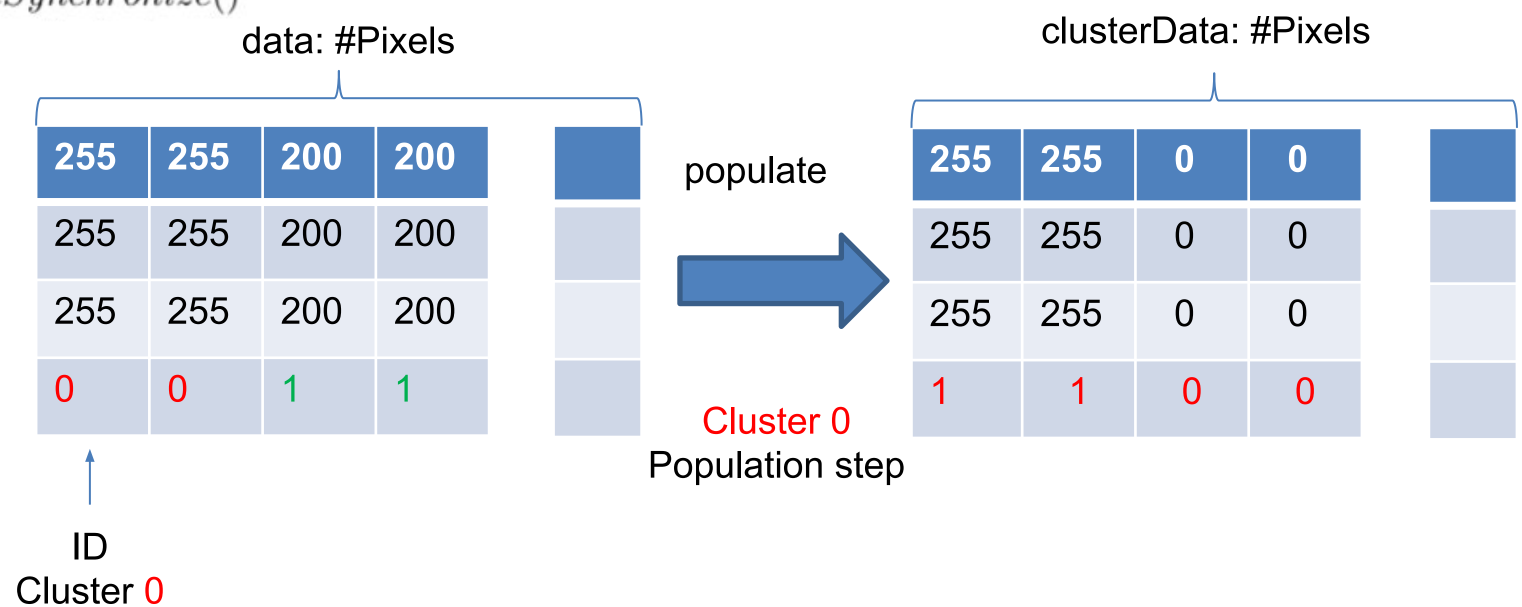
ID Cluster 1

Implementation

- fastCuda 1.0

Algorithm 6 fastCudaKmeans 1.0

Input: data, k=number_of_cluster,number_of_iterations
Output: processed data
for i=0 to k
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    $assign\_cluster(data, means, k)$
    $cudaDeviceSynchronize()$
    for cluster = 0 to k
        $populate(data, clusterData, cluster)$
        $cudaDeviceSynchronize()$

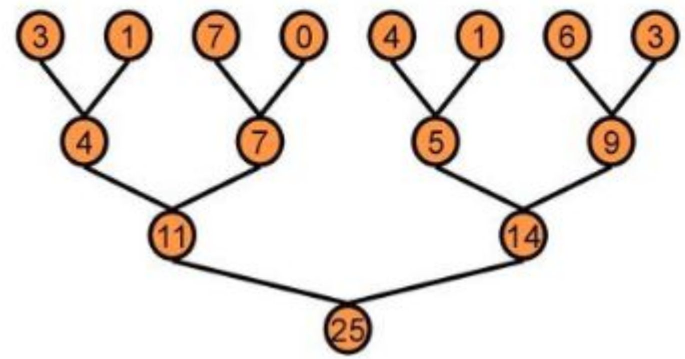


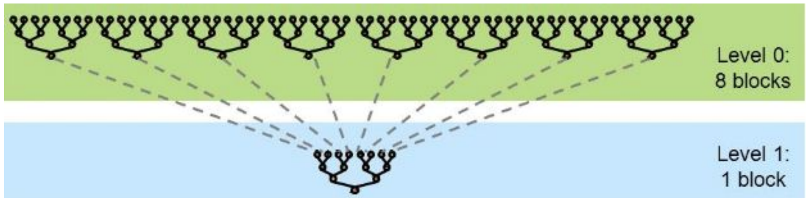
Implementation

- fastCuda 1.0

Algorithm 6 fastCudaKmeans 1.0

Input: data, k=number_of_cluster,number_of_iterations
Output: processed data
for i=0 to k
  centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
  $assign\_cluster(data, means, k)$
  $cudaDeviceSynchronize()$
  for cluster = 0 to k
    $populate(data, clusterData, cluster)$
    $cudaDeviceSynchronize()$
    $n = number\_of\_pixels$
    do
      $blocksPerGrid = n/BLOCKSIZE$
      $reduction(clusterData, tmp, n)$
      $cudaDeviceSynchronize()$
      $clusterData = tmp$
      $n = blocksPerGrid$
    while n > BLOCKSIZE





Implementation

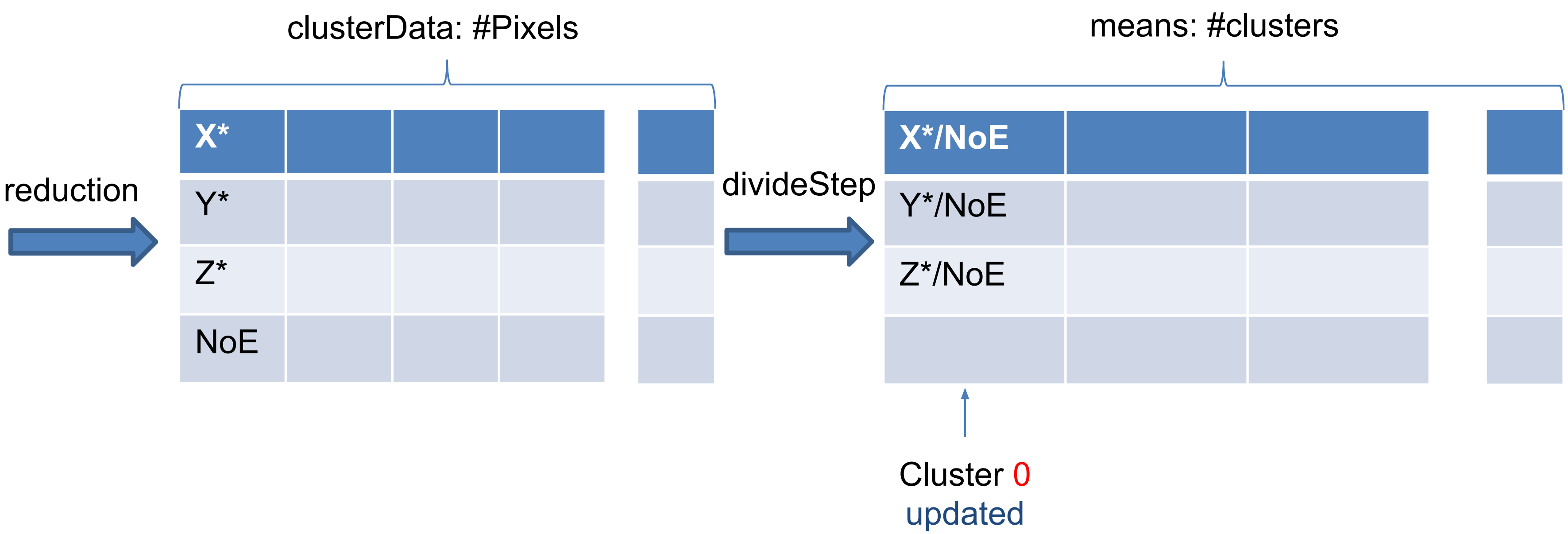
- fastCuda 1.0

Algorithm 6 fastCudaKmeans 1.0

```
Input: data, k=number_of_cluster,number_of_iterations
Output: processed data
for i=0 to k
    centroids population in means[i]
end for
for iteration = 0 to number_of_iterations
    assign_cluster(data, means, k)
    cudaDeviceSynchronize()
    for cluster = 0 to k
        populate(data, clusterData, cluster)
        cudaDeviceSynchronize()
        n = number_of_pixels
        do
            blocksPerGrid = n/BLOCKSIZE
            reduction(clusterData, tmp, n)
            cudaDeviceSynchronize()
            clusterData = tmp
            n = blocksPerGrid
        while n > BLOCKSIZE
        reduction(tmp, tmp, n)
        cudaDeviceSynchronize()
        divideStep(means, tmp, cluster, k)
        cudaDeviceSynchronize()
    end for
end for
```

Implementation

- divideStep(means,tmp,cluster,k)



Implementation

- populate 2.0 (data, clusterData, ~~cluster~~)

data: #Pixels

255	255	200	200	
255	255	200	200	
255	255	200	200	
0	0	1	1	

ID
Cluster 0

ID
Cluster 1

Populate 2.0

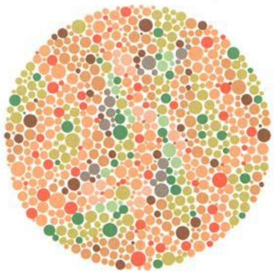
clusterData: #Pixels * #Clusters

255	255	0	0		0	0	200	200	
255	255	0	0		0	0	200	200	
255	255	0	0		0	0	200	200	
1	1	0	0		0	0	1	1	

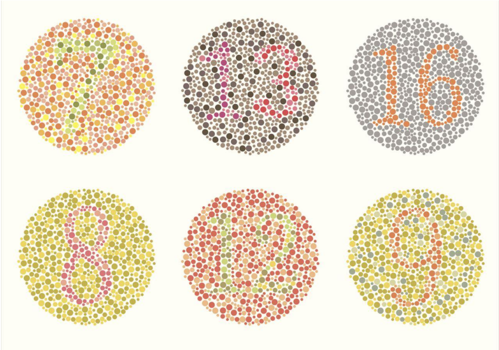
#Pixels

Dataset

We use images of different sizes :



100000 pixels



1 millions pixels

2 millions pixels

10 millions pixels

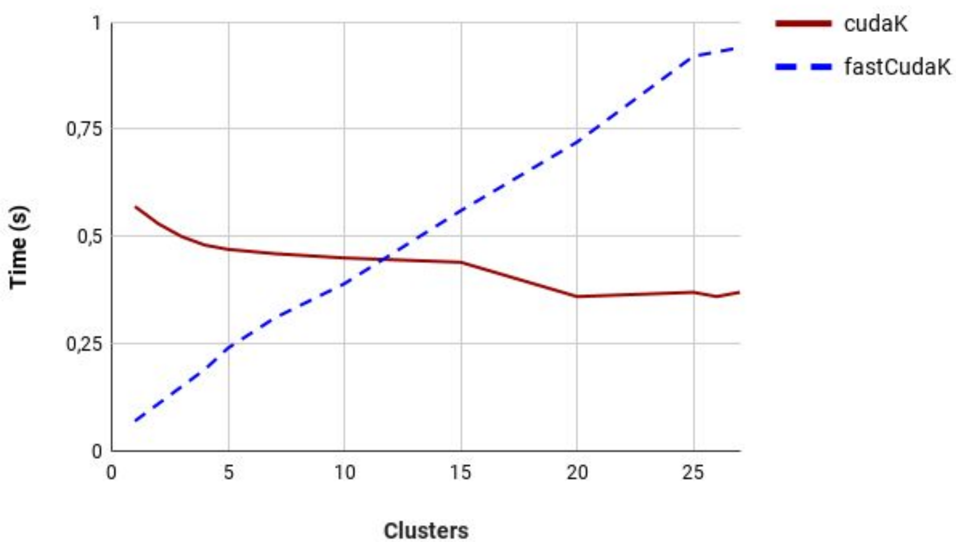
Test Platform

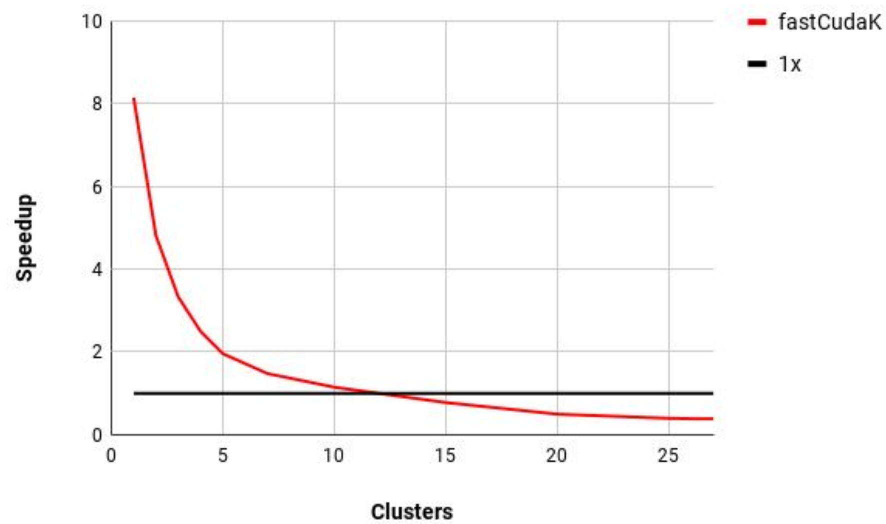
MSI GS65 with:

- i7-8750H 6 CORES 12 THREAD
- 16 GB 2400 MHz DDR4 RAM
- RTX 2060 6GB GDDR6

Results

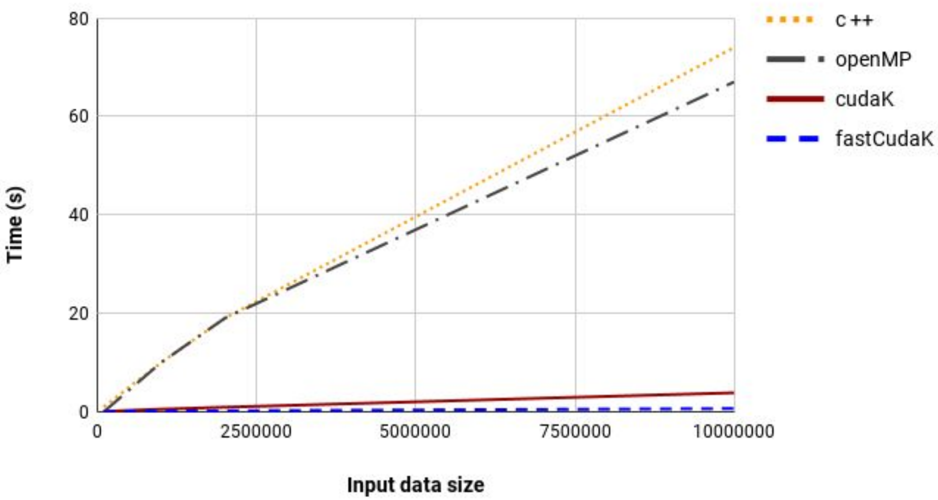
- cudaKmeans Vs fastCudaKmeans 2.0 with 2 millions pixels, 100 iterations as the number of clusters changes

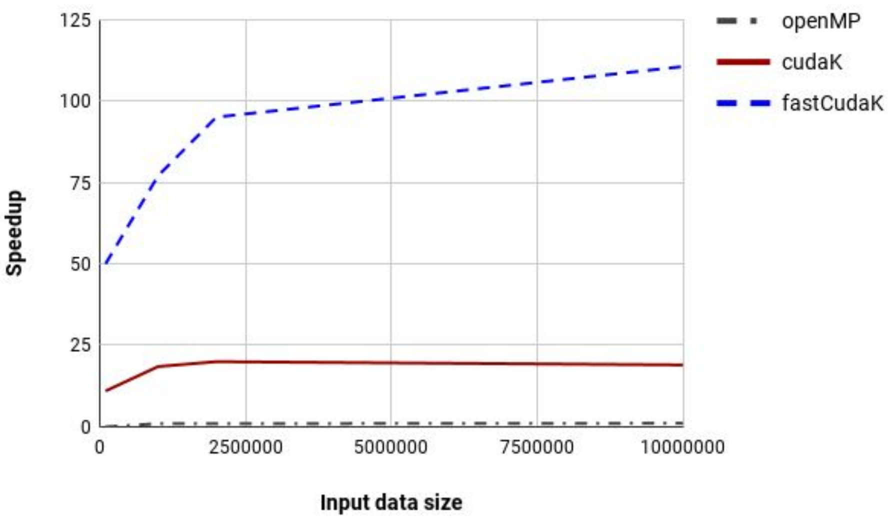




Results

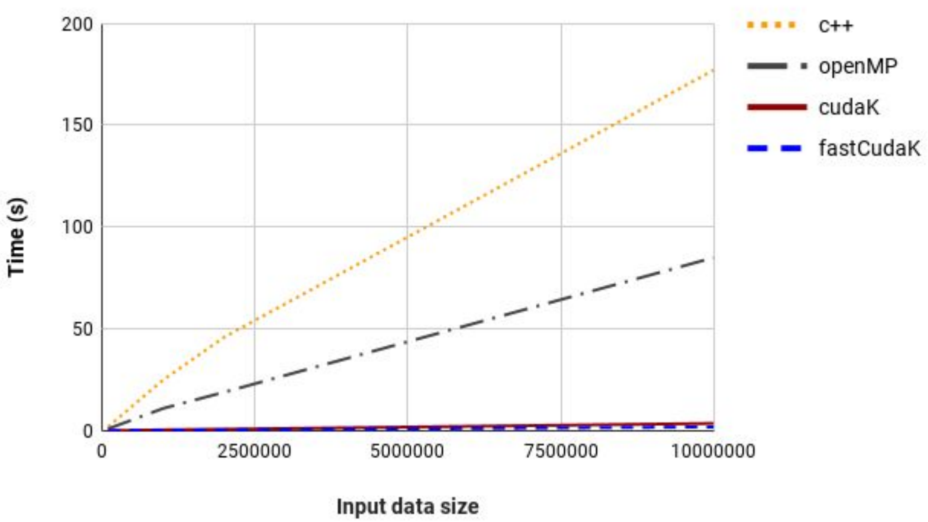
- All versions compared respect to input data size with 2 clusters and 200 iterations

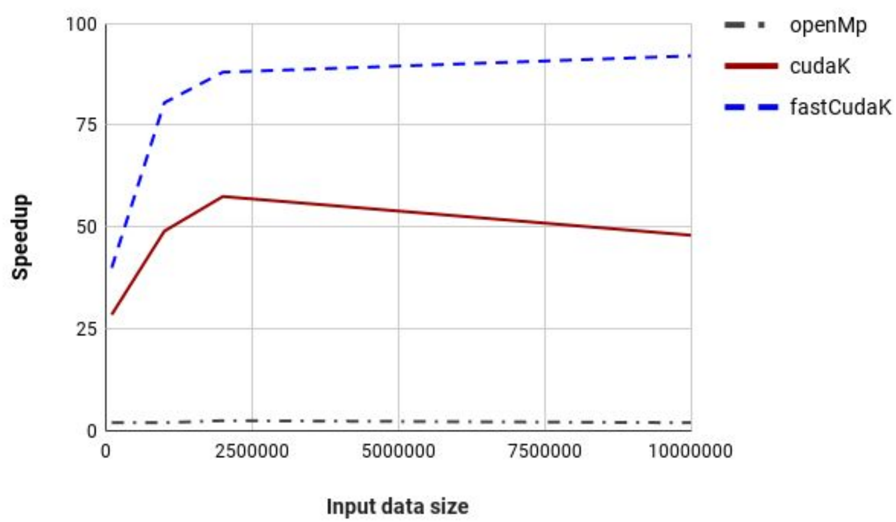




Results

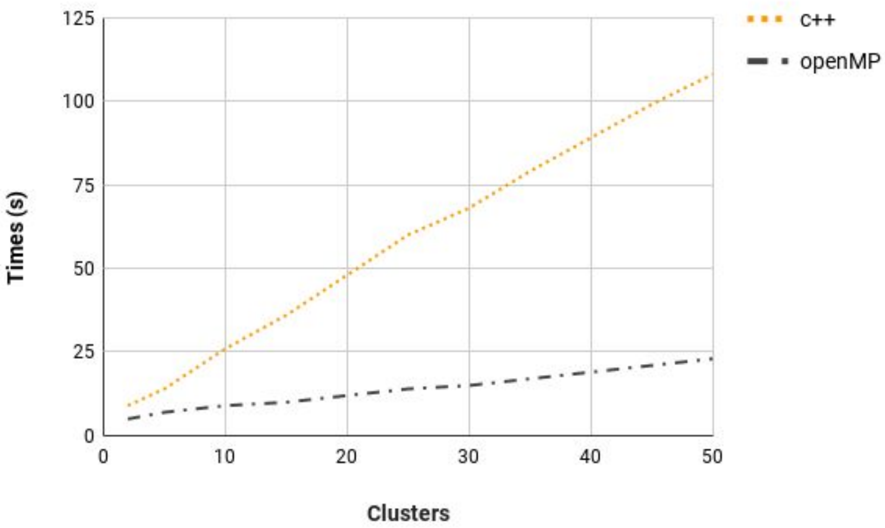
- All versions compared respect to input data size with 7 clusters and 200 iterations

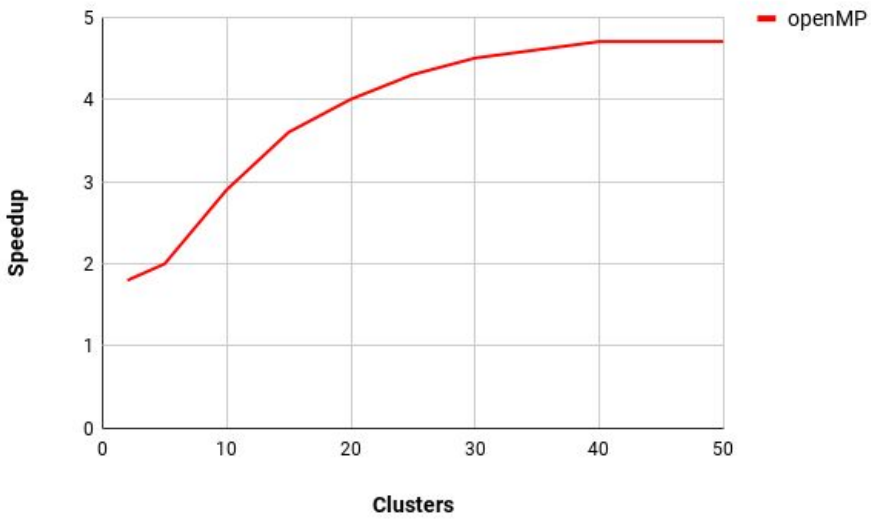




Results

- C++ Vs OpenMP with 10 millions pixels compared respect to the number of clusters changes





Conclusions

- Understand CUDA performance characteristics
- Use peak performance metrics to guide optimization
- Understand parallel algorithm complexity theory, identify type of bottleneck e.g. memory, core computation, or instruction overhead
- Optimize the algorithm, then unroll loops
- Use template parameters to generate optimal code

Thanks for the attention!