Color Quantization with Parallelized K-means

Team 23
107062228 陳劭愷
107060002 曹立元

Outline

- Introduction
- Problem Formulation
- Experimental Results
- Conclusion
- Implementation

Outline

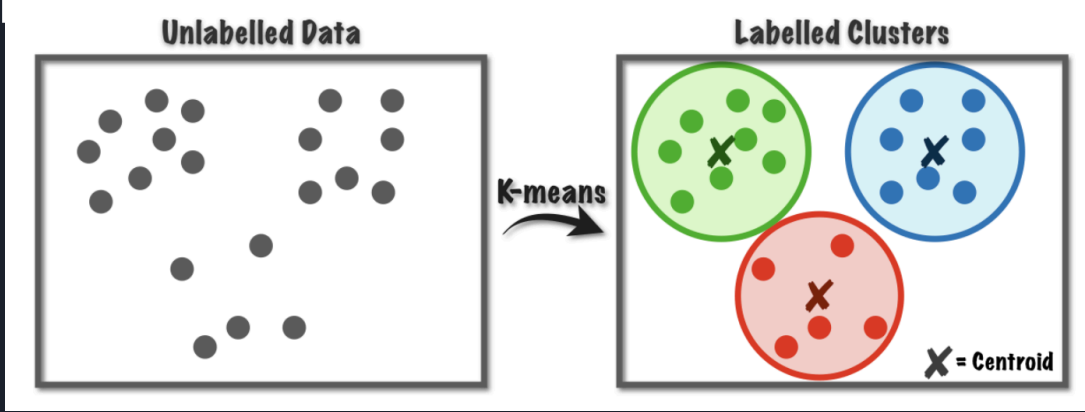
- Introduction
- Problem Formulation
- Experimental Results
- Conclusion
- Implementation

Introduction

- K-means Algorithm
- Color Quantization

K-means

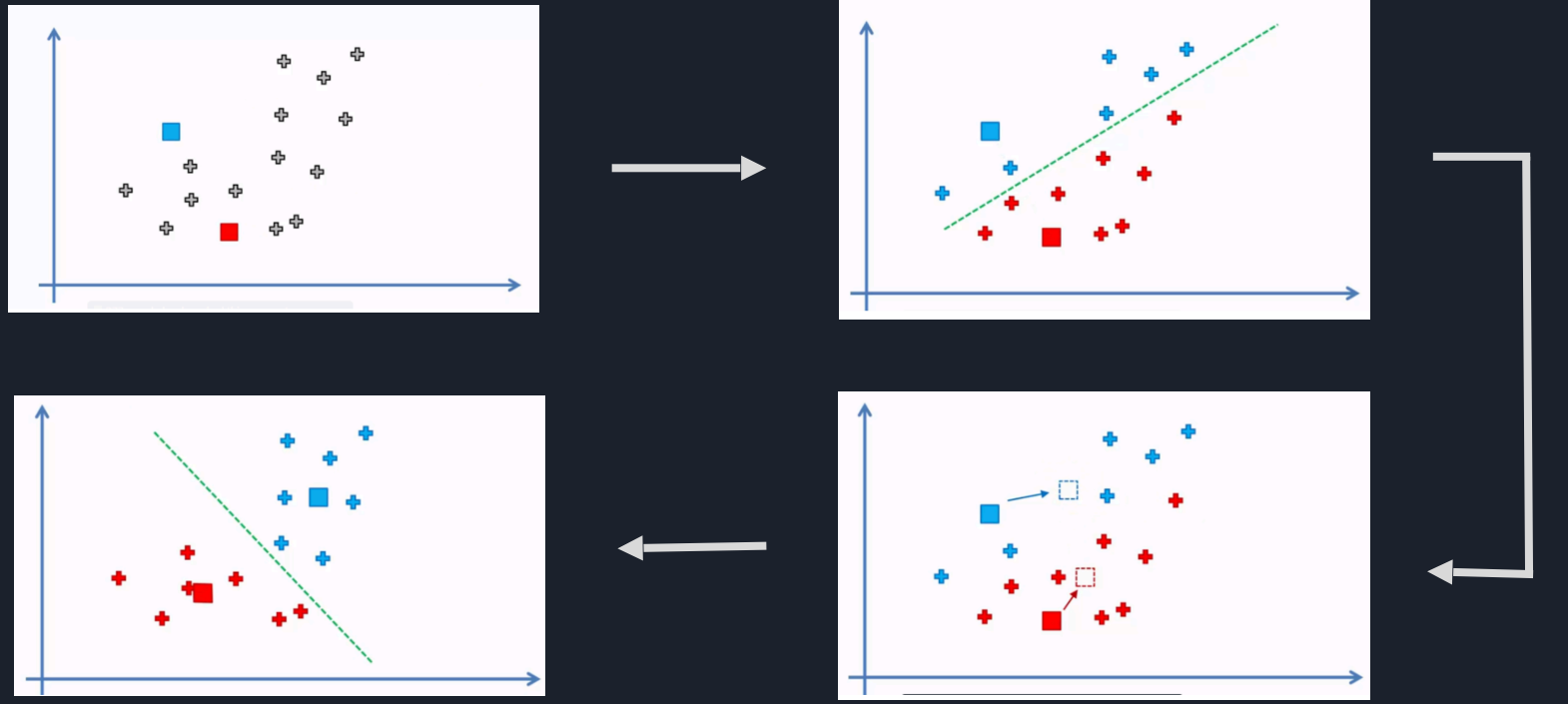
- Clustering Algorithm
- Widely used in Machine Learning
- Market / Image segmentation、Recommandation Engine



K-means

- Algorithm
  1. Decide the number of clusters (k clusters)
  2. Randomly initialize k centroids in feature space
  3. Each data point is assigned to the cluster of its nearest centroid based on the Euclidean distance
  4. For each cluster, update the centroid by taking the mean of data points assigned to this cluster
  5. Repeat step 3 and 4 until it converges

K-means Visualization



Color Quantization

- Famous Computer Vision algorithm
- Image Compression with little information loss

Original Image

After color quantization (k=2)

Outline

- Introduction
- Problem Definition
- Experimental Results
- Conclusion
- Implementation

K-means

Abundant data makes n large
=> parallelize n !!

- Algorithm & Analysis
  1. Decide the number of clusters (k clusters)
  2. Randomly assign k centroids in feature space
  3. Each data point is assigned to the cluster of its nearest centroid based on the Euclidean distance
  4. For each cluster, update the centroid by taking the mean of data points assigned to this cluster
  5. Repeat step 3 and 4 until it converges

O(nkr)

Outline

- Introduction
- Problem Formulation
- Experimental Results
- Conclusion
- Implementation

Experiment Results

- Statistical Results
  - Performance of sequential / OpenMP / CUDA
  - Speedup of multithreading
  - Issue of multithreading
  - CUDA Optimization
  - Profiling Results
  - Image compression of Color Quantization
- Qualitative Results

Experiment Results

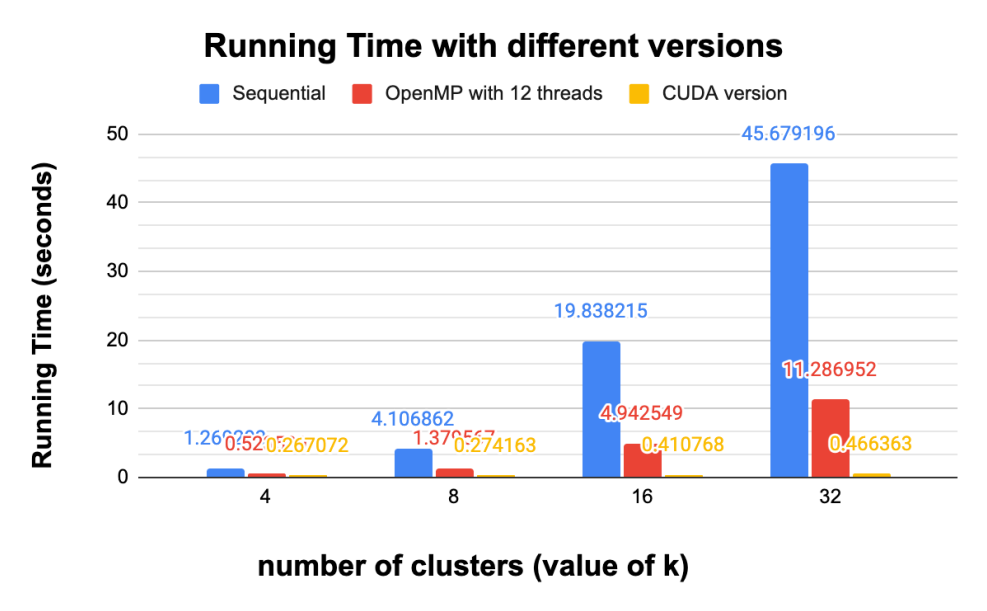
- Statistical Results
  - Performance of sequential / OpenMP / CUDA
  - Speedup of multithreading
  - Issue of multithreading
  - CUDA Optimization
  - Profiling Results
  - Image compression of Color Quantization
- Qualitative Results

Experiment Results

- Experiment Setting
  - 3 test images with diferent image size
  - Sequential / OpenMP / CUDA
  - Fixed random seed
  - Tested on apollo / hades

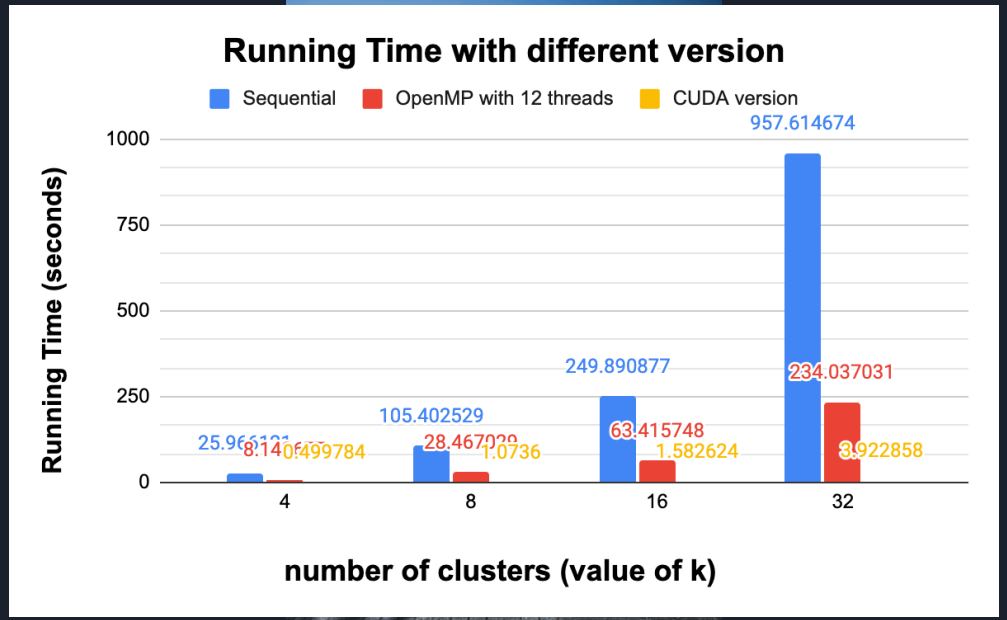
Statistical Result

- First test image (resolution = 692 x 1025, size = 1.38MB)



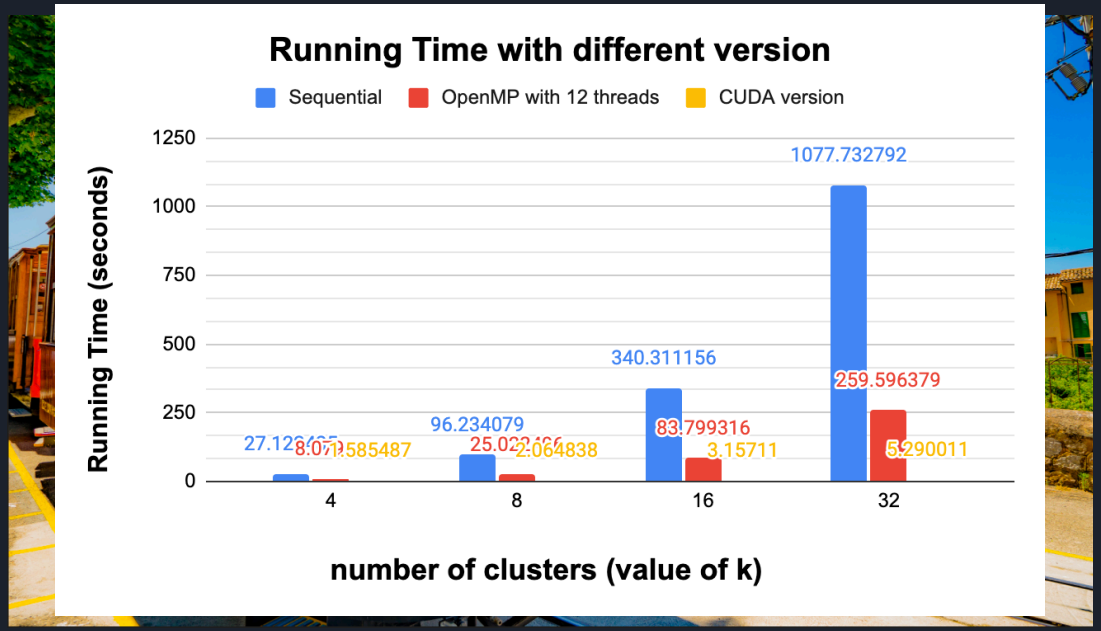
Statistical Result

- Second test image (resolution = 4000x6000, size = 25.88MB)



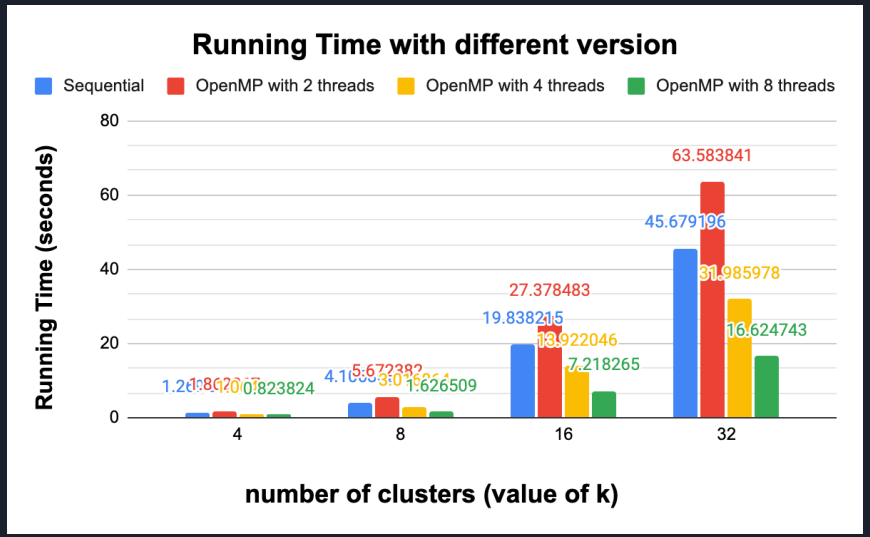
Statistical Result

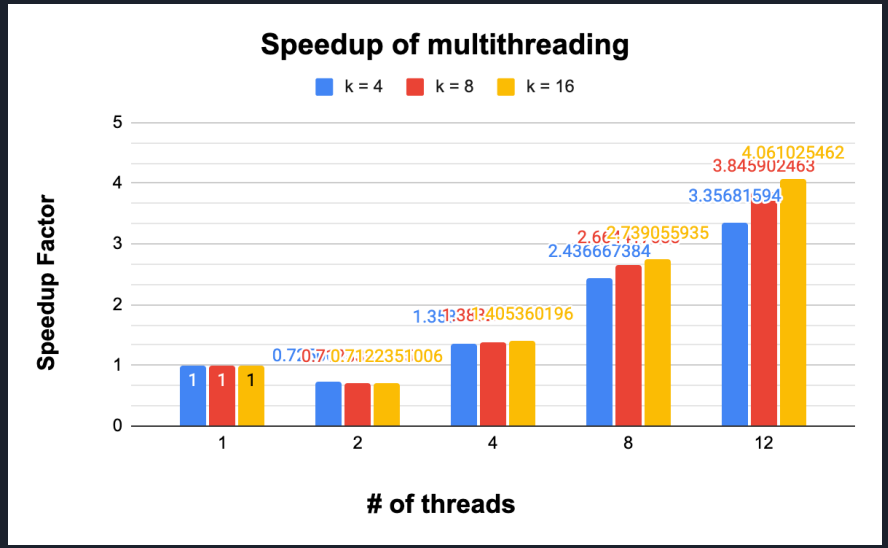
- Third test image (resolution = 3376x6000, size = 30.16MB)



Statistical Result

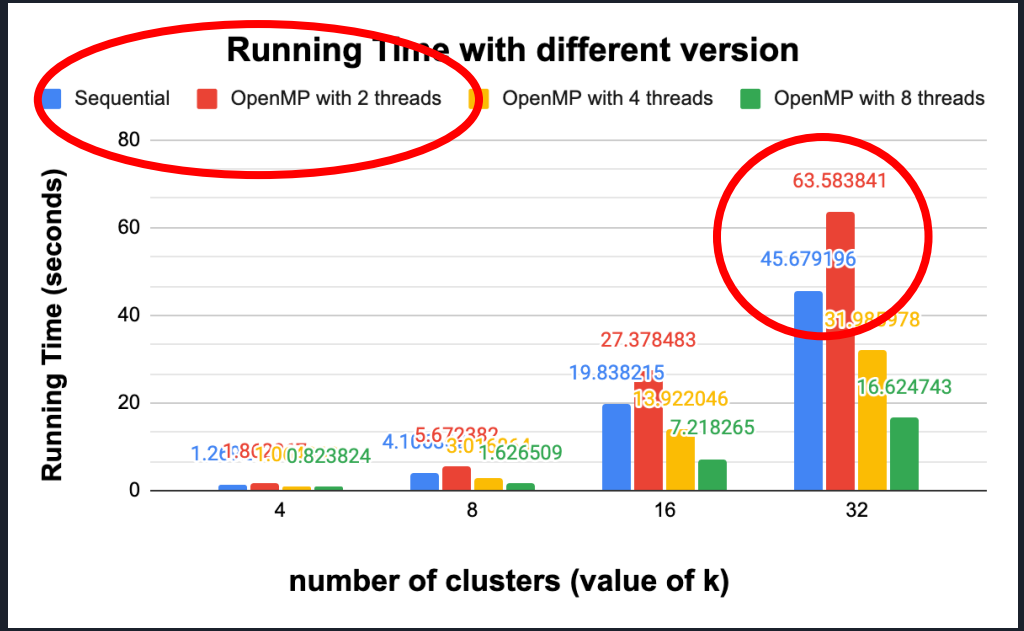
- Speedup of multithreading (tested on the largest test image)





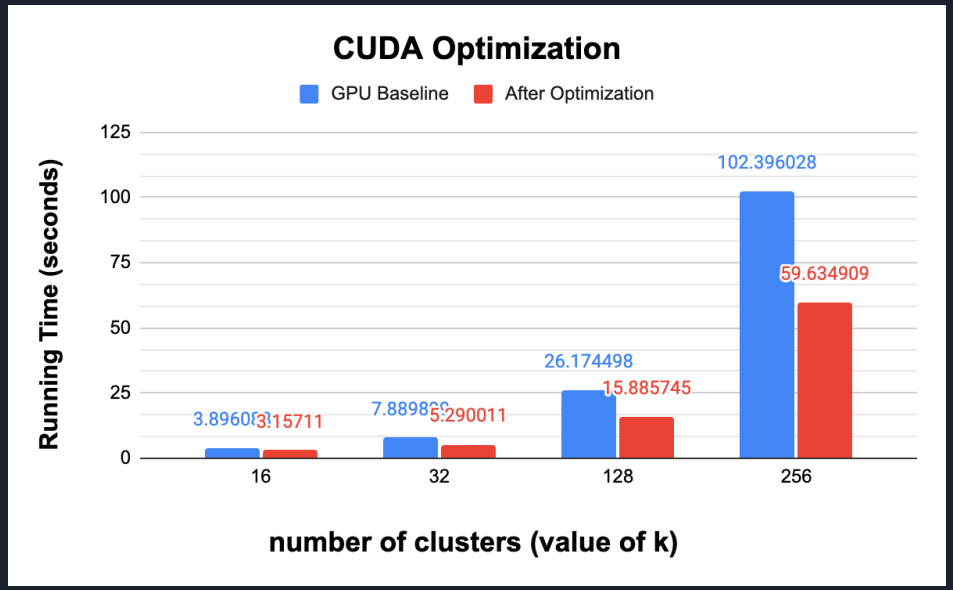
Statistical Result

- Increase computation time for better scalability



Statistical Result

- CUDA Optimization (tested on the largest test image)
  - Add shared memory, optimize reduce function

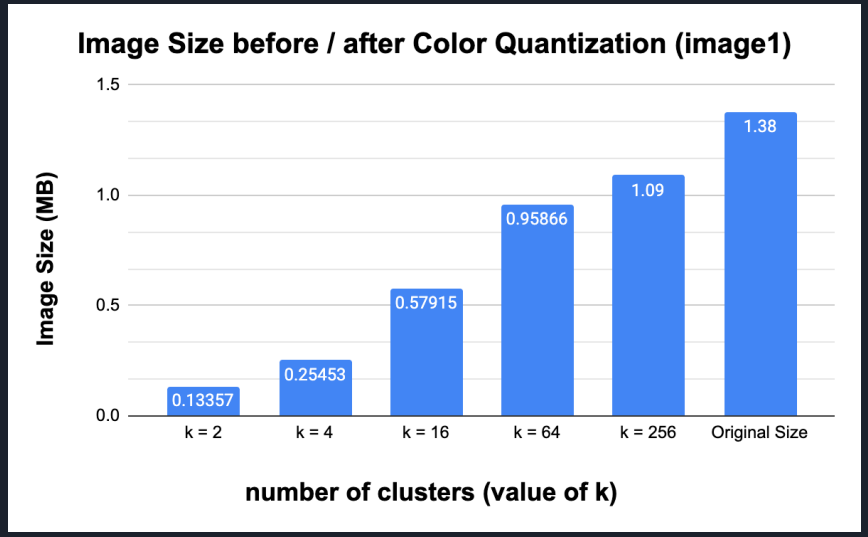


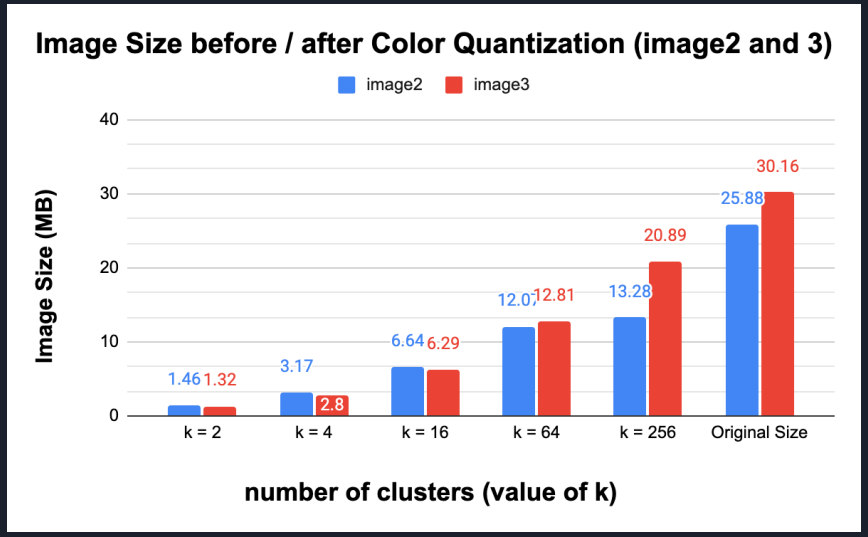
Statistical Result

- Profiling Results
  - Achieved Occupancy: 0.876
  - SM Efficiency: 99.79%
  - Global Load Throughput: 5.6863 GB/s
  - Global Store Throughput: 36.251 GB/s
  - Integer GOPS: 30.592

Statistical Result

- Image Compression by Color Quantization

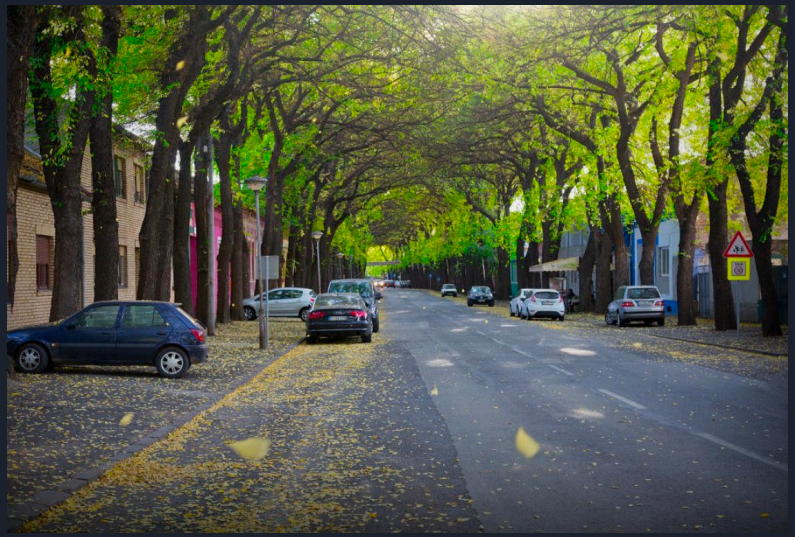




Experiment Results

- Statistical Results
  - Performance of sequential / OpenMP / CUDA
  - Speedup of multithreading
  - Issue of multithreading
  - CUDA Optimization
  - Profiling Results
  - Image compression of Color Quantization
- Qualitative Results

Qualitative Result



Original Image

Outline

- Introduction
- Problem Formulation
- Experimental Results
- Conclusion
- Implementation

Conclusion

- Successfully implement color quantization with parallelized k-means algorithm
- Implemented OpenMP / CUDA version
- About 200x speedup with our largest testing data by CUDA optimization

Implementation

```
void clustering(int n, int k, Point* points, Cluster* clusters) {
  for (int i = 0; i < k; i++) {
    clusters[i].size = 0;
  }

  #pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    int min_dist = 0x7fffffff;
    int cluster;

    for (int j = 0; j < k; ++j) {
      int dist = (
        (points[i].x - clusters[j].x) * (points[i].x - clusters[j].x) +
        (points[i].y - clusters[j].y) * (points[i].y - clusters[j].y) +
        (points[i].z - clusters[j].z) * (points[i].z - clusters[j].z)
      );

      if (min_dist > dist) {
        min_dist = dist;
        cluster = j;
      }
    }

    points[i].cluster = cluster;

    #pragma omp critical
    clusters[cluster].points[clusters[cluster].size++] = i;
  }
}
```

```
bool recentering(int n, int k, Point* points, Cluster* clusters) {
  bool done = true;

  for (int i = 0; i < k; ++i) {
    Cluster* cluster = &clusters[i];

    int sumX = 0;
    int sumY = 0;
    int sumZ = 0;

    #pragma omp parallel for reduction(+:sumX, sumY, sumZ)
    for (int j = 0; j < cluster->size; ++j) {
      Point* point = &points[cluster->points[j]];
      sumX += point->x;
      sumY += point->y;
      sumZ += point->z;
    }

    if (cluster->size > 0) {
      Cluster old = clusters[i];
      Cluster *cur = &clusters[i];

      cur->x = sumX / cluster->size;
      cur->y = sumY / cluster->size;
      cur->z = sumZ / cluster->size;

      if (old.x != cur->x || old.y != cur->y || old.z != cur->z) {
        done = false;
      }
    }
  }
```

Implementation

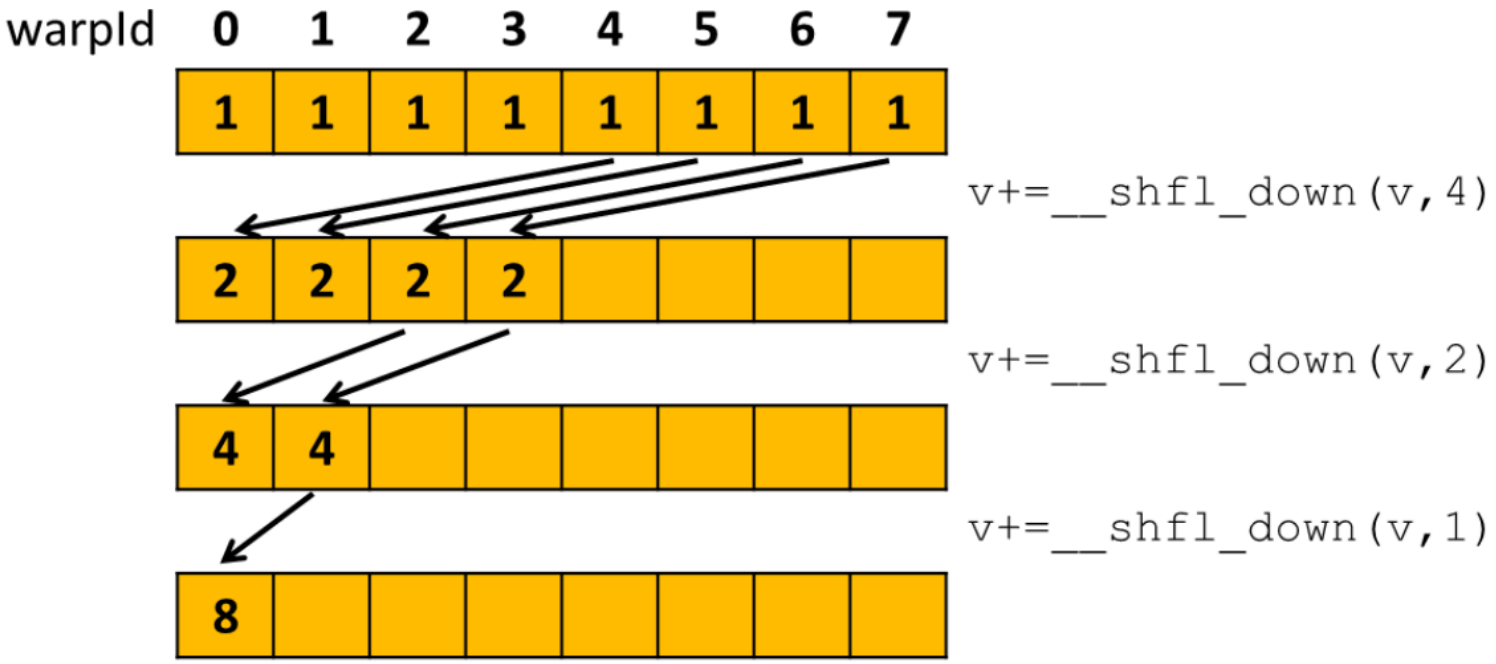
```
void clustering(int n, int k, Point* points, Cluster* clusters) {
  for (int i = 0; i < k; i++) {
    clusters[i].points.assign(clusters[i].points.size(), false);
  }

  #pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    int min_dist = 0x7fffffff;
    int cluster;

    for (int j = 0; j < k; ++j) {
      int dist = (
        (points[i].x - clusters[j].x) * (points[i].x - clusters[j].x) +
        (points[i].y - clusters[j].y) * (points[i].y - clusters[j].y) +
        (points[i].z - clusters[j].z) * (points[i].z - clusters[j].z)
      );

      if (min_dist > dist) {
        min_dist = dist;
        cluster = j;
      }
    }

    points[i].cluster = cluster;
    clusters[cluster].points[i] = true;
  }
}
```

```
bool recentering(int n, int k, Point* points, Cluster* clusters) {
  bool done = true;

  for (int i = 0; i < k; ++i) {
    int sumX = 0;
    int sumY = 0;
    int sumZ = 0;
    int count = 0;

    #pragma omp parallel for reduction(+:sumX, sumY, sumZ, count)
    for (int j = 0; j < n; ++j) {
      Point* point = &points[j];

      bool exist = clusters[i].points[j];
      sumX += point->x * exist;
      sumY += point->y * exist;
      sumZ += point->z * exist;
      count += exist;
    }

    if (count > 0) {
      Cluster old = clusters[i];
      Cluster *cur = &clusters[i];

      cur->x = sumX / count;
      cur->y = sumY / count;
      cur->z = sumZ / count;

      if (old.x != cur->x || old.y != cur->y || old.z != cur->z) {
        done = false;
      }
    }
  }

  return done;
}
```

Implementation

- Reduce sum optimization by Warp Reduce Sum strategy.
- Optimize for Pointer Aliasing & Read Only Data Cache.
- Shared memory.

Implementation



Reference

- https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/
- https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/
- https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Thank You !!