Parallel Programming Final Project

Parallel K-Means

team 29: 朱育欣 蘇勇誠

Outline

- Introduction
- Implementation
- Result
- Evaluation

Outline

- Introduction
- Implementation
- Result
- Evaluation

Image Segmentation Using Kmeans

Origin Image

k = 10

Kmeans Algorithm

1. Randomly initialize the cluster centers, denoted as c1, ..., ck.
2. For each point p, find the closest center ci and assign p to cluster i.
3. Given the points in each cluster, update center ci to be the mean of all points in cluster i.
4. Repeat step 2 and 3 until coverage.

Kmeans Algorithm

1. Randomly initialize the cluster centers, denoted as c1, ..., ck.
2. For each point p, find the closest center ci and assign p to cluster i.
3. Given the points in each cluster, update center ci to be the mean of all points in cluster i.
4. Repeat step 2 and 3 until coverage.

Parallel Step 2 and 3

Outline

- Introduction
- Implementation
- Result
- Evaluation

Implementation (omp)

1. omp critical v.s. omp atomic read / write

   OMP atomic read/write operations are faster than OMP critical sections, but the speed is similar to the non-parallel execution.

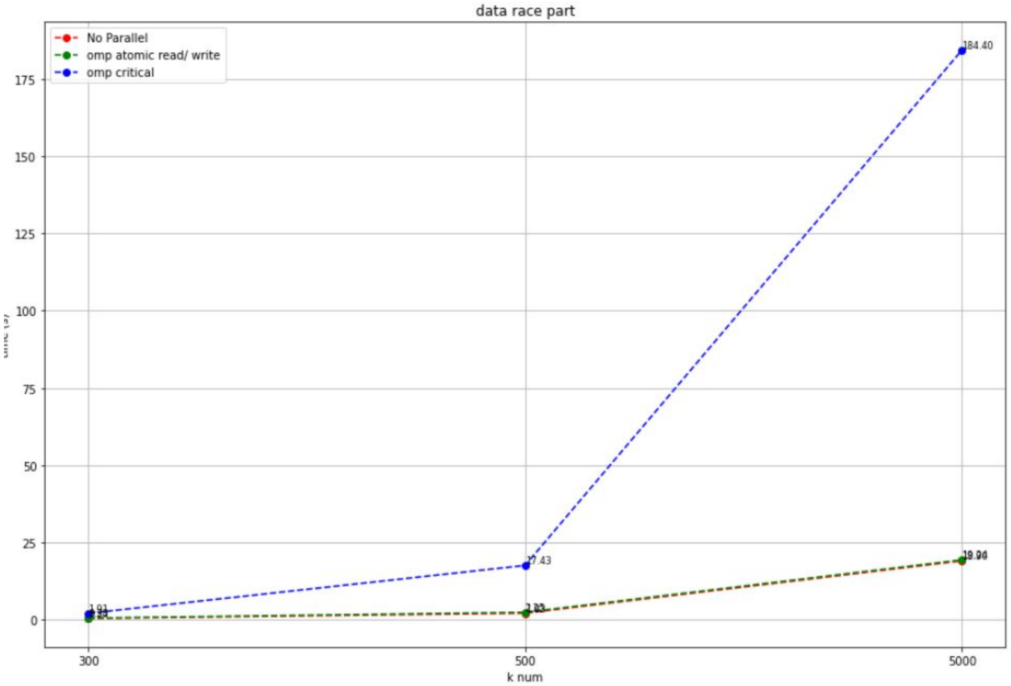
```
#pragma omp parallel for
for(int k = 0; k < num_cluster; k++) {
    dist = 0;
    dist += (val[0] - centroid[channels * k + 0]) * (val[0] - centroid[channels * k + 0]);
    dist += (val[1] - centroid[channels * k + 1]) * (val[1] - centroid[channels * k + 1]);
    dist += (val[2] - centroid[channels * k + 2]) * (val[2] - centroid[channels * k + 2]);
    dist = sqrt(dist);
    // version - 1
    #pragma omp critical
    if(dist < min_dist) {
        min_dist = dist;
        idx = k;
    }
}
pt_cluster[j + i * width] = idx;
```

```
#pragma omp parallel for
for(int k = 0; k < num_cluster; k++) {
    dist = 0;
    dist += (val[0] - centroid[channels * k + 0]) * (val[0] - centroid[channels * k + 0]);
    dist += (val[1] - centroid[channels * k + 1]) * (val[1] - centroid[channels * k + 1]);
    dist += (val[2] - centroid[channels * k + 2]) * (val[2] - centroid[channels * k + 2]);
    dist = sqrt(dist);
    // version - 2
    double temp_min_dist;
    #pragma omp atomic read
    temp_min_dist = min_dist;

    if (dist < temp_min_dist) {
        #pragma omp atomic write
        min_dist = dist;

        #pragma omp atomic write
        idx = k;
    }
}
pt_cluster[j + i * width] = idx;
```

Implementation (omp)



k = [300, 500, 5000]

no_parallel time = [0.25410, 1.93049, 18.89725]

critical time = [1.91280,17.42810, 184.40213]

atomic read/ write time = [0.34066, 2.25136, 19.24094]

OMP + SSE

```c
#pragma omp parallel for collapse(2) num_threads(threadNum)
for(int i = 0; i < height; i++) {
    for(int j = 0; j < width; j++) {
        __m128i val_r = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 0] ) ;
        __m128i val_g = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 1] ) ;
        __m128i val_b = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 2] ) ;

        int min_dist = 1000000;
        int idx = 0;

        for(int k = 0; k < num_cluster; k+=4) {

            __m128i centroid_r = _mm_set_epi32 ( (int)centroid[(channels * k )+9], (int)centroid[(channels * k )+6], (int)centroid[(channels * k )+3], (int)centroid[(channels * k )+0]);
            __m128i centroid_g = _mm_set_epi32 ( (int)centroid[ (channels * k )+10], (int)centroid[ (channels * k )+7], (int)centroid[(channels * k )+4], (int)centroid[  (channels * k )+1] ) ;
            __m128i centroid_b = _mm_set_epi32 ( (int)centroid[(channels * k )+11], (int)centroid[ (channels * k )+8], (int)centroid[(channels * k )+5], (int)centroid[ (channels * k )+2] ) ;

            __m128i result_sub_r = _mm_sub_epi32(val_r, centroid_r);
            __m128i result_square_r = _mm_mullo_epi32(result_sub_r, result_sub_r);
            __m128i result_sub_g = _mm_sub_epi32(val_g, centroid_g);
            __m128i result_square_g = _mm_mullo_epi32(result_sub_g, result_sub_g);
            __m128i result_sub_b = _mm_sub_epi32(val_b, centroid_b);
            __m128i result_square_b = _mm_mullo_epi32(result_sub_b, result_sub_b);

            __m128i dist_128 = _mm_add_epi32(result_square_r, result_square_g);
            dist_128 = _mm_add_epi32(dist_128, result_square_b);

            __m128 float_dist_128 = _mm_cvtepi32_ps(dist_128);
            __m128 sqrt_dist_128 = _mm_sqrt_ps(float_dist_128);
            __m128i int_sqrt_result = _mm_cvtps_epi32(sqrt_dist_128);

            int dist[4];
            _mm_storeu_si128((__m128i*)dist, int_sqrt_result);

            for (int ii = 0; k+ii < num_cluster && ii <4 ; ++ii) {
                if (dist[ii] < min_dist) {
                    min_dist = dist[ii];
                    idx = k+ii;
                }
            }
        }
        pt_cluster[j + i * width] = idx;
    }
}
```

1. _mm_set1_epi32

   change unsigned char into int and load pixel data into 128 bits _m128i data type

OMP + SSE

```c
#pragma omp parallel for collapse(2) num_threads(threadNum)
for(int i = 0; i < height; i++) {
    for(int j = 0; j < width; j++) {
        __m128i val_r = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 0] ) ;
        __m128i val_g = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 1] ) ;
        __m128i val_b = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 2] ) ;

        int min_dist = 1000000;
        int idx = 0;

        for(int k = 0; k < num_cluster; k+=4) {

            __m128i centroid_r = _mm_set_epi32 ( (int)centroid[(channels * k )+9], (int)centroid[(channels * k )+6], (int)centroid[(channels * k )+3], (int)centroid[(channels * k )+0]);
            __m128i centroid_g = _mm_set_epi32 ( (int)centroid[ (channels * k )+10], (int)centroid[ (channels * k )+7], (int)centroid[(channels * k )+4], (int)centroid[  (channels * k )+1] ) ;
            __m128i centroid_b = _mm_set_epi32 ( (int)centroid[(channels * k )+11], (int)centroid[ (channels * k )+8], (int)centroid[(channels * k )+5], (int)centroid[ (channels * k )+2] ) ;

            __m128i result_sub_r = _mm_sub_epi32(val_r, centroid_r);
            __m128i result_square_r = _mm_mullo_epi32(result_sub_r, result_sub_r);
            __m128i result_sub_g = _mm_sub_epi32(val_g, centroid_g);
            __m128i result_square_g = _mm_mullo_epi32(result_sub_g, result_sub_g);
            __m128i result_sub_b = _mm_sub_epi32(val_b, centroid_b);
            __m128i result_square_b = _mm_mullo_epi32(result_sub_b, result_sub_b);

            __m128i dist_128 = _mm_add_epi32(result_square_r, result_square_g);
            dist_128 = _mm_add_epi32(dist_128, result_square_b);

            __m128 float_dist_128 = _mm_cvtepi32_ps(dist_128);
            __m128 sqrt_dist_128 = _mm_sqrt_ps(float_dist_128);
            __m128i int_sqrt_result = _mm_cvtps_epi32(sqrt_dist_128);

            int dist[4];
            _mm_storeu_si128((__m128i*)dist, int_sqrt_result);

            for (int ii = 0; k+ii < num_cluster && ii <4 ; ++ii) {
                if (dist[ii] < min_dist) {
                    min_dist = dist[ii];
                    idx = k+ii;
                }
            }
        }
        pt_cluster[j + i * width] = idx;
    }
}
```

1. _mm_set1_ep32

   load centroid data to compute 4 clusters simultaneously.

   centroid_r= r_1, r_2, r_3, r_4

   centroid_g= g_1, g_2, g_3, g_4

   centroid_b= b_1, b_2, b_3, b_4

OMP + SSE

```c
#pragma omp parallel for collapse(2) num_threads(threadNum)
for(int i = 0; i < height; i++) {
    for(int j = 0; j < width; j++) {
        __m128i val_r = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 0] ) ;
        __m128i val_g = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 1] ) ;
        __m128i val_b = _mm_set1_epi32 ( (int)image_src[channels * (j + i * width) + 2] ) ;

        int min_dist = 1000000;
        int idx = 0;

        for(int k = 0; k < num_cluster; k+=4) {

            __m128i centroid_r = _mm_set_epi32 ( (int)centroid[(channels * k )+9], (int)centroid[(channels * k )+6], (int)centroid[(channels * k )+3], (int)centroid[(channels * k )+0]);
            __m128i centroid_g = _mm_set_epi32 ( (int)centroid[ (channels * k )+10], (int)centroid[ (channels * k )+7], (int)centroid[(channels * k )+4], (int)centroid[  (channels * k )+1] ) ;
            __m128i centroid_b = _mm_set_epi32 ( (int)centroid[(channels * k )+11], (int)centroid[ (channels * k )+8], (int)centroid[(channels * k )+5], (int)centroid[ (channels * k )+2] ) ;

            __m128i result_sub_r = _mm_sub_epi32(val_r, centroid_r);
            __m128i result_square_r = _mm_mullo_epi32(result_sub_r, result_sub_r);
            __m128i result_sub_g = _mm_sub_epi32(val_g, centroid_g);
            __m128i result_square_g = _mm_mullo_epi32(result_sub_g, result_sub_g);
            __m128i result_sub_b = _mm_sub_epi32(val_b, centroid_b);
            __m128i result_square_b = _mm_mullo_epi32(result_sub_b, result_sub_b);

            __m128i dist_128 = _mm_add_epi32(result_square_r, result_square_g);
            dist_128 = _mm_add_epi32(dist_128, result_square_b);

            __m128 float_dist_128 = _mm_cvtepi32_ps(dist_128);
            __m128 sqrt_dist_128 = _mm_sqrt_ps(float_dist_128);
            __m128i int_sqrt_result = _mm_cvtps_epi32(sqrt_dist_128);

            int dist[4];
            _mm_storeu_si128((__m128i*)dist, int_sqrt_result);

            for (int ii = 0; k+ii < num_cluster && ii <4 ; ++ii) {
                if (dist[ii] < min_dist) {
                    min_dist = dist[ii];
                    idx = k+ii;
                }
            }
        }
        pt_cluster[j + i * width] = idx;
    }
}
```

1. _mm_sub_epi32
2. _mm_mullo_epi32
3. _mm_add_epi32
4. _mmsqrt_ps

MPI

```
#pragma omp parallel for collapse(2) num_threads(threadNum)
for(int i = rank; i < height; i+=size) {
    for(int j = 0; j < width; j++) {
        unsigned char val[3];
        val[0] = image_src[channels * (j + i * width) + 0];
        val[1] = image_src[channels * (j + i * width) + 1];
        val[2] = image_src[channels * (j + i * width) + 2];
        int min_dist = 1000000;
        int dist, idx;
        for(int k = 0; k < num_cluster; k++) {
            dist = 0;
            dist += (val[0] - centroid[channels * k + 0]) * (val[0] - centroid[channels * k + 0]);
            dist += (val[1] - centroid[channels * k + 1]) * (val[1] - centroid[channels * k + 1]);
            dist += (val[2] - centroid[channels * k + 2]) * (val[2] - centroid[channels * k + 2]);
            dist = sqrt(dist);
            if(dist < min_dist) {
                min_dist = dist;
                idx = k;
            }
        }
        pt_cluster[j + i * width] = idx;
    }
}
```

data segmemtation: splitting based on rows, each process is allocated rows with an interval size equal to size.

MPI

sum up diastasnce and count point number in cluster.

```
for(int i = rank; i < height; i+=size) {
    for(int j = 0; j < width; j++) {
        idx = pt_cluster[j + i * width]; // get cluster id
        num_pt_cluster[idx] += 1;
        sum_dist[idx * channels + 0] += image_src[channels * (j + i * width) + 0];
        sum_dist[idx * channels + 1] += image_src[channels * (j + i * width) + 1];
        sum_dist[idx * channels + 2] += image_src[channels * (j + i * width) + 2];
    }
}
MPI_Allreduce(MPI_IN_PLACE, sum_dist, channels * num_cluster, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, num_pt_cluster, num_cluster, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

MPI

```
#pragma omp parallel for num_threads(threadNum) reduction(+:sum_val)
for(int i = rank; i < num_cluster; i+=size) {
    int dist = 0;
    dist += (new_centroid[channels * i + 0] - centroid[channels * i + 0]) * (new_centroid[channels * i + 0] - centroid[channels * i + 0]);
    dist += (new_centroid[channels * i + 1] - centroid[channels * i + 1]) * (new_centroid[channels * i + 1] - centroid[channels * i + 1]);
    dist += (new_centroid[channels * i + 2] - centroid[channels * i + 2]) * (new_centroid[channels * i + 2] - centroid[channels * i + 2]);

    sum_val += sqrt(dist);
    centroid[channels * i + 0] = new_centroid[channels * i + 0];
    centroid[channels * i + 1] = new_centroid[channels * i + 1];
    centroid[channels * i + 2] = new_centroid[channels * i + 2];
}
for (size_t process = 0; process < size; process++)
{
    if(process != rank){
        MPI_Send(centroid, channels*num_cluster, MPI_CHAR, process, UPDATE_CENTROID_TAG, MPI_COMM_WORLD);
    }
}
MPI_Barrier(MPI_COMM_WORLD);
for (size_t process = 0; process < size-1; process++)
{
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    int MPI_SOURCE = status.MPI_SOURCE;
    int MPI_TAG = status.MPI_TAG;
    MPI_Recv( tmp_centroid ,channels* num_cluster, MPI_CHAR, MPI_SOURCE, UPDATE_CENTROID_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (size_t i = MPI_SOURCE; i < num_cluster; i+=size)
    {
        centroid[channels * i + 0] = tmp_centroid[channels * i + 0];
        centroid[channels * i + 1] = tmp_centroid[channels * i + 1];
        centroid[channels * i + 2] = tmp_centroid[channels * i + 2];
    }
}
MPI_Allreduce(MPI_IN_PLACE, &sum_val, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

1. new_centroid assign to centroid

2. send assigned new_centroid to other processes

3. wait for other processes to send data

Optimization (CUDA)

1. Shared Memory

2. Reduce Bank Conflict

3. Reduce Global Atomic Operation

Optimization (CUDA) - Reduce Global Atomic Operation

```
atomicAdd(&num_pt_cluster[cluster_idx], 1);
atomicAdd(&sum_dist[cluster_idx * channels + 0], img_src0);
atomicAdd(&sum_dist[cluster_idx * channels + 1], img_src1);
atomicAdd(&sum_dist[cluster_idx * channels + 2], img_src2);
```

```
atomicAdd(&shared_num_pt_cluster[cluster_idx], 1);
atomicAdd(&shared_sum_dist[cluster_idx][0], img_src0);
atomicAdd(&shared_sum_dist[cluster_idx][1], img_src1);
atomicAdd(&shared_sum_dist[cluster_idx][2], img_src2);

for(int k = (threadIdx.x + threadIdx.y * blockDim.x); k < num_cluster; k += (blockDim.x * blockDim.y)) {
    atomicAdd(&num_pt_cluster[k], shared_num_pt_cluster[k]);
    atomicAdd(&sum_dist[k * channels + 0], shared_sum_dist[k][0]);
    atomicAdd(&sum_dist[k * channels + 1], shared_sum_dist[k][0]);
    atomicAdd(&sum_dist[k * channels + 2], shared_sum_dist[k][0]);
}
```

Image Segmentation Result

Origin Image

k = 10

Image Segmentation Result

k = 50

k = 100

Image Segmentation Result

k = 500

k = 1000

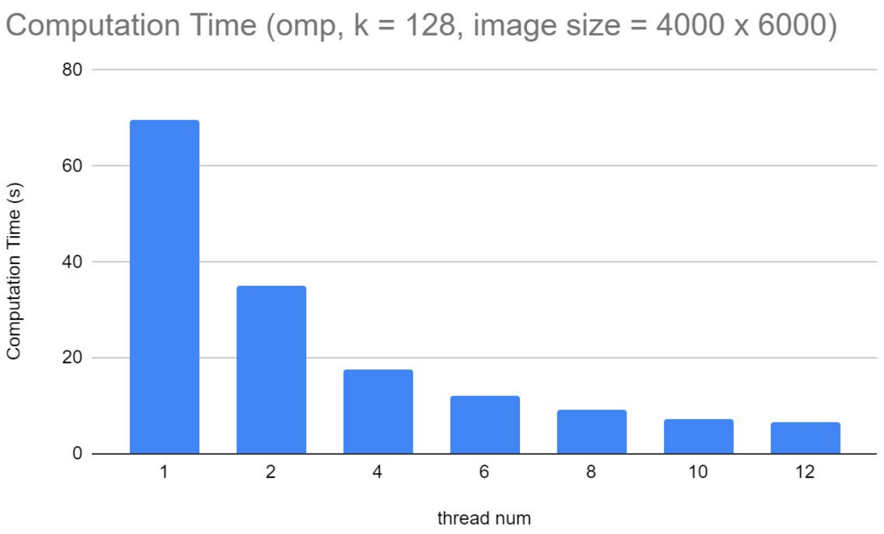
Outline

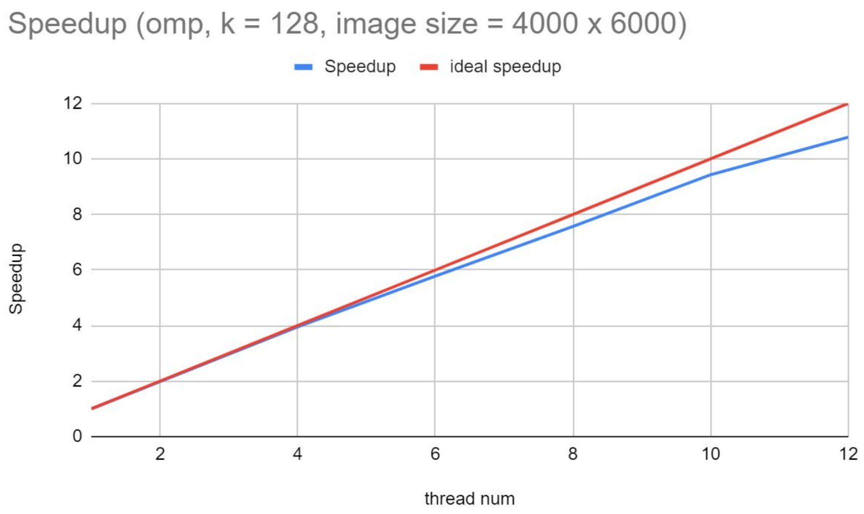
- Introduction
- Implementation
- Result
- Evaluation

Evaluation Data

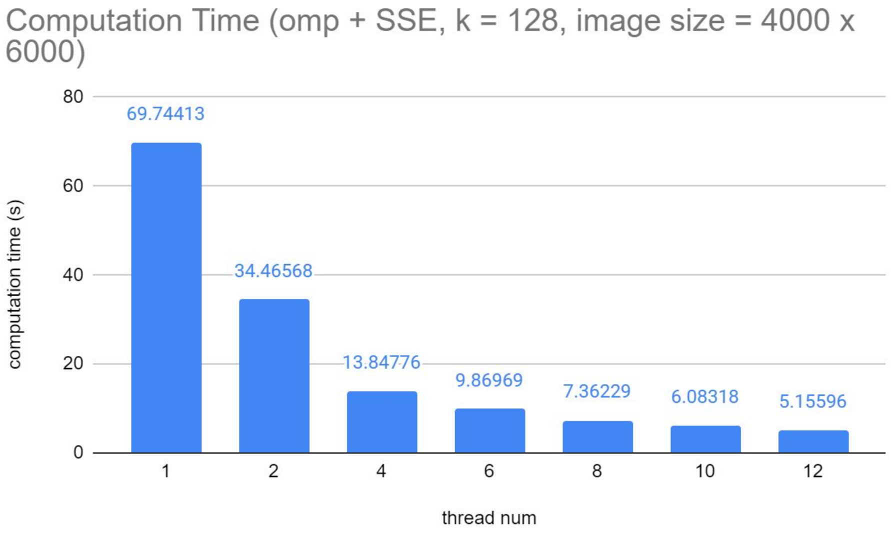
height: 4000, width: 6000

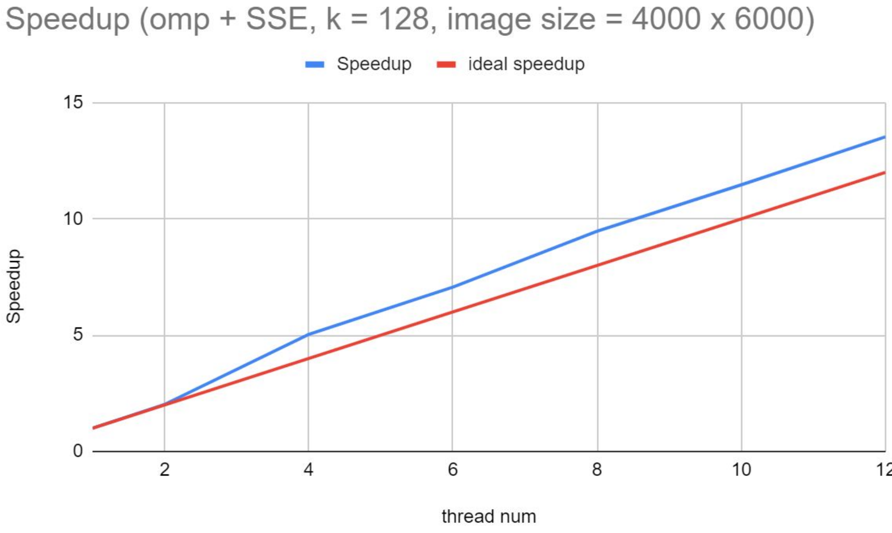
Single Node Multi-core - OMP (k = 128)



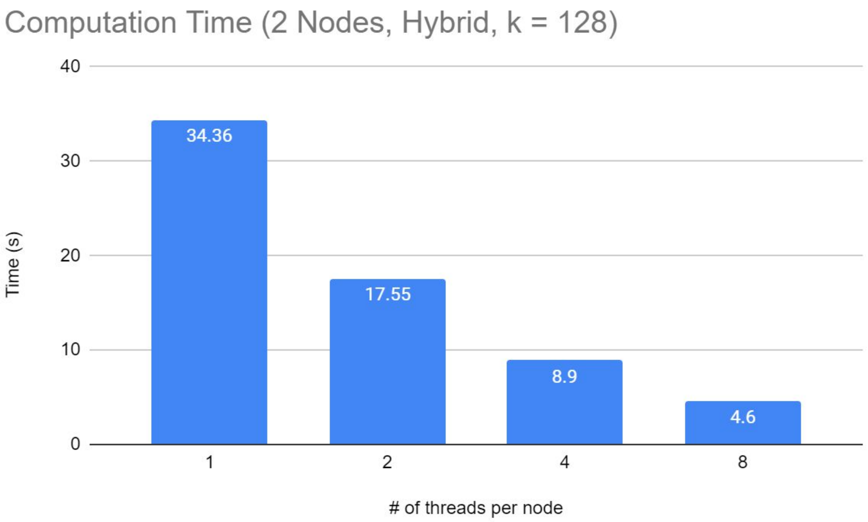


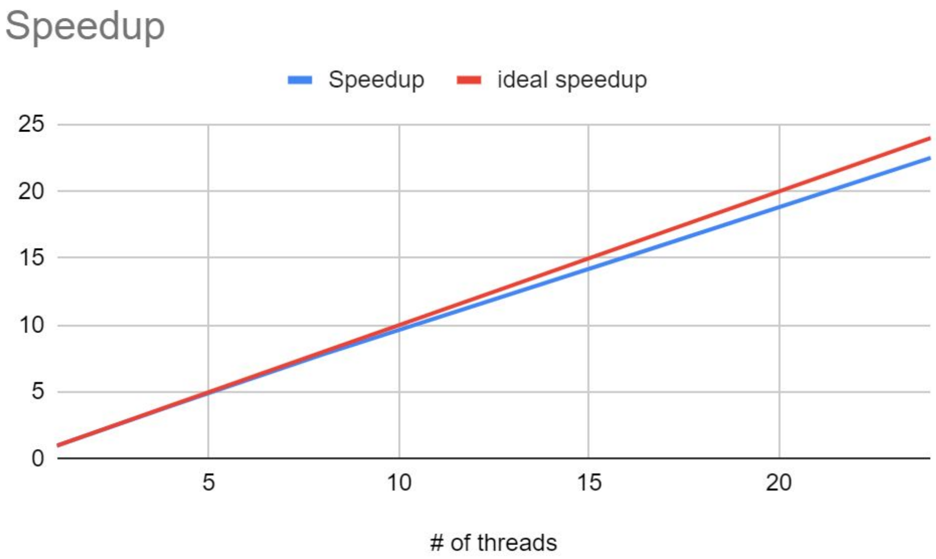
Single Node Multi-core - OMP + SSE (k=128)





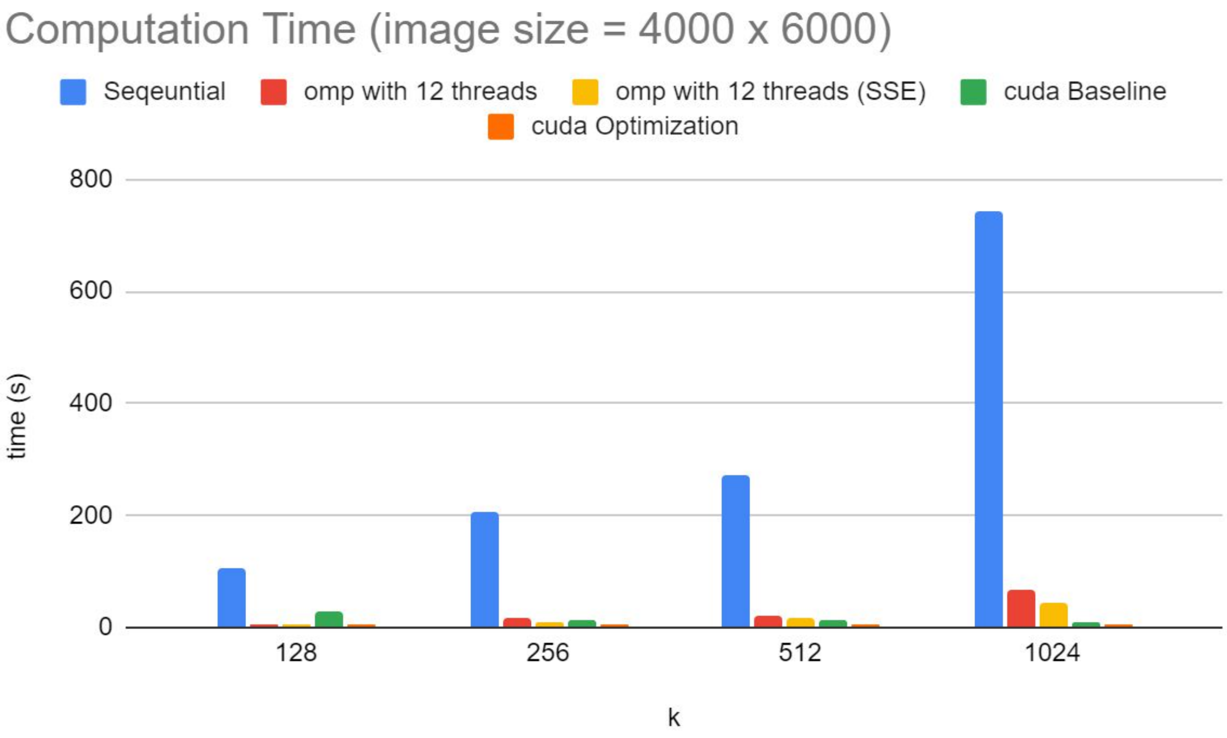
Multiple Node Multi-core - hybrid (MPI+ OMP) (k = 128)





- Experiment using 2 nodes.
- Each node create one process.
- Each process create multiple threads.

Computation Time



Computation Time

Compare to Seqeuntial Code

- omp with 12 threads achieve 12x speedup
- omp with 12 threads and SSE achieve 14.x speedup
- cuda Baseline achieve 20x speedup
- cuda Optimization achieve 60x speedup