CUDA - K-means

Anand Kamble
Department of Scientific Computing
Florida State University

1 Introduction

In this homework, we will be implementing the k-means algorithm for image segmentation using CUDA.

2 Implementation

2.1 Host-side Initialization

The host program reads the input image file (in this case, a JPEG file) using the provided JpegFile library, obtaining the image width, height, and pixel data in RGB format.

Listing 1: main.cu

```
UINT width, height;
uint8_t *hostDataBuf = JpegFile::JpegFileToRGB("test-large.jpg", &width,
    &height);
```

2.2 Device Memory Allocation

The host program allocates device memory for the input RGB data and the output segmented data using `cudaMalloc`.

Listing 2: main.cu

```
cudaMalloc(&device_colors, width * height * sizeof(Pixel));
```

2.3 Data Transfer to Device

The input RGB data is copied from the host memory to the device memory using `cudaMemcpy`.

Listing 3: main.cu

```
cudaMemcpy(device_colors, hostDataBuf, width * height * sizeof(Pixel),
    cudaMemcpyHostToDevice);
```

2.4 Kernel Launch

The k-means algorithm is implemented using three CUDA kernels.

2.4.1 Grouping Kernel

This kernel is responsible for grouping each pixel to the nearest generator (cluster center) based on the Euclidean distance

Listing 4: main.cu

```
__global__ void groupingKernel(UINT width, UINT height, int k, Pixel *colors,
    Pixel *generators, int *groupColorSum, int *groupCount){
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

```

```
    if (x < width && y < height){
        int index = y * width + x;
        Pixel pixel = colors[index];
        float minDist = INFINITY;
        int minIndex = 0;

        for (int i = 0; i < k; i++){
            Pixel gen = generators[i];
            float dist = sqrtf((pixel.r - gen.r) * (pixel.r - gen.r) +
                (pixel.g - gen.g) * (pixel.g - gen.g) + (pixel.b - gen.b) *
                (pixel.b - gen.b));
            if (dist < minDist){
                minDist = dist;
                minIndex = i;
            }
        }
        atomicAdd(&groupColorSum[minIndex * 3 + 0], (int)pixel.r);
        atomicAdd(&groupColorSum[minIndex * 3 + 1], (int)pixel.g);
        atomicAdd(&groupColorSum[minIndex * 3 + 2], (int)pixel.b);
        atomicAdd(&groupCount[minIndex], 1);
    }
}
```

2.4.2 Update Generators Kernel

This kernel is used to update the generators (cluster centers) based on the group color sums and counts.

Listing 5: main.cu

```
__global__ void updateGeneratorsKernel(int k, Pixel *generators, int
    *groupColorSum, int *groupCount)
{
    int i = threadIdx.x;
    if (i < k)
    {
        generators[i].r = (int)groupColorSum[i * 3 + 0] / groupCount[i];
        generators[i].g = (int)groupColorSum[i * 3 + 1] / groupCount[i];
        generators[i].b = (int)groupColorSum[i * 3 + 2] / groupCount[i];
    }
}
```

2.4.3 Replace Color Kernel

This kernel is used to replace the color of each pixel with the color of the group generator it belongs to, effectively segmenting the image.

Listing 6: main.cu

```
__global__ void replaceColorKernel(UINT width, UINT height, int k, Pixel
    *colors, Pixel *generators, int *groupCount)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < width && y < height)
    {
        int index = y * width + x;
        Pixel pixel = colors[index];

        float minDist = INFINITY;
        int minIndex = 0;

        for (int i = 0; i < k; i++)
```

```
        {
            Pixel gen = generators[i];
            float dist = sqrtf((pixel.r - gen.r) * (pixel.r - gen.r) +
                (pixel.g - gen.g) * (pixel.g - gen.g) + (pixel.b - gen.b) *
                (pixel.b - gen.b));
            if (dist < minDist)
            {
                minDist = dist;
                minIndex = i;
            }
        }

        colors[index] = generators[minIndex];
    }
}
```

2.5 Data Transfer to Host

After the kernel executions, the resulting segmented image data is copied from the device memory back to the host memory using `cudaMemcpy`.

Listing 7: main.cu

```
cudaMemcpy(hostDataBuf, device_colors, N * sizeof(Pixel),
    cudaMemcpyDeviceToHost);
```

2.6 Output File Writing

The host program writes the segmented image data to a new JPEG file using the JpegFile library.

Listing 8: main.cu

```
JpegFile::RGBToJpegFile("output.jpg", hostDataBuf, width, height, 100, false);
```

2.7 Memory Deallocation

The host program frees the allocated device and host memory.

Listing 9: main.cu

```
delete[] generators;
free(hostDataBuf);
cudaFree(device_colors);
cudaFree(device_generators);
cudaFree(device_groupColorSum);
cudaFree(device_groupCount);
```

2.8 Benchmarking

We are timing the program using the `cudaEventRecord` function provided by CUDA.

Listing 10: main.cu

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start); // Start recording
// ... processing part
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float elapsedTime; // Variable to store the elapsed time
cudaEventElapsedTime(&elapsedTime, start, stop);
```

3 Results

After successful execution, we get the following results:

	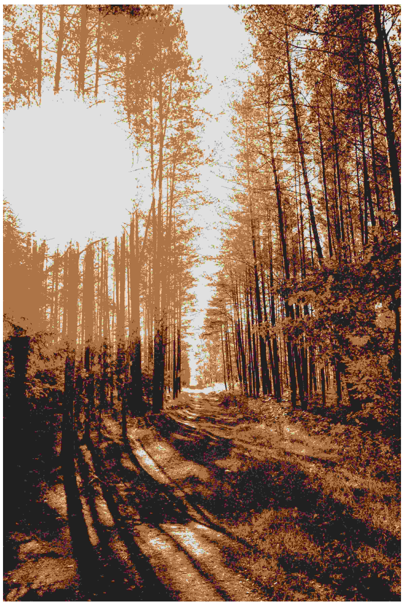	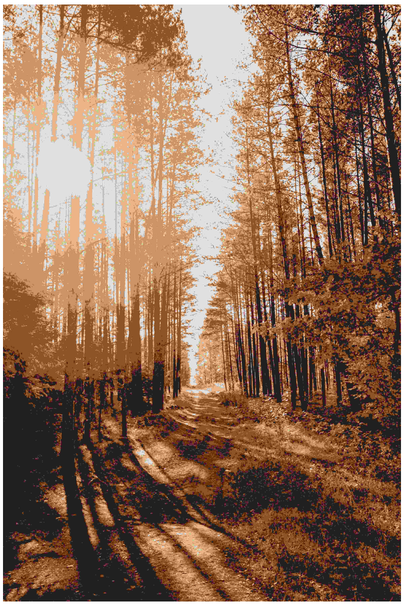
K = 2	K = 3	K = 5

Table 1: K-means

The following chart shows the time taken by CPU compared to the GPU.

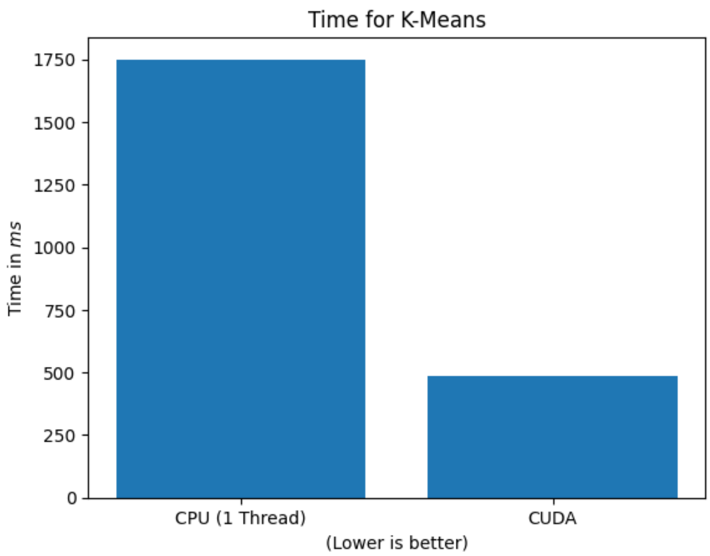

Figure 1: Caption

References

[1] NVIDIA Corporation. *K-Means Clustering Algorithm* , `https://www.nvidia.com/en-us/glossary/k-means/`. Oct 11, 2017.

[2] Renan. *Difference between cuda.h, cuda_runtime.h, cuda_runtime_api.h* , `https://stackoverflow.com/questions/6302695/difference-between-cuda-h-cuda-runtime-h-cuda-runtime-api-h`. Jun 10, 2011.