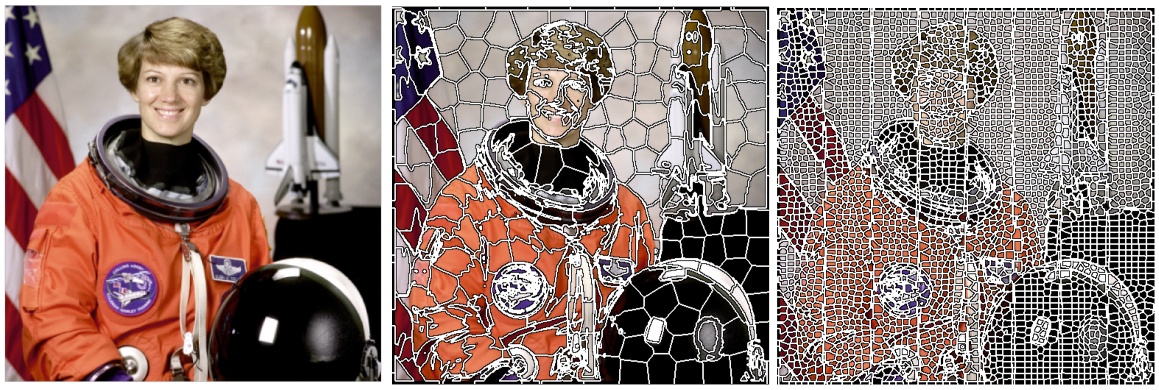
CS 7080 High Performance Computing
Homework 6
CUDA - SuperPixel

Haoxiang Zhang
PowPrint: hzny2

Due: December 12, 2021

1 Homework Description

homework 6, we need to use CUDA on Superpixel algorithm from homework 2 first part. In the Fig.1, it shows how the SLIC Superpixels alogrithm works. Once the code could run successfully, we will make the code parallelized with GPU threads.



(a) Input image (b) Output image - 256 superpixels (c) Output image - 4096 superpixels

Figure 1: SLIC Superpixels Algorithm Example

2 Homework Details

Why GPU?

Processing data in a GPU or a CPU is handled by cores. The more cores a processing unit has, the faster (and potentially more efficiently) a computer can complete tasks. GPUs use thousands of cores to process tasks in parallel. The parallel structure of the GPU is different than that of the CPU, which uses fewer cores to process tasks sequentially. This would be the reason to choose using GPU.

CUDA

For CUDA programming, we have to understand the details about CUDA. CUDA programming has HOST and DEVICE which host is the CPU and its memory and which device is the GPU and its memory. The basic processing flow is:

- Copy input data from CPU memory to GPU memory
- Load GPU program and execute caching data on chip for performance
- Copy results from GPU memory to CPU memory

GPU programming is kind of like a multi-threading programming. GPU has many SM(Stream Multiprocessors). There is a CUDA grid. This grid has multiple CUDA Blocks - multiprocessing units. Those blocks are broken up into CUDA threads. The information below is about how to get the value of the the blocks and threads:

- blockDim.x, blockDim.y, blockDim.z - To get a block's size
- blockIdx.x, blockIdx.y, blockIdx.z - To get a block's position
- threadIdx.x, threadIdx.y, threadIdx.z - To get a thread's position in a block

Therefore, we can easily to find the value at any of the position. The code below shows how to find the the value of the position.

```
// position in a block (local position)
const int xx = threadIdx.x;
const int yy = threadIdx.y;
const int zz = threadIdx.z;

// position in grid (global position)
const int x = blockDim.x * blockIdx.x + xx;
const int y = blockDim.y * blockIdx.y + yy;
const int z = blockDim.z * blockIdx.z + zz;

// position in 2 dimension space
const int 2d_pos = y * blockDim.x + x;

// position in 3 dimension space
const int 3d_pos = z * (blockDim.y * blockDim.x) + y * blockDim.x + x;
```

For the kernel function, if there is a **__global__** the beginning of the function, this means this function will be run in GPU. In the example code below, we can found that there is a <<<**grid, block**>>> when call the kernel function. The grid is telling the GPU for how many blocks you need; and block is telling the GPU for how many threads in each block.

```
#define BLOCK_WIDTH 16
#define BLOCK_HEIGHT 16
// kernel function
__global__ void kernel_gpu(unsigned char* input_img, unsigned char* output_img, int img_w, int img_h)

int main () {
    int width;
    int height;
    const dim3 grid  (grid_divide(width, BLOCK_WIDTH), grid_divide(height, BLOCK_HEIGHT), 1);
    const dim3 block (BLOCK_WIDTH, BLOCK_HEIGHT, 1);

    // to call the kernel function
    kernel_gpu<<<grid, block>>>(d_img, d_out, width, height);
}
```

How to define the size of the grid and the block

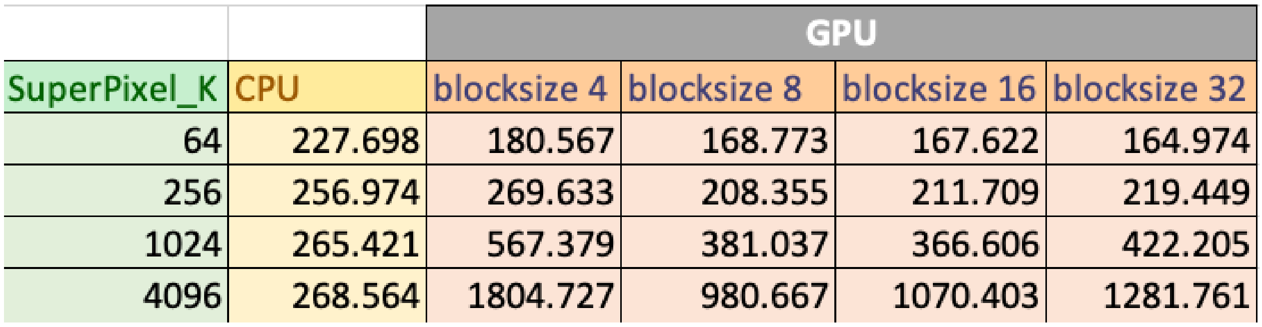
For Homework 6, I did parallelism on computing distance and getting the labels. Therefore, I setup that each thread will get the label for each pixel. For the given image - **Astronaught.png**, the image size is $512 \times 512 \times 3$ - a RGB image. However, we only need to get the label mask which is $512 \times 512 \times 1$. Usually set the **BlockWidth** and **BlockHeight** as 16. Then, `[image_width / BlockWidth, image_height / BlockHeight, 1]` will be the grid size. And `[BlockWidth, BlockHeight, 1]` will be the block size. This could make sure that each thread in blocks could deal with each pixel's label.

Implementation Tricks - Check the comment in below pseudo code area

Because I am setting the kernel function to get the new label for each pixel, I need to find all the superpixel range that the pixel within to find the closest superpixel. In Step.11, it shows some details about finding closest superpixel ID.

```
// 1. We need to add cuda.h to have the the CUDA Toolkit application programming interface.
#include <cuda.h>
#include <cuda_runtime.h>
// 2. Given Block size
#define BLOCK_WIDTH 16
#define BLOCK_HEIGHT 16
// kernel function
__global__ void parallelWork(int* lbl, double* kseedsl, double* kseedsa, double* kseedsb, double* kseedsx,
↪  double* kseedsy, double* vl, double* va, double* vb, double offset, double invwt, int width, int height,
↪  int numk)
{
    // 9. get the global position
    const int x = blockDim.x * blockIdx.x + threadIdx.x;
    const int y = blockDim.y * blockIdx.y + threadIdx.y;

    // 10. create a shared variable if need
    __shared__ double kdist[BLOCK_WIDTH*BLOCK_HEIGHT];

    // 11. find the label with min distance
    for ( n -> 0 to numk)
        findSuperPixelRange();
        if ( current pixel in the range )
            computeDistance();
            updateCloserDistance();
        fi
    endfor
    int label; // for closest superpixel ID.

    // 12. change the pixel value to the new label
    lbl[y * width + x] = label;
}

int main () {
    image = getimage();
    int width;
    int height;
    // 3. get the values for L A B channel
    double lvec[height*width], avec[height*width], bvec[height*width];
    // 4. get the kseeds using function GetLABXYSeeds()
    double kseedsl[numk], kseedsa[numk], kseedsb[numk], kseedsx[numk], kseedsy[numk];

    // 5. need the host and device memory space for all the kseeds and lab vectors
    double* d_kl = NULL;
    checkCudaErrors(cudaMalloc((void**) &d_kl,  (numk * sizeof(double))));
    // ...
    // 6. need a memory space for receiving labals from GPU and a space for label in GPU
    int* h_lbl = (int*)malloc(height * width * sizeof(int));
    int* d_lbl = NULL;
    checkCudaErrors(cudaMalloc((void**) &d_lbl, (height * width * sizeof(int))));

    // 7. now copy all the variables to GPU memory, cudaMemcpyHostToDevice is the direction of copy
    checkCudaErrors(cudaMemcpy(d_vl, &lvec, height * width * sizeof(double), cudaMemcpyHostToDevice));

    // 8. define the size of the grid and block
    const dim3 grid  (grid_divide(width, BLOCK_WIDTH), grid_divide(height, BLOCK_HEIGHT), 1);
    const dim3 block (BLOCK_WIDTH, BLOCK_HEIGHT, 1);

    // 13. to call the kernel function with grid size and block size.
    parallelWork<<<grid, block>>>(d_lbl, d_kl, d_ka, d_kb, d_kx, d_ky, d_vl, d_va, d_vb, offset, invwt,
    ↪  width, height, numk);

    // 14. after finish GPU-kernel function, copy the label back to host
    checkCudaErrors(cudaMemcpy(h_lbl, d_lbl, height * width * sizeof(int), cudaMemcpyDeviceToHost));

    // 15. update kseeds for find new label in next iteration
    updateKseeds();

    // 16. draw the contours and save image
    draw();
    save();
}
```

Timing Trends



		GPU			
SuperPixel_K	CPU	blocksize 4	blocksize 8	blocksize 16	blocksize 32
64	227.698	180.567	168.773	167.622	164.974
256	256.974	269.633	208.355	211.709	219.449
1024	265.421	567.379	381.037	366.606	422.205
4096	268.564	1804.727	980.667	1070.403	1281.761

Figure 2: Table for Timing Trends With Different Cases - 10 Iterations

In the Fig.2, we can found a problem here. GPU took more time on higher SuperPixel number, but there is no big difference for CPU. The reason I think it is because I set each thread in block doing update label, which means each thread would go K times for finding min distance. It cause the longer time. I think if we set each thread do one superpixel. Then it may get faster.

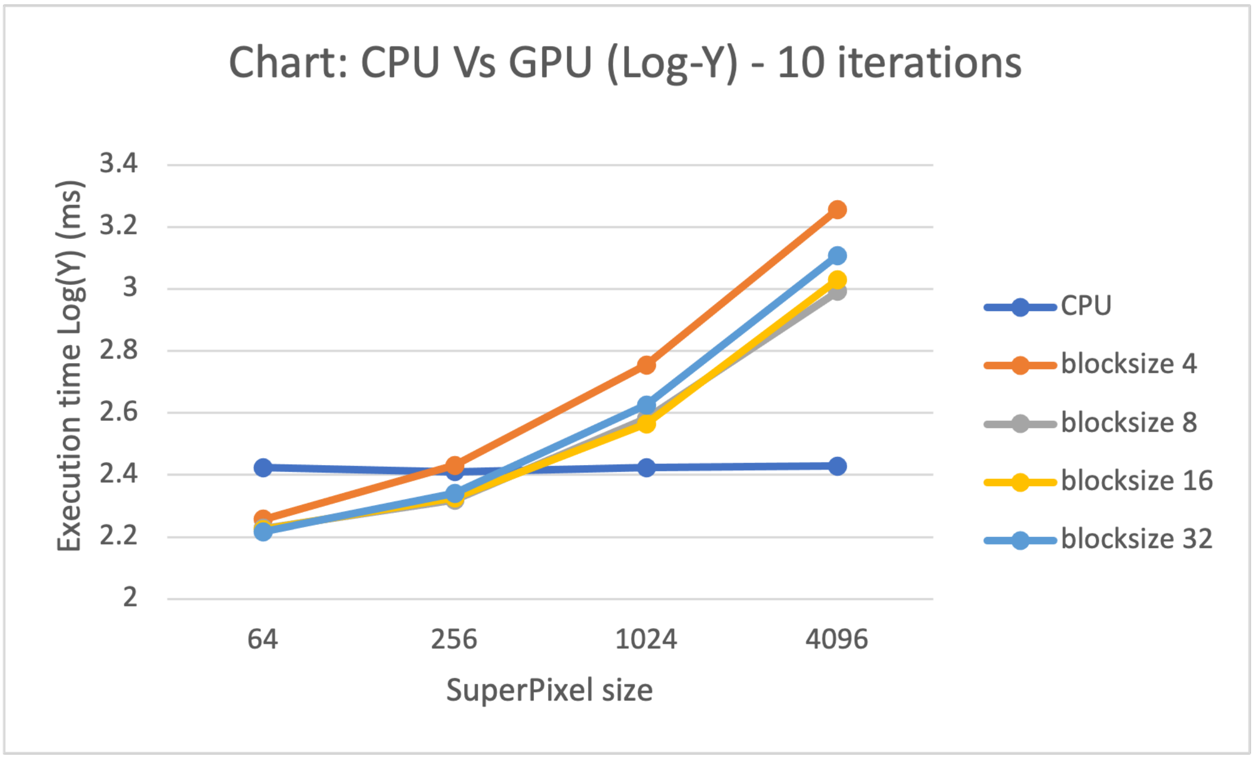


Figure 3: Chart for Timing Trends With Different Cases - 10 Iterations

For different Block Size, based on the time in Fig.3, we can found that the small block size cause more time. The reason I guess is because each block would do synchronization after all of the threads finish, which means more block number cause more time to synchronization. That's why smaller block size spend more time.

3 Lesson Learned

For homework 6, I have learned a lot of things about CUDA. First, when malloc a space, except the size of the data, also we need to times the size of the data type. Second, when define the size of the grid and block, it's better to draw a diagram to see which plan is better. Like this homework, I chose that threads update labels for each pixel. This cause the longer time than CPU did. Third, I have learned how to use the information like blockIdx, blockDim, and threadIdx to find the global postion of the input data. This would be the core thing for GPU programming. For the future work, I will do each thread do one SuperPixel and update the label rather than do one pixel and update the label.

4 Useful Links

- **Cooperative Groups: Flexible CUDA Thread Programming**, see:
  https://developer.nvidia.com/blog/cooperative-groups/
- **COOPERATIVE GROUPS (Slides)**, see:
  https://on-demand.gputechconf.com/gtc/2017/presentation/s7622-Kyrylo-perelygin-robust-and-scalable
  pdf
- **COOPERATIVE GROUPS (Video)**, see:
  https://vimeo.com/461821629
- **CUDA Runtime**, see:
  https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
- **CUDA C PROGRAMMING GUIDE**, see:
  https://nw.tsuda.ac.jp/lec/cuda/doc_v9_0/pdf/CUDA_C_Programming_Guide.pdf