Advaned Programming for HPC - Report 4

Dinh Anh Duc

November 4, 2021

Implementation

```
__global__ void grayscale2D(uchar3 *input, uchar3 *output, int width, int height) {
        int x = threadIdx.x + blockIdx.x + blockDim.x;
        int y = threadIdx.y + blockIdx.y + blockDim.y;
        if (x > width || y > height){
            printf("x, y are outside the width, height");
            return;
        }
        int tid = x + y * width;
            output[tid].x = (input[tid].x + input[tid].y + input[tid].z) / 3;
        output[tid].z = output[tid].y = output[tid].x;
}
void Labwork::labwork4_GPU() {
    // Calculate number of pixels
    int pixelCount = inputImage->width * inputImage->height;
    //char *hostInput = inputImage->buffer; // Perfect version
    char *hostInput = (char*) malloc(inputImage->width * inputImage->height * 3); // Test version
    char *hostOutput = new char[inputImage->width * inputImage->height * 3]; // Test version
    outputImage = static_cast<char *>(malloc(pixelCount * 3));
    for (int j = 0; j < 100; j++) {     // let's do it 100 times, otherwise it$
        # pragma omp parallel for
        for (int i = 0; i < pixelCount; i++) {
            outputImage[i * 3] = (char) (((int) inputImage->buffer[i * 3] +
            (int) inputImage->buffer[i * 3 + 1] + (int) inputImage->buffer[i * 3 + 2]) / 3);
            outputImage[i * 3 + 1] = outputImage[i * 3];
            outputImage[i * 3 + 2] = outputImage[i * 3];
        }
    }

    // Allocate CUDA memory
    uchar3 *devInput;
    uchar3 *devOutput;
    //cudaMalloc(&devInput, pixelCount*3); // Perfect version
    cudaMalloc(&devInput, pixelCount * sizeof(uchar3)); // Test version
    //cudaMalloc(&devOutput, pixelCount*3); // Perfect version
    cudaMalloc(&devOutput, pixelCount * sizeof(float)); // Test version

    // Copy CUDA Memory from CPU to GPU
    //cudaMemcpy(devInput, hostInput, pixelCount*3, cudaMemcpyHostToDevice); // Perfect version
    cudaMemcpy(devInput, hostInput, pixelCount * sizeof(uchar3), cudaMemcpyHostToDevice); // Test version


    // Processing
    dim3 blockSize = dim3(32, 32);
    dim3 gridSize = ((int) ((inputImage->width + blockSize.x - 1)/blockSize.x), (int)((inputImage->height
    grayscale2D<<<gridSize, blockSize>>>(devInput, devOutput, inputImage->width, inputImage->height);

    // Copy CUDA Memory from GPU to CPU
    //cudaMemcpy(outputImage, devOutput, pixelCount*3, cudaMemcpyDeviceToHost); // Perfect version
    cudaMemcpy(hostOutput, devOutput, pixelCount*sizeof(float), cudaMemcpyDeviceToHost); // Test version

    // Cleaning
    //free(hostInput);
    cudaFree(devInput);
```

```
	cudaFree(devOutput);
}
```

Result

Figure 1: Original input image

Figure 2: Output image

Exercise

Exercise 1:
Consider a GPU having the following specs (maximum numbers):

- 512 threads/block
- 1024 threads/SM
- 8 blocks/SM
- 32 threads/warp

What is the best configuration for thread blocks to implement grayscaling?

- 8 x 8
- 16 x 16
- 32 x 32

With 32 x 32 option:

- 32 x 32 = 1024 >512 threads/block (Out of limitation)
- 1024 x 8 blocks/SM = 8192 >1024 threads/SM (Out of limitation)

32 x 32 is not a good option
With 16 x 16 option:

- 16 x 16 = 256 <512 threads/block (Seem to be oke)
- 256 x 8 blocks/SM = 2048 >1024 threads/SM (Out of limitation)

16 x 16 is not a good option
With 8 x 8 option:

- 8 x 8 = 64 <512 threads/block (Seem to be oke)
- 64 x 8 blocks/SM = 512 <1024 threads/SM (Seem to be oke)

8 x 8 is not bad
Exercise 2:
Consider a device SM that can take max

- 1536 threads
- 4 blocks

Which of the following block configs would result in the most number of threads in the SM?

- 128 threads/blk
- 256 threads/blk
- 512 threads/blk
- 1024 threads/blk

With 128 threads/blk choice:

- 128 x 4 blocks = 512 threads in a SM (Not a bad option)

With 256 threads/blk choice:

- 256 x 4 blocks = 1024 threads in a SM (The best option)

With 512 threads/blk choice:

- 512 x 4 blocks = 2048 threads in a SM (Out of limitation)

With 1024 threads/blk choice:

- 1024 x 4 blocks = 4096 threads in a SM (Out of limitation)

Exercise 3:
Consider a vector addition problem

- Vector length is 2000
- Each thread produces one output
- Block size 512 threads

How many threads will be in the grid?
We need 2000 threads to execute 2000 sum opeations but we have maximum 512 threads in one block therefore we can use 4 blocks in a SM devices to calculate 2000 sum operations that means we will have 2048 threads in grid.
512 * 4 blocks/SM = 2048 threads (Not a bad option)