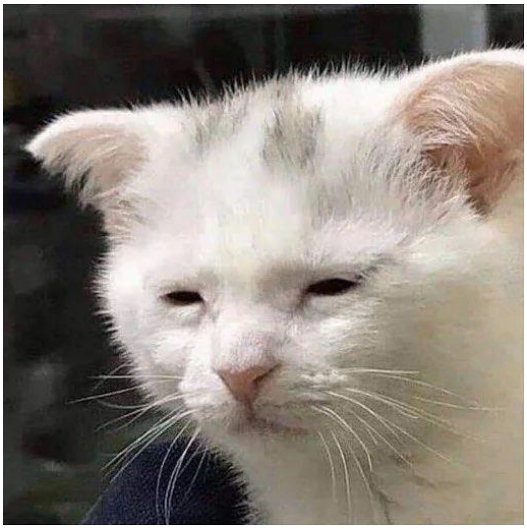
110062118
賴姿妘

HW3: Sobel



A. Optimization

(a) Serial code

First, I modified the serial version and verify the correctness. The change includes removing unnecessary computation such as *adjustX, adjustY, xBound, yBound,* using float type instead of double type, unroll the loop of two masks. The ablation result with 6.png as testcase is shown below.

improvement	**Time(s)**
origin	1.511562
double->float	1.486863
unroll mask loop	1.131843
cuda	0.52

(b) Parallel code

[1] Workload

I used 2D blocks and 2D grids to implement the code. Each block contents 32*32 threads to exploit GPU's resource. Each block will handle 28*28 region of source image, and each thread handles the computational task of each pixel. The grid size depends on the image size. It will make sure all pixel is covered by a block.

Although stripe shape block has better chance to achieve coalesced memory access, I choice rectangle block shape since it demands less margin memory and is more general to images with different size and shape.

It is obviously that there are certain proportion of idle threads, who do nothing after transmit a pixel of source image from global memory to shared memory. However, as far as my experience, if I want to process 32*32 pixels at the

same time, handling margin data would be a torture and create divergence which decreasing the performance.

Some method might let each block only process a certain channel of image to increase parallelism. Nevertheless, regarding coalesced memory access, I didn't choose this way.

I do try stream to improve the performance, but not observing any benefit.

[2] Shared Memory

There are two main items which will be frequently access in runtime. Those are mask and image block. Since the mask is constant, I saved it in constant memory instead of copying the data in every block.

In each block, each thread will first help transmit a pixel of source image block from global memory to shared memory. After the synchronization, 28*28 threads will start process the image.

[3] Code Explanation

✧ Main

I used **CudaMalloc** to allocate memory in global memory in GPU. *d_src_img* is for getting source image data, and *d_dst_img* is for storing the output.

```
unsigned char* d_src_img;
unsigned char* d_dst_img;
cudaMalloc((void**)&d_src_img, height * width * channels * sizeof(unsigned char));
cudaMalloc((void**)&d_dst_img, height * width * channels * sizeof(unsigned char));
```

Then, I used **CudaMemcpy** to copy image data from CPU to GPU's global memory. Since CPU have no work to do, I didn't use asynchronous copy method. I have tried using memory pinging, but not observing any improvement. When the transmission is done, the kernel will be launched and CPU wait for the result.

```
// Copy source image to device memory
cudaMemcpy(d_src_img, src_img, height * width * channels * sizeof(unsigned char), cudaMemcpyHostToDevice);

// Define block and grid dimensions
dim3 blockDim(32, 32);
dim3 gridDim((width + 27) / 28, (height + 27) / 28);

// Launch kernel
sobel<<<gridDim, blockDim>>>(d_src_img, d_dst_img, height, width, channels);

// Copy result from device memory to host
cudaMemcpy(dst_img, d_dst_img, height * width * channels * sizeof(unsigned char), cudaMemcpyDeviceToHost);
```

Free memory with **cudaFree** and write out image.

```
cudaFree(d_src_img);
cudaFree(d_dst_img);

write_png(argv[2], dst_img, height, width, channels);

free(src_img);
free(dst_img);
```

110062118
賴姿妘

✧ Sobel

It is the function ported to CUDA by using __global__ and kernel launching method.

First, each thread will access its thread id and map to the coordination of source image. Only the middle part of thread array is corresponding to image process task.

```
int tx = threadIdx.x;
int ty = threadIdx.y;
int x = blockIdx.x * 28 + tx -2 ;
int y = blockIdx.y * 28 + ty -2 ;
```

Each thread then do their own transmitting task. If its coordination is out of range, it will write 0 to shared image block. The **syncthreads** operation is necessary, otherwise some warps might start processing while needed information is still under transfer.

```
if (x < width && y < height) {
    shared_img[ty][tx][0] = s[channels * (width * y + x) + 0];
    shared_img[ty][tx][1] = s[channels * (width * y + x) + 1];
    shared_img[ty][tx][2] = s[channels * (width * y + x) + 2];
} else {
    shared_img[ty][tx][0] = 0;
    shared_img[ty][tx][1] = 0;
    shared_img[ty][tx][2] = 0;
}
__syncthreads();
```

The main processing parts. Unroll is used to theoretically optimization. No significant improvement results from the modification.

```
if (tx >= 2 && tx < 30 && ty >= 2 && ty < 30) {
    float val[6] = {0.0};
    #pragma unroll
    for (int v = -2; v <= 2; ++v) {
        #pragma unroll
        for (int u = -2; u <= 2; ++u) {
            val[0] += shared_img[ty+v][tx+u][2] * mask[0][u+2][v+2];
            val[1] += shared_img[ty+v][tx+u][1] * mask[0][u+2][v+2];
            val[2] += shared_img[ty+v][tx+u][0] * mask[0][u+2][v+2];
            val[3] += shared_img[ty+v][tx+u][2] * mask[1][u+2][v+2];
            val[4] += shared_img[ty+v][tx+u][1] * mask[1][u+2][v+2];
            val[5] += shared_img[ty+v][tx+u][0] * mask[1][u+2][v+2];
        }
    }
```

The remain part is quite like original one except for some minor modification such as reusing value.

110062118
賴姿妘

```
float totalR = sqrt(val[0] * val[0] + val[3] * val[3]) / SCALE;
float totalG = sqrt(val[1] * val[1] + val[4] * val[4]) / SCALE;
float totalB = sqrt(val[2] * val[2] + val[5] * val[5]) / SCALE;

unsigned char cR = min(totalR, 255.0f);
unsigned char cG = min(totalG, 255.0f);
unsigned char cB = min(totalB, 255.0f);

if (x < width && y < height) {
    int  tmp = channels * (width * y + x);
    t[tmp + 2] = cR;
    t[tmp + 1] = cG;
    t[tmp + 0] = cB;
}
```

B. **cudaMalloc** and **cudaMallocManaged**

cudaMalloc is used to allocate memory on the device (GPU) only. The allocated memory is accessible only by the device, and the host cannot directly access it. To transfer data between the host and device, explicit memory transfers using functions like **cudaMemcpy** or **cudaMemcpyAsync** are required. Last but not least, the memory allocation and deallocation are managed by the programmer.

Unlike **cudaMalloc**, **cudaMallocManaged** is used to allocate unified memory, which can be accessed by both the host (CPU) and device (GPU). The allocated memory is accessible from both the host and device code without the need for explicit memory transfers. The CUDA runtime automatically manages the data migration between the host and device as needed. When the page fault occurred the migration will happen. However, the memory allocation and deallocation are still managed by the programmer.

If I want to simplified my code or the data is too complicated or too huge but need to be accessed by both the host and device, I would choose **cudaMallocManaged** over **cudaMalloc**. On the other hand, if I want to overlap computation and communication or optimize the code, I would choose **cudaMalloc** over **cudaMallocManaged**.

C. Experiment

✧ Comparison

I conducted the experiment on 8.png to compare the program with shared memory and the one without it. Each block of the program without shared memory handle 32*32 pixels of image block.

	Shared memory	Global memory
Kernel duration (msecond)	17.23	17.46
Throughput	127.67	2.34

(GB/s)		
Ipc (inst/cycle)	0.71	0.86

The shared memory version(S) has shorter kernel duration than global memory version(G). Since using sharing memory saves time of accessing to global memory, S operates faster than G. Speaking of throughput, since shared memory has low access latency and high bandwidth compared with global memory, there is no doubt that it achieves greater throughput. Last, G has greater ipc than S, I guess it due to the extra operation for loading memory and thread synchronization.

✧ Profile

I use nsys to access following data except for CPU time. As you can see, I/O is the bottleneck.

CPU time (msecond)	72
I/O (msecond)	13756.43
Memory copy (msecond)	207.67
CUDA Memory melloc (msecond)	104.44
Kernel (msecond)	17.23