Report of COM6521 Parallel Computing with GPUs

<OpenMP/CUDA> Stage <1>

Description

The CPU code algorithms analysis of for-loops is shown as:

Image Size	Description	No. of Loops	Description	No. of Loops	Description	No. of Loops
256 x 256	tails level	8 x 8	Pixels level	3 x 3	Channels level	3
1024 x 1024	tails level	32 x 32	Pixels level	3 x 3	Channels level	3
4096 x 4096	tails level	128 x 128	Pixels level	3 x 3	Channels level	3

With analysis the serial CPU code, it can be found the outer loop is the largest with highest number of loops in all sample images, and it is a nest loops with same number of t_x and t_y for square images. Which is suitable for choosing Collapse method to optimize this level for-loops. It should have better performance than the tails to pixel and pixel to channels level in OpenMP.

In CUDA version, it is similar as OpenMP, the tails level is selected to be optimized with similar reason that larger loops should always have greater impact on the running time.

The other level loops will be remained, because for very small loops, loading the OpenMP to assign the task into threads or building CUDA kernel may spend more time than the serial code. Therefore, only tiles level task will be optimized.

It has been tested that the dynamic is faster than static method which may caused by when loading to pixel value there is three channels and possible to cause the unbalanced computing task.

Justification

OpenMP:

Private variables: t_x, t_y, p_x, p_y, ch, tile_index, tile_offset, pixel_offset
Shared variables: openmp_mosaic_sum
Schedule: Dynamic
Clause: collapse(2)

In this stage the collapse (2) is because the tails level loops is a double loops of t_x and t_y. Because all the index and loop numbers are private variables which can make sure an individual thread will be assigned a unique index number of tiles and pixel, with sum the pixel value with tile_index and ch to write the number, that can make sure there is no racing condition.

CUDA:

Similar with OpenMP version, the tiles level is optimized. The tile_index and tile_offset are

parallelized as the kernel function shown which is based on the CPU code. Then as the index in the inner loop to get unique pixel value of individual pixel and then sum as d_mosaic_sum in the device and copy them back to host to be ready in next step.

The Kernel function:

The tiles in each row in CPU code will be assigned to thread within same block, the block dimension will be assigned to the number of tiles on one row which should equal to image width/ Tile size. Therefore, the serial code can optimized by using threadIdx.x assign t_x, blockidx.x assign t_y with blockDim.x assign blockDim. Code is shown below.

```
const unsigned int tile_index = (blockDim.x * blockIdx.x + threadIdx.x) * CHANNELS;
const unsigned int tile_offset = (blockDim.x * blockIdx.x * TILE_SIZE * TILE_SIZE +
threadIdx.x * TILE_SIZE) * CHANNELS;
……
// For each pixel within the tile
    for (int p_x = 0; p_x < TILE_SIZE; ++p_x) {
        for (int p_y = 0; p_y < TILE_SIZE; ++p_y) {
            // For each colour channel
            const unsigned int pixel_offset = (p_y * d_width + p_x) * CHANNELS;
            for (int ch = 0; ch < CHANNELS; ++ch) {
                // Load pixel
                const unsigned char pixel = d_input_image_data[tile_offset +
pixel_offset + ch];
                d_mosaic_sum[tile_index + ch] += pixel;
            }
        }
    }
```

The CUDA model:

```
dim3 block = cuda_TILES_X;
dim3 grid = ((cuda_TILES_X * cuda_TILES_Y + block.x - 1) / block.x);
```

Performance:

Problem size	CPU Reference Time	Open MP stage 1	CUDA stage 1
256 x 256	0.202	0.211	0.178
1024 x 1024	2.923	0.781	0.697
2048 x 2048	12.351	2.533	1.805
4096 x 4096	64.744	10.345	29.373

The larger images than 256x256, both OpenMP and CUDA version code successful reduced the running time. In the stage 1 it seemed that OpenMP has better performance. The CUDA code may still have point to be improved and OpenMP performed well which reduced the running time for nearly 5 - 6 times but CUDA in the 4096x4096, it only speeds up for 3 times.

For small image both versions don't have great performance that might caused by the parallel task size is small the launch for OpenMP and CUDA kernel cost more time than the computation itself.

<OpenMP/CUDA> Stage <2>

Description

The CPU code algorithms analysis of for-loops is shown as:

Description	No. of Loops	Description	No. of Loops
Pixels level	3 x 3	Channels level	3
Pixels level	3 x 3	Channels level	3
Pixels level	3 x 3	Channels level	3

This stage the loop size is small also it only has limited potential to be optimized. The OpenMP and CUDA was tried to optimize this step with pixel level. For each pixel in a tile, the pixel value in 3 channels will be sum together and divided by the tiles area. The channels level for-loops will be remained. It is likely that the average is suitable for static structure because for every tiles the computation task is similar and can be assumed that they have similar computing bound.

Justification

OpenMP:

Private variables: t, ch

Schedule: Static

Clause: for

Critical section : `#pragma omp critical`

In stage 2, the static schedule was choose and the t, ch are set as private. Therefore, the loops in tile for pixel will be parallelized to get the index to get the `openmp_mosaic_sum value` calculated before. Here it is necessary to add critical section to prevent the threads write the same address of `openmp_mosaic_value` and this is able to cause race condition. Therefore a section of critical was set after the loops in tile as:

```
for (t = 0; t < openmp_TILES_X * openmp_TILES_Y; ++t) {
#pragma omp critical
```

CUDA:

The pixel channel id is assigned as follow code, which is is similar with the index (t * cpu_input_image.channels + ch) in CPU code and one thread in used for control one mosaic value in a certain tile.

The Kernel function:

```
const unsigned int tile_index_ch = (blockDim.x * blockIdx.x + threadIdx.x);
```

The CUDA model:

The size and grid are different from stage 1, initialize sizes of block and gird, in particular block =

number of tiles on row, and gird = number of tiles in whole images * channel number

```
dim3 block = cuda_TILES_X;
dim3 grid = ((cuda_TILES_X * cuda_TILES_Y *  cuda_input_image.channels + block.x - 1) / block.x);
```

Performance:

Problem size	CPU Reference Time	Open MP stage 2	CUDA stage 2
256 x 256	0.042	0.060	0.099
1024 x 1024	0.055	0.209	0.105
2048 x 2048	0.074	0.732	0.131
4096 x 4096	0.149	3.319	0.193

Generally, it can be concluded that both OpenMP and CUDA version code failed to reduce the time, this may cause that the original loops size in serial code is small, and the launch multiple threads may caused more time. The OpenMP plays very bad on larger image with more than 3 ms. Therefore, apply the CPU code here might be a better choice.

<OpenMP/CUDA> Stage <3>

Description

The CPU code algorithms analysis of for-loops is shown as:

Image Size	Description	No. of Loops	Description	No. of Loops	Description	No. of Loops
256 x 256	tails level	8 x 8	Pixels level	3 x 3	Channels level	3
1024 x 1024	tails level	32 x 32	Pixels level	3 x 3	Channels level	3
4096 x 4096	tails level	128 x 128	Pixels level	3 x 3	Channels level	3

The CPU code in stage 3 is similar to stage 1 in for-loop structure but reading the results from stage 2 to read the output_gloable_average. In this stage many threads is required to load the pre-calculated sum value so it

For OpenMP version code, the similar approach will be applied to assign the for-loops in different threads to reduce the running time.

In CUDA version, it is possible to optimize the by all threads in blocks reading the same share memory from previous steps.

Justification

OpenMP:

Private variables: t_x, t_y, p_x, p_y, ch, tile_index, tile_offset, pixel_offset

Shared variables: openmp_mosaic_sum
Schedule: Dynamic
Clause: collapse(2)

In this stage the collapse (2) is because the tails level loops is a double loops of t_x and t_y. Because all the index and loop numbers are private variables which can make sure an individual thread will be assigned a unique index number of tiles and pixel, with sum the pixel value with tile_index and ch to write the number, that can make sure there is no racing condition.

CUDA:

The threads will be assigned in x, y direction and the index will be assigned to the tile. The threads in one block(tile) loads `d_mosaic_value` to shared memory. Using shared memory brings great improvement to reduce the time. Because every thread only loads the value for one time rather than every time the threads will load the value so many times. Therefore, as the kernel function show that, the share memory will store the `d_mosaic_value` value with tile index + ch to be used to index the specific value. After assign the share memory with manual configuration, `__syncthreads()` was required to shared memory data is populated by all threads.

The Kernel function:

```
__global__ void broadcastMosaicValueToPixel(unsigned char* d_mosaic_value, unsigned char* d_output_image_data){

    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    __shared__ unsigned char shareMemory[CHANNELS];

   ......
// Assign the sum value from stage 2
    for (int ch = 0; ch < CHANNELS; ++ch) {
        d_output_image_data[pixel_global_index + ch] = shareMemory[ch];
    }
    return;
}
__syncthreads();
    // broadcast mosaic value from share memory to each pixel every thread control 3 channels for each pixel obtaining corresponding mosaic value from share memory
    for (int ch = 0; ch < CHANNELS; ++ch) {
        d_output_image_data[pixel_global_index + ch] = shareMemory[ch];
    }
    return;
}
```

The CUDA model:

The block is defined with threads in tiles (32 x 32 eg) as 2D. grid develop this to image.

```
dim3 block = dim3(TILE_SIZE, TILE_SIZE);
dim3 grid = dim3((cuda_input_image.width + block.x - 1) / block.x,
(cuda_input_image.height + block.y - 1) / block.y );
```

Performances

Problem size	CPU Reference Time	Open MP stage 3	CUDA stage 3
256 x 256	0.197	0.102	0.047
1024 x 1024	3.196	1.046	0.117
2048 x 2048	24.802	8.121	0.333
4096 x 4096	147.684	36.692	0.881

Both OpenMP and CUDA version code have good reduction on running time. Especially for the CUDA code, for the largest image 4096 x4096, it reduces the time for more than 100 times. According to the results it can be also concluded that loading shared memory is very fast with very high bandwidth. It has great ability to reduce the memory bound.