EC527
Lab8 report
Bin Xu
Seyed Reza Sajjadinasab

From part1 to part3, we compare the gpu_MMM to cpu_MMM_non_block and cpu_MMM_block. The largest
The input matrix is generated randomly. The data type is float. When comparing performance between CPU and GPU, GPU performs much better in the larger matrix size(2048). Because in the GPU parallel programming is fully utilized.
In the previous assignment, mmm_kij_block is the best version of MMM. In the below tables, mmm_kij_block is faster than the original MMM function. However, it is much slower than the GPU version of MMM.

To calculate the biggest error we have to initialize the matrix with a value that is not too big that causes the overflow and also not too small that causes a 0 error. As you can see, we divided the rand() value by 1e5.

Part1
Global

	host	best CPU MMM block	GPU_kernal_time	GPU_start_to_finish_time	Biggest error
1024 * 1024	2471.920655	2003.409405	8.245120	10.888096	49152.000000
2048 * 2048	23262.926515	22099.377953	64.435516	73.195648	98304.000000

Part2
Shared

	host	best CPU MMM block	GPU_kernal_time	GPU_start_to_finish_time	Biggest error
1024 * 1024	2415.903633	2012.938461	6.227488	9.594976	40960.000000
2048 * 2048	23984.564701	18245.560186	49.226719	58.001823	98304.000000

Part3
Unroll

From the previous two programs, the shared version of MMM is faster. So we unroll the shared kernel for part 3.

	host	best CPU MMM block	GPU_kernal_time	GPU_start_to_finish_time	Biggest error
1024 * 1024	2435.057697	1965.597739	6.223872	8.635872	3072.000000
2048 * 2048	62917.618016	16235.315186	49.319935	57.914082	8448.000000

Part4

	Block size	Grid size	GPU_kernal_time	GPU_start_to_finish_time
512 * 512	8	64	1.280576	2.240224
1024 * 1024	4	256	21.966433	30.157951
1024 * 1024	8	128	10.051712	12.649120
2048 * 2048	8	256	79.922050	99.495361
16384 * 16384	8	2048	35669.980469	36187.148438

Our GPU memory is 4GB, so the maximum row_len can be 16384(1 << 14). If the row_len is 32768(1 << 15), there is an out of memory issue. Because the overall size will be:
(1<<15) * (1<<15) * 4 * 3
Which is 12.9 GB and much larger than our GPU memory.

We find the gpu_square_matrix_mult online(https://github.com/lzhengchun/matrix-cuda/blob/master/matrix_cuda.cu), the performance is close to the shared version of our MMM.
It's actually pretty similar to our code. It just checks for the corner values so that we can have arbitrary size for our matrix rather than just a power of 2.

```
__global__ void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_result, int n)
{
    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;

    for (int sub = 0; sub < gridDim.x; ++sub)
    {
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        if(idx >= n*n)
        {
            // n may not divisible by BLOCK_SIZE
            tile_a[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }

        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if(idx >= n*n)
        {
            tile_b[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < n && col < n)
    {
        d_result[row * n + col] = tmp;
```

This is the best kernel that we have with favorable memory access pattern:

```
__global__ void MMK(int width, float* Md, float* Nd, float* Pd)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    int numOfTile = width / TILE_WIDTH;
    for (int i = 0; i < numOfTile; i++) {
        Mds[ty][tx] = Md[row * width + (i * TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[col + (i * TILE_WIDTH + ty) * width];
        __syncthreads();
        for(int j = 0; j < TILE_WIDTH; j++) {
            Pvalue += Mds[ty][j] * Nds[j][tx];
        }
        __syncthreads();
    }
    Pd[row * width + col] = Pvalue;
}
```

The below is the result for the provided kernel with a favorable memory access pattern

```
calculating results on host: 2362.200240 (msec)

calculating results on cpu_MMM_block: 1970.381145 (msec)

GPU kernel execution time: 6.160128 (msec)

GPU time: 8.795680 (msec)

Compare: 0
```

The below is one that doesn't have the favorable memory access pattern. For this, we just change the tx=threadIdx.y and ty=threadIdx.y in our code. This will change the way we access the memory (both DRAM and Shared) and as we can see it's much slower than the other version. This might be because the threadIdx.y are assigned to the successive banks and threadIdx.x have some strides. It's like each threadIdx.x has a block of threadIdx.y and as we see in the slides the second version is not a good practice to write.

```
calculating results on host: 2375.658198 (msec)

calculating results on cpu_MMM_block: 1958.663937 (msec)

GPU kernel execution time: 16.582399 (msec)

GPU time: 19.519360 (msec)

Compare: 0
```

Below is the slide we're referencing to:

Code w/ and w/o coalescing

```
// acc_array[32][32]
// block has 32 threads

int bx = blockIdx.x;
int acc = 0;

// which to use? - this one?
for (i = 0; i < 32; i++)
  acc += acc_array[bx][i];

// ...or this one?
for (i = 0; i < 32; i++)
  acc += acc_array[i][bx];
```

Recall – code is per thread
Note that threads are doing lots of work and are replicated across the block