CS5473 - Project 4

Author: Enzo Durel
Professor: Dr. Chongle Pan

March 25, 2025

GALLOGLY COLLEGE OF ENGINEERING
The UNIVERSITY of OKLAHOMA

Contents

List of Tables

1 Problem 1

Table 1: Problem 1 Runtime

Implementation	Steps on GPU	5 million trials	10 million trials
serial_mining.c	None	826.3	5643.7
gpu_mining_starter.cu	Step 1	668.4	3811.6
gpu_mining_problem1.cu	Step 1 and 2	6.7	17.7

1.1 Speedup

We have for the starter a speedup of $\frac{826.3}{668.4} = 1.24$ for 5 million trials and 1.48 for 10 million trials.

When we compute the step 2 on cuda, we've got a really good improvement, $\frac{826.3}{6.7} = 123.3$ for 5 million trials and a speedup of 318.85 for 10 million trials.

As the problem size increases in terms of trials, the speedup increases too.

2 Problem 2

Table 2: Problem 2 Runtime

Implementation	Steps on GPU	5 million trials	10 million trials
serial_mining.c	None	826.3	5643.7
gpu_mining_starter.cu	Step 1	668.4	3811.6
gpu_mining_problem1.cu	Step 1 and 2	6.7	17.7
gpu_mining_problem2.cu	Step 1, 2 and 3	9.7	17.9

2.1 Speedup

The Speedup are 0.69 and ≈ 1 for 5 and 10 million trials.

Although `gpu_mining_problem2.cu` offloads all three steps to the GPU, its runtime is comparable to (and sometimes worse than) gpu_mining_problem1.cu. This is because Step 3, the minimum hash reduction, is not computationally expensive, and offloading it incurs additional overhead such as kernel launch and memory copy latency. As a result, the full GPU offloading doesn't lead to further speedup beyond what was already achieved in Problem 1.

3 Problem 3

3.1 Problem 3 Analysis

3.1.1 Estimation without tiling

In the original CUDA convolution kernel (without tiling), each output pixel requires:

- 25 input pixel reads (for a 5×5 filter)
- 1 output pixel write
- 25 multiplications + 25 additions = 50 compute operations

Therefore, for each pixel:

- Total global memory accesses = 26
- Total compute operations = 50

Compute-to-global-memory-access ratio:

$$\frac{50}{26} \approx 1.92$$

This is relatively low and implies the kernel is memory-bound.

3.1.2 Pseudo-Code

To improve memory efficiency, we use tiling to load blocks of the input image into shared memory.

```
__global__ void tiled_convolution_kernel(int* input, int* output, int* filter,
                                         int width, int height, int filter_size)
{
    const int TILE_SIZE = 16;
    const int RADIUS = filter_size / 2;
    __shared__ int tile[TILE_SIZE + 4][TILE_SIZE + 4]; // extra 4 for 5x5 filter padding

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = blockIdx.y * TILE_SIZE + ty;
    int col = blockIdx.x * TILE_SIZE + tx;

    // Load input into shared memory with halo
    for (int dy = 0; dy < filter_size; ++dy) {
        for (int dx = 0; dx < filter_size; ++dx) {
            int global_row = row + dy - RADIUS;
            int global_col = col + dx - RADIUS;
            if (global_row >= 0 && global_row < height && global_col >= 0 && global_col < width) {
                tile[ty + dy][tx + dx] = input[global_row * width + global_col];
            } else {
                tile[ty + dy][tx + dx] = 0;
            }
```

```
        }
    }

    __syncthreads();

    // Compute convolution
    if (row < height && col < width) {
        int sum = 0;
        for (int i = 0; i < filter_size; ++i) {
            for (int j = 0; j < filter_size; ++j) {
                sum += tile[ty + i][tx + j] * filter[i * filter_size + j];
            }
        }
        output[row * width + col] = sum;
    }
}
```

3.1.3 Estimate with tiling

Assuming BLOCK_SIZE = 16 and filter size = 5:

- Shared memory loads: $(16 + 4)^2 = 400$ input reads
- Global memory writes: $16^2 = 256$
- Total memory accesses: 656
- Compute operations: 25 ops $\times$ 256 = 6400

New compute-to-global-memory-access ratio:

$$\frac{6400}{656} \approx 9.76$$

This shows a significant improvement due to reduced global memory accesses and increased reuse via shared memory.

3.2 Results

Table 3: Problem 3 Runtime

Implementation	Wall-clock runtime
Original serial program	0.010039
CUDA: Copying data from host to device	0.001752
CUDA: Launching kernel	0.000047
CUDA: Copying data from device to host	0.010149

In this case, the serial time is more efficient than the total runtime of CUDA program. The copying process takes too much time to make the parallel computation on GPU usefull.

4 Problem 4

4.1 Problem 4 Analysis (Max Pooling)

4.1.1 Estimation without tiling

- Reads 25 pixels per output
- Writes 1 output pixel
- Performs 24 comparisons

Total memory accesses: 26 Total operations: 24

$$\text{Compute-to-Memory Ratio (MaxPool)} = \frac{24}{26} \approx 0.92$$

4.1.2 Pseudo-Code

```
__global__ void tiled_maxpooling_kernel(int* input, int* output, int width, int height, int pool_size)
{
    const int TILE_SIZE = 16;
    const int RADIUS = pool_size / 2;
    __shared__ int tile[TILE_SIZE + 4][TILE_SIZE + 4];

    int tx = threadIdx.x, ty = threadIdx.y;
    int row = blockIdx.y * TILE_SIZE + ty;
    int col = blockIdx.x * TILE_SIZE + tx;

    // Load tile into shared memory with padding
    for (int dy = 0; dy < pool_size; ++dy)
        for (int dx = 0; dx < pool_size; ++dx) {
            int global_row = row + dy - RADIUS;
            int global_col = col + dx - RADIUS;
            tile[ty + dy][tx + dx] =
                (global_row >= 0 && global_row < height && global_col >= 0 && global_col < width)
                ? input[global_row * width + global_col]
                : INT_MIN;
        }

    __syncthreads();

    // Max-pooling
    if (row < height && col < width) {
        int maxVal = INT_MIN;
        for (int i = 0; i < pool_size; ++i)
            for (int j = 0; j < pool_size; ++j)
                maxVal = max(maxVal, tile[ty + i][tx + j]);
        output[row * width + col] = maxVal;
    }
```

```
}
```

4.1.3 Estimation with tiling

Assuming TILE_SIZE = 16 and filter/pool size = 5:

- Shared memory loads: $(16 + 4)^2 = 400$ per block
- Global writes: $16^2 = 256$ per block
- Total memory accesses per block = 656
- Output computations: 256 threads per block

Total compute ops: $24 \times 256 = 6144$

$$\text{Compute-to-Memory Ratio (MaxPool-Tiled)} = \frac{6144}{656} \approx 9.4$$

Using tiling significantly improves the compute-to-memory ratio:

- Max-pooling improved from **0.92 → ~9.4**

These improvements help reduce global memory bandwidth usage and increase performance by exploiting data reuse in shared memory.

4.2 Results

Table 4: Problem 4 Runtime

Implementation	Wall-clock runtime
Original serial program	0.02329700
CUDA: Copying data from host to device	0.00271099
CUDA: Running kernel 1 for convolution	0.00005400
CUDA: Running kernel 2 for max-pooling	0.00000499
CUDA: Copying data from device to host	0.01420900

In this case, the CUDA program runtime is more efficient than the serial program. The computation of 2 filters (blur and maxpooling) makes the use of GPU usefull.