

Lab 5: CUDA Matrix Multiplication

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 11:59 PM on 10/17

<https://github.com/wcharmon/ECE455/tree/main/HW05>

Overview

This lab introduces CUDA matrix multiplication kernels.

Learning Objectives

- Implement a naive CUDA kernel for matrix multiplication.
- Compare naive vs. coalesced memory access patterns.
- Apply loop unrolling to reduce loop overhead.
- Measure runtime using CUDA events.

Problem 1: Naive Matrix Multiplication

Goal: Each thread computes one element C_{ij} .

Kernel

Filename: mm_naive.cu

```
1 // Naive kernel: each thread computes one element C[i,j]
2 template <typename T>
3 __global__ void mm_kernel(T const* mat_1, T const* mat_2, T* mat_3,
4                           size_t m, size_t n, size_t p)
5 {
6     // Compute (i,j) coordinates from 2D grid
7     size_t i{blockIdx.x * blockDim.x + threadIdx.x};
8     size_t j{blockIdx.y * blockDim.y + threadIdx.y};
9
10    // Boundary check
11    if ((i >= m) || (j >= p)) return;
12
13    // Compute dot product of row i (A) and column j (B)
14    T acc_sum{0};
15    for (size_t k{0}; k < n; ++k)
16        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
17
18    mat_3[i * p + j] = acc_sum; // Write result
19 }
```

Main Function

```
1 // Host driver: run tests and measure kernel performance
2 int main()
3 {
4     const size_t num_tests{2}; // Correctness trials
5     assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
6     assert(random_multiple_test_mm_cuda<float>(num_tests));
7     assert(random_multiple_test_mm_cuda<double>(num_tests));
8     std::cout << "All tests passed!\n";
9
10    // --- Performance measurement ---
11    const size_t num_measurement_tests{2};
12    const size_t num_measurement_warmups{1};
13    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};
14
15    // Measure average latency across data types
16    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
17        m, n, p, num_measurement_tests, num_measurement_warmups);
18    float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
19        m, n, p, num_measurement_tests, num_measurement_warmups);
20    float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
21        m, n, p, num_measurement_tests, num_measurement_warmups);
22
23    // Print results
24    std::cout << "Matrix Multiplication Runtime\n";
25    std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
26    std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
27    std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
28    std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
29    return 0;
30 }
```

Slurm Script

Filename: mm_naive.slurm

```
1 #!/usr/bin/env zsh
2 #SBATCH --partition=instruction
3 #SBATCH --time=00:01:00
4 #SBATCH --ntasks=1
5 #SBATCH --cpus-per-task=1
6 #SBATCH --gpus-per-task=1
7 #SBATCH --output=mm_naive.output
8
9 cd $SLURM_SUBMIT_DIR
10 module load nvidia/cuda
11 nvcc mm_naive.cu -o mm_naive
12 ./mm_naive
```

Full code: [mm_naive.cu](#) on GitHub Gist

Problem 2: Coalesced Memory Access

Goal: Re-map threads so consecutive threads access consecutive memory addresses for better coalescing.

Kernel

Filename: mm_coalesced.cu

```
1 // Coalesced kernel: swapped x/y mapping improves memory access
2 template <typename T>
3 __global__ void mm_coalesced_kernel(T const* mat_1, T const* mat_2, T*
   mat_3,
4                                     size_t m, size_t n, size_t p)
5 {
6     size_t j{blockIdx.x * blockDim.x + threadIdx.x}; // columns -> x
7     size_t i{blockIdx.y * blockDim.y + threadIdx.y}; // rows -> y
8     if ((i >= m) || (j >= p)) return;
9
10    // Threads now traverse rows/cols in contiguous order
11    T acc_sum{0};
12    for (size_t k{0}; k < n; ++k)
13        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
14    mat_3[i * p + j] = acc_sum;
15 }
```

Main Function

```
1 // Same structure as Problem 1
2 int main()
3 {
4     const size_t num_tests{2};
5     assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
6     assert(random_multiple_test_mm_cuda<float>(num_tests));
7     assert(random_multiple_test_mm_cuda<double>(num_tests));
8     std::cout << "All tests passed!\n";
9
10    const size_t num_measurement_tests{2};
11    const size_t num_measurement_warmups{1};
12    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};
13
14    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
15        m, n, p, num_measurement_tests, num_measurement_warmups);
16    float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
17        m, n, p, num_measurement_tests, num_measurement_warmups);
18    float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
19        m, n, p, num_measurement_tests, num_measurement_warmups);
20
21    std::cout << "Matrix Multiplication Runtime\n";
22    std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
23    std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
24    std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
25    std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
```

```
26     return 0;
27 }
```

Slurm Script

Filename: mm_coalesced.slurm

```
1  #!/usr/bin/env zsh
2  #SBATCH --partition=instruction
3  #SBATCH --time=00:01:00
4  #SBATCH --ntasks=1
5  #SBATCH --cpus-per-task=1
6  #SBATCH --gpus-per-task=1
7  #SBATCH --output=mm_coalesced.output
8
9  cd $SLURM_SUBMIT_DIR
10 module load nvidia/cuda
11 nvcc mm_coalesced.cu -o mm_coalesced
12 ./mm_coalesced
```

Full code: [mm_coalesced.cu on GitHub Gist](#)

Problem 3: Loop Unrolling

Goal: Unroll the inner loop by 4 to reduce branch overhead and increase instruction-level parallelism.

Kernel

Filename: mm_unrolled.cu

```
1 // Unrolled kernel: perform 4 multiply-adds per iteration
2 template <typename T>
3 __global__ void mm_unrolled_kernel(T const* mat_1, T const* mat_2, T* mat_
4     3,
5     size_t m, size_t n, size_t p)
6 {
7     size_t j{blockIdx.x * blockDim.x + threadIdx.x};
8     size_t i{blockIdx.y * blockDim.y + threadIdx.y};
9     if ((i >= m) || (j >= p)) return;
10
11     T acc_sum{0};
12     size_t k{0};
13
14     // Main loop unrolled by 4
15     for (; k + 3 < n; k += 4) {
16         acc_sum += mat_1[i * n + (k + 0)] * mat_2[(k + 0) * p + j];
17         acc_sum += mat_1[i * n + (k + 1)] * mat_2[(k + 1) * p + j];
18         acc_sum += mat_1[i * n + (k + 2)] * mat_2[(k + 2) * p + j];
19         acc_sum += mat_1[i * n + (k + 3)] * mat_2[(k + 3) * p + j];
20     }
21
22     // Handle leftover elements (if n not multiple of 4)
23     for (; k < n; ++k)
24         acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
25
26     mat_3[i * p + j] = acc_sum;
27 }
```

Main Function

```
1 // Host driver for unrolled kernel
2 int main()
3 {
4     const size_t num_tests{2};
5     assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
6     assert(random_multiple_test_mm_cuda<float>(num_tests));
7     assert(random_multiple_test_mm_cuda<double>(num_tests));
8     std::cout << "All tests passed!\n";
9
10    const size_t num_measurement_tests{2};
11    const size_t num_measurement_warmups{1};
12    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};
13
14    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
15        m, n, p, num_measurement_tests, num_measurement_warmups);
16 }
```

```

16     float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
17         m, n, p, num_measurement_tests, num_measurement_warmups);
18     float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
19         m, n, p, num_measurement_tests, num_measurement_warmups);
20
21     std::cout << "Matrix Multiplication Runtime\n";
22     std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
23     std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
24     std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
25     std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
26     return 0;
27 }

```

Slurm Script

Filename: mm_unrolled.slurm

```

1  #!/usr/bin/env zsh
2  #SBATCH --partition=instruction
3  #SBATCH --time=00:01:00
4  #SBATCH --ntasks=1
5  #SBATCH --cpus-per-task=1
6  #SBATCH --gpus-per-task=1
7  #SBATCH --output=mm_unrolled.output
8
9  cd $SLURM_SUBMIT_DIR
10 module load nvidia/cuda
11 nvcc mm_unrolled.cu -o mm_unrolled
12 ./mm_unrolled

```

Full code: [mm_unrolled.cu](#) on GitHub Gist

Problem 4: Reflection

Task: Summarize the challenges you faced in this lab.

The hardest challenge for me during this lab was trying to figure out why the heck question 1 wasn't working because I was missing the `mm_cuda` function that a bunch of the helpers were trying to call. Also for like 30 seconds I forgot that C functions have to be in a certain order to work correctly.