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

Zachary Gunderson
https://github.com/Ztgunderson/ECE455

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`

```
// Naive kernel: each thread computes one element C[i,j]
template <typename T>
__global__ void mm_kernel(T const* mat_1, T const* mat_2, T* mat_3,
                          size_t m, size_t n, size_t p)
{
    // Compute (i,j) coordinates from 2D grid
    size_t i{blockIdx.x * blockDim.x + threadIdx.x};
    size_t j{blockIdx.y * blockDim.y + threadIdx.y};

    // Boundary check
    if ((i >= m) || (j >= p)) return;

    // Compute dot product of row i (A) and column j (B)
    T acc_sum{0};
    for (size_t k{0}; k < n; ++k)
        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];

    mat_3[i * p + j] = acc_sum; // Write result
}
```

Main Function

```
// Host driver: run tests and measure kernel performance
int main()
{
    const size_t num_tests{2};   // Correctness trials
    assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_mm_cuda<float>(num_tests));
    assert(random_multiple_test_mm_cuda<double>(num_tests));
    std::cout << "All tests passed!\n";

    // --- Performance measurement ---
    const size_t num_measurement_tests{2};
    const size_t num_measurement_warmups{1};
    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};

    // Measure average latency across data types
    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
    float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
    float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
        m, n, p, num_measurement_tests, num_measurement_warmups);

    // Print results
    std::cout << "Matrix Multiplication Runtime\n";
    std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
    std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
    std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
    std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
    return 0;
}
```

Slurm Script

Filename: mm_naive.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=mm_naive.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc mm_naive.cu -o mm_naive
./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`

```
// Coalesced kernel: swapped x/y mapping improves memory access
template <typename T>
__global__ void mm_coalesced_kernel(T const* mat_1, T const* mat_2, T* mat_3,
                                    size_t m, size_t n, size_t p)
{
    size_t j{blockIdx.x * blockDim.x + threadIdx.x}; // columns -> x
    size_t i{blockIdx.y * blockDim.y + threadIdx.y}; // rows -> y
    if ((i >= m) || (j >= p)) return;

    // Threads now traverse rows/cols in contiguous order
    T acc_sum{0};
    for (size_t k{0}; k < n; ++k)
        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
    mat_3[i * p + j] = acc_sum;
}
```

Main Function

```
// Same structure as Problem 1
int main()
{
    const size_t num_tests{2};
    assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_mm_cuda<float>(num_tests));
    assert(random_multiple_test_mm_cuda<double>(num_tests));
    std::cout << "All tests passed!\n";

    const size_t num_measurement_tests{2};
    const size_t num_measurement_warmups{1};
    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};

    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
    float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
    float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
        m, n, p, num_measurement_tests, num_measurement_warmups);

    std::cout << "Matrix Multiplication Runtime\n";
    std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
    std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
    std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
    std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
```

```
    return 0;
}
```

Slurm Script

Filename: `mm_coalesced.slurm`

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=mm_coalesced.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc mm_coalesced.cu -o mm_coalesced
./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`

```
// Unrolled kernel: perform 4 multiply-adds per iteration
template <typename T>
__global__ void mm_unrolled_kernel(T const* mat_1, T const* mat_2, T* mat_3,
                                    size_t m, size_t n, size_t p)
{
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    if ((i >= m) || (j >= p)) return;

    T acc_sum{0};
    size_t k{0};

    // Main loop unrolled by 4
    for (; k + 3 < n; k += 4) {
        acc_sum += mat_1[i * n + (k + 0)] * mat_2[(k + 0) * p + j];
        acc_sum += mat_1[i * n + (k + 1)] * mat_2[(k + 1) * p + j];
        acc_sum += mat_1[i * n + (k + 2)] * mat_2[(k + 2) * p + j];
        acc_sum += mat_1[i * n + (k + 3)] * mat_2[(k + 3) * p + j];
    }

    // Handle leftover elements (if n not multiple of 4)
    for (; k < n; ++k)
        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];

    mat_3[i * p + j] = acc_sum;
}
```

Main Function

```
// Host driver for unrolled kernel
int main()
{
    const size_t num_tests{2};
    assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_mm_cuda<float>(num_tests));
    assert(random_multiple_test_mm_cuda<double>(num_tests));
    std::cout << "All tests passed!\n";

    const size_t num_measurement_tests{2};
    const size_t num_measurement_warmups{1};
    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};

    float mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
```

```
    float mm_cuda_float_latency = measure_latency_mm_cuda<float>(
        m, n, p, num_measurement_tests, num_measurement_warmups);
    float mm_cuda_double_latency = measure_latency_mm_cuda<double>(
        m, n, p, num_measurement_tests, num_measurement_warmups);

    std::cout << "Matrix Multiplication Runtime\n";
    std::cout << "m: " << m << " n: " << n << " p: " << p << "\n";
    std::cout << "INT32: " << mm_cuda_int32_latency << " ms\n";
    std::cout << "FLOAT: " << mm_cuda_float_latency << " ms\n";
    std::cout << "DOUBLE: " << mm_cuda_double_latency << " ms\n";
    return 0;
}
```

Slurm Script

Filename: mm_unrolled.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:01:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --output=mm_unrolled.output

cd $SLURM_SUBMIT_DIR
module load nvidia/cuda
nvcc mm_unrolled.cu -o mm_unrolled
./mm_unrolled
```

Full code: mm_unrolled.cu on GitHub Gist

Problem 4: Reflection

Task: Summarize the challenges you faced in this lab.

No challenges. The code worked well, but I did notice:

cat mm_unrolled.output
All tests passed!
Matrix Multiplication Runtime
m: 1024 n: 1024 p: 1024
INT32: 1.34542 ms
FLOAT: 1.34451 ms
DOUBLE: 5.71957 ms

cat mm_coalesced.output
All tests passed!
Matrix Multiplication Runtime
m: 1024 n: 1024 p: 1024
INT32: 1.34298 ms
FLOAT: 1.34448 ms
DOUBLE: 5.73184 ms

unrolled and coalesced had around the same speed. I wonder if this is a Euler thing? It seems like Euler is not very reliable for gauging our performance. I'm concerned this might impact our results for developing our algorithm for the project