CUDA Matrix Multiplication

Introduction

CUDA is a parallel computing platform and programming language that allows software to use certain types of graphics processing unit (GPU) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU). It could significantly enhance the performance of programs that could be computed with massive parallelism.

Matrix multiplication is a typical application that could be computed with massive parallelism. In this blog post, I would like to present a “hello-world” CUDA example of matrix multiplications and its preliminary optimizations.

Matrix Multiplication

There are two common matrix multiplication forms. The ordinary matrix multiplication `mm` and the batched matrix multiplication `bmm`.

$C_{n \times p} = A_{n \times m} B_{m \times p} C_{b \times n \times p} = A_{b \times n \times m} B_{b \times m \times p} C_{n \times p} = A_{n \times m} B_{m \times p} C_{b \times n \times p} = A_{b \times n \times m} B_{b \times m \times p}$

The reader could find the specifications of `mm` and `bmm` from PyTorch documentation `torch.mm` and `torch.bmm`.

In the following example, we first implemented the `mm` and `bmm` using C++. Then we implemented the `mm` using CUDA and naturally extended the `mm` implementation to the `bmm` implementation. Finally, we verified the correctness of the `mm` and `bmm` CUDA implementations.

Naive Implementation

This is the single source code file that contains the CPU and CUDA implementations for the matrix multiplication `mm` and the batched matrix multiplication `bmm`.

mm.cu

```
#include <cassert>
#include <cstddef>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <random>
#include <stdexcept>
#include <vector>

#define BLOCK_DIM 32

#define checkCuda(val) check((val), #val, __FILE__, __LINE__)
template <typename T>
void check(T err, const char* const func, const char* const file,
           const int line)
```

```
{
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Runtime Error at: " << file << ":" << line
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

template <typename T>
std::vector<T> create_rand_vector(size_t n)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(-256, 256);

    std::vector<T> vec(n);
    for (size_t i{0}; i < n; ++i)
    {
        vec.at(i) = static_cast<T>(uniform_dist(e));
    }

    return vec;
}

// mat_1: m x n
// mat_2: n x p
// mat_3: m x p
template <typename T>
void mm(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n, size_t p)
{
    // Compute the cells in mat_3 sequentially.
    for (size_t i{0}; i < m; ++i)
    {
        for (size_t j{0}; j < p; ++j)
        {
            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;
        }
    }
}

// mat_1: b x m x n
// mat_2: b x n x p
// mat_3: b x m x p
template <typename T>
void bmm(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m, size_t n,
         size_t p)
{
    // Iterate through the batch dimension.
```

```
    for (size_t i{0}; i < b; ++i)
    {
        mm(mat_1 + i * (m * n), mat_2 + i * (n * p), mat_3 + i * (m * p), m, n,
           p);
    }
}

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)
{
    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};

    // Do not process outside the matrix.
    // Do not forget the equal sign!
    if ((i >= m) || (j >= p))
    {
        return;
    }

    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;
}

// It should be straightforward to extend a kernel to support batching.
template <typename T>
__global__ void bmm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t b,
                           size_t m, size_t n, size_t p)
{
    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};
    size_t l{blockIdx.z};

    // Do not process outside the matrix.
    // Do not forget the equal sign!
    if ((i >= m) || (j >= p))
    {
        return;
    }

    T acc_sum{0};
    for (size_t k{0}; k < n; ++k)
    {
        acc_sum += mat_1[l * m * n + i * n + k] * mat_2[l * n * p + k * p + j];
    }
    mat_3[l * m * p + i * p + j] = acc_sum;
```

```
}

template <typename T>
void mm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n,
             size_t p)
{
    dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
    dim3 blocks_per_grid(1, 1);
    blocks_per_grid.x = std::ceil(static_cast<double>(p) /
                                  static_cast<double>(threads_per_block.x));
    blocks_per_grid.y = std::ceil(static_cast<double>(m) /
                                  static_cast<double>(threads_per_block.y));
    mm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, m, n,
                                                       p);
}

template <typename T>
void bmm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m,
              size_t n, size_t p)
{
    dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
    dim3 blocks_per_grid(1, 1, 1);
    blocks_per_grid.x = std::ceil(static_cast<double>(p) /
                                  static_cast<double>(threads_per_block.x));
    blocks_per_grid.y = std::ceil(static_cast<double>(m) /
                                  static_cast<double>(threads_per_block.y));
    blocks_per_grid.z = b;
    bmm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, b,
                                                        m, n, p);
}

template <typename T>
bool allclose(std::vector<T> const& vec_1, std::vector<T> const& vec_2,
              T const& abs_tol)
{
    if (vec_1.size() != vec_2.size())
    {
        return false;
    }
    for (size_t i{0}; i < vec_1.size(); ++i)
    {
        if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol)
        {
            std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
            return false;
        }
    }
    return true;
}

template <typename T>
bool random_test_mm_cuda(size_t m, size_t n, size_t p)
{
    std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
    std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};
```

```
    std::vector<T> mat_3_vec(m * p);
    std::vector<T> mat_4_vec(m * p);
    T const* mat_1{mat_1_vec.data()};
    T const* mat_2{mat_2_vec.data()};
    T* mat_3{mat_3_vec.data()};
    T* mat_4{mat_4_vec.data()};

    mm(mat_1, mat_2, mat_3, m, n, p);

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // Copy data from host to device.
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
                         cudaMemcpyHostToDevice));

    // Run matrix multiplication on GPU.
    mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    cudaDeviceSynchronize();
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    // Copy data from device to host.
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}

template <typename T>
bool random_test_bmm_cuda(size_t b, size_t m, size_t n, size_t p)
{
    std::vector<T> const mat_1_vec{create_rand_vector<T>(b * m * n)};
    std::vector<T> const mat_2_vec{create_rand_vector<T>(b * n * p)};
    std::vector<T> mat_3_vec(b * m * p);
    std::vector<T> mat_4_vec(b * m * p);
    T const* mat_1{mat_1_vec.data()};
    T const* mat_2{mat_2_vec.data()};
```

```
    T* mat_3{mat_3_vec.data()};
    T* mat_4{mat_4_vec.data()};

    bmm(mat_1, mat_2, mat_3, b, m, n, p);

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // Copy data from host to device.
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
                         cudaMemcpyHostToDevice));

    // Run matrix multiplication on GPU.
    bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    cudaDeviceSynchronize();
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    // Copy data from device to host.
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}

template <typename T>
bool random_multiple_test_mm_cuda(size_t num_tests)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(1, 256);

    size_t m{0}, n{0}, p{0};
    bool success{false};

    for (size_t i{0}; i < num_tests; ++i)
    {
        m = static_cast<size_t>(uniform_dist(e));
```

```
        n = static_cast<size_t>(uniform_dist(e));
        p = static_cast<size_t>(uniform_dist(e));
        success = random_test_mm_cuda<T>(m, n, p);
        if (!success)
        {
            return false;
        }
    }

    return true;
}

template <typename T>
bool random_multiple_test_bmm_cuda(size_t num_tests)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(1, 256);

    size_t b{0}, m{0}, n{0}, p{0};
    bool success{false};

    for (size_t i{0}; i < num_tests; ++i)
    {
        b = static_cast<size_t>(uniform_dist(e));
        m = static_cast<size_t>(uniform_dist(e));
        n = static_cast<size_t>(uniform_dist(e));
        p = static_cast<size_t>(uniform_dist(e));
        success = random_test_bmm_cuda<T>(b, m, n, p);
        if (!success)
        {
            return false;
        }
    }

    return true;
}

template <typename T>
float measure_latency_mm_cuda(size_t m, size_t n, size_t p, size_t num_tests,
                              size_t num_warmups)
{
    cudaEvent_t startEvent, stopEvent;
    float time{0.0f};

    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));
```

```cpp
    for (size_t i{0}; i < num_warmups; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }

    checkCuda(cudaEventRecord(startEvent, 0));
    for (size_t i{0}; i < num_tests; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    float latency{time / num_tests};

    return latency;
}

template <typename T>
float measure_latency_bmm_cuda(size_t b, size_t m, size_t n, size_t p,
                               size_t num_tests, size_t num_warmups)
{
    cudaEvent_t startEvent, stopEvent;
    float time{0.0f};

    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * b * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * b * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * b * m * p));

    for (size_t i{0}; i < num_warmups; ++i)
    {
        bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    }

    checkCuda(cudaEventRecord(startEvent, 0));
```

```cpp
    for (size_t i{0}; i < num_tests; ++i)
    {
        bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    float latency{time / num_tests};

    return latency;
}

int main()
{
    constexpr size_t num_tests{10};

    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));
    assert(random_multiple_test_bmm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_bmm_cuda<float>(num_tests));
    assert(random_multiple_test_bmm_cuda<double>(num_tests));

    constexpr size_t num_measurement_tests{100};
    constexpr size_t num_measurement_warmups{10};
    size_t b{128}, m{1024}, n{1024}, p{1024};

    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)};

    float bmm_cuda_int32_latency{measure_latency_bmm_cuda<int32_t>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};
    float bmm_cuda_float_latency{measure_latency_bmm_cuda<float>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};
    float bmm_cuda_double_latency{measure_latency_bmm_cuda<double>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};

```

```
    std::cout << "Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << mm_cuda_int32_latency << " ms" << std::endl;
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << mm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << mm_cuda_double_latency << " ms" << std::endl;

    std::cout << "Batched Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "b: " << b << " "
              << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << bmm_cuda_int32_latency << " ms" << std::endl;
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << bmm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << bmm_cuda_double_latency << " ms" << std::endl;
}
```

Run Naive Example

Building and running the example requires an NVIDIA GPU. We also used NVIDIA official Docker container to set up the building environment.

To start the Docker container, please run the following command on the host computer.

```
$ docker run -it --rm --gpus all --cap-add=SYS_PTRACE --security-opt seccomp=unconfined -v $(pwd):/m
```

To build and run the application, please run the following command in the Docker container.

```
$ cd /mnt/
$ nvcc mm.cu -o mm -std=c++14
$ ./mm
Matrix Multiplication CUDA Latency
m: 1024 n: 1024 p: 1024
INT32: 1.11436 ms
FLOAT: 0.98451 ms
DOUBLE: 4.10433 ms
Batched Matrix Multiplication CUDA Latency
b: 128 m: 1024 n: 1024 p: 1024
INT32: 125.26781 ms
FLOAT: 124.67697 ms
DOUBLE: 487.87039 ms
```

We should expect no assertion error or any other kind of error for build and execution. The latencies were measured on a NVIDIA RTX 3090 GPU.

Matrix Multiplication Optimizations

The CUDA kernel optimization is usually all about how to accelerate the data traffic without affecting the number of math operations. To get the CUDA kernel fully optimized for GPU, the user would have to be very experienced with low-level GPU features and specifications and CUDA programming. But this does not prevent us from doing some preliminary optimization based on some shallow understandings of GPU.

Make Matrix Multiplication More Math-Bound

GPU is very friendly with math-bound operations. According to my previous blog post "Math-Bound VS Memory-Bound Operations", if the number of operations remains the same and the number of memory IO bytes gets reduced, the operation will become more math-bound. That is to say, we want

$$\frac{N_{op}}{N_{byte}} > \frac{BW_{math}}{BW_{mem}}$$

In our matrix multiplication naive CUDA implementation,

We have to do mnp multiplication and additions, $2mnp$ reads from memory, and mp writes to memory. We could ignore the mp writes from memory IO because the $2mnp$ reads is usually much more than the mp writes.

Suppose we are doing FP32 matrix multiplication,

$$\frac{N_{op}}{N_{byte}} = \frac{2 \times mnp}{2mnp \times 4} = \frac{1}{4}$$

For a modern GPU such as NVIDIA RTX 3090, for FP32 math,

$$\frac{BW_{math}}{BW_{mem}} = \frac{35.58}{0.936} = 38.0$$

We could see that the naive CUDA matrix multiplication implementation does not get even close to math-bound. Since N_{op} should be a constant in matrix multiplication, let's see if we could reduce N_{byte} by caching.

Ideally, if we could cache the two full operand matrices $A^{n \times m}$ and $B^{m \times p}$, we could make the matrix multiplication most math-bound. However, since the caching size is limited and the implementation is supposed to support matrix multiplications with all different sizes, caching the full matrices is not technically possible.

Matrix Multiplication Decomposition

It is possible to decompose matrix multiplication mm into smaller matrix multiplications.

$$A = \begin{bmatrix} A_{1,1}^{d \times d} & A_{1,2}^{d \times d} & \cdots & A_{1,n/d}^{d \times d} \\ A_{2,1}^{d \times d} & A_{2,2}^{d \times d} & \cdots & A_{2,n/d}^{d \times d} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m/d,1}^{d \times d} & A_{m/d,2}^{d \times d} & \cdots & A_{m/d,n/d}^{d \times d} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{1,1}^{d \times d} & B_{1,2}^{d \times d} & \cdots & B_{1,p/d}^{d \times d} \\ B_{2,1}^{d \times d} & B_{2,2}^{d \times d} & \cdots & B_{2,p/d}^{d \times d} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n/d,1}^{d \times d} & B_{n/d,2}^{d \times d} & \cdots & B_{n/d,p/d}^{d \times d} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{1,1}^{d \times d} & C_{1,2}^{d \times d} & \cdots & C_{1,p/d}^{d \times d} \\ C_{2,1}^{d \times d} & C_{2,2}^{d \times d} & \cdots & C_{2,p/d}^{d \times d} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m/d,1}^{d \times d} & C_{m/d,2}^{d \times d} & \cdots & C_{m/d,p/d}^{d \times d} \end{bmatrix}$$

$$C_{i,j}^{d \times d} = \sum_{k=1}^{n/d} A_{i,k}^{d \times d} B_{k,j}^{d \times d}$$

The decomposition does not alter the number of operations Nop.

Nop=2d3(nd)(mdpd)=2mnp

Because small matrices Ad×di,k and Bd×dk,j could be cached, the memory IO bytes could be reduced, and the overall matrix multiplication could become more math bound. Let’s calculate how much memory IO bytes is needed in this case.

Nbyte=2d2×4×(nd)(mdpd)=8mnpd

Therefore,

NopNbyte=2mnp8mnpd=d4

Notice that when d=1, the matrix multiplication falls back to the naive matrix multiplication. When d becomes larger, the implementation becomes more math-bound.

Optimized Implementation

The following implementation decomposed the matrix multiplication into multiple small matrix multiplications. The source code could be found on GitHub.

mm_optimization.cu

```
#include <cassert>
#include <cstddef>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <random>
#include <stdexcept>
#include <vector>

#define BLOCK_DIM 32

#define checkCuda(val) check((val), #val, __FILE__, __LINE__)
template <typename T>
void check(T err, const char* const func, const char* const file,
           const int line)
{
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Runtime Error at: " << file << ":" << line
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

template <typename T>
std::vector<T> create_rand_vector(size_t n)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(-256, 256);

```

```
    std::vector<T> vec(n);
    for (size_t i{0}; i < n; ++i)
    {
        vec.at(i) = static_cast<T>(uniform_dist(e));
    }

    return vec;
}

// mat_1: m x n
// mat_2: n x p
// mat_3: m x p
template <typename T>
void mm(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n, size_t p)
{
    // Compute the cells in mat_3 sequentially.
    for (size_t i{0}; i < m; ++i)
    {
        for (size_t j{0}; j < p; ++j)
        {
            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;
        }
    }
}

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)
{
    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};

    // Do not process outside the matrix.
    // Do not forget the equal sign!
    if ((i >= m) || (j >= p))
    {
        return;
    }

    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;
}

template <typename T>
```

```
__global__ void mm_kernel_optimized(T const* mat_1, T const* mat_2, T* mat_3,
                                    size_t m, size_t n, size_t p)
{
    __shared__ T mat_1_tile[BLOCK_DIM][BLOCK_DIM];
    __shared__ T mat_2_tile[BLOCK_DIM][BLOCK_DIM];

    T acc_sum{0};

    for (size_t tile_idx{0};
         tile_idx < ceilf(static_cast<float>(n) / BLOCK_DIM); ++tile_idx)
    {
        size_t i{blockIdx.y * blockDim.y + threadIdx.y};
        size_t j{tile_idx * blockDim.x + threadIdx.x};
        if ((i < m) && (j < n))
        {
            mat_1_tile[threadIdx.y][threadIdx.x] = mat_1[i * n + j];
        }
        else
        {
            mat_1_tile[threadIdx.y][threadIdx.x] = 0;
        }
        i = tile_idx * blockDim.y + threadIdx.y;
        j = blockIdx.x * blockDim.x + threadIdx.x;
        if ((i < n) && (j < p))
        {
            mat_2_tile[threadIdx.y][threadIdx.x] = mat_2[i * p + j];
        }
        else
        {
            mat_2_tile[threadIdx.y][threadIdx.x] = 0;
        }
        __syncthreads();
        for (size_t k{0}; k < BLOCK_DIM; ++k)
        {
            acc_sum += mat_1_tile[threadIdx.y][k] * mat_2_tile[k][threadIdx.x];
        }
        __syncthreads();
    }

    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};

    if ((i < m) && (j < p))
    {
        mat_3[i * p + j] = acc_sum;
    }
}

template <typename T>
void mm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n,
             size_t p,
             void (*f)(T const*, T const*, T*, size_t, size_t, size_t))
{
```

```
    dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
    dim3 blocks_per_grid(1, 1);
    blocks_per_grid.x = std::ceil(static_cast<double>(p) /
                                  static_cast<double>(threads_per_block.x));
    blocks_per_grid.y = std::ceil(static_cast<double>(m) /
                                  static_cast<double>(threads_per_block.y));
    f<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, m, n, p);
}

template <typename T>
bool allclose(std::vector<T> const& vec_1, std::vector<T> const& vec_2,
              T const& abs_tol)
{
    if (vec_1.size() != vec_2.size())
    {
        return false;
    }
    for (size_t i{0}; i < vec_1.size(); ++i)
    {
        if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol)
        {
            std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
            return false;
        }
    }
    return true;
}

template <typename T>
bool random_test_mm_cuda(size_t m, size_t n, size_t p,
                         void (*f)(T const*, T const*, T*, size_t, size_t,
                                   size_t))
{
    std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
    std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};
    std::vector<T> mat_3_vec(m * p);
    std::vector<T> mat_4_vec(m * p);
    T const* mat_1{mat_1_vec.data()};
    T const* mat_2{mat_2_vec.data()};
    T* mat_3{mat_3_vec.data()};
    T* mat_4{mat_4_vec.data()};

    mm(mat_1, mat_2, mat_3, m, n, p);

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // Copy data from host to device.
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
```

```
                         cudaMemcpyHostToDevice));

    // Run matrix multiplication on GPU.
    mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
    cudaDeviceSynchronize();
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
    // Copy data from device to host.
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}

template <typename T>
bool random_multiple_test_mm_cuda(size_t num_tests,
                                  void (*f)(T const*, T const*, T*, size_t,
                                            size_t, size_t))
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(1, 256);

    size_t m{0}, n{0}, p{0};
    bool success{false};

    for (size_t i{0}; i < num_tests; ++i)
    {
        m = static_cast<size_t>(uniform_dist(e));
        n = static_cast<size_t>(uniform_dist(e));
        p = static_cast<size_t>(uniform_dist(e));
        success = random_test_mm_cuda<T>(m, n, p, f);
        if (!success)
        {
            return false;
        }
    }

    return true;
}

template <typename T>
float measure_latency_mm_cuda(size_t m, size_t n, size_t p, size_t num_tests,
                              size_t num_warmups,
```

```
                              void (*f)(T const*, T const*, T*, size_t, size_t,
                                        size_t))
{
    cudaEvent_t startEvent, stopEvent;
    float time{0.0f};

    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));

    for (size_t i{0}; i < num_warmups; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
    }

    checkCuda(cudaEventRecord(startEvent, 0));
    for (size_t i{0}; i < num_tests; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    float latency{time / num_tests};

    return latency;
}

int main()
{
    constexpr size_t num_tests{10};

    assert(random_multiple_test_mm_cuda<int32_t>(num_tests, mm_kernel));
    assert(random_multiple_test_mm_cuda<float>(num_tests, mm_kernel));
    assert(random_multiple_test_mm_cuda<double>(num_tests, mm_kernel));
```

```

    assert(
        random_multiple_test_mm_cuda<int32_t>(num_tests, mm_kernel_optimized));
    assert(random_multiple_test_mm_cuda<float>(num_tests, mm_kernel_optimized));
    assert(
        random_multiple_test_mm_cuda<double>(num_tests, mm_kernel_optimized));

    constexpr size_t num_measurement_tests{100};
    constexpr size_t num_measurement_warmups{10};
    const size_t m{1024}, n{1024}, p{1024};

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

    std::cout << "Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << mm_cuda_int32_latency << " ms" << std::endl;
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << mm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << mm_cuda_double_latency << " ms" << std::endl;

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

    std::cout << "Optimized Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << mm_cuda_int32_latency << " ms" << std::endl;
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << mm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << mm_cuda_double_latency << " ms" << std::endl;
}
```

Run Optimized Example

In the same Docker container, build and run the following application. We could see that the latency of INT32 and FP32 matrix multiplication got improved too different degrees.

```
$ nvcc mm_optimization.cu -o mm_optimization --std=c++14
$ ./mm_optimization
Matrix Multiplication CUDA Latency
m: 1024 n: 1024 p: 1024
INT32: 1.04373 ms
FLOAT: 1.02149 ms
DOUBLE: 3.83370 ms
Optimized Matrix Multiplication CUDA Latency
m: 1024 n: 1024 p: 1024
INT32: 0.84207 ms
FLOAT: 0.81759 ms
DOUBLE: 3.95231 ms
```

Miscellaneous

There are more subtle factors affecting the performance and there are more optimization opportunities to further optimize the matrix multiplication implementation. But those requires more thorough understanding of GPU and CUDA.

References