

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`.

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

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

```
1  #include <cassert>
2  #include <cstdint>
3  #include <cstdint>
4  #include <iomanip>
5  #include <iostream>
6  #include <random>
7  #include <stdexcept>
8  #include <vector>
9
10 #define BLOCK_DIM 32
11
12 #define checkCuda(val) check((val), #val, __FILE__, __LINE__)
13 template <typename T>
14 void check(T err, const char* const func, const char* const file,
15           const int line)
```

```

16 {
17     if (err != cudaSuccess)
18     {
19         std::cerr << "CUDA Runtime Error at: " << file << ":" << line
20             << std::endl;
21         std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
22         std::exit(EXIT_FAILURE);
23     }
24 }
25
26 template <typename T>
27 std::vector<T> create_rand_vector(size_t n)
28 {
29     std::random_device r;
30     std::default_random_engine e(r());
31     std::uniform_int_distribution<int> uniform_dist(-256, 256);
32
33     std::vector<T> vec(n);
34     for (size_t i{0}; i < n; ++i)
35     {
36         vec.at(i) = static_cast<T>(uniform_dist(e));
37     }
38
39     return vec;
40 }
41
42 // mat_1: m x n
43 // mat_2: n x p
44 // mat_3: m x p
45 template <typename T>
46 void mm(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n, size_t p)
47 {
48     // Compute the cells in mat_3 sequentially.
49     for (size_t i{0}; i < m; ++i)
50     {
51         for (size_t j{0}; j < p; ++j)
52         {
53             T acc_sum{0};
54             for (size_t k{0}; k < n; ++k)
55             {
56                 acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
57             }
58             mat_3[i * p + j] = acc_sum;
59         }
60     }
61 }
62
63 // mat_1: b x m x n
64 // mat_2: b x n x p
65 // mat_3: b x m x p
66 template <typename T>
67 void bmm(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m, size_t n,
68     size_t p)
69 {
70     // Iterate through the batch dimension.

```

```

71     for (size_t i{0}; i < b; ++i)
72     {
73         mm(mat_1 + i * (m * n), mat_2 + i * (n * p), mat_3 + i * (m * p), m, n,
74            p);
75     }
76 }
77
78 template <typename T>
79 __global__ void mm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t m,
80                           size_t n, size_t p)
81 {
82     // 2D block and 2D thread
83     // Each thread computes one cell in mat_3.
84     size_t i{blockIdx.y * blockDim.y + threadIdx.y};
85     size_t j{blockIdx.x * blockDim.x + threadIdx.x};
86
87     // Do not process outside the matrix.
88     // Do not forget the equal sign!
89     if ((i ≥ m) || (j ≥ p))
90     {
91         return;
92     }
93
94     T acc_sum{0};
95     for (size_t k{0}; k < n; ++k)
96     {
97         acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
98     }
99     mat_3[i * p + j] = acc_sum;
100 }
101
102 // It should be straightforward to extend a kernel to support batching.
103 template <typename T>
104 __global__ void bmm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t b,
105                            size_t m, size_t n, size_t p)
106 {
107     // 2D block and 2D thread
108     // Each thread computes one cell in mat_3.
109     size_t i{blockIdx.y * blockDim.y + threadIdx.y};
110     size_t j{blockIdx.x * blockDim.x + threadIdx.x};
111     size_t l{blockIdx.z};
112
113     // Do not process outside the matrix.
114     // Do not forget the equal sign!
115     if ((i ≥ m) || (j ≥ p))
116     {
117         return;
118     }
119
120     T acc_sum{0};
121     for (size_t k{0}; k < n; ++k)
122     {
123         acc_sum += mat_1[l * m * n + i * n + k] * mat_2[l * n * p + k * p + j];
124     }
125     mat_3[l * m * p + i * p + j] = acc_sum;

```

```

126 }
127
128 template <typename T>
129 void mm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n,
130             size_t p)
131 {
132     dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
133     dim3 blocks_per_grid(1, 1);
134     blocks_per_grid.x = std::ceil(static_cast<double>(p) /
135                                 static_cast<double>(threads_per_block.x));
136     blocks_per_grid.y = std::ceil(static_cast<double>(m) /
137                                 static_cast<double>(threads_per_block.y));
138     mm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, m, n,
139                                                         p);
140 }
141
142 template <typename T>
143 void bmm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m,
144              size_t n, size_t p)
145 {
146     dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
147     dim3 blocks_per_grid(1, 1, 1);
148     blocks_per_grid.x = std::ceil(static_cast<double>(p) /
149                                 static_cast<double>(threads_per_block.x));
150     blocks_per_grid.y = std::ceil(static_cast<double>(m) /
151                                 static_cast<double>(threads_per_block.y));
152     blocks_per_grid.z = b;
153     bmm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, b,
154                                                         m, n, p);
155 }
156
157 template <typename T>
158 bool allclose(std::vector<T> const& vec_1, std::vector<T> const& vec_2,
159              T const& abs_tol)
160 {
161     if (vec_1.size() != vec_2.size())
162     {
163         return false;
164     }
165     for (size_t i{0}; i < vec_1.size(); ++i)
166     {
167         if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol)
168         {
169             std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
170             return false;
171         }
172     }
173     return true;
174 }
175
176 template <typename T>
177 bool random_test_mm_cuda(size_t m, size_t n, size_t p)
178 {
179     std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
180     std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};

```

```

181     std::vector<T> mat_3_vec(m * p);
182     std::vector<T> mat_4_vec(m * p);
183     T const* mat_1{mat_1_vec.data()};
184     T const* mat_2{mat_2_vec.data()};
185     T* mat_3{mat_3_vec.data()};
186     T* mat_4{mat_4_vec.data()};
187
188     mm(mat_1, mat_2, mat_3, m, n, p);
189
190     T *d_mat_1, *d_mat_2, *d_mat_4;
191
192     // Allocate device buffer.
193     checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
194     checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
195     checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));
196
197     // Copy data from host to device.
198     checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
199                          cudaMemcpyHostToDevice));
200     checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
201                          cudaMemcpyHostToDevice));
202
203     // Run matrix multiplication on GPU.
204     mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
205     cudaDeviceSynchronize();
206     cudaError_t err{cudaGetLastError()};
207     if (err != cudaSuccess)
208     {
209         std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
210                 << std::endl;
211         std::cerr << cudaGetErrorString(err) << std::endl;
212         std::exit(EXIT_FAILURE);
213     }
214
215     // Copy data from device to host.
216     checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
217                          cudaMemcpyDeviceToHost));
218
219     // Free device buffer.
220     checkCuda(cudaFree(d_mat_1));
221     checkCuda(cudaFree(d_mat_2));
222     checkCuda(cudaFree(d_mat_4));
223
224     return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
225 }
226
227 template <typename T>
228 bool random_test_bmm_cuda(size_t b, size_t m, size_t n, size_t p)
229 {
230     std::vector<T> const mat_1_vec{create_rand_vector<T>(b * m * n)};
231     std::vector<T> const mat_2_vec{create_rand_vector<T>(b * n * p)};
232     std::vector<T> mat_3_vec(b * m * p);
233     std::vector<T> mat_4_vec(b * m * p);
234     T const* mat_1{mat_1_vec.data()};
235     T const* mat_2{mat_2_vec.data()};

```

```

236     T* mat_3{mat_3_vec.data()};
237     T* mat_4{mat_4_vec.data()};
238
239     bmm(mat_1, mat_2, mat_3, b, m, n, p);
240
241     T *d_mat_1, *d_mat_2, *d_mat_4;
242
243     // Allocate device buffer.
244     checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
245     checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
246     checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));
247
248     // Copy data from host to device.
249     checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
250                          cudaMemcpyHostToDevice));
251     checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
252                          cudaMemcpyHostToDevice));
253
254     // Run matrix multiplication on GPU.
255     bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
256     cudaDeviceSynchronize();
257     cudaError_t err{cudaGetLastError()};
258     if (err != cudaSuccess)
259     {
260         std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
261                   << std::endl;
262         std::cerr << cudaGetErrorString(err) << std::endl;
263         std::exit(EXIT_FAILURE);
264     }
265
266     // Copy data from device to host.
267     checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
268                          cudaMemcpyDeviceToHost));
269
270     // Free device buffer.
271     checkCuda(cudaFree(d_mat_1));
272     checkCuda(cudaFree(d_mat_2));
273     checkCuda(cudaFree(d_mat_4));
274
275     return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
276 }
277
278 template <typename T>
279 bool random_multiple_test_mm_cuda(size_t num_tests)
280 {
281     std::random_device r;
282     std::default_random_engine e(r());
283     std::uniform_int_distribution<int> uniform_dist(1, 256);
284
285     size_t m{0}, n{0}, p{0};
286     bool success{false};
287
288     for (size_t i{0}; i < num_tests; ++i)
289     {
290         m = static_cast<size_t>(uniform_dist(e));

```

```

291     n = static_cast<size_t>(uniform_dist(e));
292     p = static_cast<size_t>(uniform_dist(e));
293     success = random_test_mm_cuda<T>(m, n, p);
294     if (!success)
295     {
296         return false;
297     }
298 }
299
300     return true;
301 }
302
303 template <typename T>
304 bool random_multiple_test_bmm_cuda(size_t num_tests)
305 {
306     std::random_device r;
307     std::default_random_engine e(r());
308     std::uniform_int_distribution<int> uniform_dist(1, 256);
309
310     size_t b{0}, m{0}, n{0}, p{0};
311     bool success{false};
312
313     for (size_t i{0}; i < num_tests; ++i)
314     {
315         b = static_cast<size_t>(uniform_dist(e));
316         m = static_cast<size_t>(uniform_dist(e));
317         n = static_cast<size_t>(uniform_dist(e));
318         p = static_cast<size_t>(uniform_dist(e));
319         success = random_test_bmm_cuda<T>(b, m, n, p);
320         if (!success)
321         {
322             return false;
323         }
324     }
325
326     return true;
327 }
328
329 template <typename T>
330 float measure_latency_mm_cuda(size_t m, size_t n, size_t p, size_t num_tests,
331                               size_t num_warmups)
332 {
333     cudaEvent_t startEvent, stopEvent;
334     float time{0.0f};
335
336     checkCuda(cudaEventCreate(&startEvent));
337     checkCuda(cudaEventCreate(&stopEvent));
338
339     T *d_mat_1, *d_mat_2, *d_mat_4;
340
341     // Allocate device buffer.
342     checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
343     checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
344     checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));
345

```

```

346     for (size_t i{0}; i < num_warmups; ++i)
347     {
348         mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
349     }
350
351     checkCuda(cudaEventRecord(startEvent, 0));
352     for (size_t i{0}; i < num_tests; ++i)
353     {
354         mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
355     }
356     checkCuda(cudaEventRecord(stopEvent, 0));
357     checkCuda(cudaEventSynchronize(stopEvent));
358     cudaError_t err{cudaGetLastError()};
359     if (err != cudaSuccess)
360     {
361         std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
362             << std::endl;
363         std::cerr << cudaGetErrorString(err) << std::endl;
364         std::exit(EXIT_FAILURE);
365     }
366     checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));
367
368     // Free device buffer.
369     checkCuda(cudaFree(d_mat_1));
370     checkCuda(cudaFree(d_mat_2));
371     checkCuda(cudaFree(d_mat_4));
372
373     float latency{time / num_tests};
374
375     return latency;
376 }
377
378 template <typename T>
379 float measure_latency_bmm_cuda(size_t b, size_t m, size_t n, size_t p,
380                               size_t num_tests, size_t num_warmups)
381 {
382     cudaEvent_t startEvent, stopEvent;
383     float time{0.0f};
384
385     checkCuda(cudaEventCreate(&startEvent));
386     checkCuda(cudaEventCreate(&stopEvent));
387
388     T *d_mat_1, *d_mat_2, *d_mat_4;
389
390     // Allocate device buffer.
391     checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * b * m * n));
392     checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * b * n * p));
393     checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * b * m * p));
394
395     for (size_t i{0}; i < num_warmups; ++i)
396     {
397         bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
398     }
399
400     checkCuda(cudaEventRecord(startEvent, 0));

```



```

401     for (size_t i{0}; i < num_tests; ++i)
402     {
403         bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
404     }
405     checkCuda(cudaEventRecord(stopEvent, 0));
406     checkCuda(cudaEventSynchronize(stopEvent));
407     cudaError_t err{cudaGetLastError()};
408     if (err != cudaSuccess)
409     {
410         std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
411                 << std::endl;
412         std::cerr << cudaGetErrorString(err) << std::endl;
413         std::exit(EXIT_FAILURE);
414     }
415     checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));
416
417     // Free device buffer.
418     checkCuda(cudaFree(d_mat_1));
419     checkCuda(cudaFree(d_mat_2));
420     checkCuda(cudaFree(d_mat_4));
421
422     float latency{time / num_tests};
423
424     return latency;
425 }
426
427 int main()
428 {
429     constexpr size_t num_tests{10};
430
431     assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
432     assert(random_multiple_test_mm_cuda<float>(num_tests));
433     assert(random_multiple_test_mm_cuda<double>(num_tests));
434     assert(random_multiple_test_bmm_cuda<int32_t>(num_tests));
435     assert(random_multiple_test_bmm_cuda<float>(num_tests));
436     assert(random_multiple_test_bmm_cuda<double>(num_tests));
437
438     constexpr size_t num_measurement_tests{100};
439     constexpr size_t num_measurement_warmups{10};
440     size_t b{128}, m{1024}, n{1024}, p{1024};
441
442     float mm_cuda_int32_latency{measure_latency_mm_cuda<int32_t>(
443         m, n, p, num_measurement_tests, num_measurement_warmups)};
444     float mm_cuda_float_latency{measure_latency_mm_cuda<float>(
445         m, n, p, num_measurement_tests, num_measurement_warmups)};
446     float mm_cuda_double_latency{measure_latency_mm_cuda<double>(
447         m, n, p, num_measurement_tests, num_measurement_warmups)};
448
449     float bmm_cuda_int32_latency{measure_latency_bmm_cuda<int32_t>(
450         b, m, n, p, num_measurement_tests, num_measurement_warmups)};
451     float bmm_cuda_float_latency{measure_latency_bmm_cuda<float>(
452         b, m, n, p, num_measurement_tests, num_measurement_warmups)};
453     float bmm_cuda_double_latency{measure_latency_bmm_cuda<double>(
454         b, m, n, p, num_measurement_tests, num_measurement_warmups)};
455

```

```

456     std::cout << "Matrix Multiplication CUDA Latency" << std::endl;
457     std::cout << "m: " << m << " "
458         << "n: " << n << " "
459         << "p: " << p << std::endl;
460     std::cout << "INT32: " << std::fixed << std::setprecision(5)
461         << mm_cuda_int32_latency << " ms" << std::endl;
462     std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
463         << mm_cuda_float_latency << " ms" << std::endl;
464     std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
465         << mm_cuda_double_latency << " ms" << std::endl;
466
467     std::cout << "Batched Matrix Multiplication CUDA Latency" << std::endl;
468     std::cout << "b: " << b << " "
469         << "m: " << m << " "
470         << "n: " << n << " "
471         << "p: " << p << std::endl;
472     std::cout << "INT32: " << std::fixed << std::setprecision(5)
473         << bmm_cuda_int32_latency << " ms" << std::endl;
474     std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
475         << bmm_cuda_float_latency << " ms" << std::endl;
476     std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
477         << bmm_cuda_double_latency << " ms" << std::endl;
478 }

```

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.

```

1 | $ 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.

```

1 | $ cd /mnt/
2 | $ nvcc mm.cu -o mm -std=c++14
3 | $ ./mm
4 | Matrix Multiplication CUDA Latency
5 | m: 1024 n: 1024 p: 1024
6 | INT32: 1.11436 ms
7 | FLOAT: 0.98451 ms
8 | DOUBLE: 4.10433 ms
9 | Batched Matrix Multiplication CUDA Latency
10 | b: 128 m: 1024 n: 1024 p: 1024
11 | INT32: 125.26781 ms
12 | FLOAT: 124.67697 ms
13 | 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

$$\text{NopNbyte} \gg \text{BW}_{\text{math}} \text{BW}_{\text{mem}} \text{NopNbyte} \gg \text{BW}_{\text{math}} \text{BW}_{\text{mem}}$$

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,

$$\text{NopNbyte} = 2 \times mnp \times 4 = 14$$

For a modern GPU such as [NVIDIA RTX 3090](#), for FP32 math,

$$\text{BW}_{\text{math}} \text{BW}_{\text{mem}} = 35.580.936 = 38.0$$

We could see that the naive CUDA matrix multiplication implementation does not get even close to math-bound. Since Nop should be a constant in matrix multiplication, let's see if we could reduce Nbyte 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.

$$A = [A_{d \times d1}, 1A_{d \times d1}, 2 \dots A_{d \times d1}, n/d A_{d \times d2}, 1A_{d \times d2}, 2 \dots A_{d \times d2}, n/d \dots A_{d \times dm/d}, 1A_{d \times dm/d}, 2 \dots A_{d \times dm/d}, n/d]$$

$$B = [B_{d \times d1}, 1B_{d \times d1}, 2 \dots B_{d \times d1}, p/d B_{d \times d2}, 1B_{d \times d2}, 2 \dots B_{d \times d2}, p/d \dots B_{d \times dn/d}, 1B_{d \times dn/d}, 2 \dots B_{d \times dn/d}, p/d]$$

$$C = [C_{d \times d1}, 1C_{d \times d1}, 2 \dots C_{d \times d1}, p/d C_{d \times d2}, 1C_{d \times d2}, 2 \dots C_{d \times d2}, p/d \dots C_{d \times dm/d}, 1C_{d \times dm/d}, 2 \dots C_{d \times dm/d}, p/d]$$

$$C_{d \times di}, j = n/d \sum_k = 1 A_{d \times di}, k B_{d \times dk}, j$$

The decomposition does not alter the number of operations Nop .

$$Nop = 2d^3(nd)(mdpd) = 2mnp$$

Because small matrices $A_{d \times d, k}$ and $B_{d \times d, j}$ 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.

$$Nbyte = 2d^2 \times 4 \times (nd)(mdpd) = 8mnpd$$

Therefore,

$$NopNbyte = 2mnp8mnpd = d^4$$

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

```
1  #include <cassert>
2  #include <cstddef>
3  #include <cstdint>
4  #include <iomanip>
5  #include <iostream>
6  #include <random>
7  #include <stdexcept>
8  #include <vector>
9
10 #define BLOCK_DIM 32
11
12 #define checkCuda(val) check((val), #val, __FILE__, __LINE__)
13 template <typename T>
14 void check(T err, const char* const func, const char* const file,
15           const int line)
16 {
17     if (err != cudaSuccess)
18     {
19         std::cerr << "CUDA Runtime Error at: " << file << ":" << line
20                 << std::endl;
21         std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
22         std::exit(EXIT_FAILURE);
23     }
24 }
25
26 template <typename T>
27 std::vector<T> create_rand_vector(size_t n)
28 {
29     std::random_device r;
30     std::default_random_engine e(r());
31     std::uniform_int_distribution<int> uniform_dist(-256, 256);
32 }
```

```

33     std::vector<T> vec(n);
34     for (size_t i{0}; i < n; ++i)
35     {
36         vec.at(i) = static_cast<T>(uniform_dist(e));
37     }
38
39     return vec;
40 }
41
42 // mat_1: m x n
43 // mat_2: n x p
44 // mat_3: m x p
45 template <typename T>
46 void mm(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n, size_t p)
47 {
48     // Compute the cells in mat_3 sequentially.
49     for (size_t i{0}; i < m; ++i)
50     {
51         for (size_t j{0}; j < p; ++j)
52         {
53             T acc_sum{0};
54             for (size_t k{0}; k < n; ++k)
55             {
56                 acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
57             }
58             mat_3[i * p + j] = acc_sum;
59         }
60     }
61 }
62
63 template <typename T>
64 __global__ void mm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t m,
65                          size_t n, size_t p)
66 {
67     // 2D block and 2D thread
68     // Each thread computes one cell in mat_3.
69     size_t i{blockIdx.y * blockDim.y + threadIdx.y};
70     size_t j{blockIdx.x * blockDim.x + threadIdx.x};
71
72     // Do not process outside the matrix.
73     // Do not forget the equal sign!
74     if ((i ≥ m) || (j ≥ p))
75     {
76         return;
77     }
78
79     T acc_sum{0};
80     for (size_t k{0}; k < n; ++k)
81     {
82         acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
83     }
84     mat_3[i * p + j] = acc_sum;
85 }
86
87 template <typename T>

```

```

88 __global__ void mm_kernel_optimized(T const* mat_1, T const* mat_2, T* mat_3,
89                                     size_t m, size_t n, size_t p)
90 {
91     __shared__ T mat_1_tile[BLOCK_DIM][BLOCK_DIM];
92     __shared__ T mat_2_tile[BLOCK_DIM][BLOCK_DIM];
93
94     T acc_sum{0};
95
96     for (size_t tile_idx{0};
97         tile_idx < ceilf(static_cast<float>(n) / BLOCK_DIM); ++tile_idx)
98     {
99         size_t i{blockIdx.y * blockDim.y + threadIdx.y};
100        size_t j{tile_idx * blockDim.x + threadIdx.x};
101        if ((i < m) && (j < n))
102        {
103            mat_1_tile[threadIdx.y][threadIdx.x] = mat_1[i * n + j];
104        }
105        else
106        {
107            mat_1_tile[threadIdx.y][threadIdx.x] = 0;
108        }
109        i = tile_idx * blockDim.y + threadIdx.y;
110        j = blockIdx.x * blockDim.x + threadIdx.x;
111        if ((i < n) && (j < p))
112        {
113            mat_2_tile[threadIdx.y][threadIdx.x] = mat_2[i * p + j];
114        }
115        else
116        {
117            mat_2_tile[threadIdx.y][threadIdx.x] = 0;
118        }
119        __syncthreads();
120        for (size_t k{0}; k < BLOCK_DIM; ++k)
121        {
122            acc_sum += mat_1_tile[threadIdx.y][k] * mat_2_tile[k][threadIdx.x];
123        }
124        __syncthreads();
125    }
126
127    // 2D block and 2D thread
128    // Each thread computes one cell in mat_3.
129    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
130    size_t j{blockIdx.x * blockDim.x + threadIdx.x};
131
132    if ((i < m) && (j < p))
133    {
134        mat_3[i * p + j] = acc_sum;
135    }
136 }
137
138 template <typename T>
139 void mm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n,
140             size_t p,
141             void (*f)(T const*, T const*, T*, size_t, size_t, size_t))
142 {

```

```

143     dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
144     dim3 blocks_per_grid(1, 1);
145     blocks_per_grid.x = std::ceil(static_cast<double>(p) /
146                                 static_cast<double>(threads_per_block.x));
147     blocks_per_grid.y = std::ceil(static_cast<double>(m) /
148                                 static_cast<double>(threads_per_block.y));
149     f<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, m, n, p);
150 }
151
152 template <typename T>
153 bool allclose(std::vector<T> const& vec_1, std::vector<T> const& vec_2,
154              T const& abs_tol)
155 {
156     if (vec_1.size() != vec_2.size())
157     {
158         return false;
159     }
160     for (size_t i{0}; i < vec_1.size(); ++i)
161     {
162         if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol)
163         {
164             std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
165             return false;
166         }
167     }
168     return true;
169 }
170
171 template <typename T>
172 bool random_test_mm_cuda(size_t m, size_t n, size_t p,
173                          void (*)(T const*, T const*, T*, size_t, size_t,
174                                   size_t))
175 {
176     std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
177     std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};
178     std::vector<T> mat_3_vec(m * p);
179     std::vector<T> mat_4_vec(m * p);
180     T const* mat_1{mat_1_vec.data()};
181     T const* mat_2{mat_2_vec.data()};
182     T* mat_3{mat_3_vec.data()};
183     T* mat_4{mat_4_vec.data()};
184
185     mm(mat_1, mat_2, mat_3, m, n, p);
186
187     T *d_mat_1, *d_mat_2, *d_mat_4;
188
189     // Allocate device buffer.
190     checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
191     checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
192     checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));
193
194     // Copy data from host to device.
195     checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
196                         cudaMemcpyHostToDevice));
197     checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),

```

```

198         cudaMemcpyHostToDevice));
199
200     // Run matrix multiplication on GPU.
201     mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
202     cudaDeviceSynchronize();
203     cudaError_t err{cudaGetLastError()};
204     if (err != cudaSuccess)
205     {
206         std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
207             << std::endl;
208         std::cerr << cudaGetErrorString(err) << std::endl;
209         std::exit(EXIT_FAILURE);
210     }
211     // Copy data from device to host.
212     checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
213         cudaMemcpyDeviceToHost));
214
215     // Free device buffer.
216     checkCuda(cudaFree(d_mat_1));
217     checkCuda(cudaFree(d_mat_2));
218     checkCuda(cudaFree(d_mat_4));
219
220     return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
221 }
222
223 template <typename T>
224 bool random_multiple_test_mm_cuda(size_t num_tests,
225     void (*f)(T const*, T const*, T*, size_t,
226         size_t, size_t))
227 {
228     std::random_device r;
229     std::default_random_engine e(r());
230     std::uniform_int_distribution<int> uniform_dist(1, 256);
231
232     size_t m{0}, n{0}, p{0};
233     bool success{false};
234
235     for (size_t i{0}; i < num_tests; ++i)
236     {
237         m = static_cast<size_t>(uniform_dist(e));
238         n = static_cast<size_t>(uniform_dist(e));
239         p = static_cast<size_t>(uniform_dist(e));
240         success = random_test_mm_cuda<T>(m, n, p, f);
241         if (!success)
242         {
243             return false;
244         }
245     }
246
247     return true;
248 }
249
250 template <typename T>
251 float measure_latency_mm_cuda(size_t m, size_t n, size_t p, size_t num_tests,
252     size_t num_warmups,

```



```

253         void (*f)(T const*, T const*, T*, size_t, size_t,
254                   size_t))
255     {
256         cudaEvent_t startEvent, stopEvent;
257         float time{0.0f};
258
259         checkCuda(cudaEventCreate(&startEvent));
260         checkCuda(cudaEventCreate(&stopEvent));
261
262         T *d_mat_1, *d_mat_2, *d_mat_4;
263
264         // Allocate device buffer.
265         checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
266         checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
267         checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));
268
269         for (size_t i{0}; i < num_warmups; ++i)
270         {
271             mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
272         }
273
274         checkCuda(cudaEventRecord(startEvent, 0));
275         for (size_t i{0}; i < num_tests; ++i)
276         {
277             mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p, f);
278         }
279         checkCuda(cudaEventRecord(stopEvent, 0));
280         checkCuda(cudaEventSynchronize(stopEvent));
281         cudaError_t err{cudaGetLastError()};
282         if (err != cudaSuccess)
283         {
284             std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
285                         << std::endl;
286             std::cerr << cudaGetErrorString(err) << std::endl;
287             std::exit(EXIT_FAILURE);
288         }
289         checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));
290
291         // Free device buffer.
292         checkCuda(cudaFree(d_mat_1));
293         checkCuda(cudaFree(d_mat_2));
294         checkCuda(cudaFree(d_mat_4));
295
296         float latency{time / num_tests};
297
298         return latency;
299     }
300
301 int main()
302 {
303     constexpr size_t num_tests{10};
304
305     assert(random_multiple_test_mm_cuda<int32_t>(num_tests, mm_kernel));
306     assert(random_multiple_test_mm_cuda<float>(num_tests, mm_kernel));
307     assert(random_multiple_test_mm_cuda<double>(num_tests, mm_kernel));

```

```

308
309     assert(
310         random_multiple_test_mm_cuda<int32_t>(num_tests, mm_kernel_optimized));
311     assert(random_multiple_test_mm_cuda<float>(num_tests, mm_kernel_optimized));
312     assert(
313         random_multiple_test_mm_cuda<double>(num_tests, mm_kernel_optimized));
314
315     constexpr size_t num_measurement_tests{100};
316     constexpr size_t num_measurement_warmups{10};
317     const size_t m{1024}, n{1024}, p{1024};
318
319     float mm_cuda_int32_latency{measure_latency_mm_cuda<int32_t>(
320         m, n, p, num_measurement_tests, num_measurement_warmups, mm_kernel)};
321     float mm_cuda_float_latency{measure_latency_mm_cuda<float>(
322         m, n, p, num_measurement_tests, num_measurement_warmups, mm_kernel)};
323     float mm_cuda_double_latency{measure_latency_mm_cuda<double>(
324         m, n, p, num_measurement_tests, num_measurement_warmups, mm_kernel)};
325
326     std::cout << "Matrix Multiplication CUDA Latency" << std::endl;
327     std::cout << "m: " << m << " "
328         << "n: " << n << " "
329         << "p: " << p << std::endl;
330     std::cout << "INT32: " << std::fixed << std::setprecision(5)
331         << mm_cuda_int32_latency << " ms" << std::endl;
332     std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
333         << mm_cuda_float_latency << " ms" << std::endl;
334     std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
335         << mm_cuda_double_latency << " ms" << std::endl;
336
337     mm_cuda_int32_latency = measure_latency_mm_cuda<int32_t>(
338         m, n, p, num_measurement_tests, num_measurement_warmups,
339         mm_kernel_optimized);
340     mm_cuda_float_latency = measure_latency_mm_cuda<float>(
341         m, n, p, num_measurement_tests, num_measurement_warmups,
342         mm_kernel_optimized);
343     mm_cuda_double_latency = measure_latency_mm_cuda<double>(
344         m, n, p, num_measurement_tests, num_measurement_warmups,
345         mm_kernel_optimized);
346
347     std::cout << "Optimized Matrix Multiplication CUDA Latency" << std::endl;
348     std::cout << "m: " << m << " "
349         << "n: " << n << " "
350         << "p: " << p << std::endl;
351     std::cout << "INT32: " << std::fixed << std::setprecision(5)
352         << mm_cuda_int32_latency << " ms" << std::endl;
353     std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
354         << mm_cuda_float_latency << " ms" << std::endl;
355     std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
356         << mm_cuda_double_latency << " ms" << std::endl;
357 }

```

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.

```
1 $ nvcc mm_optimization.cu -o mm_optimization --std=c++14
2 $ ./mm_optimization
3 Matrix Multiplication CUDA Latency
4 m: 1024 n: 1024 p: 1024
5 INT32: 1.04373 ms
6 FLOAT: 1.02149 ms
7 DOUBLE: 3.83370 ms
8 Optimized Matrix Multiplication CUDA Latency
9 m: 1024 n: 1024 p: 1024
10 INT32: 0.84207 ms
11 FLOAT: 0.81759 ms
12 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