

Matrix Multiplication & Pybind

by Daniel Sanei

Project Proposal

- High performance CUDA-accelerated matrix multiplication using C++ and CUDA
- Use Pybind11 to bind CUDA kernels from C++ to Python
- Allows coding using Python (more accessible syntax) while leveraging CUDA's parallel processing capabilities by abstracting the higher complexity of C++

Background & Motivation

- Become more proficient in CUDA
- Solidify understanding of GPU architecture, memory optimization techniques
- Learn how to bind Python with C++

Project Proposal Goals

- 1 Implement matrix multiplication code using CUDA kernels in C++
- 2 Use Pybind11 to bind C++ to Python
- 3 Create a Python script to initiate the driver code
- 4 Compare this result with a baseline CPU matrix multiplication
- 5 Analyze potential areas for memory optimization

All Successful!

Project Structure

- final_project
- Python2Cuda_example

Matrix Multiplication

```
cuda_bind.cu ➔ X
cuda_bind (Global Scope) ⚡ launch_matrix_multiplication_shared_memory

1 // Imports
2 #include <stdio.h>
3 #include <chrono>
4
5 // GPU initializations
6 #define N 1024           // 1024 x 1024 matrix
7 #define BLOCK_SIZE 16    // where 1024 / 16 = 64 blocks => 64 x 64 blocks
8 #define THREADS_PER_BLOCK 256 // 16 x 16 = 256 threads, 32 threads per warp => 8 warps
9
10 // Perform matrix multiplication on GPU using global memory
11 __global__ void matrix_multiplication_global_memory(int* input1, int* input2, int* output)
12 {
13     // get rows, columns
14     int row = threadIdx.y + blockIdx.y * blockDim.y;
15     int col = threadIdx.x + blockIdx.x * blockDim.x;
16
17     // GPU matrix multiplication
18     if (row < N && col < N) {
19         int sum = 0;
20         for (int k = 0; k < N; k++) {
21             sum += input1[row * N + k] * input2[k * N + col];
22         }
23         output[row * N + col] = sum;
24     }
25 }
```

Matrix Multiplication

```
27     // Perform matrix multiplication on GPU using shared memory
28     __global__ void matrix_multiplication_shared_memory(int* input1, int* input2, int* output)
29     {
30         // define tile sizes
31         __shared__ int tile1[BLOCK_SIZE][BLOCK_SIZE];
32         __shared__ int tile2[BLOCK_SIZE][BLOCK_SIZE];
33
34         // get rows, columns
35         int row = threadIdx.y + blockIdx.y * blockDim.y;
36         int col = threadIdx.x + blockIdx.x * blockDim.x;
37
38         // GPU matrix multiplication
39         int sum = 0;
40         int tile_size = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
41         for (int s = 0; s < tile_size; s++) {
42
43             // load tiles A, B into shared memory
44             if (row < N && (s * BLOCK_SIZE + threadIdx.x) < N &&
45                 col < N && (s * BLOCK_SIZE + threadIdx.y) < N) {
46                 tile1[threadIdx.y][threadIdx.x] = input1[row * N + (s * BLOCK_SIZE + threadIdx.x)];
47                 tile2[threadIdx.y][threadIdx.x] = input2[(s * BLOCK_SIZE + threadIdx.y) * N + col];
48             }
49             __syncthreads();
50
51             // perform matrix multiplication
52             for (int k = 0; k < BLOCK_SIZE; k++) {
53                 sum += tile1[threadIdx.y][k] * tile2[k][threadIdx.x];
54             }
55             __syncthreads();
56         }
57
58         // transfer result from shared to global memory
59         if (row < N && col < N) {
60             output[row * N + col] = sum;
61         }
62     }
63 }
```

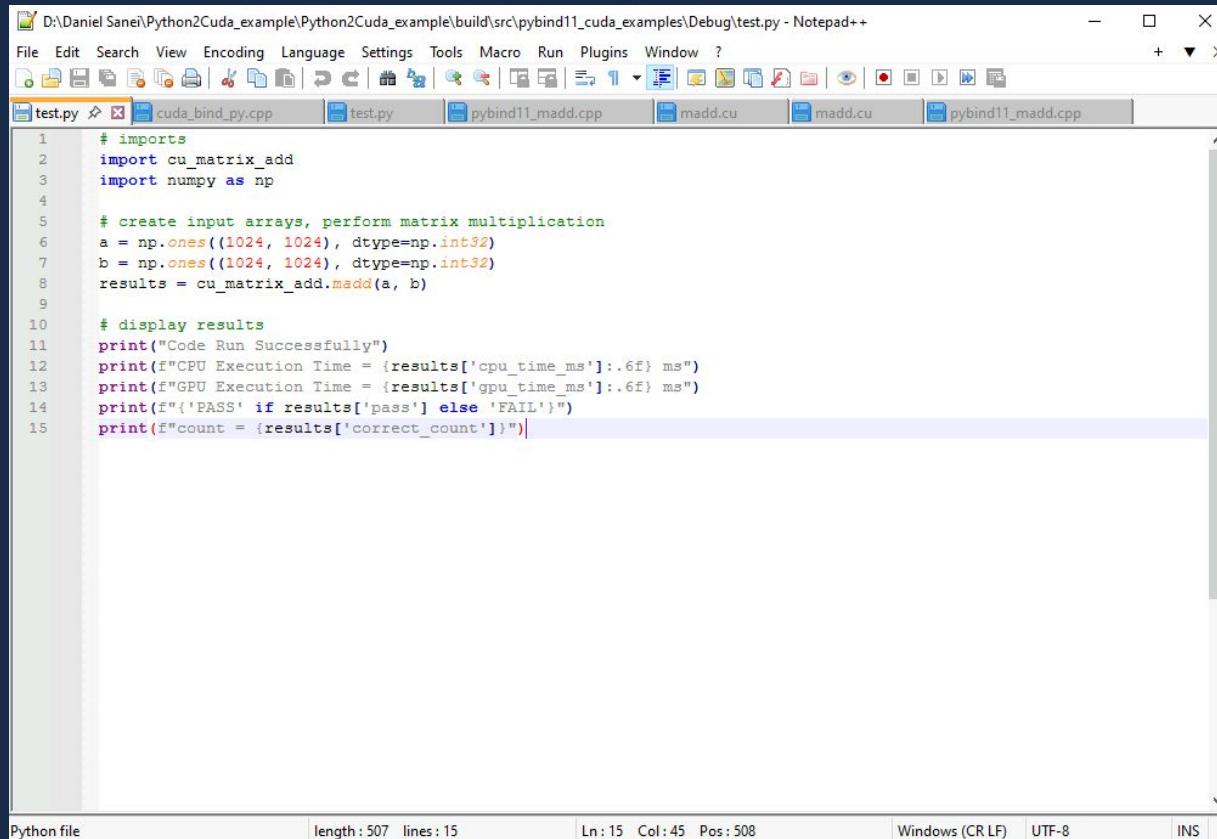
Matrix Multiplication

```
64 // Driver code
65
66 #include <cuda.h>
67 #include <math.h>
68 #include <sys/time.h>
69 #include <chrono>
70
71 // variables for matrices
72 int* input1, * input2, * output, * reference; // CPU (host)
73 int* gpu_input1, * gpu_input2, * gpu_output; // GPU (device)
74
75 // initialize block size
76 int size = (N * N) * sizeof(int);
77
78 // allocate memory for host matrices
79 input1 = (int*)malloc(size);
80 input2 = (int*)malloc(size);
81 output = (int*)malloc(size);
82 reference = (int*)malloc(size);
83
84 // initialize input matrices with randomly generated values
85 for (int i = 0; i < (N * N); i++) {
86     input1[i] = rand() % 10;
87     input2[i] = rand() % 10;
88     output[i] = 0;
89     reference[i] = 0;
90 }
91
92 // perform CPU matrix multiplication (measure execution time)
93 auto cpuStartTime = std::chrono::high_resolution_clock::now();
94 for (int row = 0; row < N; row++) {
95     for (int col = 0; col < N; col++) {
96         int sum = 0;
97         for (int k = 0; k < N; k++) {
98             sum += input1[row * N + k] * input2[k * N + col];
99         }
100        reference[row * N + col] = sum;
101    }
102 }
103
104 // determine CPU execution time
105 std::chrono::duration<float, std::milli> cpuExecutionTime = cpuEndTime - cpuStartTime;
106 printf("CPU Execution Time = %f ms\n", cpuExecutionTime.count());
107
108 // allocate memory for device matrices
109 cudaMalloc((void**)&gpu_input1, size);
110 cudaMalloc((void**)&gpu_input2, size);
111 cudaMalloc((void**)&gpu_output, size);
112
113 // copy input matrices to device
114 cudaMemcpy(gpu_input1, input1, size, cudaMemcpyHostToDevice);
115 cudaMemcpy(gpu_input2, input2, size, cudaMemcpyHostToDevice);
116
117 // define blocks, threads
118 // 2D block size, 16 x 16 = 256 threads
119 // total blocks, each is 16 x 16
120 dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
121 dim3 numBlocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);
122
123
124 // initialize timing event for measurement
125 cudaEvent_t gpuStartTime;
126 cudaEventCreate(&gpuStartTime);
127 cudaEvent_t gpuEndtime;
128 cudaEventCreate(&gpuEndtime);
129
130 // launch kernel function on GPU (measure execution time)
131 cudaEventRecord(gpuStartTime); // timestamp on GPU
132 matrix_multiplication_shared_memory << < numBlocks, threadsPerBlock >> (gpu_input1, gpu_input2, gpu_output);
133 cudaDeviceSynchronize();
134 cudaEventRecord(gpuEndtime); // timestamp on GPU
135
136 // copy resulting matrix back to host
137 cudaMemcpy(output, gpu_output, size, cudaMemcpyDeviceToHost);
138 cudaEventSynchronize(gpuEndtime); // ensure data transfer is complete before measuring end time
139
140 // determine GPU execution time
141 float ms = 0;
142 cudaEventElapsedTime(&ms, gpuStartTime, gpuEndtime);
143 printf("GPU Execution Time: %.6f ms\n", ms);
144
145 // clean up events
146 cudaEventDestroy(gpuStartTime);
147 cudaEventDestroy(gpuEndtime);
148
149 // compare device results with host reference
150 bool pass = true;
151 int correct_count = 0;
152 for (int i = 0; i < (N * N); i++) { // check all N*N elements in array (as 1D matrix)
153     if (reference[i] != output[i]) {
154         pass = false;
155     }
156     else if (reference[i] == output[i]) {
157         correct_count++;
158     }
159 }
160
161 // display comparison results
162 if (pass)
163     printf("PASS\n");
164 else
165     printf("FAIL\n");
166 printf("count = %d", correct_count);
167
168
```

Matrix Multiplication

```
164     // memory deallocation
165     free(input1);
166     free(input2);
167     free(output);
168     free(reference);
169     cudaFree(gpu_input1);
170     cudaFree(gpu_input2);
171     cudaFree(gpu_output);
172
173     // Return
174     return 0;
175 }
176
177 extern "C" void launch_matrix_multiplication_shared_memory(int* input1, int* input2, int* output) {
178     dim3 threadsPerBlock(16, 16);
179     dim3 numBlocks((1024 + 16 - 1) / 16, (1024 + 16 - 1) / 16);
180     matrix_multiplication_shared_memory<<<numBlocks, threadsPerBlock>>>(input1, input2, output);
181     cudaDeviceSynchronize();
182 }
```

Pybind



The screenshot shows a Notepad++ window with the following details:

- File Path: D:\Daniel Sanei\Python2Cuda_example\Python2Cuda_example\build\src\pybind11_cuda_examples\Debug\test.py
- File Type: Python file
- Length: 507 lines: 15
- Line: 15 Col: 45 Pos: 508
- Encoding: Windows (CR LF)
- Character Set: UTF-8
- Mode: INS

```
1 # imports
2 import cu_matrix_add
3 import numpy as np
4
5 # create input arrays, perform matrix multiplication
6 a = np.ones((1024, 1024), dtype=np.int32)
7 b = np.ones((1024, 1024), dtype=np.int32)
8 results = cu_matrix_add.madd(a, b)
9
10 # display results
11 print("Code Run Successfully")
12 print(f"CPU Execution Time = {results['cpu_time_ms']:.6f} ms")
13 print(f"GPU Execution Time = {results['gpu_time_ms']:.6f} ms")
14 print(f"{'PASS' if results['pass'] else 'FAIL'}")
15 print(f"count = {results['correct_count']}")|
```

Pybind

```
1 // Imports
2 #include <pybind11/pybind11.h>
3 #include <pybind11/numpy.h>
4 #include <chrono>
5 #include <cuda_runtime.h>
6
7 // declare external CUDA function (matrix multiplication using GPU shared memory)
8 extern "C" void launch_matrix_multiplication_shared_memory(int* input1, int* input2, int* output);
9
10 // keyword for pybind functionality
11 namespace py = pybind11;
12
13 // python wrapper function for matrix multiplication
14 py::dict madd_wrapper(py::array_t<int> a1, py::array_t<int> a2)
15 {
16     // matrix size N * N (1024 * 1024)
17     const int N = 1024;
18
19     // check matrix dimensions
20     if (a1.ndim() != 2 || a2.ndim() != 2)
21         throw std::runtime_error("Number of dimensions must be two");
22
23     // check matrix size
24     if (a1.shape(0) != N || a1.shape(1) != N ||
25         a2.shape(0) != N || a2.shape(1) != N)
26         throw std::runtime_error("Input matrices must be 1024x1024");
27
28     // get numpy array buffers
29     auto buf1 = a1.request();
30     auto buf2 = a2.request();
31
32     // create pointers for matrix data in host (both input matrices)
33     int* A = (int*)buf1.ptr;
34     int* B = (int*)buf2.ptr;
35
36     // allocate memory for host matrices
37     int* reference = new int[N * N];
38     int* output_cpu = new int[N * N];
```

Pybind

```
40 // CPU matrix multiplication (for reference)
41 auto cpuStartTime = std::chrono::high_resolution_clock::now(); // start of execution time
42 for (int row = 0; row < N; row++) {
43     for (int col = 0; col < N; col++) {
44         int sum = 0;
45         for (int k = 0; k < N; k++) {
46             sum += A[row * N + k] * B[k * N + col];
47         }
48         reference[row * N + col] = sum;
49     }
50 }
51 auto cpuEndTime = std::chrono::high_resolution_clock::now(); // end of execution time
52 std::chrono::duration<float, std::milli> cpuExecutionTime = cpuEndTime - cpuStartTime; // total execution
53
54 // declare GPU matrix pointers
55 int* d_input1, * d_input2, * d_output;
56 int size = N * N * sizeof(int);
57
58 // allocate memory for device matrices
59 cudaMalloc(&d_input1, size);
60 cudaMalloc(&d_input2, size);
61 cudaMalloc(&d_output, size);
62
63 // copy input matrices to device
64 cudaMemcpy(d_input1, A, size, cudaMemcpyHostToDevice);
65 cudaMemcpy(d_input2, B, size, cudaMemcpyHostToDevice);
66
67 // initialize timing event for measurement
68 cudaEvent_t gpuStartTime, gpuEndTime;
69 cudaEventCreate(&gpuStartTime);
70 cudaEventCreate(&gpuEndTime);
71
72 // launch kernel function on GPU (measure execution time)
73 cudaEventRecord(gpuStartTime);
74 launch_matrix_multiplication_shared_memory(d_input1, d_input2, d_output);
75 cudaEventRecord(gpuEndTime);
76 cudaEventSynchronize(gpuEndTime);
77
78 // determine GPU execution time
79 float gpuMs = 0;
80 cudaEventElapsedTime(&gpuMs, gpuStartTime, gpuEndTime);
```

Pybind

```
82     // copy resulting matrix back to host
83     auto result = py::array(py::buffer_info(
84         nullptr, sizeof(int), py::format_descriptor<int>::value, 2, { N, N },
85         { sizeof(int) * N, sizeof(int) }
86     ));
87     auto buf3 = result.request();
88     int* C = (int*)buf3.ptr;
89     cudaMemcpy(C, d_output, size, cudaMemcpyDeviceToHost);
90
91     // compare device results to host reference
92     bool pass = true;
93     int correct_count = 0;
94     for (int i = 0; i < N * N; i++) {    // check all N * N elements in array (as 1D matrix)
95         if (reference[i] != C[i]) {
96             pass = false;
97         }
98         else {
99             correct_count++;
100        }
101    }
102
103    // clean up memory, events
104    delete[] reference;
105    delete[] output_cpu;
106    cudaFree(d_input1);
107    cudaFree(d_input2);
108    cudaFree(d_output);
109    cudaEventDestroy(gpuStartTime);
110    cudaEventDestroy(gpuEndTime);
111
112    // display results
113    py::dict results;
114    results["cpu_time_ms"] = py::float_(cpuExecutionTime.count());
115    results["gpu_time_ms"] = py::float_(gpuMs);
116    results["pass"] = py::bool_(pass);
117    results["correct_count"] = py::int_(correct_count);
118
119    // return results
120    return results;
121 }
122
123 // define python module for pybind11
124 PYBIND11_MODULE(cu_matrix_add, m) {
125     m.doc() = "PyBind11 plugin for CUDA matrix multiplication";
126     m.def("madd", &madd_wrapper, "Perform matrix multiplication on GPU, return result and execution time");
127 }
```

Result - Pure CUDA

Result - CUDA with Pybind

The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Code Editor:** The main window displays the file `pybind11_madd.cpp`. The code implements a CUDA kernel for matrix multiplication and a Python binding for it. It includes imports for `pybind11`, `cuda`, and `cublas`. The CUDA kernel is named `madd` and performs shared memory matrix multiplication. The Python binding defines a function `madd` that takes two matrices and a result matrix, and returns a dictionary containing execution time, memory usage, and correctness information.
- Solution Explorer:** Shows the project structure for `Python2Cuda_example`, which contains three projects: `cu_matrix_add`, `cu_matrix_sub`, and `cu_matrix_dot`.
- Properties:** A floating window showing properties for the selected item.
- Status Bar:** Shows build output: "Done building project 'cu_matrix.add.vcxproj'." and "Rebuild All: 0 Succeeded, 0 Failed, 1 Skipped".
- Taskbar:** Shows the Windows taskbar with various pinned icons and the system tray.

A photograph of a modern concrete building with a green roof and large glass windows. A small, blue-painted wooden house is balanced precariously on the edge of the building's roof. The house has a dark shingled roof, a chimney, and several windows. The scene is set against a backdrop of a cloudy sky at dusk or dawn, with warm orange and pink hues. Some trees are visible in the foreground.

Thank You!