Matrix Multiplication & Pybind

by Daniel Sanei

ECE 277 | February 17, 2025

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

cuda_bind | (Global Scope) | launch_matrix_multiplication_shared_memory

```
// Imports
#include <stdio.h>
#include <chrono>

// GPU initializations
#define N 1024                      // 1024 x 1024 matrix
#define BLOCK_SIZE 16               // where 1024 / 16 = 64 blocks => 64 x 64 blocks
#define THREADS_PER_BLOCK 256       // 16 x 16 = 256 threads, 32 threads per warp => 8 warps

// Perform matrix multiplication on GPU using global memory
__global__ void matrix_multiplication_global_memory(int* input1, int* input2, int* output)
{
    // get rows, columns
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;

    // GPU matrix multiplication
    if (row < N && col < N) {
        int sum = 0;
        for (int k = 0; k < N; k++) {
            sum += input1[row * N + k] * input2[k * N + col];
        }
        output[row * N + col] = sum;
    }
}

```

Matrix Multiplication

```
// Perform matrix multiplication on GPU using shared memory
__global__ void matrix_multiplication_shared_memory(int* input1, int* input2, int* output)
{
    // define tile sizes
    __shared__ int tile1[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile2[BLOCK_SIZE][BLOCK_SIZE];

    // get rows, columns
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;

    // GPU matrix multiplication
    int sum = 0;
    int tile_size = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for (int s = 0; s < tile_size; s++) {

        // load tiles A, B into shared memory
        if (row < N && (s * BLOCK_SIZE + threadIdx.x) < N &&
            col < N && (s * BLOCK_SIZE + threadIdx.y) < N) {
            tile1[threadIdx.y][threadIdx.x] = input1[row * N + (s * BLOCK_SIZE + threadIdx.x)];
            tile2[threadIdx.y][threadIdx.x] = input2[(s * BLOCK_SIZE + threadIdx.y) * N + col];
        }
        __syncthreads();

        // perform matrix multiplication
        for (int k = 0; k < BLOCK_SIZE; k++) {
            sum += tile1[threadIdx.y][k] * tile2[k][threadIdx.x];
        }
        __syncthreads();
    }

    // transfer result from shared to global memory
    if (row < N && col < N) {
        output[row * N + col] = sum;
    }
}
```

Matrix Multiplication

```
// Driver code
int main()
{
    // variables for matrices
    int* input1, * input2, * output, * reference;      // CPU (host)
    int* gpu_input1, * gpu_input2, * gpu_output;      // GPU (device)

    // initialize block size
    int size = (N * N) * sizeof(int);

    // allocate memory for host matrices
    input1 = (int*)malloc(size);
    input2 = (int*)malloc(size);
    output = (int*)malloc(size);
    reference = (int*)malloc(size);

    // initialize input matrices with randomly generated values
    for (int i = 0; i < (N * N); i++) {
        input1[i] = rand() % 10;
        input2[i] = rand() % 10;
        output[i] = 0;
        reference[i] = 0;
    }

    // perform CPU matrix multiplication (measure execution time)
    auto cpuStartTime = std::chrono::high_resolution_clock::now();
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += input1[row * N + k] * input2[k * N + col];
            }
            reference[row * N + col] = sum;
        }
    }
    auto cpuEndTime = std::chrono::high_resolution_clock::now();

    // determine CPU execution time
    std::chrono::duration<float, std::milli> cpuExecutionTime = cpuEndTime - cpuStartTime;
    printf("CPU Execution Time = %f ms\n", cpuExecutionTime.count());

    // allocate memory for device matrices
    cudaMalloc((void**)&gpu_input1, size);
    cudaMalloc((void**)&gpu_input2, size);
    cudaMalloc((void**)&gpu_output, size);

    // copy input matrices to device
    cudaMemcpy(gpu_input1, input1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_input2, input2, size, cudaMemcpyHostToDevice);

    // define blocks, threads
        // 2D block size, 16 x 16 = 256 threads
        // total blocks, each is 16 x 16
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

    // initialize timing event for measurement
    cudaEvent_t gpuStartTime;
    cudaEventCreate(&gpuStartTime);
    cudaEvent_t gpuEndTime;
    cudaEventCreate(&gpuEndTime);

    // launch kernel function on GPU (measure execution time)
    cudaEventRecord(gpuStartTime);  // timestamp on GPU
    matrix_multiplication_shared_memory << < numBlocks, threadsPerBlock >> > (gpu_input1, gpu_input2, gpu_o
    cudaDeviceSynchronize();
    cudaEventRecord(gpuEndTime);    // timestamp on GPU

    // copy resulting matrix back to host
    cudaMemcpy(output, gpu_output, size, cudaMemcpyDeviceToHost);
    cudaEventSynchronize(gpuEndTime);   // ensure data transfer is complete before measuring end time

    // determine GPU execution time
    float ms = 0;
    cudaEventElapsedTime(&ms, gpuStartTime, gpuEndTime);
    printf("GPU Execution Time: %.6f ms\n", ms);

    // clean up events
    cudaEventDestroy(gpuStartTime);
    cudaEventDestroy(gpuEndTime);

    // compare device results with host reference
    bool pass = true;
    int correct_count = 0;
    for (int i = 0; i < (N * N); i++) {       // check all N*N elements in array (as 1D matrix)
        if (reference[i] != output[i]) {
            pass = false;
        }
        else if (reference[i] == output[i]) {
            correct_count++;
        }
    }

    // display comparison results
    if (pass)
        printf("PASS\n");
    else
        printf("FAIL\n");
    printf("count = %d", correct_count);
```

Matrix Multiplication

```
    // memory deallocation
    free(input1);
    free(input2);
    free(output);
    free(reference);
    cudaFree(gpu_input1);
    cudaFree(gpu_input2);
    cudaFree(gpu_output);

    // Return
    return 0;
}

extern "C" void launch_matrix_multiplication_shared_memory(int* input1, int* input2, int* output) {
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((1024 + 16 - 1) / 16, (1024 + 16 - 1) / 16);
    matrix_multiplication_shared_memory<<<numBlocks, threadsPerBlock>>>(input1, input2, output);
    cudaDeviceSynchronize();
}
```

Pybind

D:\Daniel Sanei\Python2Cuda_example\Python2Cuda_example\build\src\pybind11_cuda_examples\Debug\test.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

test.py | cuda_bind_py.cpp | test.py | pybind11_madd.cpp | madd.cu | madd.cu | pybind11_madd.cpp

```python
# imports
import cu_matrix_add
import numpy as np

# create input arrays, perform matrix multiplication
a = np.ones((1024, 1024), dtype=np.int32)
b = np.ones((1024, 1024), dtype=np.int32)
results = cu_matrix_add.madd(a, b)

# display results
print("Code Run Successfully")
print(f"CPU Execution Time = {results['cpu_time_ms']:.6f} ms")
print(f"GPU Execution Time = {results['gpu_time_ms']:.6f} ms")
print(f"{'PASS' if results['pass'] else 'FAIL'}")
print(f"count = {results['correct_count']}")
```

Python file | length : 507 lines : 15 | Ln : 15 Col : 45 Pos : 508 | Windows (CR LF) | UTF-8 | INS

Pybind

```cpp
// Imports
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include <chrono>
#include <cuda_runtime.h>

// declare external CUDA function (matrix multiplication using GPU shared memory)
extern "C" void launch_matrix_multiplication_shared_memory(int* input1, int* input2, int* output);

// keyword for pybind functionality
namespace py = pybind11;

// python wrapper function for matrix multiplication
py::dict madd_wrapper(py::array_t<int> a1, py::array_t<int> a2)
{
    // matrix size N * N (1024 * 1024)
    const int N = 1024;

    // check matrix dimensions
    if (a1.ndim() != 2 || a2.ndim() != 2)
        throw std::runtime_error("Number of dimensions must be two");

    // check matrix size
    if (a1.shape(0) != N || a1.shape(1) != N ||
        a2.shape(0) != N || a2.shape(1) != N)
        throw std::runtime_error("Input matrices must be 1024x1024");

    // get numpy array buffers
    auto buf1 = a1.request();
    auto buf2 = a2.request();

    // create pointers for matrix data in host (both input matrices)
    int* A = (int*)buf1.ptr;
    int* B = (int*)buf2.ptr;

    // allocate memory for host matrices
    int* reference = new int[N * N];
    int* output_cpu = new int[N * N];
```

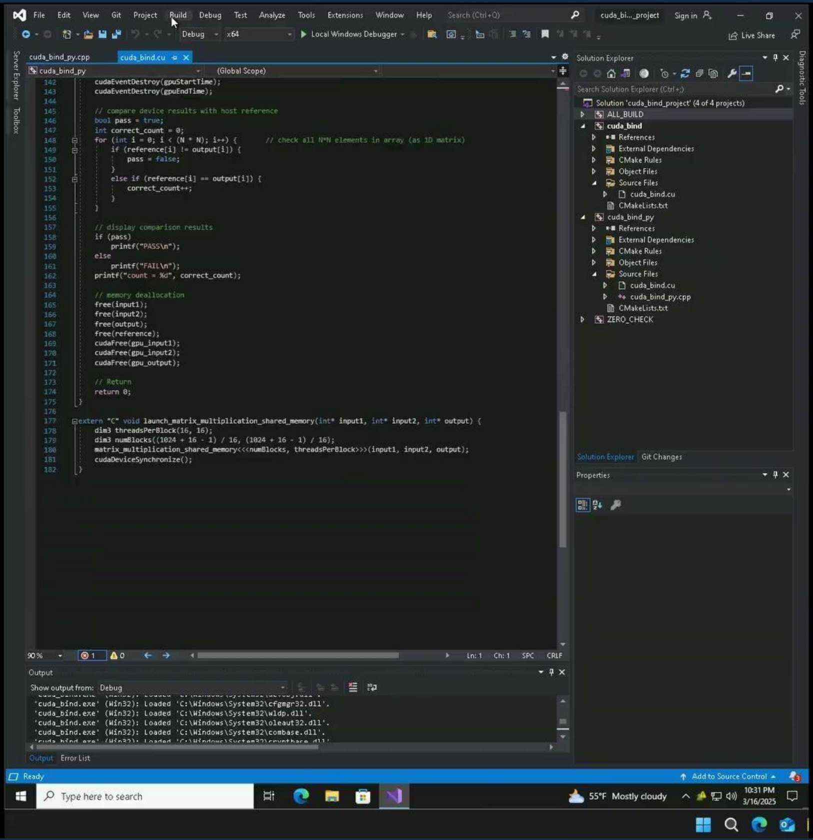
Pybind

```
// CPU matrix multiplication (for reference)
auto cpuStartTime = std::chrono::high_resolution_clock::now();  // start of execution time
for (int row = 0; row < N; row++) {
    for (int col = 0; col < N; col++) {
        int sum = 0;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        reference[row * N + col] = sum;
    }
}
auto cpuEndTime = std::chrono::high_resolution_clock::now();    // end of execution time
std::chrono::duration<float, std::milli> cpuExecutionTime = cpuEndTime - cpuStartTime;  // total execution

// declare GPU matrix pointers
int* d_input1, * d_input2, * d_output;
int size = N * N * sizeof(int);

// allocate memory for device matrices
cudaMalloc(&d_input1, size);
cudaMalloc(&d_input2, size);
cudaMalloc(&d_output, size);

// copy input matrices to device
cudaMemcpy(d_input1, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_input2, B, size, cudaMemcpyHostToDevice);

// initialize timing event for measurement
cudaEvent_t gpuStartTime, gpuEndTime;
cudaEventCreate(&gpuStartTime);
cudaEventCreate(&gpuEndTime);

// launch kernel function on GPU (measure execution time)
cudaEventRecord(gpuStartTime);
launch_matrix_multiplication_shared_memory(d_input1, d_input2, d_output);
cudaEventRecord(gpuEndTime);
cudaEventSynchronize(gpuEndTime);

// determine GPU execution time
float gpuMs = 0;
cudaEventElapsedTime(&gpuMs, gpuStartTime, gpuEndTime);
```

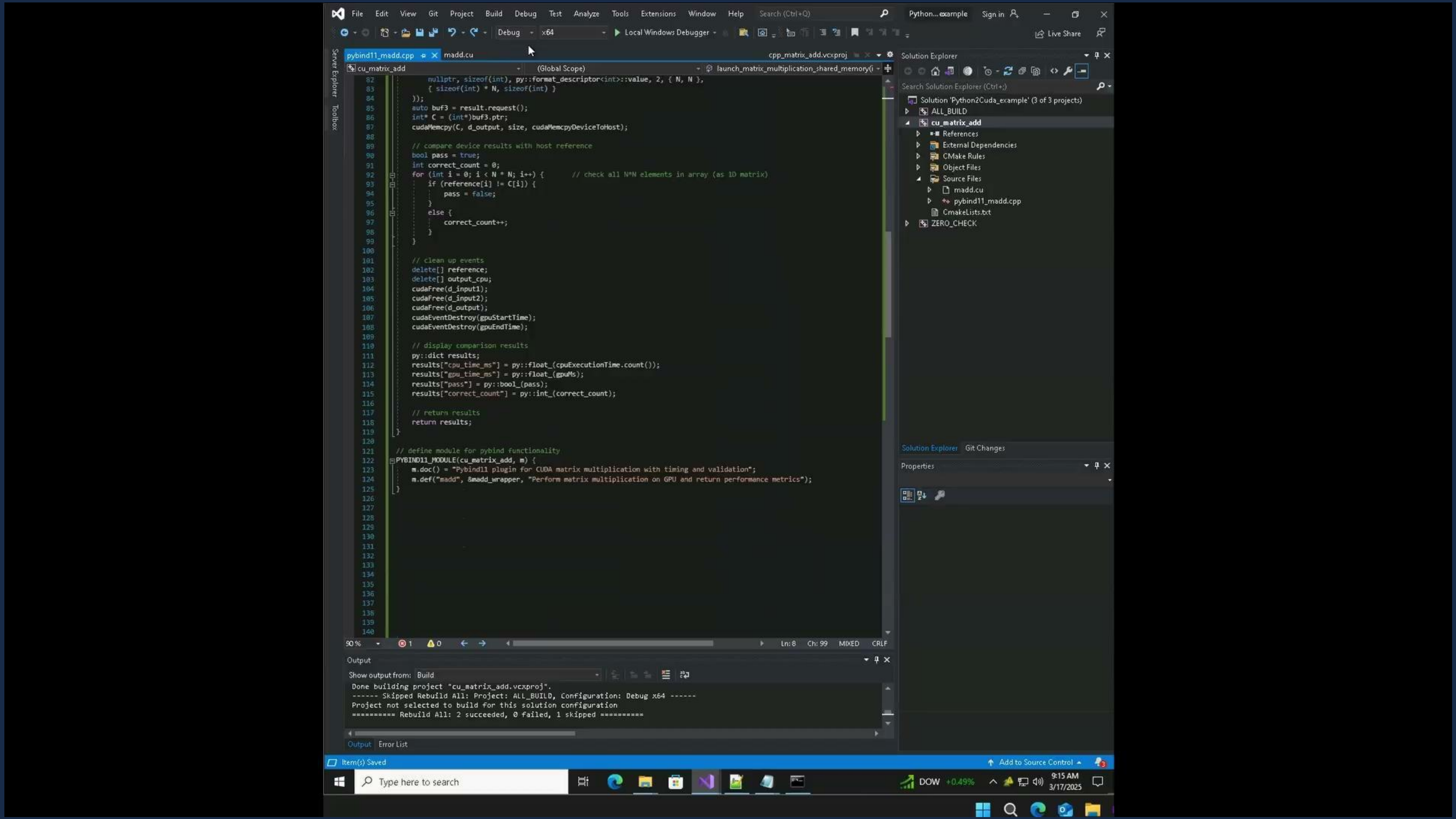
Pybind

```
    // copy resulting matrix back to host
    auto result = py::array(py::buffer_info(
        nullptr, sizeof(int), py::format_descriptor<int>::value, 2, { N, N },
        { sizeof(int) * N, sizeof(int) }
    ));
    auto buf3 = result.request();
    int* C = (int*)buf3.ptr;
    cudaMemcpy(C, d_output, size, cudaMemcpyDeviceToHost);

    // compare device results to host reference
    bool pass = true;
    int correct_count = 0;
    for (int i = 0; i < N * N; i++) {   // check all N * N elements in array (as 1D matrix)
        if (reference[i] != C[i]) {
            pass = false;
        }
        else {
            correct_count++;
        }
    }

    // clean up memory, events
    delete[] reference;
    delete[] output_cpu;
    cudaFree(d_input1);
    cudaFree(d_input2);
    cudaFree(d_output);
    cudaEventDestroy(gpuStartTime);
    cudaEventDestroy(gpuEndTime);

    // display results
    py::dict results;
    results["cpu_time_ms"] = py::float_(cpuExecutionTime.count());
    results["gpu_time_ms"] = py::float_(gpuMs);
    results["pass"] = py::bool_(pass);
    results["correct_count"] = py::int_(correct_count);

    // return results
    return results;
}

// define python module for pybind11
PYBIND11_MODULE(cu_matrix_add, m) {
    m.doc() = "Pybind11 plugin for CUDA matrix multiplication";
    m.def("madd", &madd_wrapper, "Perform matrix multiplication on GPU, return result and execution time");
}
```

Result – Pure CUDA



Result - CUDA with Pybind



UC San Diego

Thank You!