CUDA Program and Custom Python Library Findings

Aditya Venkataramani

Objectives

1. Write and execute a C program performing large matrix operations on a CPU.
2. Measure CPU execution performance with different matrix sizes.
3. Port the program to CUDA and execute it on a GPU.
4. Deploy a GPU-enabled virtual machine on Google Cloud and run CUDA programs.
5. Optimize CUDA code for better GPU performance.
6. Compare performance results between CPU, naïve CUDA, optimized CUDA, and cuBLAS implementations.
7. Analyze performance scaling as the problem size grows.
8. Create shared libraries using CUDA and make use of the acceleration from Python.

Matrix Multiplication on the CPU

The CPU implementation uses a standard triple-nested loop compiled with optimization level `-O2`. I observed that execution time increases rapidly as matrix size grows. These results establish a baseline for GPU comparison.

```
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % gcc matrix_cpu.c -o matrix_cpu -O2
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matrix_cpu 512
CPU execution time (N=512): 0.153796 seconds
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matrix_cpu 1024
CPU execution time (N=1024): 1.304792 seconds
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matrix_cpu 2048
CPU execution time (N=2048): 20.267226 seconds
```

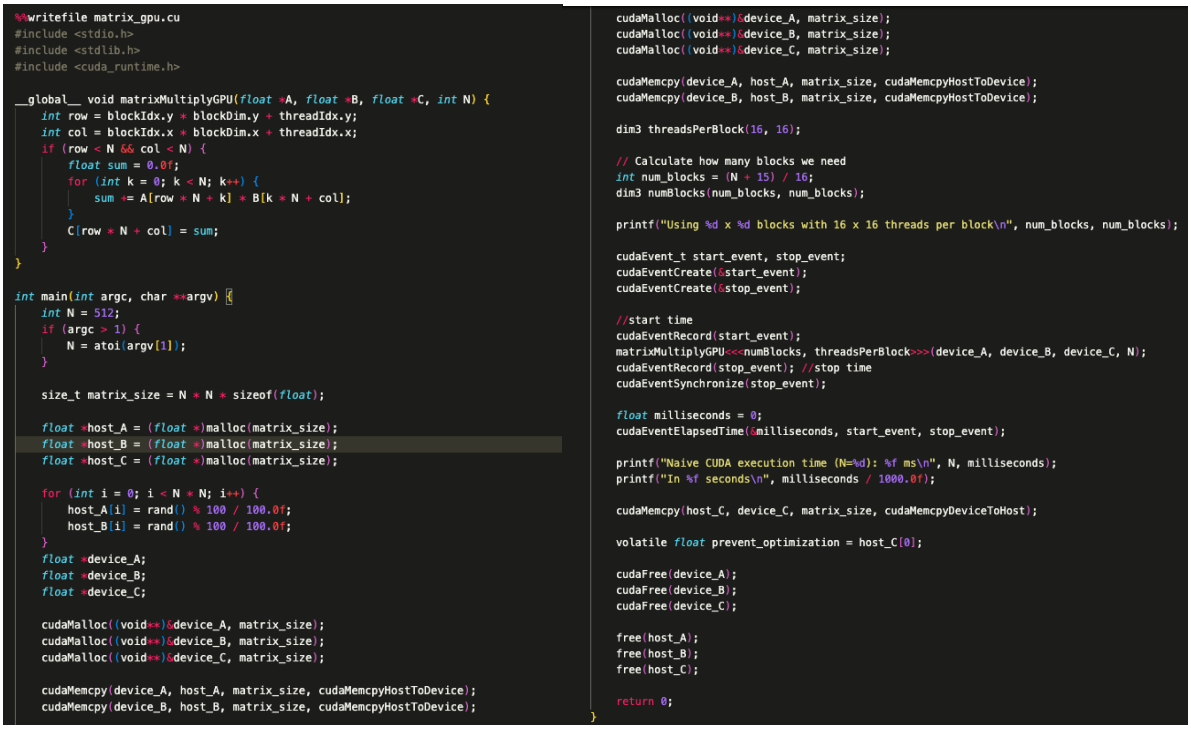
```
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % gcc matvec_cpu.c -o matvec_cpu
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matvec_cpu 512
CPU execution time (N=512): 0.001675 seconds
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matvec_cpu 1024
CPU execution time (N=1024): 0.006250 seconds
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % ./matvec_cpu 2048
CPU execution time (N=2048): 0.017114 seconds
```

I also implemented matrix multiplication with Python (not required by the assignment) to compare Python runtime with C. Due to the overhead that comes with Python, C clearly beats Python in runtime.

```
(base) adityavenkat@Adityas-MacBook-Air-2 cpu % python matrix_cpu.py
Matrix multiplication of size 512x512 completed in 7.810238 seconds.
Matrix multiplication of size 1024x1024 completed in 65.458638 seconds.
Matrix multiplication of size 2048x2048 completed in 839.488353 seconds.
```

```
(base) adityavenkat@usc-guestwireless-upc-new175 cpu % python matvec_cpu.py
Matrix-vector multiplication of size 512x512 completed in 0.0109889507 seconds.
Matrix-vector multiplication of size 1024x1024 completed in 0.0462450981 seconds.
Matrix-vector multiplication of size 2048x2048 completed in 0.1884961128 seconds.
(base) adityavenkat@usc-guestwireless-upc-new175 cpu %
```

Introduction to CUDA Programming



```
%%writefile matrix_gpu.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void matrixMultiplyGPU(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

int main(int argc, char **argv) {
    int N = 512;
    if (argc > 1) {
        N = atoi(argv[1]);
    }

    size_t matrix_size = N * N * sizeof(float);

    float *host_A = (float *)malloc(matrix_size);
    float *host_B = (float *)malloc(matrix_size);
    float *host_C = (float *)malloc(matrix_size);

    for (int i = 0; i < N * N; i++) {
        host_A[i] = rand() % 100 / 100.0f;
        host_B[i] = rand() % 100 / 100.0f;
    }
    float *device_A;
    float *device_B;
    float *device_C;

    cudaMalloc((void**)&device_A, matrix_size);
    cudaMalloc((void**)&device_B, matrix_size);
    cudaMalloc((void**)&device_C, matrix_size);

    cudaMemcpy(device_A, host_A, matrix_size, cudaMemcpyHostToDevice);
    cudaMemcpy(device_B, host_B, matrix_size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&device_A, matrix_size);
    cudaMalloc((void**)&device_B, matrix_size);
    cudaMalloc((void**)&device_C, matrix_size);

    cudaMemcpy(device_A, host_A, matrix_size, cudaMemcpyHostToDevice);
    cudaMemcpy(device_B, host_B, matrix_size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);

    // Calculate how many blocks we need
    int num_blocks = (N + 15) / 16;
    dim3 numBlocks(num_blocks, num_blocks);

    printf("Using %d x %d blocks with 16 x 16 threads per block\n", num_blocks, num_blocks);

    cudaEvent_t start_event, stop_event;
    cudaEventCreate(&start_event);
    cudaEventCreate(&stop_event);

    //start time
    cudaEventRecord(start_event);
    matrixMultiplyGPU<<<numBlocks, threadsPerBlock>>>(device_A, device_B, device_C, N);
    cudaEventRecord(stop_event); //stop time
    cudaEventSynchronize(stop_event);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start_event, stop_event);

    printf("Naive CUDA execution time (N=%d): %f ms\n", N, milliseconds);
    printf("In %f seconds\n", milliseconds / 1000.0f);

    cudaMemcpy(host_C, device_C, matrix_size, cudaMemcpyDeviceToHost);

    volatile float prevent_optimization = host_C[0];

    cudaFree(device_A);
    cudaFree(device_B);
    cudaFree(device_C);

    free(host_A);
    free(host_B);
    free(host_C);

    return 0;
}
```

Naïve CUDA code

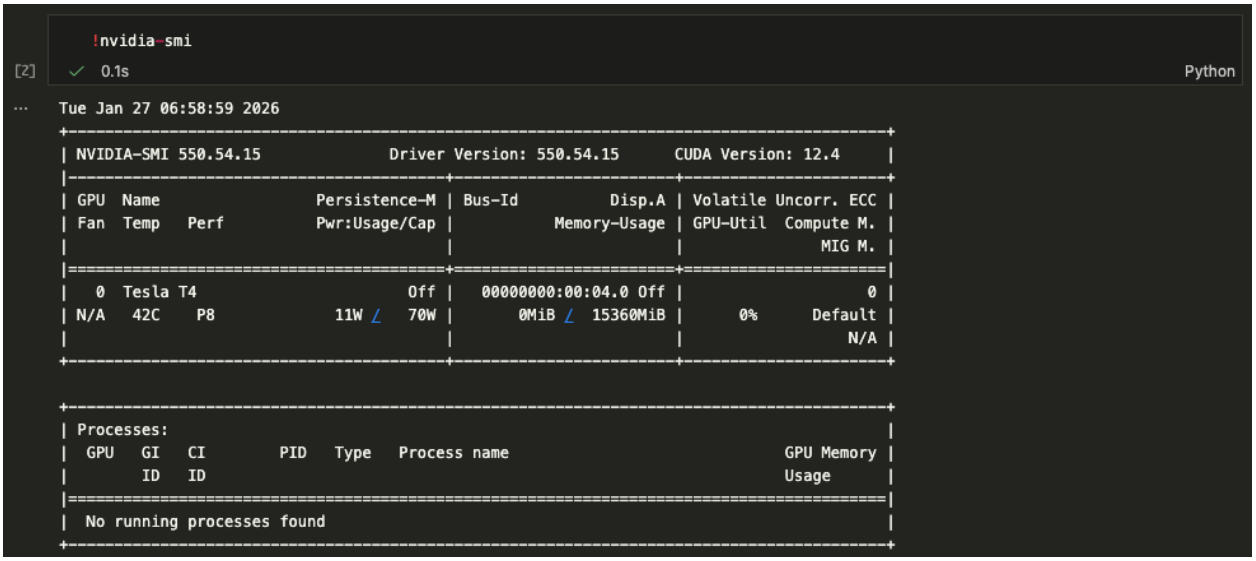
I tested the runtimes for N=512, 1024, 2048, 4096, and 8192. The program allocates memory on the CPU for matrices A, B, and C and fills A and B with random numbers. Then it allocates memory on the GPU and copies A and B from the CPU's memory to the GPU's memory. The program sets up a grid of 16 by 16 threads per block and calculates the number of blocks needed to cover the whole matrix using the formula (N + 15) / 16.

The `matrixMultiplyGPU` function defines a GPU kernel where each thread computes a single element of the output matrix C. Each thread determines its row and column based on its block and thread indices, and calculates the dot product of that row of A and column of B to produce the value in C.

The elapsed time includes the time to launch the GPU kernel and perform the matrix multiplication, and it is printed in milliseconds. After the kernel finishes, the resulting matrix C is copied back to the CPU. A volatile read is used to guarantee that it completes the matrix multiplication. At the end, all memory in the GPU and CPU is freed.

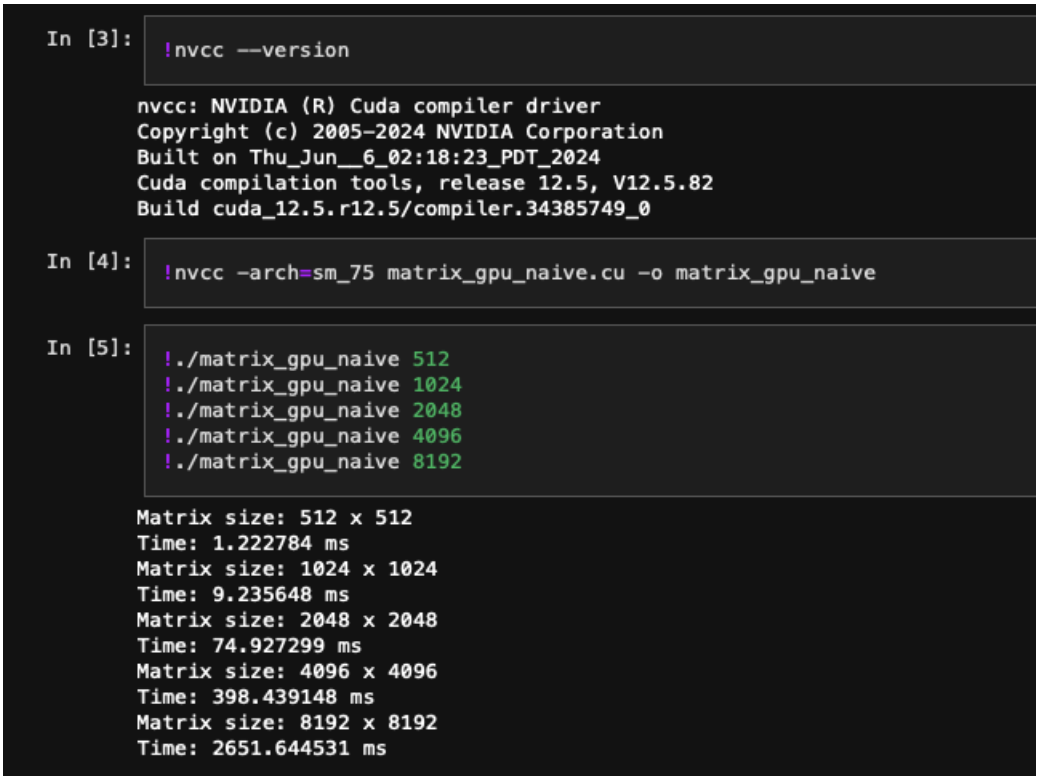
In the naive CUDA version, each GPU thread computes a single output matrix element using global memory only. I observed that the first run (N=512) takes the longest despite being the smallest N due the GPU initialization overhead (memory allocation, load kernels, GPU clock ramp-up, etc.). All later runs reuse an already-initialized CUDA context, so they're much faster and have similar runtimes despite the increases in matrix sizes. Thus, I can see that parallelism significantly reduces execution time compared to the CPU.

Running CUDA on Google Cloud



I used an NVIDIA Tesla T4 GPU in Google Colab. Since creating a VM in Google Cloud Compute Engine is costly, I first tested CUDA (GPU performance) in Colab, which is a quick set up and freely accessible to students through Colab Pro, offering better GPUs and resources.

Run Naïve CUDA Program



Naïve CUDA: The GPU runtimes increase with N, but the GPU runtime is significantly faster than CPU runtime.

Optimizing CUDA Code

Optimized (Memory Tiled) CUDA Code:

```
//start time
cudaEventRecord(start_event);
matrixMultiplyTiled<<<numBlocks, threadsPerBlock>>>(device_A, device_B, device_C, N);
cudaEventRecord(stop_event); //stop time
cudaEventSynchronize(stop_event);
```

```
%%writefile matrix_gpu_tiled.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#define TILE 16

__global__ void matmul_tiled(float *a, float *b, float *c, int n) {
    __shared__ float tile_a[TILE][TILE];
    __shared__ float tile_b[TILE][TILE];

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row = blockIdx.y * TILE + ty;
    int col = blockIdx.x * TILE + tx;

    float sum = 0.0f;

    for (int m = 0; m < (n + TILE - 1) / TILE; m++) {
        if (row < n && (m * TILE + tx) < n)
            tile_a[ty][tx] = a[row * n + m * TILE + tx];
        else
            tile_a[ty][tx] = 0.0f;

        if (col < n && (m * TILE + ty) < n)
            tile_b[ty][tx] = b[(m * TILE + ty) * n + col];
        else
            tile_b[ty][tx] = 0.0f;

        __syncthreads();

        for (int k = 0; k < TILE; k++)
            sum += tile_a[ty][k] * tile_b[k][tx];

        __syncthreads();
    }

    if (row < n && col < n)
        c[row * n + col] = sum;
}
```

```
int main(int argc, char **argv) {
    int n = 512;
    if (argc > 1) {
        n = atoi(argv[1]);
    }

    printf("Matrix size: %d x %d\n", n, n);

    size_t size = n * n * sizeof(float);

    float *a = (float *)malloc(size);
    float *b = (float *)malloc(size);
    float *c = (float *)malloc(size);

    for (int i = 0; i < n * n; i++) {
        a[i] = rand() % 100 / 100.0f;
        b[i] = rand() % 100 / 100.0f;
    }

    float *d_a, *d_b, *d_c;

    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    dim3 threads(16, 16);
    dim3 blocks((n + 15) / 16, (n + 15) / 16);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    matmul_tiled<<<blocks, threads>>>(d_a, d_b, d_c, n);
    cudaEventRecord(stop);
```

```
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Time: %f ms\n", milliseconds);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

```
In [7]: !nvcc -arch=sm_75 matrix_gpu_tiled.cu -o matrix_gpu_tiled

In [8]: !./matrix_gpu_tiled 512
        !./matrix_gpu_tiled 1024
        !./matrix_gpu_tiled 2048
        !./matrix_gpu_tiled 4096
        !./matrix_gpu_tiled 8192

Matrix size: 512 x 512
Time: 0.322944 ms
Matrix size: 1024 x 1024
Time: 2.198464 ms
Matrix size: 2048 x 2048
Time: 46.329983 ms
Matrix size: 4096 x 4096
Time: 292.058533 ms
Matrix size: 8192 x 8192
Time: 1739.154175 ms
```

Optimized CUDA: The optimized kernel uses shared memory tiling to reduce global memory access and improve memory reuse. Again, I saw that GPU runtime increases as N increases. However, for each N, the optimized CUDA outperforms (is faster) than Naïve CUDA.

Performance Comparison

Implementation	N=512	N=1024	N=2048	N=4096	N=8192
CPU (C)	0.4900 sec	3.710 sec	37.630 sec	~10 min	
Naïve CUDA	1.223 ms	9.236 ms	74.927 ms	398.439 ms	2651.644 ms
Optimized CUDA	0.323 ms	2.198 ms	46.330 ms	292.059 ms	1739.154 ms

Compute **speedup = CPU time / GPU time**.

Implementation	N=512	N=1024	N=2048	N=4096
Naïve CUDA Speedup	401	402	502	1506
Optimized CUDA Speedup	1517	1688	812	2054

Using cuBLAS Library

```
%%writefile matrix_gpu_cublas.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>

int main(int argc, char **argv) {
    int N;
    if (argc > 1) {
        N = atoi(argv[1]);
    } else {
        N = 1024; // default size
    }

    size_t size = N * N * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 100 / 100.0f;
    }

    for (int i = 0; i < N * N; i++) {
        h_B[i] = rand() % 100 / 100.0f;
    }

    float *d_A;
    float *d_B;
    float *d_C;

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cublasHandle_t handle;
    cublasCreate(&handle);

    float alpha = 1.0f;
    float beta  = 0.0f;
```

```
    cudaEvent_t start;
    cudaEvent_t stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    cublasSgemm(handle,
                CUBLAS_OP_N, CUBLAS_OP_N,
                N, N, N,
                &alpha,
                d_B, N,
                d_A, N,
                &beta,
                d_C, N);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float ms;
    cudaEventElapsedTime(&ms, start, stop);
    printf("cuBLAS SGEMM time (N=%d): %f ms\n", N, ms);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    volatile float sink = h_C[0];

    cublasDestroy(handle);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

```
In [10]: !nvcc matrix_gpu_cublas.cu -o matrix_gpu_cublas -lcublas

In [11]: !./matrix_gpu_cublas 512
         !./matrix_gpu_cublas 1024
         !./matrix_gpu_cublas 2048
         !./matrix_gpu_cublas 4096
         !./matrix_gpu_cublas 8192

cuBLAS SGEMM time (N=512): 5.489536 ms
cuBLAS SGEMM time (N=1024): 6.142016 ms
cuBLAS SGEMM time (N=2048): 11.465792 ms
cuBLAS SGEMM time (N=4096): 53.734783 ms
cuBLAS SGEMM time (N=8192): 293.303375 ms
```

Implementation	N=512	N=1024	N=2048	N=4096	N=8192
CPU (C)	0.4900 sec	3.710 sec	37.630 sec	~10 min	
Naïve CUDA	1.223 ms	9.236 ms	74.927 ms	398.439 ms	2651.644 ms
Optimized CUDA	0.323 ms	2.198 ms	46.330 ms	292.059 ms	1739.154 ms
cuBLAS	5.489 ms	6.142 ms	11.466 ms	53.735 ms	293.303 ms

Analysis / Observations

Performance changed as matrix size increased:
As the matrix size N increases, runtime increases significantly across all implementations. The CPU scales very poorly, drastically growing from under one second at N=512 to approximately 10 minutes at N=4096. While GPU runtimes also increase with matrix size N, GPU performance remains several orders of magnitude faster than the CPU (hundreds to thousands of times faster). This demonstrates the GPU's superior ability to handle large-scale parallel computation, especially as problem size grows.

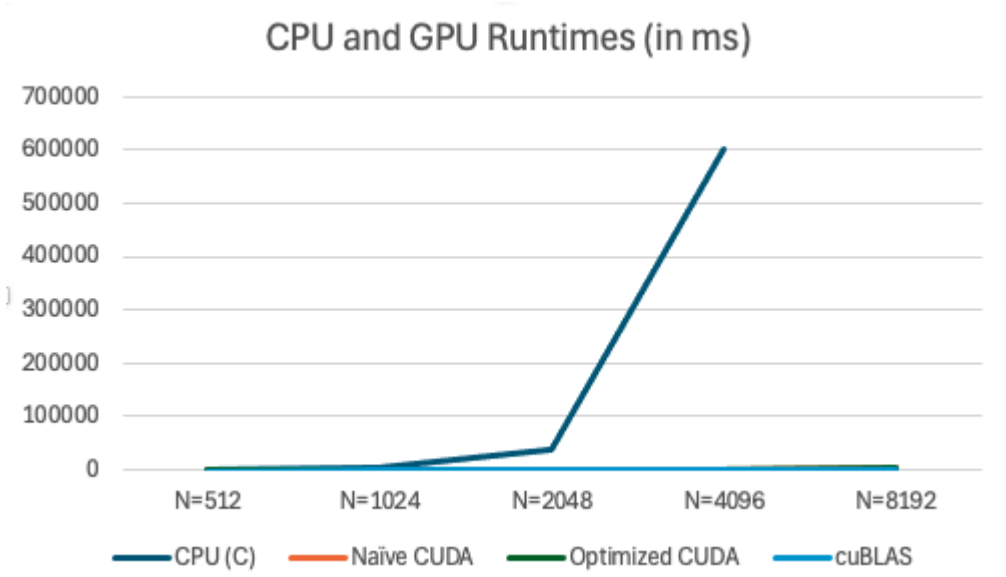
GPU significantly outperforms the CPU…but when?
The GPU already significantly outperforms the CPU when N=512, and the gap only widens as N increases. It goes from being about 400x faster at N=512 to a couple thousand times faster at N=4096. The advantages of GPUs become more pronounced at larger N.

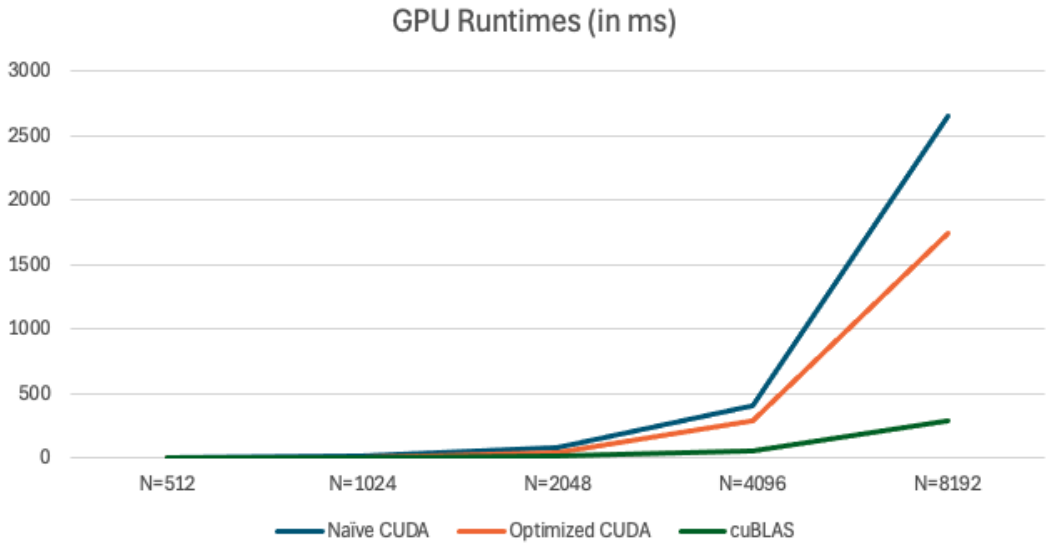
Speedup gained by tiling optimization vs. naïve CUDA:
The tiling optimization provides noticeable speedup over the naïve CUDA implementation across all tested matrix sizes. For smaller matrices, the optimized kernel is approximately 3-4x faster than the naïve version. For larger matrices, the improvement remains significant, with speedups ranging from roughly 1.3x to 2x. The improvement is due to more efficient memory access patterns and better use of shared memory, which reduce global memory latency.

Optimized kernel vs cuBLAS performance:
For smaller matrix sizes (N=512 and 1024), the optimized CUDA kernel outperforms cuBLAS, likely due to lower overhead and kernel launch costs dominating at small problem sizes. However for larger matrices (N=2048, 4096, and 8192), cuBLAS drastically outperformed the optimized kernel. As matrix size increases, cuBLAS scales much more efficiently, demonstrating its superior optimization for large workloads.

cuBLAS still outperforming hand-written kernels

cuBLAS is highly optimized by NVIDIA and tuned specifically for each GPU architecture. It leverages advanced techniques such as architecture-specific optimizations, highly efficient memory tiling strategies, instruction-level parallelism, and low-level hardware features that are difficult to replicate in hand-written kernels. Additionally, cuBLAS benefits from extensive testing and continuous optimization, allowing it to achieve near-peek hardware performance that is challenging to match with custom implementations.


CPU and GPU Runtimes (in ms)
700000
600000
500000
400000
300000
200000
100000
0
N=512
N=1024
N=2048
N=4096
N=8192
CPU (C)
Naïve CUDA
Optimized CUDA
cuBLAS


GPU Runtimes (in ms)
3000
2500
2000
1500
1000
500
0
N=512
N=1024
N=2048
N=4096
N=8192
Naïve CUDA
Optimized CUDA
cuBLAS

Creating a Shared Library and Using it in Python

```
%%writefile matrix_lib.cu
#include <cuda_runtime.h>
#include <stdio.h>
#define TILE_WIDTH 16

__global__ void matrixMultiplyTiled(float *A, float *B, float *C, int N) {
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Row = blockIdx.y * TILE_WIDTH + ty;
    int Col = blockIdx.x * TILE_WIDTH + tx;

    float Pvalue = 0.0f;

    int numTiles = (N + TILE_WIDTH - 1) / TILE_WIDTH;
    for (int m = 0; m < numTiles; ++m) {
        int aCol = m * TILE_WIDTH + tx;
        if (Row < N && aCol < N) {
            ds_A[ty][tx] = A[Row * N + aCol];
        } else {
            ds_A[ty][tx] = 0.0f;
        }

        int bRow = m * TILE_WIDTH + ty;
        if (Col < N && bRow < N) {
            ds_B[ty][tx] = B[bRow * N + Col];
        } else {
            ds_B[ty][tx] = 0.0f;
        }

        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += ds_A[ty][k] * ds_B[k][tx];
        }

        __syncthreads();
    }

    if (Row < N && Col < N) {
        C[Row * N + Col] = Pvalue;
    }
}
```

```
extern "C" void gpu_matrix_multiply(float *h_A, float *h_B, float *h_C, int N) {
    size_t size = N * N * sizeof(float);

    float *d_A;
    float *d_B;
    float *d_C;

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    dim3 block(TILE_WIDTH, TILE_WIDTH);

    int numBlocksX = (N + TILE_WIDTH - 1) / TILE_WIDTH;
    int numBlocksY = (N + TILE_WIDTH - 1) / TILE_WIDTH;
    dim3 grid(numBlocksX, numBlocksY);

    matrixMultiplyTiled<<<grid, block>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Above includes images of a CUDA kernel for multiplying two NxN matrices (A and B) and storing the result in C. This step drives from the logic I implemented in prior steps of using tiling

and shared memory to reduce global memory accesses and thus improve performance. The kernel loops over all tiles needed to compute the output element, and each iteration loads a tile from A and B into shared memory, synchronizes, then does a partial sum.

I added in a host wrapper function ‘extern "C" void gpu_matrix_multiply’ which is a C-style function for Python operations.

```
lib = ctypes.cdll.LoadLibrary("./libmatrix.so")
lib.gpu_matrix_multiply.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    ctypes.c_int
]

N = 512

for i in range(5):
    print(f"\nTesting with matrix size N = {N}")

    A = np.random.rand(N, N).astype(np.float32)
    B = np.random.rand(N, N).astype(np.float32)
    C = np.zeros((N, N), dtype=np.float32)

    start = time.time()
    lib.gpu_matrix_multiply(A.ravel(), B.ravel(), C.ravel(), N)
    end = time.time()
    elapsed_time = end - start

    print(f"CUDA library time (N={N}): {elapsed_time * 1000:.4f} milliseconds")

    N = N * 2
```

2.1s Python

```
Testing with matrix size N = 512
CUDA library time (N=512): 1.9896 milliseconds

Testing with matrix size N = 1024
CUDA library time (N=1024): 3.7229 milliseconds

Testing with matrix size N = 2048
CUDA library time (N=2048): 20.6728 milliseconds

Testing with matrix size N = 4096
CUDA library time (N=4096): 75.4676 milliseconds

Testing with matrix size N = 8192
CUDA library time (N=8192): 290.0529 milliseconds
```

The shared library shown in the screenshot above, loads the compiled CUDA shared library “libmatrix.so” to ensure I was able to call the function via Python script. I added a simple for loop, to loop over the matrix sizes, starting with a 512x512 matrix and it doubles in size up until 8192x8192. As expected, the results also show as matrix size increases computation time increases. The times are comparable to the results I received when running the compiled cuBLAS code by itself.

Adding Custom Functions to the Shared Library - Convolution Function

```
%%writefile conv_gpu.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void convolution2D_GPU(
    unsigned char *image,
    int *kernel,
    int *output,
    int width,
    int height
) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int sum = 0;
    int k = 1;  // 3x3 kernel radius

    for (int ky = -k; ky <= k; ky++) {
        for (int kx = -k; kx <= k; kx++) {
            int ix = x + kx;
            int iy = y + ky;
            if (ix >= 0 && ix < width && iy >= 0 && iy < height) {
                int pixel = image[iy * width + ix];
                int weight = kernel[(ky + k) * 3 + (kx + k)];
                sum += pixel * weight;
            }
        }
    }
    output[y * width + x] = sum;
}

int main() {
    int width = 512;
    int height = 512;
    size_t img_size = width * height * sizeof(unsigned char);
    size_t out_size = width * height * sizeof(int);

    unsigned char *h_img = (unsigned char*)malloc(img_size);
    int *h_out = (int*)malloc(out_size);

    int h_kernel[9] = {
        -1, -1, -1,
        -1,  8, -1,
        -1, -1, -1
    };

    for (int i = 0; i < width * height; i++)
        h_img[i] = rand() % 256;

    unsigned char *d_img;
    int *d_kernel, *d_out;

    cudaMalloc(&d_img, img_size);
    cudaMalloc(&d_kernel, 9 * sizeof(int));
    cudaMalloc(&d_out, out_size);

    cudaMemcpy(d_img, h_img, img_size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernel, h_kernel, 9 * sizeof(int), cudaMemcpyHostToDevice);

    dim3 block(16, 16);
    dim3 grid((width + 15) / 16, (height + 15) / 16);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    convolution2D_GPU<<<grid, block>>>(d_img, d_kernel, d_out, width, height);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms;
    cudaEventElapsedTime(&ms, start, stop);
    printf("CUDA convolution time: %f ms\n", ms);

    cudaMemcpy(h_out, d_out, out_size, cudaMemcpyDeviceToHost);

    cudaFree(d_img);
    cudaFree(d_kernel);
    cudaFree(d_out);
    free(h_img);
    free(h_out);

    return 0;
}
```

The conv_gpu.cu code above performs a 2D convolution on a grayscale image using a 3x3 filter (kernel) for edge detection, running the computation on the GPU.

- **convolution2D_GPU kernel:**
  - Each GPU thread computes the convolution result for one pixel in the output image. It multiplies the 3x3 region of the input image (centered at the current pixel) by the kernel and sums the results. Edge handling is done by checking bounds.
- **main function:**
  - Allocates and fills a random 512x512 image on the CPU.
  - Defines a 3x3 edge detection kernel.
  - Allocates memory on the GPU for the image, kernel, and output.
  - Copies data to the GPU.
  - Launches the CUDA kernel to compute the convolution in parallel.
  - Measures and prints the time taken.
  - Copies the result back to the CPU and frees all memory.

Overall, this code demonstrates how to accelerate 2D convolution (a common image processing operation) using CUDA, with each thread handling one output pixel.

Porting the Convolution Function to CUDA

```
[8]  !nvcc -arch=sm_75 conv_gpu.cu -o conv_gpu
     ✓ 1.1s

[9]  !./conv_gpu 512
     !./conv_gpu 1024
     !./conv_gpu 2048
     !./conv_gpu 4096
     !./conv_gpu 8192
     ✓ 3.5s

...  CUDA convolution time: 0.163936 ms
     CUDA convolution time: 0.192672 ms
     CUDA convolution time: 0.490048 ms
     CUDA convolution time: 1.577568 ms
     CUDA convolution time: 5.937184 ms
```

These results show the CUDA convolution running directly as a compiled C++ executable.

Adding and Using the Convolution to your Custom Python Library

```
%%writefile conv_lib.cu
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void convolution2D_GPU(
    unsigned char *image,
    int *kernel,
    int *output,
    int width,
    int height
) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int sum = 0;
    int k = 1; // 3x3 kernel radius

    for (int ky = -k; ky <= k; ky++) {
        for (int kx = -k; kx <= k; kx++) {
            int ix = x + kx;
            int iy = y + ky;

            if (ix >= 0 && ix < width && iy >= 0 && iy < height) {
                int pixel = image[iy * width + ix];
                int weight = kernel[(ky + k) * 3 + (kx + k)];
                sum += pixel * weight;
            }
        }
    }

    output[y * width + x] = sum;
}

extern "C" void gpu_convolution(
    unsigned char *h_image,
    int *h_kernel,
    int *h_output,
    int width,
    int height
) {
    size_t img_size = width * height * sizeof(unsigned char);
    size_t out_size = width * height * sizeof(int);

    unsigned char *d_image;
    int *d_kernel;
    int *d_output;

    cudaMalloc((void**)&d_image, img_size);
    cudaMalloc((void**)&d_kernel, 9 * sizeof(int));
    cudaMalloc((void**)&d_output, out_size);

    cudaMemcpy(d_image, h_image, img_size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernel, h_kernel, 9 * sizeof(int), cudaMemcpyHostToDevice);

    dim3 block(16, 16);
    dim3 grid((width + 15) / 16, (height + 15) / 16);

    convolution2D_GPU<<<grid, block>>>(d_image, d_kernel, d_output, width, height);
    cudaDeviceSynchronize();

    cudaMemcpy(h_output, d_output, out_size, cudaMemcpyDeviceToHost);

    cudaFree(d_image);
    cudaFree(d_kernel);
    cudaFree(d_output);
}
```

```
In [24]:
if './libmatrix.so' in sys.modules:
    del sys.modules['./libmatrix.so']

lib_path = os.path.abspath('./libmatrix.so')
lib = ctypes.CDLL(lib_path)

gpu_conv_func = lib.gpu_convolution
gpu_conv_func.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.uint8, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.int32, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.int32, ndim=1, flags="C_CONTIGUOUS"),
    ctypes.c_int,
    ctypes.c_int
]

width = 512
height = 512

for i in range(5):
    print(f"\nTesting with image size: {width} x {height}")

    image = np.random.randint(0, 256, size=(height, width), dtype=np.uint8)

    kernel = np.array([
        -1, -1, -1,
        -1,  8, -1,
        -1, -1, -1
    ], dtype=np.int32)

    output = np.zeros((height, width), dtype=np.int32)

    start = time.time()
    gpu_conv_func(
        image.ravel(),
        kernel,
        output.ravel(),
        width,
        height
    )
    end = time.time()
    elapsed_time = end - start

    print(f"CUDA convolution time ({width}x{height}): {elapsed_time * 1000:.4f} milliseconds")

    width = width * 2
    height = height * 2

Testing with image size: 512 x 512
CUDA convolution time (512x512): 1.4246 milliseconds

Testing with image size: 1024 x 1024
CUDA convolution time (1024x1024): 3.8025 milliseconds

Testing with image size: 2048 x 2048
CUDA convolution time (2048x2048): 13.1280 milliseconds

Testing with image size: 4096 x 4096
CUDA convolution time (4096x4096): 51.6033 milliseconds

Testing with image size: 8192 x 8192
CUDA convolution time (8192x8192): 190.6698 milliseconds
```

The above code implements the 2D image convolution using CUDA. I wrote the GPU kernel that applied the 3x3 filter to a grayscale image, and wrapped it in a function callable from Python. In Python, I loaded the shared library, generate random images, and call the CUDA function for different image sizes. The CUDA kernel output avoids any Python overhead and calls the GPU kernel with minimal delay, and thus the Python time output is far slower than that of the CUDA output in the previous screenshot.