Custom CUDA Library for CNN Pre-Processing

Technical Report · GPU Matrix Multiplication & Image Convolution

Shamik Basu

Jan 2026 - Feb 2026

Repository: https://github.com/ShamikOfficial/CUDA-Accelerated-Python-Library

Development Environment

For development, I connected Visual Studio Code to Google Colab using the Colab extension, which let us work in VS Code while running on Colab's remote NVIDIA Tesla T4 GPU. Alternatively, the project can be run directly in Google Colab on the same Tesla T4, which can be easier for uploading the images used in Module 2.

```
# check if GPU is available
!nvidia-smi
```

✓ 0.1s

```
Fri Jan 30 23:30:31 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   34C    P8              9W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

```
# check nvcc version
!nvcc --version
```

✓ 0.1s

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Jun__6_02:18:23_PDT_2024
Cuda compilation tools, release 12.5, V12.5.82
Build cuda_12.5.r12.5/compiler.34385749_0
```

Module 1: Matrix Multiplication

(1) Matrix Multiplication on the CPU

We implemented matrix multiplication in matrix_cpu.c and measured CPU execution time. As the example code was provided, we focused on understanding the C and CUDA syntax and the structure of the example.

The matrixMultiplyCPU() function computes the N x N matrix multiplication and times the CPU run. It allocates host memory for three N x N float arrays, initializes A and B with random values, and uses clock() to record the start and end of the multiplication before freeing the memory. We placed this code in matrix_cpu.c.

matrix_cpu.c was compiled with gcc matrix_cpu.c -o matrix_cpu -O2. To collect runtimes across multiple sizes, we wrote a helper, run_time_comparasion(), that takes a list of N values and an executable name and records the runtime for each. We measured N = 64, 128, 256, 512, 1024, and 2048.

```python
def run_time_comparison(n_list, command = './matrix_cpu'):
    time_list = []

    for n in n_list:
        run_cmd = subprocess.run([command, str(n)], capture_output=True, text=True)
        output = run_cmd.stdout
        print(output)
        time_str = output.split(':')[-1].strip().replace('second', 'sec')
        time_list.append(time_str)

    return time_list
```

0.0s Python

```python
n_list = [2**k for k in range(6, 12)]
cpu_times = run_time_comparison(n_list, command = './matrix_cpu')
```

1m 24.5s Python

(2) Naive CUDA Kernel

In this section, I implemented a CUDA kernel where each thread computes one element of the output matrix.

A CUDA kernel matrixMultiplyGPU is defined to compute matrix multiplication on the GPU. Each thread computes its global row and column indices using blockIdx, blockDim, and threadIdx, and then calculates one output element C by adding the dot product of row row in A and column col in B.

In the main() function, the matrix dimension N is read from the given N and the required memory size for N*N matrix is calculated as a size variable. And then the host memory pointers h_A, h_B and h_C are defined, where h_A and h_B are initialized with random values and h_C is set as 0s. The corresponding device memory pointers d_A, d_B and d_C are set as null and are allocated memory on GPU using cudaMalloc. After that, the h_A and h_B are copied from host to device using cudaMemcpy. The block size is set as 16*16 and grid size is calculated by ceiling division from the matrix dimension. A warm-up kernel launch is executed to reduce the overhead of the first launch, followed by error checking and synchronization. Kernel execution time is then measured using CUDA events by recording timestamps of starting and ending points of the kernel execution. Finally, the events are destroyed and both device and host memory are freed.

Implementation note: Compiling with !nvcc matrix_naive_gpu.cu -o matrix_naive_gpu produced the error "Kernel launch error: the provided PTX was compiled with an unsupported toolchain."

The error is because the PTX generated by nvcc is not compatible with the GPU driver on the execution machine, so the kernel cannot be loaded or JIT-compiled. Adding -arch=sm_75 compiles the code for the target GPU architecture, ensuring matching device code is generated and reducing PTX compatibility issues, which makes the kernel launch succeed.

The fix was to compile the CUDA file with the command below:

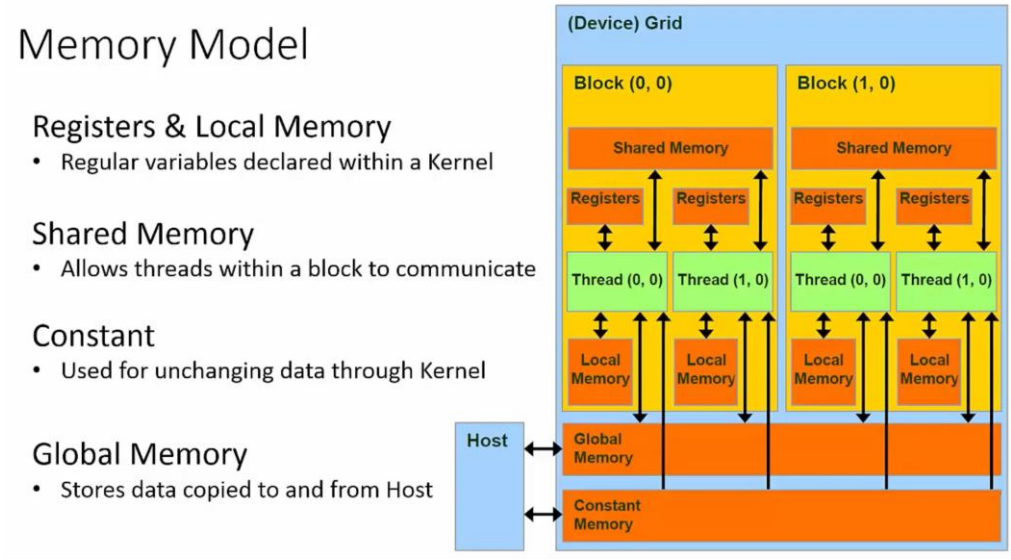
!nvcc -O2 -arch=sm_75 matrix_naive_gpu.cu -o matrix_naive_gpu

All the CUDA files followed are compiled with the -arch=sm_75 parameter.

(3) Optimizing CUDA Code

In this section, I introduced tiling with shared memory to reduce global memory accesses. Shared memory is faster to access than global memory because it is on-chip storage located within each Streaming Multiprocessor (SM). Since global memory is stored on device DRAM, shared memory offers much lower latency and it can be accessed by all threads inside the block. In naive CUDA, many threads repeatedly load the same A and B elements from global memory to execute matrix multiplication, which causes repeated access to global memory.

With a tile width of 16, each block loads a 16*16 tile of A and B into shared memory once, and all threads in the block can load these cached values on the chip. This reduces redundant global memory traffic and improves overall throughput, leading to shorter GPU execution time.



(4) Performance Comparison

We saved all the runtimes of CPU and GPU into a DataFrame and calculated the speedup = CPU time / GPU time.

The performance tables for this stage are shown below; a more detailed discussion follows the cuBLAS implementation.

Implementation	N=64	N=128	N=256	N=512	N=1024	N=2048
CPU (C)	0.0004 sec	0.0032 sec	0.0212 sec	0.2060 sec	3.4606 sec	80.1934 sec
Naive CUDA	0.0155 ms	0.0324 ms	0.1867 ms	1.1592 ms	9.1939 ms	67.0700 ms
Optimized CUDA	0.0102 ms	0.0369 ms	0.0696 ms	0.7476 ms	5.8040 ms	46.2909 ms

	N	Speedup (CPU / Naive CUDA)	Speedup (CPU / Optimized CUDA)
0	64	25.806452	39.215686
1	128	98.765432	86.720867
2	256	113.551152	304.597701
3	512	177.708765	275.548422
4	1024	376.401745	596.243970
5	2048	1195.667213	1732.379366

(5) Using cuBLAS Library

CuBLAS is NVIDIA's GPU-accelerated implementation of the standard Basic Linear Algebra Subprograms (BLAS) library. It provides optimized routines for linear algebra operations on NVIDIA GPUs using CUDA, such as the single-precision general matrix multiply(cublasSgemm).

In this section, I use cublasSgemm to calculate matrix multiplication instead of hand written CUDA kernel. In the main() function, a cuBLAS handle is created with cublasCreate. Constant variables alpha and beta are defined for the GEMM operation. A warm-up cublasSgemm call is executed to avoid first-call overhead, followed by cudaDeviceSynchronize(). The cuBLAS execution time is measured using CUDA events by recording start and stop around the cublasSgemm call and computing the elapsed time in milliseconds.

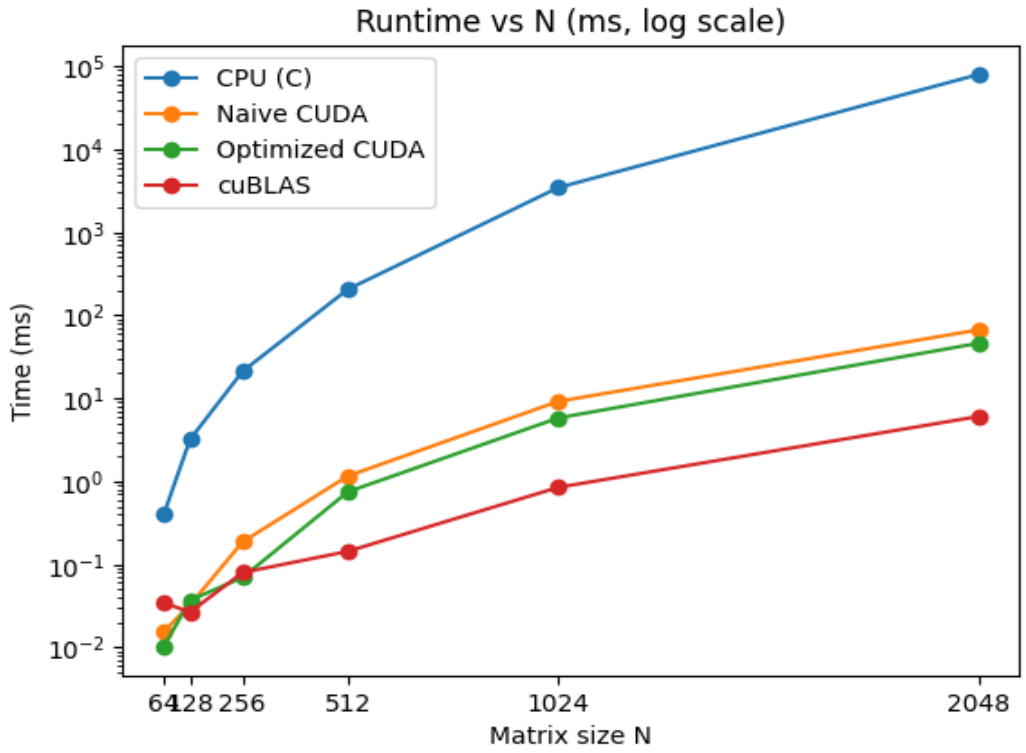
For compiling, use command: !nvcc -O2 -arch=sm_75 matrix_cublas_gpu.cu -o matrix_cublas_gpu -lcublas

(6) Overall Performance Comparison

In this section, I saved all the CPU and GPU's runtimes into a Dataframe. After converting the seconds into milliseconds, we calculated all the speedup. Besides, we plotted the execution time vs. matrix size for all CPU and GPU implementations.

Implementation	N=64	N=128	N=256	N=512	N=1024	N=2048
CPU (C)	0.0004 sec	0.0032 sec	0.0212 sec	0.2060 sec	3.4606 sec	80.1934 sec
Naive CUDA	0.0155 ms	0.0324 ms	0.1867 ms	1.1592 ms	9.1939 ms	67.0700 ms
Optimized CUDA	0.0102 ms	0.0369 ms	0.0696 ms	0.7476 ms	5.8040 ms	46.2909 ms
cuBLAS	0.0344 ms	0.0266 ms	0.0797 ms	0.1434 ms	0.8465 ms	6.0680 ms

	N	Speedup (CPU / Naive CUDA)	Speedup (CPU / Optimized CUDA)	Speedup (CPU / cuBLAS)
0	64	25.806452	39.215686	11.627907
1	128	98.765432	86.720867	120.300752
2	256	113.551152	304.597701	265.997491
3	512	177.708765	275.548422	1436.541144
4	1024	376.401745	596.243970	4088.127584
5	2048	1195.667213	1732.379366	13215.787739



Analysis Questions:

1. How does performance change as matrix size increases?

All implementations slow down as the matrix size increases, but they scale very differently. The CPU degrades the most, slowing faster than the expected quadratic rate at large sizes, likely due to cache effects. The GPU implementations scale far more smoothly, and cuBLAS scales best - its effective throughput keeps improving as the matrices grow.

2. At what point does the GPU significantly outperform the CPU?

The GPU outperforms the CPU across every size we tested, and the advantage grows with matrix size. By medium sizes the GPU is already orders of magnitude faster, and the gap widens sharply for large matrices where the CPU struggles most.

3. How much speedup is gained by tiling optimization vs. naïve CUDA?

Tiling shows little benefit at very small sizes, where its overhead can make it slower. Beyond a certain size the shared-memory optimization takes effect and produces a clear improvement. Tiling is most effective within a specific size range and shows little difference outside it.

4. How close is your optimized kernel to cuBLAS performance?

For small matrices our optimized kernel is competitive with cuBLAS and is occasionally slightly faster. As the matrices grow, cuBLAS becomes significantly faster, and the gap is largest for the heaviest workloads.

5. Why might cuBLAS still outperform hand-written kernels?

cuBLAS has years of optimization work behind it, with engineers who understand every detail of GPU architecture. It uses specialized hardware like tensor cores, adapts algorithms to different GPU generations, and applies low-level optimizations that are hard to replicate without deep domain expertise.

(7) Creating a Shared Library and Using it in Python

The code wraps the optimized CUDA matrix multiplication kernel into a shared library (libmatrix.so) callable from Python. The extern "C" wrapper function gpu_matrix_multiply() handles GPU memory allocation, data transfers, and kernel execution. Python uses ctypes to load the library, declare argument types (NumPy float32 arrays and an integer), and call the function with flattened NumPy arrays, enabling GPU acceleration from Python without rewriting CUDA code.

Key Components:

CUDA Library (matrix_lib.cu): Contains the tiled matrix multiplication kernel and an extern "C" wrapper that manages GPU memory and execution.

Compilation: Uses nvcc -Xcompiler -fPIC -shared to create a position-independent shared library (.so file).

Python Interface: Uses ctypes to load the library, specify argument types matching NumPy arrays, and call the CUDA function directly from Python, achieving GPU acceleration with minimal overhead.

Module 2: Image Convolution & Custom Python Library

Step 1: Convolution Function

Convolution filter (kernel) is a N by N matrix that is used for image processing. The image processing is done by a weighted sum of a pixel value and its neighborhood NxN pixel values to create a new pixel value. This feature makes this a good candidate for using CUDA since GPU is great for performing weighted sum with big data.

The library supports three filter families:

- **Edge Detection:** Edge detection is one of three supported convolution filters.
- **Gaussian Blur:** We chose Gaussian blur over a simple box blur because its effect is more visually noticeable.
- **Sharpen:** We selected the kernel weights from the Wikipedia article on image-processing kernels: https://en.wikipedia.org/wiki/Kernel_(image_processing).

Image Preprocessing

We used the Python Pillow library to resize the image to 1024 x 1024 and the convert("L") method to convert it to 8-bit grayscale (monochrome).

Code for preprocessing:

```
from PIL import Image
img = Image.open("image.jpg").convert("L").resize((1024, 1024))
img.save("input.pgm")
```

First Image before resizing and grayscaling	Resized (1024, 1024) Monochrome Image
	

The Code to Perform Convolution Function:

We implemented convolution with nested loops: for each pixel we gather its N x N neighborhood and apply the filter weights as a weighted sum, then store the result. The helper get_pixel_zero_ped() handles boundaries, returning zero for neighbors that fall outside the image. The window size is set by the weight length (default M=512, configurable via CLI arguments), advancing to a new row each time the weight length is reached.

Example Code for N=3 edge Detection:

```
'
// EDGE N=3:
static void edge_n3(const uint8_t *image, uint8_t *out, int W, int H) {
    const int32_t K[9] = {
         0, -1,  0,
        -1,  4, -1,
         0, -1,  0
    };
    int r = 1;

    for (int y = 0; y < H; y++) {
        for (int x = 0; x < W; x++) {
            int32_t sum = 0;

            for (int ky = 0; ky < 3; ky++) {
                for (int kx = 0; kx < 3; kx++) {
                    int ix = x + (kx - r);
                    int iy = y + (ky - r);
                    uint8_t p = get_pixel_zero_pad(image, W, H, ix, iy);
                    sum += (int32_t)p * K[ky * 3 + kx];
                }
            }

            if (sum < 0) sum = -sum;
            out[y * W + x] = clamp_to_u8(sum);
        }
    }
}
```

Filters:

We use three filters: edge detection, sharpen, and Gaussian blur. The 3x3 weights come directly from Wikipedia; for 5x5 and 7x7 we extended the same patterns. Edge detection uses a Laplacian-style kernel (large positive center with negative surrounds), while the sharpen and Gaussian-blur weights follow standard references. The Gaussian blur is normalized to preserve overall brightness.

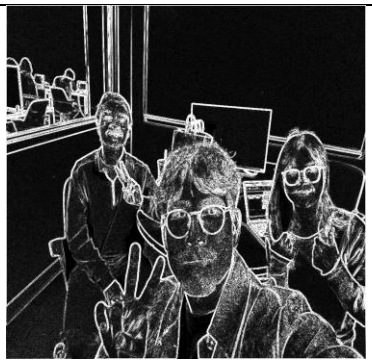
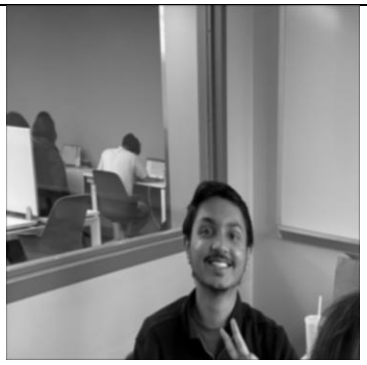
The active filter is selected with a mode string such as "edge_n3" or "sharpen_n7", so different kernels can be tested by changing a single CLI argument.

Performance of CPU only:

```
Compute time (convolution only): mode=edge_n3, N=3, M=256 -> 1.202 ms
Compute time (convolution only): mode=edge_n5, N=5, M=256 -> 2.388 ms
Compute time (convolution only): mode=edge_n7, N=7, M=256 -> 7.079 ms
Compute time (convolution only): mode=sharpen_n3, N=3, M=256 -> 1.394 ms
Compute time (convolution only): mode=sharpen_n5, N=5, M=256 -> 2.861 ms
Compute time (convolution only): mode=sharpen_n7, N=7, M=256 -> 4.659 ms
Compute time (convolution only): mode=blur_n7, N=7, M=256 -> 7.633 ms
Compute time (convolution only): mode=blur_n3, N=3, M=256 -> 1.347 ms
Compute time (convolution only): mode=blur_n5, N=5, M=256 -> 2.562 ms
Compute time (convolution only): mode=edge_n3, N=3, M=512 -> 5.388 ms
Compute time (convolution only): mode=edge_n5, N=5, M=512 -> 9.901 ms
Compute time (convolution only): mode=edge_n7, N=7, M=512 -> 27.389 ms
Compute time (convolution only): mode=sharpen_n3, N=3, M=512 -> 4.492 ms
Compute time (convolution only): mode=sharpen_n5, N=5, M=512 -> 10.415 ms
Compute time (convolution only): mode=sharpen_n7, N=7, M=512 -> 18.694 ms
Compute time (convolution only): mode=blur_n3, N=3, M=512 -> 7.518 ms
Compute time (convolution only): mode=blur_n5, N=5, M=512 -> 10.343 ms
Compute time (convolution only): mode=blur_n7, N=7, M=512 -> 27.197 ms
Compute time (convolution only): mode=edge_n3, N=3, M=1024 -> 20.132 ms
Compute time (convolution only): mode=edge_n5, N=5, M=1024 -> 39.494 ms
Compute time (convolution only): mode=edge_n7, N=7, M=1024 -> 111.012 ms
Compute time (convolution only): mode=sharpen_n3, N=3, M=1024 -> 16.895 ms
Compute time (convolution only): mode=sharpen_n5, N=5, M=1024 -> 40.281 ms
Compute time (convolution only): mode=sharpen_n7, N=7, M=1024 -> 76.936 ms
Compute time (convolution only): mode=blur_n3, N=3, M=1024 -> 22.251 ms
Compute time (convolution only): mode=blur_n5, N=5, M=1024 -> 55.309 ms
Compute time (convolution only): mode=blur_n7, N=7, M=1024 -> 195.043 ms
```

The time increases as M increases. The time also increases as N increases. The time is not equal for all filters.

Displaying Images to prove Functions work Properly:

Edge n3 with 1024 size	
Edge n5 with 1024 size	
Sharp n5 1024 size	
Blur n7 with 512 size: The image is cropped to the top-left corner so it fits the smaller 512 size we defined.	

Step 2: Porting the Convolution Function to CUDA

Why this is a good CUDA Candidate:

Convolution is highly parallel because each output pixel can be computed independently - the same operation is applied to every pixel. GPUs are designed for exactly this kind of repeated multiply-and-accumulate across large arrays, so CUDA is a good fit.

This uses essentially the same kernel logic: GPU memory is allocated with cudaMalloc, and each CUDA thread computes one output pixel (x, y) by looping over its N x N neighborhood, multiplying by the assigned filter weights, applying boundary checks, and clipping the result to 0-255.

Performance: Comparison of CPU and GPU

```
Collected 9 CPU modes and 9 CUDA modes
```

Filter	M=256	M=512	M=1024
edge_n3	CPU: 1.198 ms, CUDA: 0.217 ms (Speedup: 5.52x)	CPU: 5.216 ms, CUDA: 0.253 ms (Speedup: 20.62x)	CPU: 20.372 ms, CUDA: 0.189 ms (Speedup: 107.79x)
edge_n5	CPU: 2.509 ms, CUDA: 0.268 ms (Speedup: 9.36x)	CPU: 10.228 ms, CUDA: 0.275 ms (Speedup: 37.19x)	CPU: 39.561 ms, CUDA: 0.308 ms (Speedup: 128.44x)
edge_n7	CPU: 7.384 ms, CUDA: 0.234 ms (Speedup: 31.56x)	CPU: 28.293 ms, CUDA: 0.284 ms (Speedup: 99.62x)	CPU: 113.929 ms, CUDA: 0.442 ms (Speedup: 257....
sharpen_n3	CPU: 1.063 ms, CUDA: 0.221 ms (Speedup: 4.81x)	CPU: 4.169 ms, CUDA: 0.248 ms (Speedup: 16.81x)	CPU: 17.123 ms, CUDA: 0.259 ms (Speedup: 66.11x)
sharpen_n5	CPU: 2.626 ms, CUDA: 0.242 ms (Speedup: 10.85x)	CPU: 12.244 ms, CUDA: 0.266 ms (Speedup: 46.03x)	CPU: 45.814 ms, CUDA: 0.392 ms (Speedup: 116.87x)
sharpen_n7	CPU: 5.140 ms, CUDA: 0.254 ms (Speedup: 20.24x)	CPU: 19.389 ms, CUDA: 0.302 ms (Speedup: 64.20x)	CPU: 81.197 ms, CUDA: 0.451 ms (Speedup: 180.04x)
blur_n3	CPU: 1.452 ms, CUDA: 0.297 ms (Speedup: 4.89x)	CPU: 5.618 ms, CUDA: 0.286 ms (Speedup: 19.64x)	CPU: 22.442 ms, CUDA: 0.270 ms (Speedup: 83.12x)
blur_n5	CPU: 2.580 ms, CUDA: 0.354 ms (Speedup: 7.29x)	CPU: 10.989 ms, CUDA: 0.323 ms (Speedup: 34.02x)	CPU: 43.981 ms, CUDA: 0.382 ms (Speedup: 115.13x)
blur_n7	CPU: 7.280 ms, CUDA: 0.376 ms (Speedup: 19.36x)	CPU: 29.059 ms, CUDA: 0.358 ms (Speedup: 81.17x)	CPU: 115.875 ms, CUDA: 0.522 ms (Speedup: 221....

```
--- Summary Statistics ---
Average speedup: 67.05x
Minimum speedup: 4.81x
Maximum speedup: 257.76x
```

Step 3: Adding and Using the Convolution in the Custom Python Library

Compiles the CUDA convolution kernels into a shared library (libconvo.so) that Python can call via ctypes. This lets Python scripts use GPU-accelerated convolution (edge detection, sharpening, blur) without rewriting the CUDA code, enabling performance comparisons between CPU, CUDA executables, and Python-bound CUDA functions.

GPU Performance per function in different architecture:

	Mode	M	C (ms)	CUDA Exec (ms)	Python+CUDA (ms)	Speedup (C vs CUDA Exec)	Speedup (C vs Python+CUDA)
0	edge_n3	256	1.217	0.206	76.478	5.91x	0.02x
1	sharpen_n3	256	1.027	0.219	0.691	4.69x	1.49x
2	blur_n3	256	1.418	0.206	0.603	6.88x	2.35x
3	edge_n3	512	4.888	0.214	0.953	22.84x	5.13x
4	sharpen_n3	512	4.004	0.235	0.851	17.04x	4.71x
5	blur_n3	512	5.419	0.233	0.829	23.26x	6.53x
6	edge_n3	1024	22.758	0.197	2.249	115.52x	10.12x
7	sharpen_n3	1024	16.311	0.200	1.869	81.55x	8.73x
8	blur_n3	1024	21.644	0.202	1.536	107.15x	14.09x

```
=== Summary ===
Performance comparison: C program vs CUDA executable vs Python+CUDA library
Note: Python+CUDA includes overhead of Python/ctypes interface
```