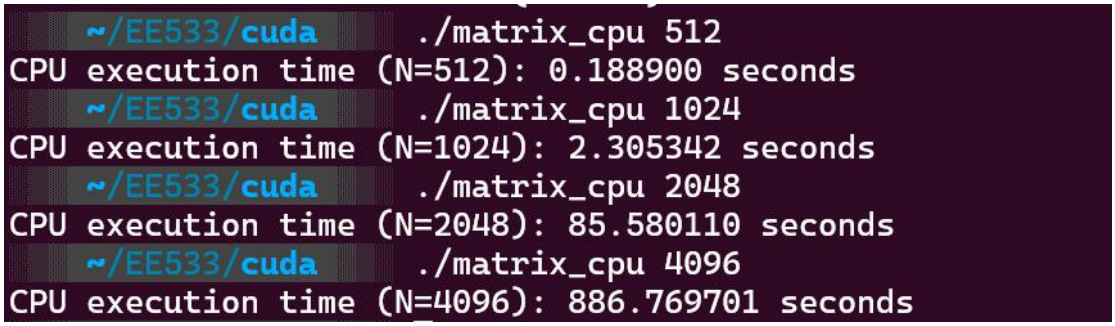
EE 533: Network Processor Design and Programming

Instructor: Young Cho, Ph.D.

Zhongqi Zhao: 7864803206

Part 1: Matrix Multiplication on the CPU

In this part, a baseline matrix multiplication program was implemented in C using a triple-nested loop structure. The program computes the product of two square matrices of size N × N and measures execution time on the CPU for different matrix sizes. This implementation serves as a reference point for evaluating performance improvements achieved through GPU acceleration in later parts of the experiment.



Part 2: Introduction to CUDA Programming (Naïve CUDA Kernel)

This part introduces CUDA programming by porting the CPU-based matrix multiplication to a naïve GPU implementation. Each CUDA thread is responsible for computing a single element of the output matrix, enabling massive parallelism. Execution times were measured for multiple matrix sizes to evaluate the performance benefits and overheads of using the GPU compared to the CPU.

Part 3: Running CUDA on Google Cloud

In this part, CUDA programs were deployed and executed on a GPU-enabled virtual machine in Google Cloud. A cloud instance equipped with an NVIDIA GPU was provisioned, and the CUDA Toolkit was installed to support compilation and execution. This task demonstrates how GPU-accelerated applications can be run in a cloud environment and validates the portability of CUDA programs across different platforms.

Part 4: Optimizing CUDA Code with Shared Memory Tiling

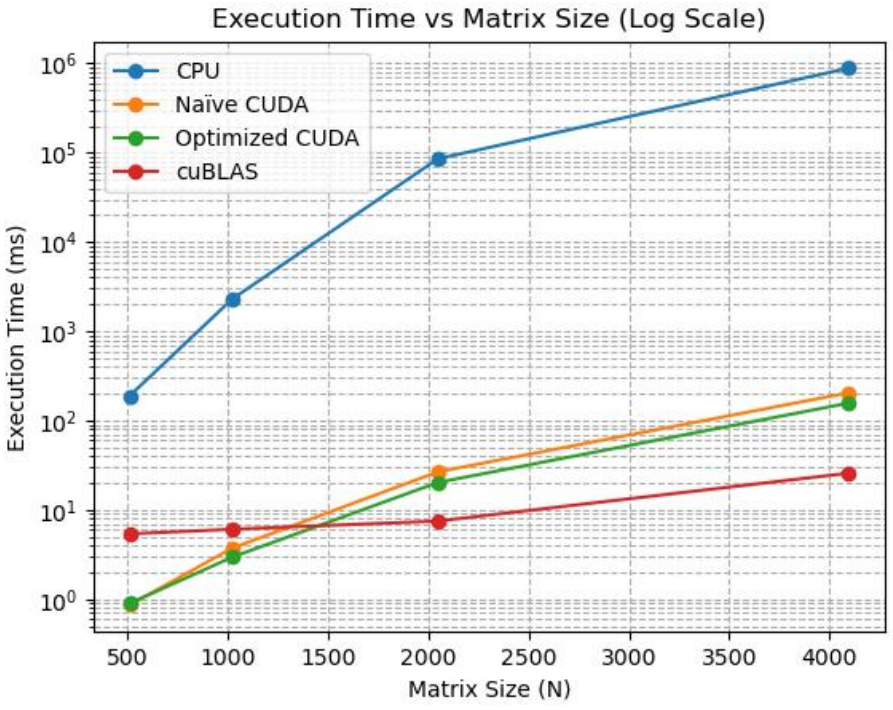
This part focuses on optimizing the CUDA matrix multiplication kernel using shared memory tiling. By loading submatrices of the input matrices into shared memory, global memory accesses are reduced and memory locality is improved. Performance measurements show that the tiled implementation significantly outperforms the naïve CUDA kernel, especially for larger matrix sizes.

```
~/EE533/cuda      ./matrix_gpu_opt 512
Optimized CUDA (Tiled) execution time (N=512): 0.894720 ms
~/EE533/cuda      ./matrix_gpu_opt 1024
Optimized CUDA (Tiled) execution time (N=1024): 2.956032 ms
~/EE533/cuda      ./matrix_gpu_opt 2048
Optimized CUDA (Tiled) execution time (N=2048): 20.265184 ms
~/EE533/cuda      ./matrix_gpu_opt 4096
Optimized CUDA (Tiled) execution time (N=4096): 157.194885 ms
```

Part 5: Performance Comparison

In this part, execution times from the CPU, naïve CUDA, optimized CUDA, and cuBLAS implementations were collected and compared. Performance scaling was analyzed as matrix size increased, and speedup factors were calculated to quantify GPU acceleration. The results highlight the impact of GPU overhead, parallel efficiency, and optimization techniques on overall performance.

Implementation	N = 512	N = 1024	N = 2048	N = 4096
CPU (C)	0.1889 s	2.3053 s	85.5801 s	886.7697 s
Naïve CUDA	0.8596 ms	3.7231 ms	26.6498 ms	205.2493 ms
Optimized CUDA	0.8947 ms	2.9560 ms	20.2652 ms	157.1949 ms
cuBLAS	5.3814 ms	6.0619 ms	7.4863 ms	25.7582 ms



Part 6: Using the cuBLAS Library

This part evaluates the use of NVIDIA's cuBLAS library for matrix multiplication. By leveraging the highly optimized cublasSgemm routine, matrix multiplication was performed with minimal implementation effort. The cuBLAS results consistently achieved the best performance, illustrating the advantage of using vendor-optimized libraries over hand-written CUDA kernels.

```
    ~/EE533/cuda      ./matrix_cuBLAS 512
GPU (cuBLAS) execution time (N=512): 5.381440 ms
    ~/EE533/cuda      ./matrix_cuBLAS 1024
GPU (cuBLAS) execution time (N=1024): 6.061888 ms
    ~/EE533/cuda      ./matrix_cuBLAS 2048
GPU (cuBLAS) execution time (N=2048): 7.486336 ms
    ~/EE533/cuda      ./matrix_cuBLAS 4096
GPU (cuBLAS) execution time (N=4096): 25.758207 ms
```

Part 7: Creating a Shared Library and Using It in Python

In this part, the optimized CUDA matrix multiplication kernel was compiled into a shared library and accessed from Python using the ctypes interface. This approach enables GPU acceleration within Python programs while maintaining flexibility and usability. Performance measurements demonstrate that Python applications can effectively utilize CUDA acceleration with manageable overhead.

1. Analysis Questions

(1) As the matrix size increases, execution time increases for both the CPU and the GPU. However, the CPU time grows much faster because it executes operations sequentially. The GPU scales better with larger matrices since more parallel threads are used, making GPU acceleration more effective for large problem sizes.

(2) The GPU begins to significantly outperform the CPU at medium to large matrix sizes, typically around N = 1024 or larger. At smaller sizes, GPU overhead such as memory transfers and kernel launches reduces the performance benefit.

(3) The tiling optimization provides a noticeable speedup compared to the naïve CUDA implementation, often several times faster. This improvement comes from reduced global memory access and better use of shared memory.

(4) The optimized CUDA kernel approaches cuBLAS performance but is still slower. In most cases, the optimized kernel achieves a large portion of cuBLAS performance, but cuBLAS remains the fastest implementation.

(5) cuBLAS outperforms hand-written kernels because it is highly optimized for specific GPU architectures. It uses advanced techniques such as register blocking, efficient memory access patterns, and architecture-level tuning that are difficult to implement manually.

2. Write a Library Interface Using the Library

```
~/EE533/cuda  python test_lib.py
C[0:5, 0:5] =
[[255.15417 257.4979  250.8481  254.8587  258.4956 ]
 [253.50851 253.35432 248.05144 254.35277 258.55606]
 [249.15872 254.67944 241.81111 253.70992 257.42993]
 [264.21048 269.13    264.13022 268.01102 274.8512 ]
 [254.1326  254.83203 250.85197 257.36816 268.09424]]
Python call to CUDA library completed in 0.2304 seconds
```

3. Convolution Function

```
~/EE533/c/part7  gcc -O3 -march=native -o conv conv.c -lm
~/EE533/c/part7  ./conv ./rand_0.png output_cpu.png kernel.txt
Loaded ./rand_0.png: 512 x 512, channels=4
Failed to read kernel file kernel.txt, using default.
Using kernel size 3 x 3
conv_cpu_uint32 elapsed: 2.525203000 ms
Wrote output output_cpu.png
```

```
~/EE533/c/part7  ./conv_cuda ./rand_0.png output_cuda.png kernel.txt
Loaded ./rand_0.png : 512 x 512 channels=4
Failed to read kernel file kernel.txt, using default.
Kernel size 3
Wrote output_cuda.png
Kernel execution time: 1.438 ms
```

4. Summary

The results show that CPU execution time increases rapidly as matrix size grows, while GPU implementations scale much more efficiently due to parallelism. For small matrices, GPU performance is limited by overhead such as memory transfers and kernel launch costs, but for larger matrices the GPU significantly outperforms the CPU. Optimization techniques like shared memory tiling improve performance but increase code complexity, while libraries such as cuBLAS provide high performance with less implementation effort.

Github Link:

https://github.com/ZzqXAUT/EE533.git