CSI 4650 Final Project

Matrix Multiplication

Group Members: Willow Connelly, Rumana Yeasmin, Emma Gabos

Project Overview

Single-threaded vs Parallelized Computation

- Compare single-threaded and parallelized methods for matrix multiplication.
- Investigate how different methods and hardware (CPU vs GPU) affect computation time.

Benchmarking includes latency analysis for varying matrix size, matrix generation and utilizing various batch sizes to compare processing time on the CPU vs GPU.

Single-threaded vs Parallelized Computation

- **Focus Area**
  - Sequential vs Parallel Execution
  - Performance Scaling with Matrix Size
  - CPU vs GPU Efficiency

- **Why This Project?**
  - Matrix multiplication is core to applications in machine learning, data analytics, and scientific computing.
  - Optimizing this operation significantly impacts performance in high-demand computing tasks.

Info & Use cases

- **Machine Learning and AI:**
  - Accelerating training for deep learning models via GPU-based tensor operations.
- **Scientific Computing:**
  - Solving large systems of equations in physics and engineering simulations.
- **Data Analytics:**
  - Speeding up large-scale data transformation and computation pipelines.
- **Graphics and Gaming:**
  - Real-time rendering and physics simulations using GPUs for optimal performance.
- **Autonomous Vehicles:**
  - Self-driving systems use parallelized matrix operations for sensor fusion, path planning, and real-time environment mapping, enabling safe and efficient navigation.

Benchmarking Methodology

Benchmarking Goals:

- Measure and compare computational latency for matrix multiplication across different matrix sizes.
- Investigate the impact of parallelization (CPU and GPU) on performance.
- Explore the trade-offs between sequential and parallel execution for different hardware and data sizes.

Tools and Technologies

Tools and Platforms

- Google Colab:
  - Cloud-based environment for executing and testing code.
  - Provides access to GPUs for parallel computation experiments.

Environment:

- CPU: Intel Xeon
- Physical cores: 1
- Logical processors (threads): 2
- GPU: NVIDIA Tesla T4 - 2560 CUDA cores

Libraries

- PyTorch
- itertools
- time and timeit
- matplotlib
- psutil

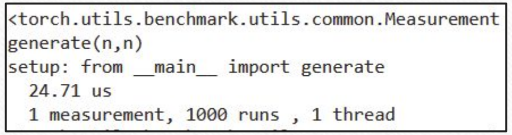
Code Snippet

```
def naive_mult(A, B):
    row_a, col_a = A.shape
    _, col_b = B.shape
    C = torch.zeros(row_a, col_b, device=A.device)

    for i in range(row_a):
        for j in range(col_b):
            for k in range(col_a):
                C[i, j] += A[i, k] * B[k, j]
    return C


def broadcast_mult(A,B):
    resultMat = (A[:, :, None] * B[None, :, :]).sum(dim=1)
    return resultMat
```

Experimental Results - Matrix Generation

Generated from normal distribution

Device	Time (ms)
CPU	6.0
GPU	0.025



```
<torch.utils.benchmark.utils.common.Measurement
generate(n,n)
setup: from __main__ import generate
  24.71 us
  1 measurement, 1000 runs , 1 thread
```

Generated from large number set

Device	Time (ms)
CPU	14.0
GPU	0.04

Many ranges of integers were tested with varying difference in range and varying number values, but no significant time difference was found.

All tests done for a single 1000x1000 matrix. Each measurement is the result of averaging 5 runs of 1000 iterations. All measurements performed with PyTorch's Benchmarking methods.

Experimental Results - Transfering Matrix

This test was to see the time difference between moving our matrices from the CPU onto the GPU and vice versa.

The previous test generated the tensors on the specified processor directly; therefore, no transferring was needed.

Device	Time (ms)
CPU to GPU	1.0
GPU to CPU	0.88

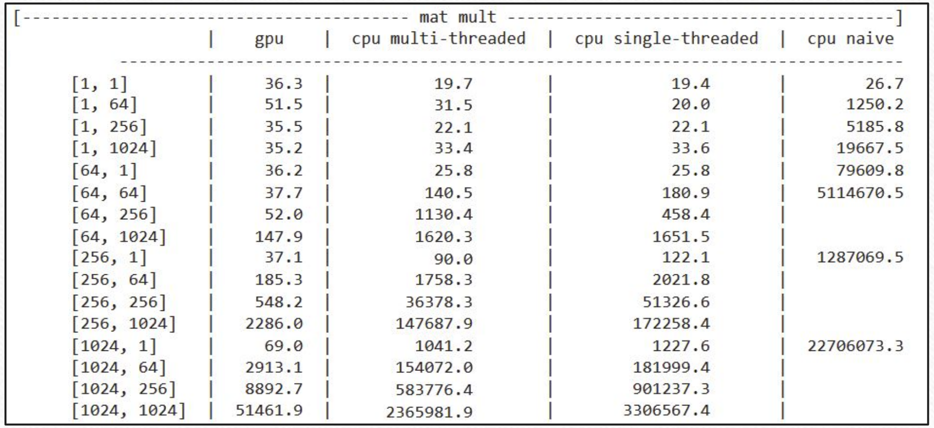
All tests done for a single 1000x1000 matrix. Each measurement is the result of averaging 5 runs of 1000 iterations. All measurements performed with PyTorch's Benchmarking methods.

Code Snippet

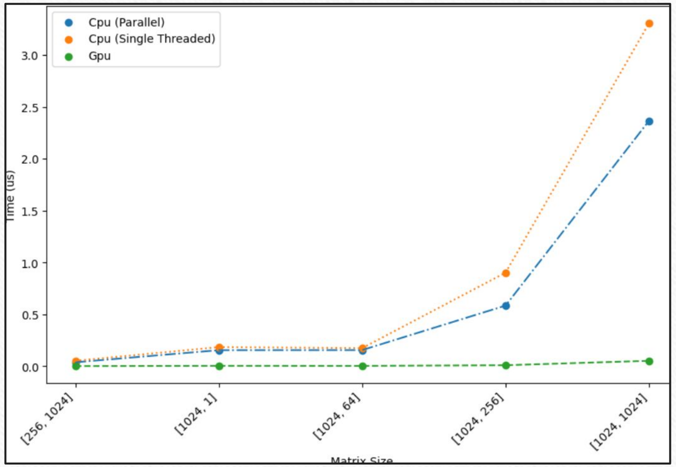
```
results = []
sizes = [1,64,256,1024]

for b, n in product(sizes, sizes):
    label = 'mat mult'
    sub_label = f'[{b}, {n}]'
    # Generate tensors
    x = torch.rand((b,n),device='cuda')
    y = x.to('cpu')
    # GPU - test
    results.append(benchmark.Timer(
        stmt='mult(x, x.T)',
        setup='from __main__ import mult',
        globals={'x': x},
        label=label,
        sub_label=sub_label,
        description='gpu',
    ).blocked_autorange(min_run_time=1))
    ...
```

Experimental Results - Matrix Multiplication

- Testing matrix multiplication with varying matrix sizes, both square and non-square.
  - Each matrix was multiplied by the transpose of itself.



```
[--------------------------------------- mat mult ---------------------------------------]
                  |    gpu    |  cpu multi-threaded  |  cpu single-threaded  |  cpu naive
          ------------------------------------------------------------------------------
     [1, 1]       |     36.3  |          19.7        |           19.4        |       26.7
     [1, 64]      |     51.5  |          31.5        |           20.0        |     1250.2
     [1, 256]     |     35.5  |          22.1        |           22.1        |     5185.8
     [1, 1024]    |     35.2  |          33.4        |           33.6        |    19667.5
     [64, 1]      |     36.2  |          25.8        |           25.8        |    79609.8
     [64, 64]     |     37.7  |         140.5        |          180.9        |  5114670.5
     [64, 256]    |     52.0  |        1130.4        |          458.4        |
     [64, 1024]   |    147.9  |        1620.3        |         1651.5        |
     [256, 1]     |     37.1  |          90.0        |          122.1        |  1287069.5
     [256, 64]    |    185.3  |        1758.3        |         2021.8        |
     [256, 256]   |    548.2  |       36378.3        |        51326.6        |
     [256, 1024]  |   2286.0  |      147687.9        |       172258.4        |
     [1024, 1]    |     69.0  |        1041.2        |         1227.6        | 22706073.3
     [1024, 64]   |   2913.1  |      154072.0        |       181999.4        |
     [1024, 256]  |   8892.7  |      583776.4        |       901237.3        |
     [1024, 1024] |  51461.9  |     2365981.9        |      3306567.4        |
```



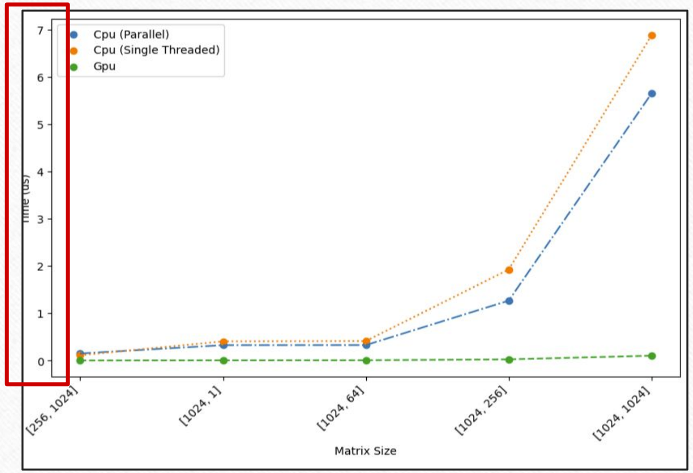
Timing values shown in microseconds (us) which is equal to 1e-6 seconds.

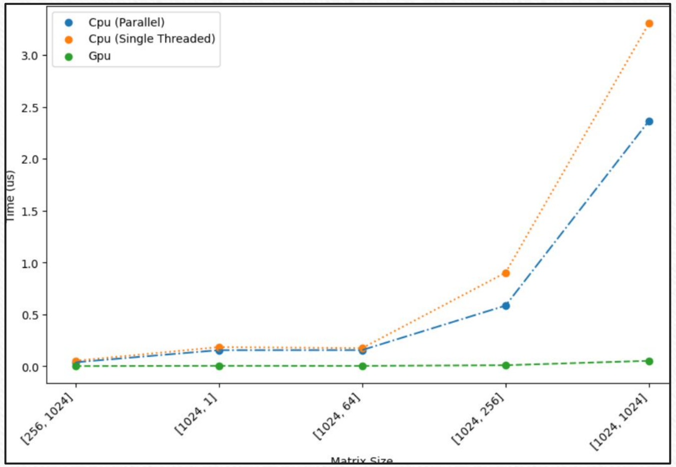
Group 6

CSI 4650, Fall 2024

Experimental Results - Matrix Multiplication

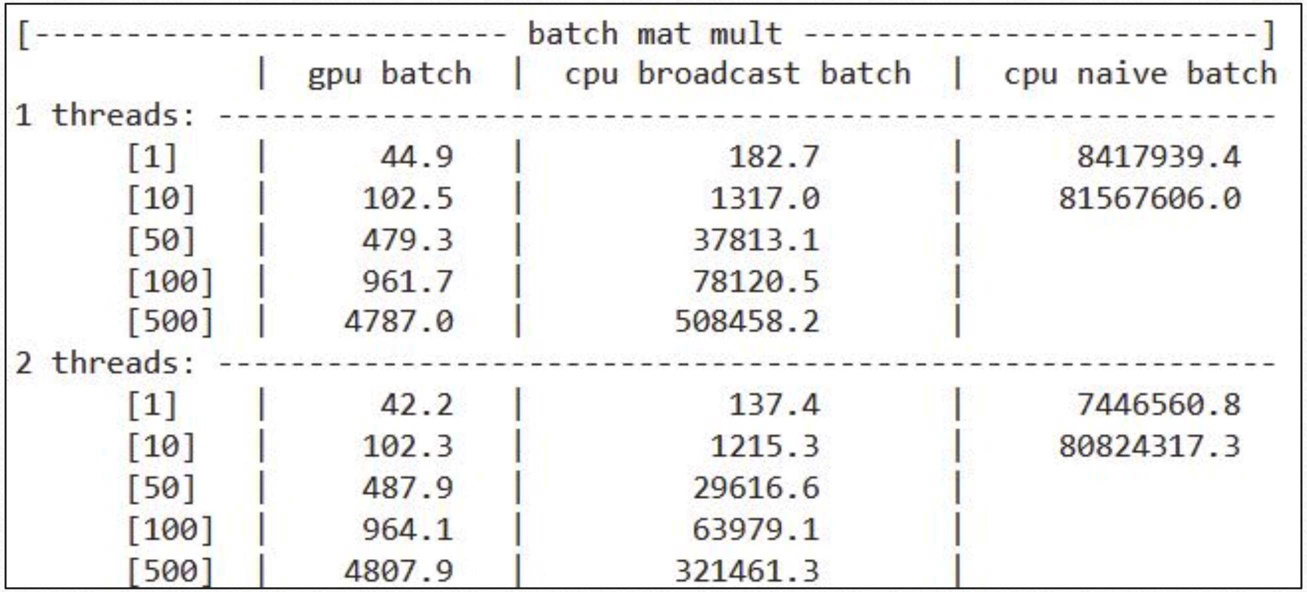
- Testing matrix multiplication with varying matrix sizes, both square and non-square.
    - Each matrix was multiplied by the transpose of itself.





Timing values shown in microseconds (us) which is equal to 1e-6 seconds.

Experimental Results - Batch Matrix Mult



```
[------------------------- batch mat mult ------------------------]
             |  gpu batch  |  cpu broadcast batch  |  cpu naive batch
1 threads: -------------------------------------------------------
      [1]    |      44.9   |            182.7      |     8417939.4
      [10]   |     102.5   |           1317.0      |    81567606.0
      [50]   |     479.3   |          37813.1      |
      [100]  |     961.7   |          78120.5      |
      [500]  |    4787.0   |         508458.2      |
2 threads: -------------------------------------------------------
      [1]    |      42.2   |            137.4      |     7446560.8
      [10]   |     102.3   |           1215.3      |    80824317.3
      [50]   |     487.9   |          29616.6      |
      [100]  |     964.1   |          63979.1      |
      [500]  |    4807.9   |         321461.3      |
```

Performance Analysis by Batch Size

- **Small Batches**: Overhead dominates, making single-threading more efficient.
- **Larger Batches**: Multi-threading starts to benefit but gains are limited by hardware constraints.
- **Thread Scaling**: Performance may degrade with more threads (e.g., 2) due to contention.

Challenges and Insights

- Figuring out Google Colab’s environment specs
- Correctly timing/benchmarking the code. Using PyTorch’s timing functions was necessary.
- Building a parallelizable yet efficient matrix multiplication function that doesn’t just use pytorch’s or NumPy’s built-in matmult functions.

Future Updates

- Use better environment. One with a better GPU and more CPU threads to test on.
- Explore additional GPU optimization techniques.
- Integrate more matrix multiplication algorithms.
- Automate matrix size and iteration count testing for dynamic benchmarking.

Questions?