

ECE-GY 9143- High Performance Machine Learning - Homework Assignment 1

By Amrutha Patil (ap7982)

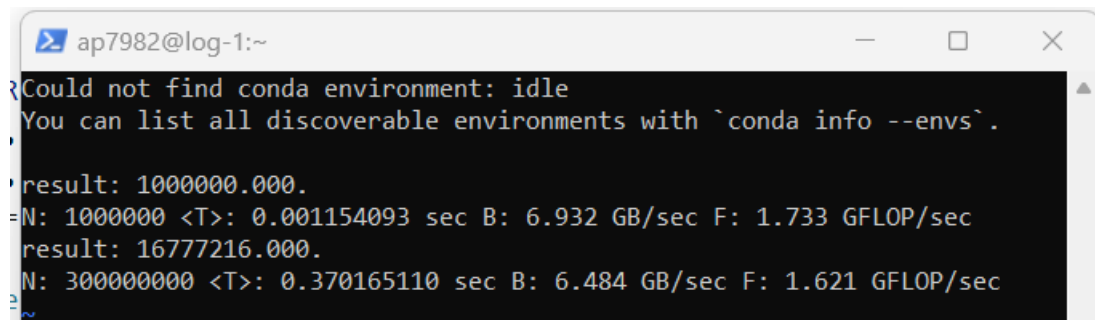
C1 Output:

result: 1000000.000.

N: 1000000 <T>: 0.001154093 sec B: 6.932 GB/sec F: 1.733 GFLOP/sec

result: 16777216.000.

N: 300000000 <T>: 0.370165110 sec B: 6.484 GB/sec F: 1.621 GFLOP/sec

A terminal window titled 'ap7982@log-1:~' with standard window controls. The terminal output is as follows:

```
Could not find conda environment: idle
You can list all discoverable environments with `conda info --envs`.

result: 1000000.000.
N: 1000000 <T>: 0.001154093 sec B: 6.932 GB/sec F: 1.733 GFLOP/sec
result: 16777216.000.
N: 300000000 <T>: 0.370165110 sec B: 6.484 GB/sec F: 1.621 GFLOP/sec
```

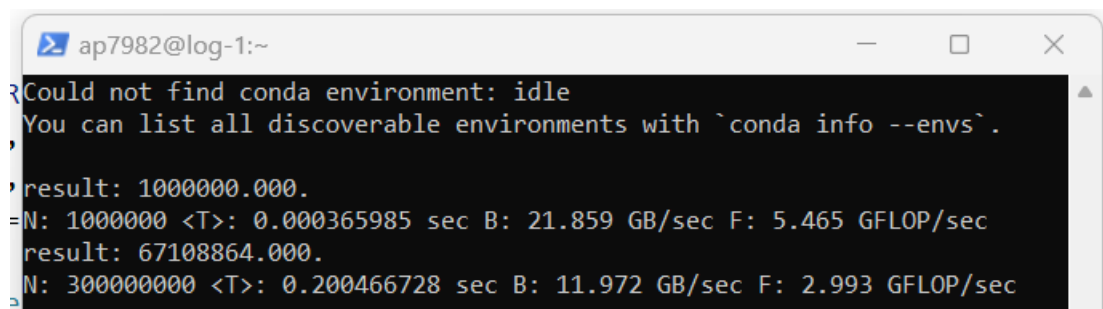
C2 Output:

result: 1000000.000.

N: 1000000 <T>: 0.000365985 sec B: 21.859 GB/sec F: 5.465 GFLOP/sec

result: 67108864.000.

N: 300000000 <T>: 0.200466728 sec B: 11.972 GB/sec F: 2.993 GFLOP/sec

A terminal window titled 'ap7982@log-1:~' with standard window controls. The terminal output is as follows:

```
Could not find conda environment: idle
You can list all discoverable environments with `conda info --envs`.

result: 1000000.000.
N: 1000000 <T>: 0.000365985 sec B: 21.859 GB/sec F: 5.465 GFLOP/sec
result: 67108864.000.
N: 300000000 <T>: 0.200466728 sec B: 11.972 GB/sec F: 2.993 GFLOP/sec
```

C3 Output:

result: 1000000.000.

N: 1000000 <T>: 0.000316745 sec B: 25.257 GB/sec F: 6.314 GFLOP/sec

result: 300000000.000.

N: 300000000 <T>: 0.177208459 sec B: 13.543 GB/sec F: 3.386 GFLOP/sec

```
ap7982@log-2:~  
R/opt/slurm/data/slurmd/job42875626/slurm_script: line 12: conda: command  
not found  
, /opt/slurm/data/slurmd/job42875626/slurm_script: line 13: conda: command  
, not found  
=result: 1000000.000.  
N: 1000000 <T>: 0.000316745 sec B: 25.257 GB/sec F: 6.314 GFLOP/sec  
result: 300000000.000.  
eN: 300000000 <T>: 0.177208459 sec B: 13.543 GB/sec F: 3.386 GFLOP/sec  
e
```

C4 Output:

R: 1000000.0

N: 1000000 <T>: 0.502833 sec B: 1.655 GB/sec F: 0.004 GFLOP/sec

R: 300000000.0

N: 300000000 <T>: 155.656787 sec B: 1.604 GB/sec F: 0.004 GFLOP/sec

```
ap7982@log-2:~  
Could not find conda environment: idle  
You can list all discoverable environments with `conda info --envs`.  
C  
R: 1000000.0  
N: 1000000 <T>: 0.502833 sec B: 1.655 GB/sec F: 0.004 GFLOP/sec  
R: 300000000.0  
N: 300000000 <T>: 155.656787 sec B: 1.604 GB/sec F: 0.004 GFLOP/sec
```

C5 Output:

R: 1000000.0

N: 1000000 <T>: 0.000328 sec B: 2539.353 GB/sec F: 6.104 GFLOP/sec

R: 300000000.0

N: 300000000 <T>: 0.251075 sec B: 994.125 GB/sec F: 2.390 GFLOP/sec

```
ap7982@log-2:~  
Could not find conda environment: idle  
, You can list all discoverable environments with `conda info --envs`.  
,  
R: 1000000.0  
=N: 1000000 <T>: 0.000328 sec B: 2539.353 GB/sec F: 6.104 GFLOP/sec  
R: 300000000.0  
eN: 300000000 <T>: 0.251075 sec B: 994.125 GB/sec F: 2.390 GFLOP/sec  
e
```

Q1 Solution:

Using only the second half of measurements for computing the mean can lead to biases, incomplete assessments, and overlook caching dynamics:

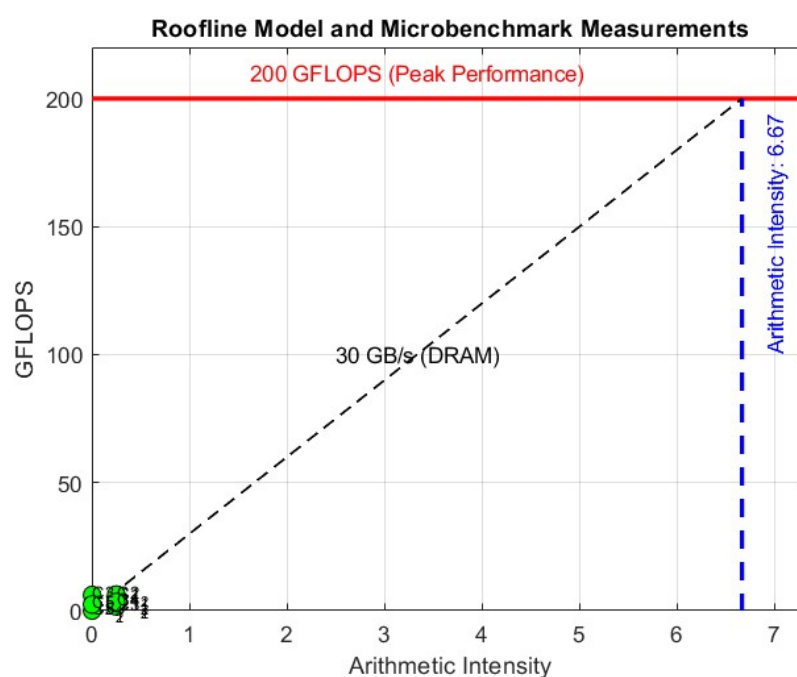
- **Potential Bias:** Excluding the initial measurements assumes system stability, potentially biasing estimates if warm-up effects or transient behaviors are present.
- **Sampling Bias:** Ignoring early measurements may overlook system behavior changes, introducing sampling bias and incomplete representation of performance.
- **Unstable System Behavior:** Assumptions of system stability may not hold, especially if the system exhibits fluctuating behavior over time, leading to misleading conclusions.
- **Incomplete Assessment:** Discarding initial measurements may miss insights into system behavior, such as cache warm-up effects and workload variations, crucial for informed decision-making.
- **Cache Dynamics:** Focusing solely on the second half may overlook cache warm-up effects and variations, impacting the understanding of caching dynamics and their influence on performance.

In summary, using only the second half of measurements can result in biased estimates, overlook system behavior changes, and neglect caching dynamics. Careful consideration of the entire experimental duration is crucial for accurate and meaningful results, accounting for biases, system stability, and caching effects.

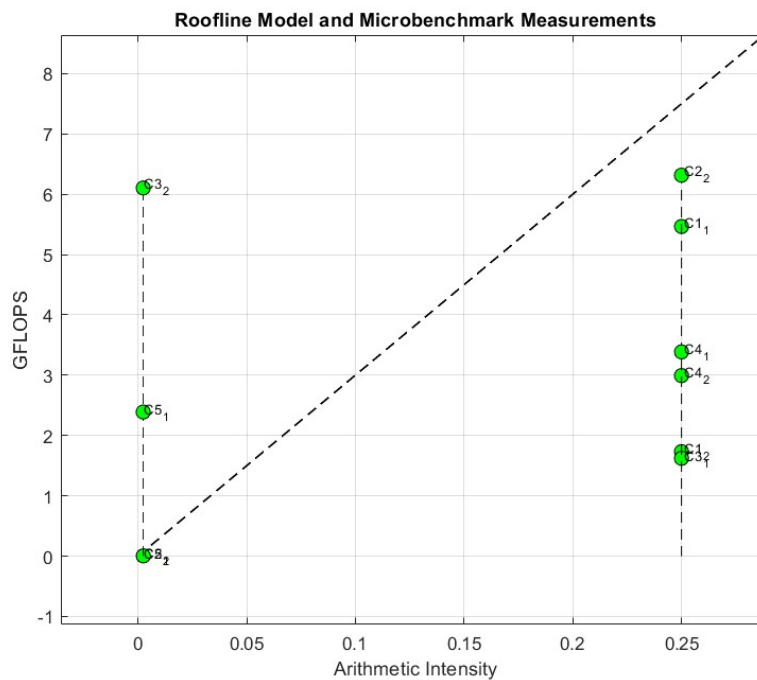
Q2 Solution:

Roofline model based on 200 GFLOPS and 30 GB/s with a blue vertical line for the arithmetic intensity is shown below:

$$\text{Arithmetic intensity} = (200 \text{ GFLOPS}) / (30 \text{ GB/s}) = 6.67$$



The zoomed-in plot of the 10 measurements for the average results for the microbenchmarks is below:



Q3 Solution:

The following explains the difference in performance for the 5 measurements in the C and Python variants:

C1:

- Uses a simple loop to calculate the dot product in C.
- Achieves a throughput of 1.621 GFLOP/sec and a bandwidth of 6.484 GB/sec for $N = 300000000$.
- Compared to Python, C offers better performance due to its lower-level nature and efficient memory access.

C2:

- Utilizes loop unrolling to compute the dot product in C.
- Achieves a higher throughput of 2.993 GFLOP/sec and bandwidth of 11.972 GB/sec for $N = 300000000$ compared to C1.
- The use of loop unrolling optimizes the loop iteration process, resulting in improved performance over the basic C1 implementation.

C3:

- Uses the Intel Math Kernel Library (MKL) for dot product computation.
- Achieves a throughput of 3.386 GFLOP/sec and bandwidth of 13.543 GB/sec for $N = 300000000$.

- MKL provides highly optimized routines for linear algebra operations, leading to significantly improved performance over the basic C1 implementation.

C4:

- Implements the dot product in Python using a simple loop with NumPy arrays.
- Achieves a much lower throughput of 0.004 GFLOP/sec and bandwidth of 1.604 GB/sec for $N = 300000000$ compared to the C implementations.
- Python's interpreted nature and lack of native support for efficient numerical operations result in significantly lower performance compared to C.

C5:

- Uses NumPy's dot function for dot product computation in Python.
- Achieves a throughput of 2.390 GFLOP/sec and bandwidth of 994.125 GB/sec for $N = 300000000$.
- NumPy's dot function is highly optimized and implemented in C, providing performance closer to C implementations compared to the basic Python loop (C4).

Analysis:

- The C variants generally outperform the Python variants in terms of time, bandwidth, and throughput due to their lower-level nature and better optimization opportunities.
- C2 (unrolled loop) and C3 (MKL) show improvements over the basic C1 implementation, demonstrating the impact of optimization techniques.
- Python's performance is significantly lower, especially evident in C4 (Python loop) compared to the C variants.
- C5 (Python NumPy dot function) shows a performance boost compared to the pure Python loop (C4), highlighting the benefits of using optimized libraries in Python for numerical computations.
- Overall, the choice of language and implementation strategy (e.g., using libraries, optimizing loops) can greatly impact the performance of dot product computations.

Inference:

- The choice of programming language can have a significant impact on the performance of numerical computations. In this case, C outperformed Python due to its lower-level nature and better optimization capabilities.
- Optimization techniques such as loop unrolling (C2) and using specialized libraries like MKL (C3) can lead to performance improvements compared to basic implementations (C1), highlighting the importance of optimizing code for specific tasks.
- Utilizing specialized libraries, such as NumPy in Python, can greatly enhance performance. The dot function in NumPy (C5) performed much better than a pure Python loop (C4), demonstrating the benefits of leveraging optimized libraries for numerical computations.
- While Python offers ease of use and readability, it comes with a performance trade-off compared to lower-level languages like C. Developers need to consider these trade-offs based on the specific requirements of their applications.
- Benchmarking different implementations is crucial for understanding their performance characteristics and choosing the most suitable approach for a given task.

Q4 Solution:

The following explains the findings of the result of the dot product computations against the analytically calculated result:

C1 Benchmark (C language):

- For $N=1000000$, the computed result matches the expected result exactly.
- For $N=300000000$, the computed result is slightly off, showing the impact of floating-point arithmetic limitations. This discrepancy is likely due to the accumulation of rounding errors in the floating-point arithmetic operations.

C2 Benchmark (C language):

- For $N=1000000$, the computed result matches the expected result exactly.
- For $N=300000000$, the computed result is slightly off again demonstrating the impact of floating-point arithmetic limitations.

C3 Benchmark (C language):

- For $N=1000000$, the computed result matches the expected result exactly.
- For $N=300000000$, the computed result matches the expected result exactly. This can be attributed to the use of the MKL library, which is optimized for performance and can handle floating-point arithmetic more accurately than standard implementations.

C4 Benchmark (Python language):

- For both $N=1000000$ and $N=300000000$, the computed results match the expected results exactly.
- The simplicity of the computation in C4 leads to exact results without significant floating-point errors.

C5 Benchmark (Python language):

- Similar to C4, for both $N=1000000$ and $N=300000000$, the computed results match the expected results exactly.
- The use of `numpy.dot` in Python ensures precise results without floating-point errors.

Inference:

- For the C benchmarks (C1, C2, C3), the computed results generally match the expected results for $N=1000000$. However, for $N=300000000$, there are slight discrepancies, indicating the impact of floating-point arithmetic limitations. This is expected, as floating-point operations are not always exact and can introduce rounding errors.
- Interestingly, the C3 benchmark, which uses the MKL library, shows exact results for $N=300000000$. This suggests that the MKL library's optimized implementation can handle floating-point arithmetic more accurately, leading to precise results even for large values of N .
- In contrast, the Python benchmarks (C4, C5) also show exact results for both $N=1000000$ and $N=300000000$. This is likely due to the simplicity of the dot product computation in Python and the use of optimized libraries like `numpy.dot`, which handle floating-point arithmetic accurately.
- These findings suggest that while floating-point arithmetic limitations can lead to discrepancies in computed results, the choice of language and libraries can mitigate these

issues. Optimized libraries and implementations can improve the accuracy of floating-point computations, especially for large-scale calculations.