Denzel Farmer

High Performance ML

Homework 1 Report

C1 Output

```
N: 1000000 <T>: 0.001198 sec B: 6.676 GB/sec F: 1.669 GFLOP/sec
N: 300000000 <T>: 0.469145 sec B: 5.116 GB/sec F: 1.279 GFLOP/sec
```

C2 Output

```
N: 1000000 <T>: 0.000404 sec B: 19.778 GB/sec F: 4.944 GFLOP/sec
N: 300000000 <T>: 0.271302 sec B: 8.846 GB/sec F: 2.212 GFLOP/sec
```

C3 Output

```
N: 1000000 <T>: 0.000097 sec B: 82.832 GB/sec F: 20.708 GFLOP/sec
N: 300000000 <T>: 0.050211 sec B: 47.798 GB/sec F: 11.950 GFLOP/sec
```

C4 Output

```
N: 1000000 <T>: 0.247236 sec B: 0.032 GB/sec F: 0.008 GFLOP/sec
N: 300000000 <T>: 82.260015 sec B: 0.029 GB/sec F: 0.007 GFLOP/sec
```

C5 Output

```
N: 1000000 <T>: 0.000332 sec B: 24.115 GB/sec F: 6.029 GFLOP/sec
N: 300000000 <T>: 0.194324 sec B: 12.351 GB/sec F: 3.088 GFLOP/sec
```

Question 1

Explain the rationale and expected consequence of only using the second half of the measurements for the computation of the mean execution time.

A. Using only the second half of measurements for computing the mean execution time minimizes transient effects like cache warming and leads to a focus on the system's steady-state performance. This approach provides a more accurate representation of how the machine behaves during long-running calculations that fully utilize its resources, which is more valuable than measuring ‘cold’ performance.

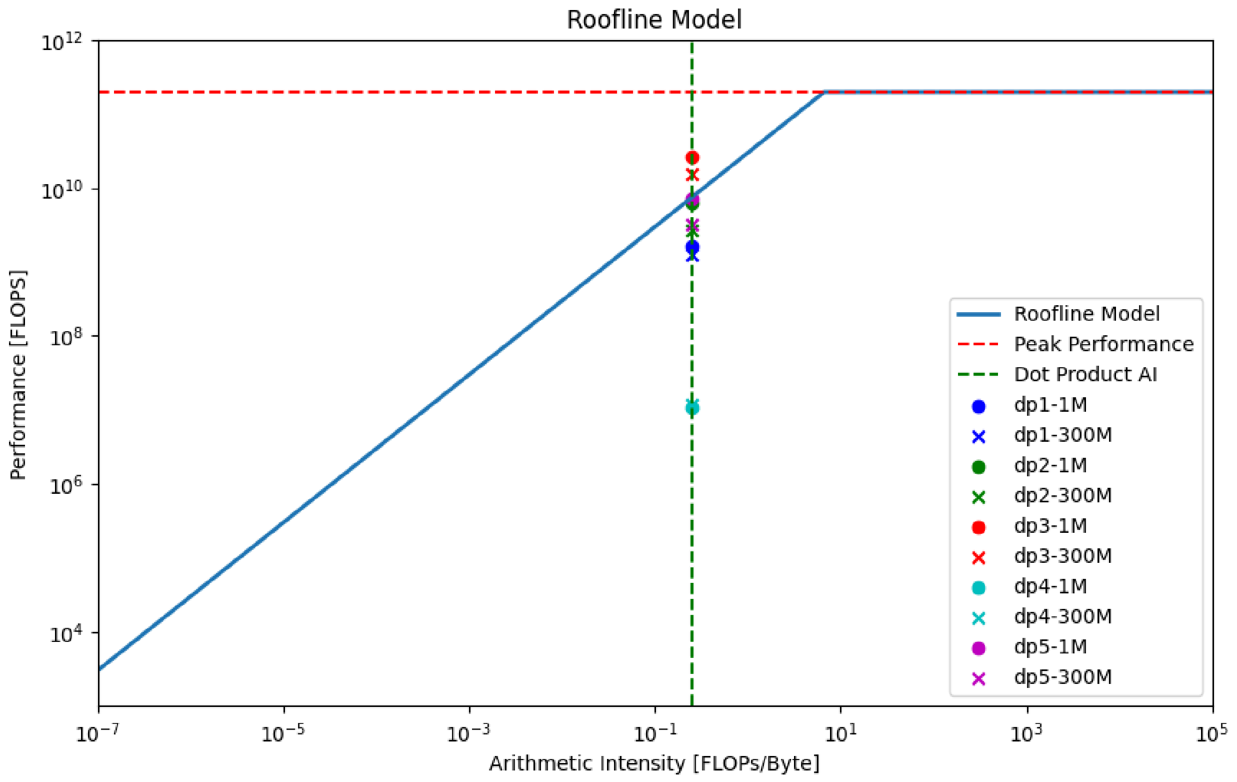
Moreover, explain what type of mean is appropriate for the calculations, and why.

B. The appropriate type of mean for these measurements is the **harmonic mean** because bandwidth and throughput are **rates** measured in units of bytes per second and operations per second, respectively. The harmonic mean places more emphasis on lower values, effectively penalizing measurements of reduced bandwidth or throughput, which can more severely impact overall performance than higher values can improve it. So, using the harmonic mean ensures that the calculated average more accurately reflects the system's true performance.

Question 2

Draw a roofline model based on a peak performance of 200 GFLOPS and memory band-width of 30 GB/s. Add a vertical line for the arithmetic intensity. Plot points for the 10 measurements for the average results for each microbenchmark. The roofline model must be "plotted" using matplotlib or an equivalent package.

A. Roofline Model:



Note: Roofline plotted on log-log scale

AI = (total flops) / (total bytes)

Total flops = count * size * 2

Total bytes = count * size * 2 * 4

AI = (count * size * 2) / (count * size * 2 * 4) = ¼

Based on your plotted measurements, explain clearly whether the computations are compute or memory bound, and why. Discuss the underlying reasons for why these computations differ or don't across each microbenchmark.

B. All computations are memory-bound, because the AI is below the threshold (~6.66 flops/byte) where the machine/algorithm pair transitions from being memory-bound to compute-bound. This value remains constant across all microbenchmarks since they perform the same number of floating-point operations on the same amount of data—the specific dot product algorithm chosen affects performance but not the AI.
   The computations that perform better are the ones optimized to take maximum effect of memory bandwidth, for example by overlapping FLOP execution with memory fetching, or prefetching. Specifically, the 'unrolled' C example demonstrates that fetching more memory per iteration improves performance significantly by reducing time spent waiting for individual memory loads. The maximally optimized library code performs the best, even outperforming the roofline model, indicating that it may be able to reduce the number of required bytes / operation, changing the 'true' arithmetic intensity.

Lastly, identify any microbenchmarks that underperform relative to the roofline, and explain the algorithmic bottlenecks responsible for this performance gap.

C. The naive Python implementation significantly underperforms relative to the roofline due to interpreter overhead and lack of compile-time optimizations. Because each loop iteration over the input vectors requires switching to the interpreter, each loop iteration consumes more cycles than required. The frequent switching to interpreter instructions also likely harms cache hit rates and prefetching, as instructions unrelated to the vector calculation are interposed. Far fewer of the FLOPs likely happens concurrently with fetching memory.
   In contrast, the NumPy Python implementation does not suffer from these bottlenecks because it likely uses optimized C routines for vector operations rather than looping through the vectors naively. Each vector operation is almost as efficient as in the optimized C library code version.
   The non-library C implementation does not underform the roofline as significantly, but does underperform compared to both the 'unrolled' implementation (because this implementation fetches more memory per iteration) and the optimized library implementation (because the library implementation has advanced optimizations, that

may actually alter the arithmetic intensity of the algorithm). The C library version also performs so well because it may use advanced hardware features like SIMD instructions.

Question 3

Using the N = 300000000 simple loop as the baseline, explain the the difference in performance for the 5 measurements in the C and Python variants. Explain why this occurs by considering the underlying algorithms used.

A. Naive C: 1.278 GFLOP/sec
   Unrolled C: 2.212 GFLOP/sec
   Library C: 11.950 GFLOP/sec
   Naive Python: 0.007 GFLOP/sec
   Numpy Python: 3.008 GFLOP/sec

   The largest performance gap is between the naive Python implementation and the other results. This is almost certainly because (as explained in Q2) this implementation is entirely interpreted, so every arithmetic operation requires overhead from the interpreter itself (which harms both cycle count and likely cache hit rates). Single operation overhead becomes extremely impactful as the vector size increases, which explains why the relative naive Python performance is worse in the N=300M case than the N=1M case. In addition, the Python code cannot benefit from compiler optimization like other versions of the algorithm–these optimizations may reduce the amount of memory transfer required, which is especially important considering the algorithm is memory bound on this machine.

   This gap is eliminated by the NumPy python code, and it even outperforms the naive and unrolled C implementations. This is because the Numpy library both offloads heavy computation to compiled C routines, and those routines likely take advantage of similar optimizations as found in the library-based C implementation. The per-line overhead on the remaining ‘calling’ code in Python is negligible.

Question 4

Check the result of the dot product computations against the analytically calculated result. Explain your findings, and why the results occur. (Hint: Floating point operations are not exact.)

When the input vectors are filled with 1s, the results for both ‘naive’ C implementations (dp1 and dp2) are exactly correct, until their result hits the float maximum that can be incremented, 16777216.0. A 32-bit float stores a 23-bit mantissa, and 16777216 is the largest value that can be stored in 23 bits. While a 32 bit float can store larger values, it cannot increment larger (since the mantissa cannot take up the required 24 bits), so the result becomes incorrect.

On the other hand, the library-based dp3 C code likely uses internal computations that are more complex than simple incrementing and as a result can produce near-correct values for much larger results. However, since the fundamental limitation on the mantissa size still exists, they lose some precision. For example, dp3 with size=1010000001 produces result 1010000000 (when the result should clearly be 1010000001).

For ‘fill values’ other than whole numbers (like 0.334, for example), the non-integer multiplication on each iteration introduces imprecision into the floating point calculations at much lower values for both ‘naive’ and library based C examples. For example, with a fill value of 0.334 and a size of 100, the result produced by dp1 is 11.155604 rather than the expected value of 11.155600.

While both Python examples also suffer from imprecision for non-integer fill values, for integer fill values the maximum ‘float’ that can be incremented is higher, likely because Python dynamically chooses an underlying type larger than 32-bits (or at least, with a mantissa larger than 23 bits).