**Coding Results**

E2 standard, 16 vCPUs, 64GB RAM, N = 1000000, Reps = 1000:

```
./dp1 1000000 1000
N: 1000000   <T>: 0.001511 sec   B: 5.295617 GB/sec   F: 1323904171.854163 FLOP/sec
Dot Product Result: 1000000.000000

./dp2 1000000 1000
N: 1000000   <T>: 0.000450 sec   B: 17.775787 GB/sec   F: 4443946840.859510 FLOP/sec
Dot Product Result: 1000000.000000

./dp3 1000000 1000
N: 1000000   <T>: 0.000066 sec   B: 120.574941 GB/sec   F: 30143735272.990257 FLOP/sec
Dot Product Result: 1000000.000000

python3 dp4.py 1000000 1000
N: 1000000   <T>: 0.40264037209400794 sec   B: 0.039737694252538446 GB/sec   F:
4967211.781567305 FLOP/sec
Dot Product Result: 1000000.0

python3 dp5.py 1000000 1000
N: 1000000   <T>: 0.00024069227000083669 sec   B: 66.47492252012835 GB/sec   F:
8309365315.016043 FLOP/sec
Dot Product Result: 1000000.0
```

E2 standard, 16 vCPUs, 64GB RAM, N = 300000000, Reps = 20

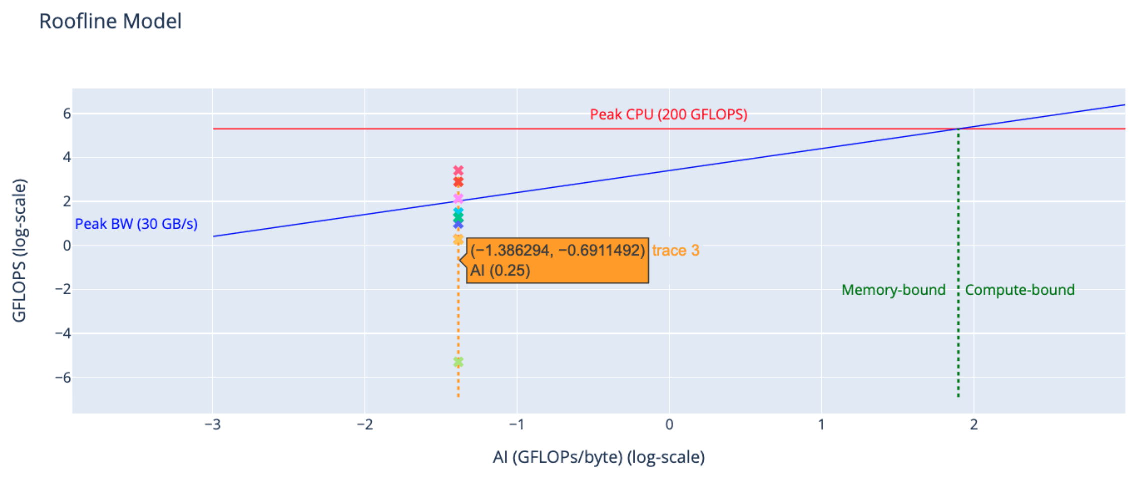
```
./dp1 300000000 20
N: 300000000   <T>: 0.463500 sec   B: 5.177999 GB/sec   F: 1294499738.378762 FLOP/sec
Dot Product Result: 16777216.000000

./dp2 300000000 20
N: 300000000   <T>: 0.221922 sec   B: 10.814623 GB/sec   F: 2703655673.126110 FLOP/sec
Dot Product Result: 67108864.000000

./dp3 300000000 20
N: 300000000   <T>: 0.033396 sec   B: 71.864192 GB/sec   F: 17966048067.832813 FLOP/sec
Dot Product Result: 300000000.000000

python3 dp4.py 300000000 20
N: 300000000   <T>: 108.77106600069992 sec   B: 0.04412938271625575 GB/sec   F:
5516172.839531968 FLOP/sec
Dot Product Result: 300000000.0
```

```
python3 dp5.py 300000000 20
N: 300000000   <T>: 0.16866607580022902 sec   B: 28.458597718756454 GB/sec   F: 3557324714.8445563 FLOP/sec
Dot Product Result: 300000000.0
```

**Questions**

1. During the first half of measurements for computing the average execution time, there could be outlier execution times due to the program needing to perform optimization, prefetching, and caching. This distribution of execution times could have a larger standard deviation and a skewed distribution which could affect the mean average time. It's better to take the mean of the second half of measurements to get a more representative distribution and omit outlier execution times due to the optimization/caching reason mentioned above.

2.



Note that the AI is 0.25, which is displayed as ~-1.38 on the plot since it's in log scale. Also, many of the experiments result in GFLOPS greater than that allowed by the peak BW. This probably happens because I use a GCP instance with 16 vCPUs and 64GB RAM, which has a higher peak CPU and peak BW than that stated in the problem.

3. The python dot product implementation using loops (dp4) is by far the slowest, due to the fact that there is no vectorization of operations and the python interpreter is much slower than C. The next slowest is the C dot product implementation using loops (dp1). Once again, this code doesn't make use of vectorization. The third slowest is the C implementation using loops but with more operations per loop (dp2). This is closer to vectorization, which speeds up the computation. The fourth slowest is the python implementation using numpy (dp5). This has the advantage of full vectorization, and many of numpy's functions are implemented using C. However, the python interpreter still has to interpret the python code and then compile C code, which requires more "middle men" than having only C code. Finally, the C implementation using MKL is the fastest (dp3). MKL is optimized to work with Intel CPUs and leverages vectorization

making it the fastest.

4. In the small case (n=1000000), the analytical result is 1000000. In all programs, the number is small enough to avoid a float overflow. In the large case (n=300000000), the analytical result is 300000000. In dp1, the result is 16777216.000000. Due to the limited prevision, adding 1 to this number makes it stay the same, and even in future iterations the number remains as is. In dp2, the result is 67108864.000000. 4 is added to the cumulative sum in each iteration, so when 4 is added to 67108864.000000, once again the float type cannot handle a larger value, so the value remains as 67108864.000000 for all future iterations. In dp3, this is implemented using mkl which does vectorized operations and prevents this underflow/overflow behavior. In dp4 and dp5, python uses doubles (double precision floats) which prevents this behavior.