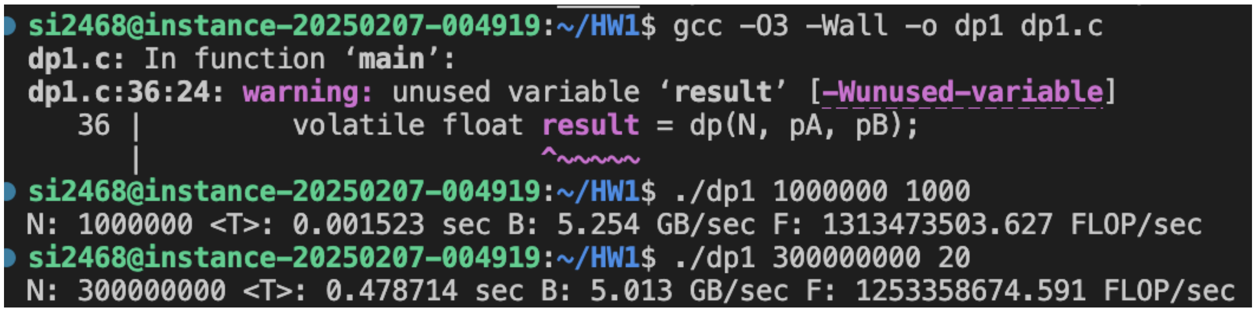
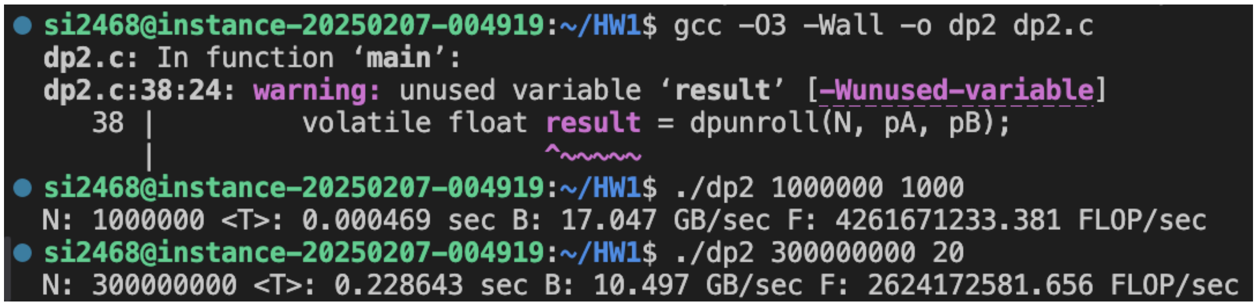
Coding Outputs

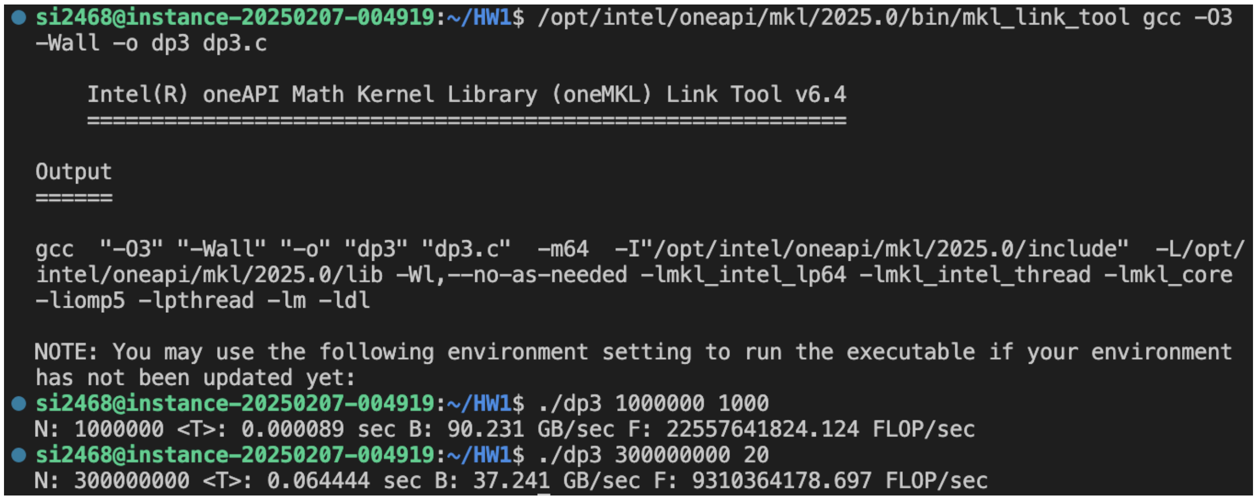
C1


```
si2468@instance-20250207-004919:~/HW1$ gcc -O3 -Wall -o dp1 dp1.c
dp1.c: In function ‘main’:
dp1.c:36:24: warning: unused variable ‘result’ [-Wunused-variable]
   36 |         volatile float result = dp(N, pA, pB);
      |                        ^~~~~~
si2468@instance-20250207-004919:~/HW1$ ./dp1 1000000 1000
N: 1000000 <T>: 0.001523 sec B: 5.254 GB/sec F: 1313473503.627 FLOP/sec
si2468@instance-20250207-004919:~/HW1$ ./dp1 300000000 20
N: 300000000 <T>: 0.478714 sec B: 5.013 GB/sec F: 1253358674.591 FLOP/sec
```

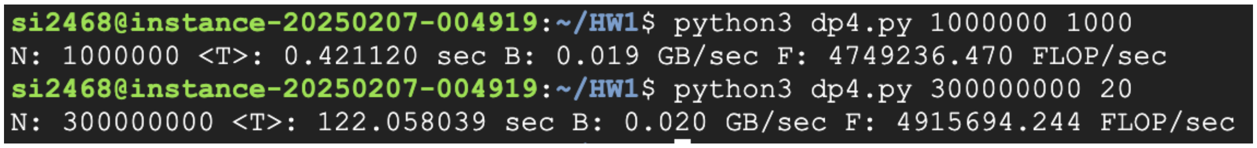
C2


```
si2468@instance-20250207-004919:~/HW1$ gcc -O3 -Wall -o dp2 dp2.c
dp2.c: In function ‘main’:
dp2.c:38:24: warning: unused variable ‘result’ [-Wunused-variable]
   38 |         volatile float result = dpunroll(N, pA, pB);
      |                        ^~~~~~
si2468@instance-20250207-004919:~/HW1$ ./dp2 1000000 1000
N: 1000000 <T>: 0.000469 sec B: 17.047 GB/sec F: 4261671233.381 FLOP/sec
si2468@instance-20250207-004919:~/HW1$ ./dp2 300000000 20
N: 300000000 <T>: 0.228643 sec B: 10.497 GB/sec F: 2624172581.656 FLOP/sec
```

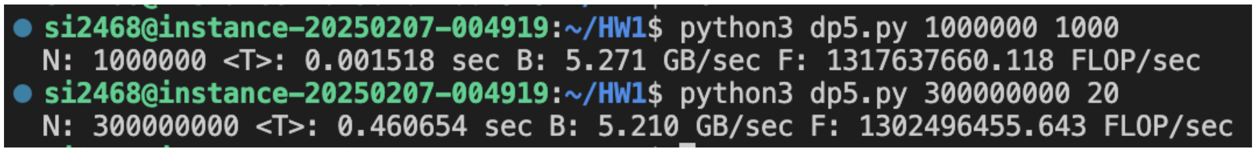
C3


```
si2468@instance-20250207-004919:~/HW1$ /opt/intel/oneapi/mkl/2025.0/bin/mkl_link_tool gcc -O3
-Wall -o dp3 dp3.c

    Intel(R) oneAPI Math Kernel Library (oneMKL) Link Tool v6.4
    ===========================================================

Output
======

gcc  "-O3" "-Wall" "-o" "dp3" "dp3.c"  -m64  -I"/opt/intel/oneapi/mkl/2025.0/include"  -L/opt/
intel/oneapi/mkl/2025.0/lib -Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core
-liomp5 -lpthread -lm -ldl

NOTE: You may use the following environment setting to run the executable if your environment
has not been updated yet:
si2468@instance-20250207-004919:~/HW1$ ./dp3 1000000 1000
N: 1000000 <T>: 0.000089 sec B: 90.231 GB/sec F: 22557641824.124 FLOP/sec
si2468@instance-20250207-004919:~/HW1$ ./dp3 300000000 20
N: 300000000 <T>: 0.064444 sec B: 37.241 GB/sec F: 9310364178.697 FLOP/sec
```

C4


```
si2468@instance-20250207-004919:~/HW1$ python3 dp4.py 1000000 1000
N: 1000000 <T>: 0.421120 sec B: 0.019 GB/sec F: 4749236.470 FLOP/sec
si2468@instance-20250207-004919:~/HW1$ python3 dp4.py 300000000 20
N: 300000000 <T>: 122.058039 sec B: 0.020 GB/sec F: 4915694.244 FLOP/sec
```

C5

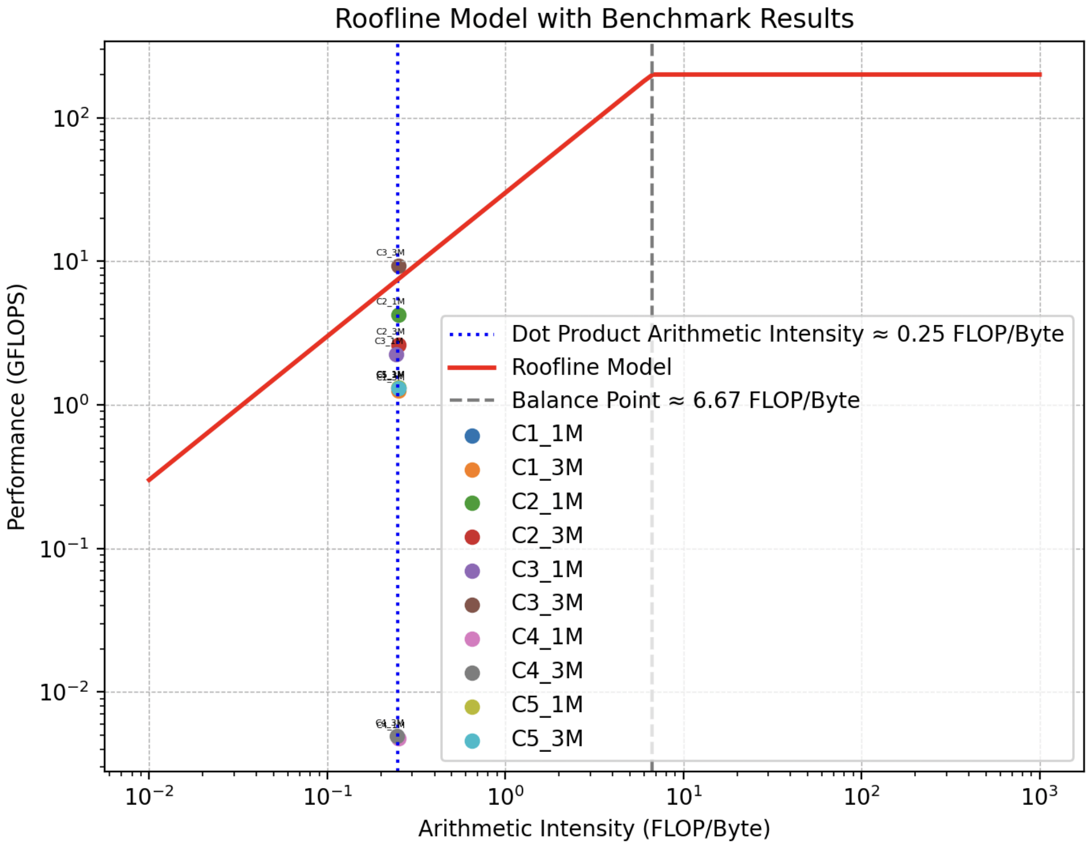

```
● si2468@instance-20250207-004919:~/HW1$ python3 dp5.py 1000000 1000
  N: 1000000 <T>: 0.001518 sec B: 5.271 GB/sec F: 1317637660.118 FLOP/sec
● si2468@instance-20250207-004919:~/HW1$ python3 dp5.py 300000000 20
  N: 300000000 <T>: 0.460654 sec B: 5.210 GB/sec F: 1302496455.643 FLOP/sec
```

Q1: Explain the rationale and expected consequence of only using the second half of the measurements for the computation of the mean execution time. Moreover, explain what type of mean is appropriate for the calculations, and why.

From an architectural standpoint, the rationale behind using the second half of the measurements is to get an estimate after the cache has been warmed up and the branch predictors have been “trained”. Initially, the cache will have cold misses and will get filled up with values by the middle of the program, or at least somewhat filled up. Additionally, the branch predictors of modern processors will better understand which branches to take speculatively ahead of time to more efficiently perform out of order processing. Thus, the expected consequence is that the mean time of the second half of readings will be more representative of a CPU’s performance ceiling, and will be a bit faster than the mean time of all the readings.

The arithmetic mean is appropriate for the calculations. This is because we are measuring times, rather than throughput, and because the sum of the values represents something physical - the total time it takes to run all of the computations. Additionally, the values in the latter half of each experiment will be similar due to cache usage and branch prediction warming up, so there is no need to use harmonic mean.

Q2: Draw a roofline model based on a peak performance of 200 GFLOPS and memory bandwidth 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. 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. Lastly, identify any microbenchmarks that underperform relative to the roofline, and explain the algorithmic bottlenecks responsible for this performance gap.



Based on the plotted measurements, the computations are memory-bound. This is because the points lie to the left of the balance point, and for the most part, underneath the roofline. There is one point that is above the roofline, but I would hypothesize that this is probably due to the fact that we made an assumption that the peak bandwidth of the system was 30 GB/s, when in reality it might be more than that. If a point lies to the left of the balance point, it indicates that the algorithm that the point came from is constrained by memory bandwidth and is memory-bound.

For N = 1000000, the performances are ranked as follows:

1. C2
2. C3
3. C5
4. C1
5. C4

For N = 300000000, the performances are ranked as follows:

1. C3
2. C2
3. C5
4. C1
5. C4

We see that for size N = 1 million, C3 is worse than C2, but for N = 300M, C3 is better than C2. C3 uses a library to perform the dot product with SIMD operations and employs parallelism. Parallelism suffers from an overhead that will only be worth it if the problem is of large enough size. Thus, 1 million elements is probably not a large enough array for this parallelism to be worth it, but 300 million is enough.

In both cases, we see an improvement from C1 to C2. This is because by unrolling the loop in C1, C2 suffers from less branching, condition checking, and iterator incrementing, which results in overall less assembly instructions and faster runtime. Furthermore, it is likely that the unrolled implementation benefits from the cache due to spatial locality and the fact that caches operate on cache lines of size greater than one float.

In both cases, C4 is worse than C5. This is because C4 is using the barebones Python implementation with a for loop, with Python overhead (interpreted language) as well as lack of parallelization and compiler optimizations. This is because C5 uses numpy.dot, which takes advantage of parallelizable hardware to perform computations in parallel with SIMD. It is essentially the Pythonic version of C3, with some extra overhead.

Finally, C5 beats C1 in both cases. This means that the optimizations of numpy.dot() compensates for the Python overhead. The fastest Python version beats the slowest C version, but nothing more.

Everything except the C3-3M version underperforms with respect to the roofline. The algorithmic reasons for this are manyfold. The Python implementations have significant overhead. Some implementations in both languages do not employ SIMD-style computing, which means that they cannot perform as many operations with respect to the amount of memory being loaded in.

Q3: 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.

DP4 and DP1 are the same algorithms implemented in Python and C respectively. The reasons why the Python version is so much slower are due to multiple factors. One of the reasons is that the Python version is interpreted line by line, which adds overhead when compared to the C version, which is compiled first into machine code. Another reason is that the compilation of the C program is done with the -O3 flag, which ensures that the compiler uses many optimization techniques to improve the performance of the C code, such as loop unrolling, which Python does not do inherently.

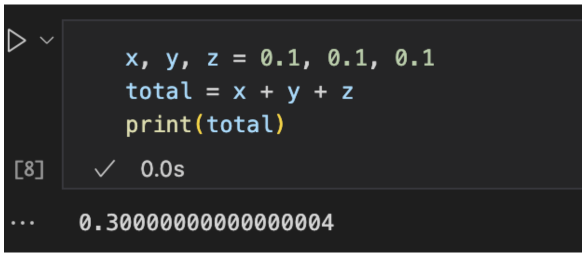
DP3 and DP5 are implementations of the same algorithm in C and Python respectively. Here, the C program invokes BLAS to use a heavily optimized dot product function that can compute the dot product of two vectors using SIMD. The Python version uses numpy.dot, which internally uses BLAS to do the same thing. The reason why the C version is still faster is because the Python interface acts as a wrapper, and the Python function calls incur lots of overhead.

DP2 is a C implementation that falls in between DP1 and DP3. It unrolls the loop in DP1, which allows for less condition checking and iterator increases, and thus increases performance by lowering the number of assembly instructions required.

Q4: Check the result of the dot product computations against the analytically calculated result. Explain your findings, and why the results occur. (Hint: Floating point operation are not exact.

When I perform the dot product for N = 300000000, I get a value of 16777216.000000. The answer should be 300000000. The reason for this is that float variables have 32 bits, and with the way the numbers are represented (only 23 mantissa bits), the maximum value for an exact integer in a float representation is 2 ^ 24 = 16777216. So the input size is too big for the float variable to be represented exactly. Thus we lose information due to lack of precision in np32 data types. Increasing to a 64 bit representation yields the right answer due to increased precision.

As a side note, if we were to perform the following operations:



```
x, y, z = 0.1, 0.1, 0.1
total = x + y + z
print(total)
```

```
[8]  ✓  0.0s
```

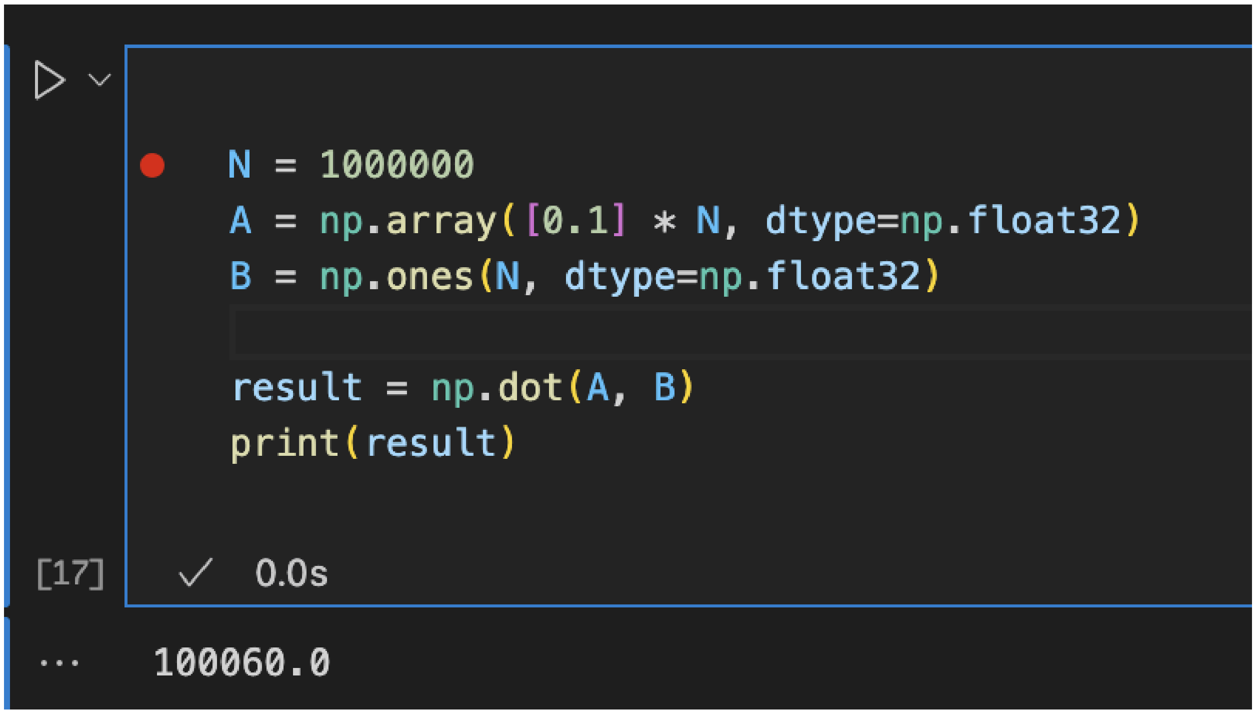
```
0.30000000000000004
```

We would not get 0.3, but rather 0.30000…..4.

This is because it is not possible to represent 0.1 exactly with floating point representations, so the exact value is off. The only values that can be represented with perfect accuracy are numbers that can be written as sums of powers of 2, assuming that there are an appropriate number of exponent bits in the floating point representation to account for it. Thus it easier to see the lack of precision if we use decimals who we know can not be represented by sums of powers of 2 (like the example above).

In our current implementations, we are performing the addition of many 1.0's, which can be written as the sum of powers of 2. Thus, it is unlikely we will incur any floating point precision loss by summing and multiplying them. However, if we changed the values in one array to 0.1, and kept the values in the other array at 1.0, we will see that our result is off because adding many (0.1 * 1.0 = 0.1) together will result in floating point precision loss.

This is with an enforced precision of float 32 for N = 1000000. We get an answer that is significantly off of the expected answer.

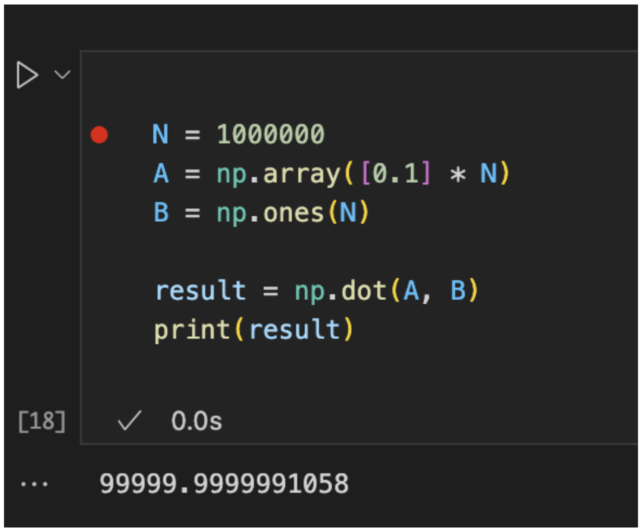


```
N = 1000000
A = np.array([0.1] * N, dtype=np.float32)
B = np.ones(N, dtype=np.float32)

result = np.dot(A, B)
print(result)
```

```
[17]  ✓  0.0s
```

```
100060.0
```

If we don’t enforce the types:



```
N = 1000000
A = np.array([0.1] * N)
B = np.ones(N)

result = np.dot(A, B)
print(result)
```

```
99999.9999991058
```

We get a much closer answer. This is because we have more precision, but it is not enough to make up for the fact that 0.1 cannot be represented completely accurately.