Output of the Programs

C1

```
pkh2120@instance-20250921-001503:~$ ./dp1 1000000 1000
N: 1000000 <T>: 0.001531 sec B: 5.226 GB/sec F: 1306572476.484 FLOP/sec
Dot Product Result: 1000000.000000

pkh2120@instance-20250921-001503:~$ ./dp1 300000000 20
N: 300000000 <T>: 0.470168 sec B: 5.105 GB/sec F: 1276139060.669 FLOP/sec
Dot Product Result: 16777216.000000
```

C2

```
pkh2120@instance-20250921-001503:~$ ./dp2 1000000 1000
N: 1000000 <T>: 0.000841 sec B: 9.518 GB/sec F: 2379464971.733 FLOP/sec
Dot product result: 1000000.000000

pkh2120@instance-20250921-001503:~$ ./dp2 300000000 20
N: 300000000 <T>: 0.236008 sec B: 10.169 GB/sec F: 2542291554.957 FLOP/sec
Dot product result: 67108864.000000
```

C3

```
pkh2120@instance-20250921-001503:~$ ./dp3 1000000 1000
N: 1000000 <T>: 0.000086 sec B: 93.295 GB/sec F: 23323663337.679 FLOP/sec
Dot product result: 1000000.000000

pkh2120@instance-20250921-001503:~$ ./dp3 300000000 20
N: 300000000 <T>: 0.053415 sec B: 44.931 GB/sec F: 11232733640.898 FLOP/sec
Dot product result: 300000000.000000
```

C4

```
pkh2120@instance-20250921-001503:~$ python3 dp4.py 1000000 1000
N: 1000000 <T>: 0.277011 sec B: 0.029 GB/sec F: 7219940.395 FLOP/sec
Dot product result: 1000000.0

pkh2120@instance-20250921-001503:~$ python3 dp4.py 300000000 20
N: 300000000 <T>: 94.458942 sec B: 0.025 GB/sec F: 6351966.144 FLOP/sec
Dot product result: 16777216.0
```

C5

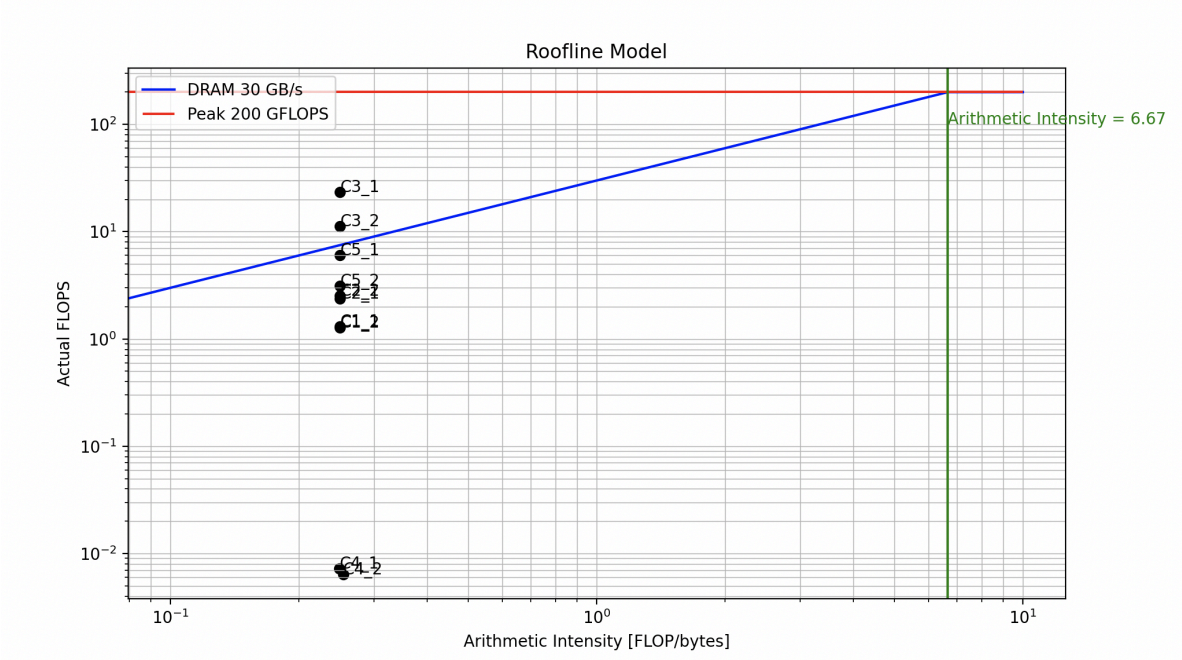
```
pkh2120@instance-20250921-001503:~$ python3 dp5.py 1000000 1000
N: 1000000 <T>: 0.000333 sec B: 24.028 GB/sec F: 6007038847.873 FLOP/sec
Dot product result: 1000000.0

pkh2120@instance-20250921-001503:~$ python3 dp5.py 300000000 20
N: 300000000 <T>: 0.192057 sec B: 12.496 GB/sec F: 3124080208.627 FLOP/sec
Dot product result: 300000000.0
```

Q1
The first half of the measurements might include the startup effects(frequent cache misses since there's no relevant data in the CPU cache yet). Whereas the second half of the measurements benefit from a warm cache(frequent cache hits since a lot of the necessary data for computation is loaded in the caches now). So only using the second half of the measurements for the computation of the mean execution time eliminates the penalties of the startup effects which better reflects the system's true computational performance. I guess the expected consequence would be that the measurement for the mean execution time might be too optimistic since it doesn't include the startup effects. The appropriate type of mean for the calculations is the arithmetic mean since it gives us the expected/average time for a single run which is what we want to use for the mean execution time, bandwidth, and throughput. The harmonic mean is only useful when we are averaging ratios inversely related to total performance and the geometric mean is only useful when we have values that tend to have large fluctuations. In our case, the total time for each repetition isn't inversely related to total performance and doesn't fluctuate too much which is why the harmonic mean and geometric mean aren't really appropriate.

Q2
_1 = output for array size 1,000,000 and 1,000 repetitions
_2 = output for array size 300,000,000 and 20 repetitions

(C1_1) = (C1_1_x, C1_1_y) = (0.25001386844316875, 1.3065724764839999)
(C1_2) = (C1_2_x, C1_2_y) = (0.24997826849539664, 1.276139060669)
(C2_1) = (C2_1_x, C2_1_y) = (0.24999631978703507, 2.379464971733)
(C2_2) = (C2_2_x, C2_2_y) = (0.2500040864349493, 2.542291554957)
(C3_1) = (C3_1_x, C3_1_y) = (0.24999907109361702, 23.323663337679)
(C3_2) = (C3_2_x, C3_2_y) = (0.24999963590612273, 11.232733640898001)
(C4_1) = (C4_1_x, C4_1_y) = (0.24896346189655172, 0.007219940395)
(C4_2) = (C4_2_x, C4_2_y) = (0.25407864576, 0.006351966144)
(C5_1) = (C5_1_x, C5_1_y) = (0.2500016167751373, 6.007038847873)
(C5_2) = (C5_2_x, C5_2_y) = (0.25000641874415813, 3.124080208627)



The code used to generate this roofline model plot can be seen in the rooflineModel.py file which I also provided in this tar-archive. These computations are obviously all memory bound since all the plotted points(both measurements for the average results for each benchmark) are in the region to the left of the arithmetic intensity line where x = 6.67. The actual FLOPs for C2 are slightly better than C1 due to the for loop having less iterations and the CPU's ability to make use of parallelization in each iteration of the for loop. C2 loops N // 4 times and performs 4 multiplications and 4 add operations per iteration. So C2 performs less iterations of its for loop and on top of that, the CPU can parallelize the 4 multiplication operations in each iteration of the for loop in C2 which makes better use of the CPU and cache and improves the performance of the overall system even more. C3 performs the best and displays the highest values for actual FLOPs because the underlying function it calls is highly optimized for calculating dot products and contains a ton of advanced optimizations(vectorization, high cache bandwidth, etc.) that make it way faster than any trivial solution we can write in C code. Even though the algorithm for C4 is the same as the algorithm for C1, it performs significantly slower than any other benchmarks due to the overhead caused by Python and its interpreter which is why it performed the worst and displays the lowest values for actual FLOPs. And although C5 calls an underlying function that is also highly optimized for calculating dot products and contains a ton of advanced optimizations(vectorization, high cache bandwidth, etc.), it probably didn't perform as well as C3 also due to the overhead caused by Python and its interpreter. The arithmetic intensity for all the measurements is around 0.25 so we can assume that any measurement below 0.25 x (30 GB / s) = 7.5 GFLOPS is underperforming relative to the roofline. According to the graph and measurements shown above, only benchmark C3 doesn't underperform relative to the roofline. Obviously C1 underperforms due to the absence of any type of optimizations like vectorization, threading, etc. C2 also still underperforms due to its partial optimizations which still don't make full use of the CPU and caching available. C4 obviously underperforms due to the absence of any type of optimizations like vectorization, threading, etc. and the overhead caused by Python and its interpreter on top of that. Lastly, the C5 microbenchmark gets close to the roofline but still underperforms due to the Python overhead in the underlying implementation of the np.dot function which might be causing the system to not be able to make full use of the CPU and caching available.

Q3

C1 and C4 obviously perform the worst out of all the micro benchmarks across all metrics(mean execution time, bandwidth, and throughput) due to the absence of any type of optimizations in the algorithm like vectorization, threading, etc. C4 performs significantly worse than C1 due to the significant overhead caused by Python and its interpreter. C2 shows better performance than C1 across all metrics(mean execution time, bandwidth, and throughput) for a few reasons. First, C1 loops N times and performs 1 multiplication and add operation per iteration. C2 loops N // 4 times and performs 4 multiplications and 4 add operations per iteration. So C2 performs less iterations of its for loop. In addition, the CPU can parallelize the 4 multiplication operations in each iteration of the for loop in C2 which makes better use of the CPU and cache and improves the performance of the overall system even more. C3 performs the best across all metrics(mean execution time, bandwidth, and throughput) because the underlying function it calls is highly optimized for calculating dot products and contains a ton of advanced optimizations(vectorization, high cache bandwidth, etc.) that make it way faster than any trivial solution we can write in C code. Although C5 has the 2nd best performance across all metrics(mean execution time, bandwidth, and throughput) and invokes the highly optimized np.dot function to calculate the dot product, its metrics are still pretty close to the less optimal micro benchmarks like C2 due to the significant Python overhead in the underlying implementation of the np.dot function which might be causing the system to not be able to make

full use of the CPU and caching available. It's pretty surprising how much of a cost using higher level languages like Python as opposed to lower level languages like C proves to be in the overall performance of these benchmarks.

Q4

All the benchmarks obviously return the correct dot product when evaluated with the input size of 1,000,000 since 1,000,000 is well within the single-precision floating point format bounds. For C1, when evaluated with the input size of 300,000,000, since 16,777,216 is the largest float value that can be represented in single-precision floating point format with increment of 1 for the dot product in each iteration of the for loop, C1 will return 16,777,216 as the dot product between the two 'float' (32 bit) arrays instead of the expected analytical result 300,000,000. When evaluated with the input size of 300,000,000, since 67,108,864 is the largest float value that can be represented in single-precision floating point format with increment of 4 for the dot product in each iteration of the for loop, C2 will return 67,108,864 as the dot product between the two 'float' (32 bit) arrays instead of the expected analytical result 300,000,000. For C3, the cblas_sdot function in the mkl_cblas.h library probably promotes values being multiplied or added to double-precision floating point format elements('dtype=np.float64') which is why it's able to return the correct result for the dot product without any estimation or precision errors even though the input arrays store single-precision floating point format elements('dtype=np.float32'). Even though variables declared as float in Python have double-precision floating point format, the NumPy arrays initialized for the problem contain single-precision floating point format elements('dtype=np.float32') so all the computations still happen in single precision. As a result, the C4 microbenchmark will return 16,777,216 as the dot product between the two 'float' (32 bit) arrays instead of the expected analytical result 300,000,000 since 16,777,216 is the largest float value that can be represented in single-precision floating point format with increment of 1 for the dot product in each iteration of the for loop. For C5, the np.dot function probably also promotes values being multiplied or added to double-precision floating point format elements('dtype=np.float64') which is why it's able to return the correct result for the dot product without any estimation or precision errors even though the input arrays store single-precision floating point format elements('dtype=np.float32').