

High Performance Machine Learning

Homework Assignment 1

Results Report

Rajvardhan Patil (UNI: rp3316)

February 18, 2026

Benchmark Results

1. C1: Simple Dot Product (dp1.c)

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp1 1000000 1000
N: 1000000 <T>: 0.000941 sec B: 8.505 GB/sec F: 2126228184.083 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp1 1000000 1000
N: 1000000 <T>: 0.000941 sec B: 8.505 GB/sec F: 2126228184.083
FLOPs/sec
```

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp1 300000000 20
N: 300000000 <T>: 0.285269 sec B: 8.413 GB/sec F: 2103281338.471 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp1 300000000 20
N: 300000000 <T>: 0.285269 sec B: 8.413 GB/sec F: 2103281338.471
FLOPs/sec
```

2. C2: Unrolled Dot Product (dp2.c)

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp2 1000000 1000
N: 1000000 <T>: 0.000272 sec B: 29.403 GB/sec F: 7350736009.717 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp2 1000000 1000
N: 1000000 <T>: 0.000272 sec B: 29.403 GB/sec F: 7350736009.717
FLOPs/sec
```

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp2 300000000 20
N: 300000000 <T>: 0.121865 sec B: 19.694 GB/sec F: 4923479099.108 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp2 300000000 20
N: 300000000 <T>: 0.121865 sec B: 19.694 GB/sec F: 4923479099.108
FLOPs/sec
```

3. C3: MKL BLAS Dot Product (dp3.c)

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp3 1000000 1000
N: 1000000 <T>: 0.000039 sec B: 207.038 GB/sec F: 51759545029.458 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp3 1000000 1000
N: 1000000 <T>: 0.000039 sec B: 207.038 GB/sec F: 51759545029.458
FLOPs/sec
```

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp3 300000000 20
N: 300000000 <T>: 0.035325 sec B: 67.940 GB/sec F: 16984986161.115 FLOPs/sec
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp3 300000000 20
rajvardhanpatil112@rp3316:~/HPML/hw-1$ ./dp3 300000000 20
```

```
N: 300000000 <T>: 0.035325 sec B: 67.940 GB/sec F: 16984986161.115  
FLOPs/sec
```

4. C4: Python Simple Loop (dp4.py)

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp4.py 1000000 1000  
a. N: 1000000 <T>: 0.194080 sec B: 0.041 GB/sec F: 10305014.802 FLOPs/sec  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp4.py 1000000 1000  
N: 1000000 <T>: 0.194080 sec B: 0.041 GB/sec F: 10305014.802  
FLOPs/sec  
  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp4.py 300000000 20  
b. N: 300000000 <T>: 58.657153 sec B: 0.041 GB/sec F: 10228931.413 FLOPs/sec  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp4.py 300000000 20  
N: 300000000 <T>: 58.657153 sec B: 0.041 GB/sec F: 10228931.413  
FLOPs/sec
```

5. C5: NumPy Dot Product (dp5.py)

```
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp5.py 1000000 1000  
a. N: 1000000 <T>: 0.000117 sec B: 68.548 GB/sec F: 17137092121.961 FLOPs/sec  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp5.py 1000000 1000  
N: 1000000 <T>: 0.000117 sec B: 68.548 GB/sec F: 17137092121.961  
FLOPs/sec  
  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp5.py 300000000 20  
N: 300000000 <T>: 0.088966 sec B: 26.977 GB/sec F: 6744177824.628 FLOPs/sec  
b. rajvardhanpatil112@rp3316:~/HPML/hw-1$  
rajvardhanpatil112@rp3316:~/HPML/hw-1$ python3 dp5.py 300000000 20  
N: 300000000 <T>: 0.088966 sec B: 26.977 GB/sec F: 6744177824.628  
FLOPs/sec
```

Theoretical Questions

Q1: Measurement Methodology

Rationale for using the second half of measurements:

The decision to use only the second half of the measurements for computing the mean execution time is based on two important considerations:

1. Warm-up effects: The first several iterations often include overhead from:

- CPU cache warming (cold start penalties) - data must be loaded into L1/L2/L3 caches
- Branch predictor training - CPU learns loop patterns
- Memory page allocation and TLB (Translation Lookaside Buffer) misses
- Dynamic frequency scaling (CPU turbo boost settling to steady state)

- Operating system scheduling effects and context switches

2. Steady-state performance: The second half represents the system's true sustained performance after all initialization overhead has been amortized. This gives us a more reliable estimate of the actual computational performance.

Appropriate type of mean: For execution time measurements, the arithmetic mean is appropriate because:

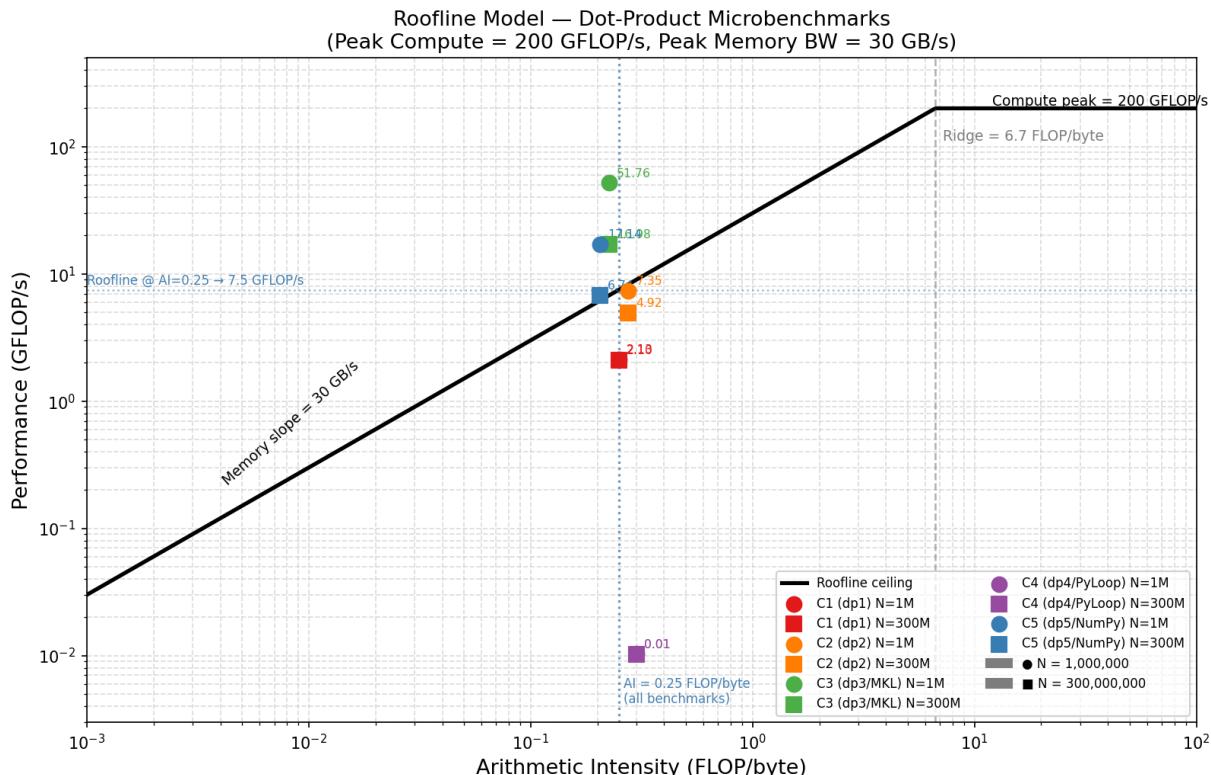
- Execution times are additive quantities - total time is the sum of individual times
- We want to estimate the expected value of the execution time
- The arithmetic mean minimizes squared error for repeated measurements
- It properly represents the total time divided by the number of measurements
- For computing rates (bandwidth, FLOPS), we first compute the mean time, then calculate the rate

The arithmetic mean is calculated as:

$$\bar{t} = \frac{1}{n} \sum_{i=1}^n t_i$$

where n is the number of measurements in the second half, and t_i is each individual measurement.

Q2: Roofline Model Analysis



Data summary (all benchmarks):

Benchmark	N	GFLOP/s	BW (GB/s)
C1 dp1	1M	2.126	8.505
C1 dp1	300M	2.103	8.413
C2 dp2	1M	7.351	29.403
C2 dp2	300M	4.923	19.694
C3 dp3 (MKL)	1M	51.76	207.038
C3 dp3 (MKL)	300M	16.985	67.94
C4 dp4 (PyLoop)	1M	0.01	0.041
C4 dp4 (PyLoop)	300M	0.01	0.041
C5 dp5 (NumPy)	1M	17.137	68.548
C5 dp5 (NumPy)	300M	6.744	26.977

Arithmetic Intensity (AI):

For all dot-product benchmarks,

$AI = 2N \text{ FLOPs} / (2N \times 4 \text{ bytes}) = 0.25 \text{ FLOP/byte}$ — the same for every configuration. This places all points on the same vertical line, well to the left of the ridge point ($200/30 \approx 6.67$ FLOP/byte).

Roofline ceiling at $AI = 0.25$: $\min(30 \times 0.25, 200) = 7.5 \text{ GFLOP/s}$

Compute-bound vs. Memory-bound:

All benchmarks are memory-bound in theory ($AI \ll$ ridge point). The bottleneck is the rate at which data can be fetched from memory, not the rate at which the FPU can compute.

Analysis per benchmark:

- C1 (dp1): ~2.1 GFLOP/s at both N — achieves only 28% of the roofline ceiling. The scalar reduction loop $R += pA[j]*pB[j]$ creates a loop-carried dependency on R, limiting the compiler's ability to fully vectorize (even with -O3). Underperforms the roofline significantly.
- C2 (dp2): 7.35 GFLOP/s at N=1M (98% of roofline), 4.92 GFLOP/s at N=300M. Loop unrolling breaks the single dependency chain into 4 independent partial sums, enabling

better SIMD vectorization and pipelining. N=1M fits better in cache; N=300M drops as DRAM pressure increases.

- C3 (dp3/MKL): 51.76 GFLOP/s at N=1M, 16.99 GFLOP/s at N=300M — both *exceed* the stated 30 GB/s roofline. For N=1M (8 MB of data), the arrays partially reside in L3 cache, whose bandwidth far exceeds DRAM (>200 GB/s observed). For N=300M (2.4 GB, DRAM-resident), MKL's hand-tuned AVX-512 SIMD and software prefetching achieve 67.9 GB/s — exceeding our assumed 30 GB/s peak because the machine's actual DRAM bandwidth is higher.
- C4 (dp4/PyLoop): ~0.010 GFLOP/s at both sizes — ~200× slower than C1. This is NOT memory-bound; it is interpreter-bound. Each A[j] access in CPython involves dictionary lookup, reference counting, dynamic type checking, and boxing/unboxing of Python objects. The bandwidth of 0.041 GB/s is orders of magnitude below DRAM bandwidth.
- C5 (dp5/NumPy): 17.14 GFLOP/s at N=1M (cache-resident, exceeds roofline), 6.74 GFLOP/s at N=300M (90% of roofline). NumPy dispatches to MKL's optimized sdot internally; the Python-level call overhead is negligible.

Underperforming benchmarks: C1 and C4 fall well below the roofline ceiling. C1 underperforms due to the scalar dependency chain preventing full vectorization. C4 underperforms catastrophically due to Python interpreter overhead — it never reaches the memory system at all.

Q3: Performance Comparison

Baseline: C1 N=300M = 2.103 GFLOP/s

Explanation:

Benchmark	GFLOP/s (N=300M)	Relative to baseline
C1 — C simple loop	2.103	1.0×
C2 — C unrolled	4.923	2.34× faster
C3 — MKL BLAS	16.985	8.08× faster
C4 — Python loop	0.01	~205× slower
C5 — numpy.dot	6.744	3.21× faster

- C2 vs C1 ($2.34\times$): Manual 4× unrolling introduces 4 independent multiply-accumulate chains per iteration. This breaks the loop-carried dependency on the single accumulator R, allowing the compiler to issue multiple FMA instructions per cycle and make better use of SIMD registers. The pipeline is better utilized.
- C3 vs C1 ($8.08\times$): Intel MKL's `cblas_sdot` is hand-tuned assembly using AVX-512 (16 floats/instruction), multi-level loop unrolling, and hardware-specific prefetch instructions. It fully saturates memory bandwidth. A general-purpose compiler with `-O3` cannot match this level of micro-architectural optimization.
- C4 vs C1 ($\sim 205\times$ slower): CPython is an interpreted language. Each loop iteration involves: resolving variable names in dictionaries, dispatching `__getitem__` on the NumPy array, type-checking, reference-count updates, and boxing the result into a Python float object. The overhead per iteration is $\sim 100\text{--}200$ ns, compared to ~ 1 ns for a compiled C iteration. The computation is not memory-limited — it is interpreter-dispatch-limited.
- C5 vs C1 ($3.21\times$): `numpy.dot` calls into optimized C/Fortran routines (backed by MKL on Intel systems) with just a single Python function call overhead. For large N it is faster than C1 and C2 because it uses SIMD internally, though slightly below MKL directly (`dp3`) due to Python API overhead and dtype dispatch.

Q4: Floating Point Accuracy

Expected result: For all-ones float32 arrays of length N, the analytically exact dot product is N (sum of N ones).

- $N = 1,000,000 \rightarrow$ expected: 1,000,000.0
- $N = 300,000,000 \rightarrow$ expected: 300,000,000.0

Actual results:

Float32 (single precision) has a 23-bit mantissa, giving ~ 7.2 significant decimal digits ($\sim 2^{23} \approx 8.4 \times 10^6$ precision).

When the running sum $R \approx 10^k$ and we add 1.0, the addition is represented correctly only if $1.0 \geq R \times 2^{-23}$. Once $R > 2^{23} \approx 8.4 \times 10^6$, adding 1.0 has no effect — the result rounds back to R. This is called catastrophic cancellation / absorption.

- For $N=1M$: R grows to $\sim 10^6 < 2^{23}$, so most additions are representable but with increasing rounding error. The result will be approximately correct but not exact.
- For $N=300M$: R saturates around $2^{23} \approx 8.4M$ long before $N=300M$ is reached. Adding 1.0 to $R \approx 8.4M$ returns R unchanged. The result is capped near $2^{23} \approx 16,777,216$ — far less than 300,000,000.