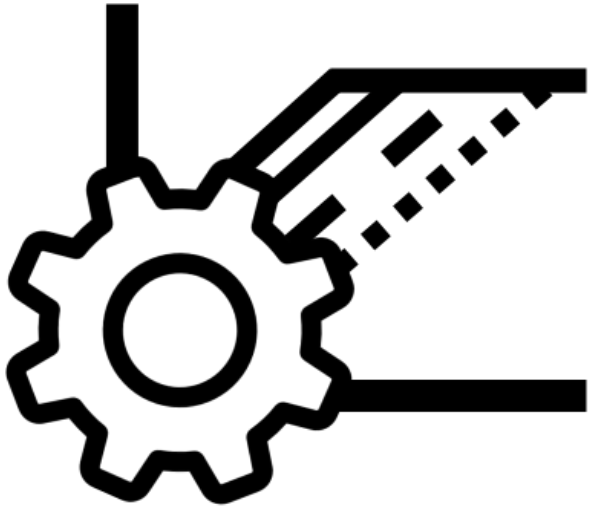
The CARM Tool

https://github.com/champ-hub/carm-roofline

The CARM Tool

Features:

- Microbenchmark generation
  - x86_64, AARCH64, RISCV64
  - ISA extensions (AVX512, Neon, RVV, etc.)
- Application profiling
  - PAPI, DBI (Dynamic Binary Instrumentation)
- Plotting
  - Web-based interface



Matrix Multiplication

Application:

- DGEMM: C = A · B
- Single-thread
- Double-precision (8 bytes)
- Square N=2048 matrices

Architecture:

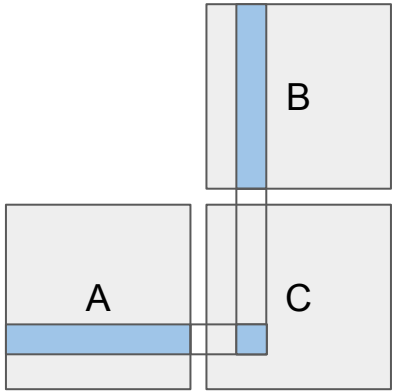
- MareNostrum 5
- Xeon Max 9480
- Caches:
  - L1 – 48 KiB
  - L2 – 2 MiB
  - L3 – **225 MiB**

2048^2	Elements
8	Bytes
× 3	Matrices
96	**MiB**

Optimizing Matrix Multiplication

Naive GEMM implementation

- Every iteration:
  - 2 loads, 1 store (24 bytes)
  - 2 operations (mul, add)

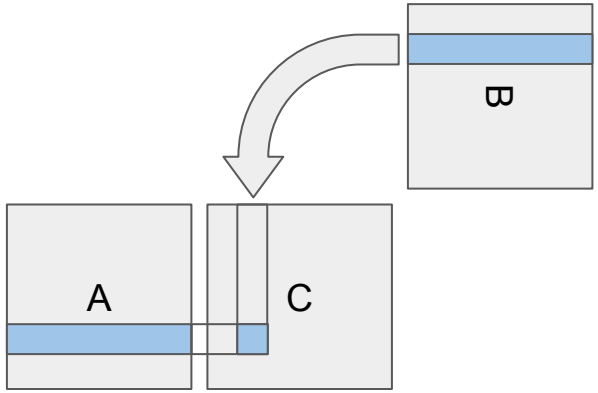



```
1  void gemm_naive(double *A, double *B, double *C)
2  {
3      for (int i = 0; i < N; i++) {
4          for (int j = 0; j < N; j++) {
5              for (int k = 0; k < N; k++) {
6                  C[i * N + j] += A[i * N + k] * B[k * N + j];
7              }
8          }
9      }
10 }
```

Optimizing Matrix Multiplication

Transposing the B matrix

- Column-major order in B
  - Contiguous memory accesses
  - Better locality

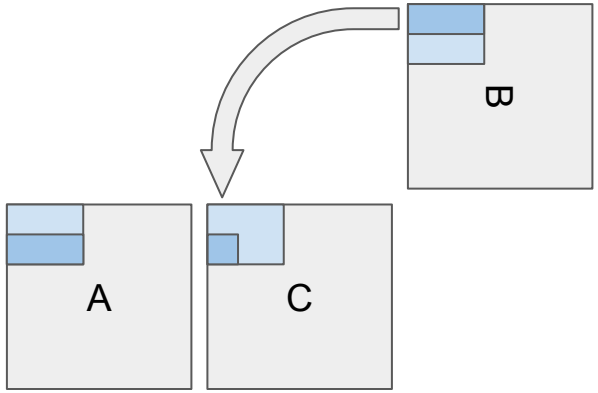



```
1  void gemm_transposed(double *A, double *B_T, double *C)
2  {
3      for (int i = 0; i < N; i++) {
4          for (int j = 0; j < N; j++) {
5              for (int k = 0; k < N; k++) {
6                  C[i * N + j] += A[i * N + k] * B_T[j * N + k];
7              }
8          }
9      }
10 }
```

Optimizing Matrix Multiplication

Matrix blocking

- Better cache locality
  - Reuse data in higher level caches
  - 32-element block – 24 KiB < L1
  - Higher effective memory bandwidth




```
1  void gemm_blocked(double *A, double *B, double *C)
2  {
3      for (int i0 = 0; i0 < N; i0 += BLOCK_SIZE) {
4          for (int j0 = 0; j0 < N; j0 += BLOCK_SIZE) {
5              for (int k0 = 0; k0 < N; k0 += BLOCK_SIZE) {
6                  for (int i = i0; i < i0 + BLOCK_SIZE; i++) {
7                      for (int j = j0; j < j0 + BLOCK_SIZE; j++) {
8                          for (int k = k0; k < k0 + BLOCK_SIZE; k++) {
9                              C[i * N + j] += A[i * N + k] * B[j * N + k];
10                         }
```

Optimizing Matrix Multiplication

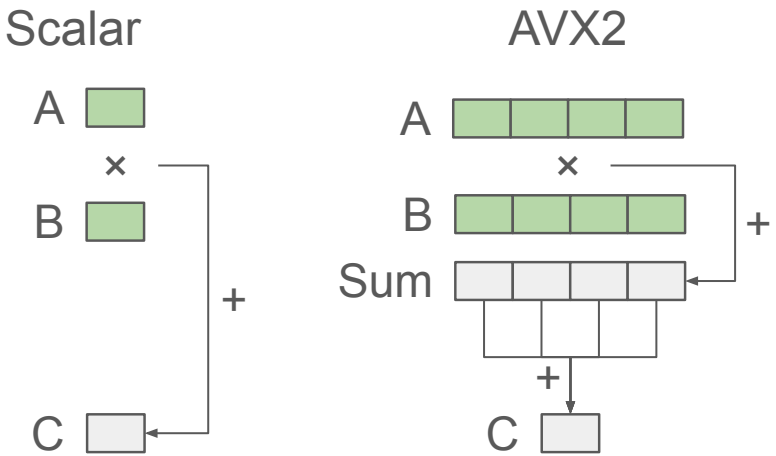
restrict keyword

- Tells the compiler the pointers don't alias
  - "No other pointer will access this pointer's data"
  - Avoids redundant loads (register reuse)

```
1  void gemm_blocked_restrict(double *restrict A, double *restrict B, double *restrict C)
2  {
3      for (int i0 = 0; i0 < N; i0 += BLOCK_SIZE) {
4          for (int j0 = 0; j0 < N; j0 += BLOCK_SIZE) {
5              for (int k0 = 0; k0 < N; k0 += BLOCK_SIZE) {
6                  for (int i = i0; i < i0 + BLOCK_SIZE; i++) {
7                      for (int j = j0; j < j0 + BLOCK_SIZE; j++) {
8                          for (int k = k0; k < k0 + BLOCK_SIZE; k++) {
9                              C[i * N + j] += A[i * N + k] * B[j * N + k];
10                         }
```

Optimizing Matrix Multiplication

- AVX2 vectorization (256-bit)
  - Load 4-element vectors of A and B
  - Vector fused multiply-add
  - Horizontal reduction (add the 4 elements)
  - Store in C




```
1  // abridged
2  void gemm_blocked_restrict_avx2(double *restrict A, ...)
3  {
4      // 3 block loops ...
5      // inner i,j loops ...
6      __m256d vsum = _mm256_setzero_pd();
7
8      int k = k0;
9      for (; k <= k0 + BLOCK_SIZE - 4; k += 4) {
10         __m256d va = _mm256_loadu_pd(&A[i * N + k]);
11         __m256d vb = _mm256_loadu_pd(&B[j * N + k]);
12
13         vsum = _mm256_fmadd_pd(va, vb, vsum);
14     }
15
16     /* horizontal reduction */
17     __m128d low = _mm256_castpd256_pd128(vsum);
18     __m128d high = _mm256_extractf128_pd(vsum, 1);
19     __m128d sum2 = _mm_add_pd(low, high);
20     sum2 = _mm_hadd_pd(sum2, sum2);
21
22     double sum = _mm_cvtsd_f64(sum2);
23
24     C[i * N + j] += sum;
25 }
```

Optimizing Matrix Multiplication

- OpenBLAS call
  - Register blocking / 16x4 microkernel
  - **AVX-512** assembly
  - Inner-loop unrolling
  - Data-prefetching

```
1  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, N, N, N, 1.0, A, N, B_T, N, 0.0, C, N);
```

Optimizing Matrix Multiplication – Summary

- Memory-bound?
  - Improve locality (↑**Bandwidth**↑)
  - Reduce redundant loads (↑**AI**↑)

- Blocking and transposition
  - Leverage higher caches
  - ↑**Bandwidth**↑ 10×
- Register reuse
  - Memory-bound ➔ Compute-bound
  - ↑**AI**↑ 5.5×

- Compute-bound?
  - Vectorize (↑**Peak-performance**↑)

- AVX2 / AVX-512 Vectorization
  - ↑**Peak-performance**↑ 8×

166× Speedup

STREAM

Memory bandwidth benchmark

Copy

`C[i] = A[i]`

16 bytes, 0 ops

AI = 0

Scale

`C[i] = scalar * A[i]`

16 bytes, 1 operation

AI = 0.0625

Add

`C[i] = A[i] + B[i]`

24 bytes, 1 operation

AI = 0.0417

Triad

`C[i] = A[i] + scalar * B[i]`

24 bytes, 2 operations

AI = 0.0833

STREAM

Memory bandwidth benchmark

- 56 Threads
- 100 M elements
- 800 MB > 225 MiB L3

Scale

```
C[i] = scalar * A[i]
```

16 bytes, 1 operation

AI = 0.0625

Add

```
C[i] = A[i] + B[i]
```

24 bytes, 1 operation

AI = 0.0417

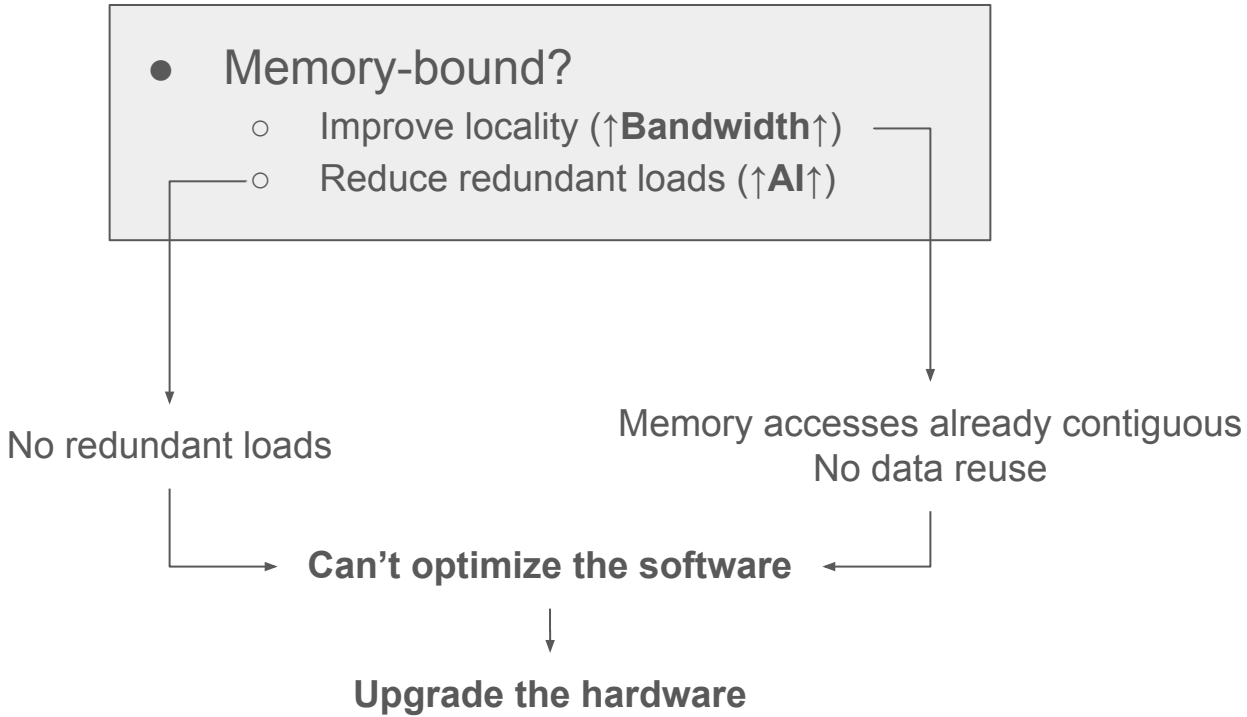
Triad

```
C[i] = A[i] + scalar * B[i]
```

24 bytes, 2 operations

AI = 0.0833

Optimizing STREAM



The CARM Tool

Software

Easily identify the bottleneck

Profiling + CARM
=
Powerful optimization hints

Hardware

Easily visualize the performance

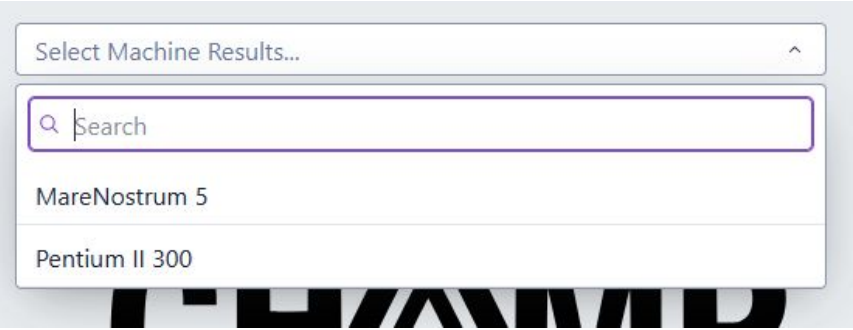
Application AI + CARM
=
Fast performance prediction

Plotting

From the CLI, open the web interface:

```
python ResultsGUI.py
```

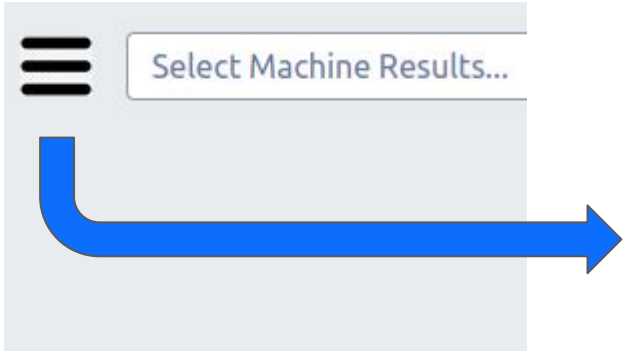
- Outputs a link to open in a browser



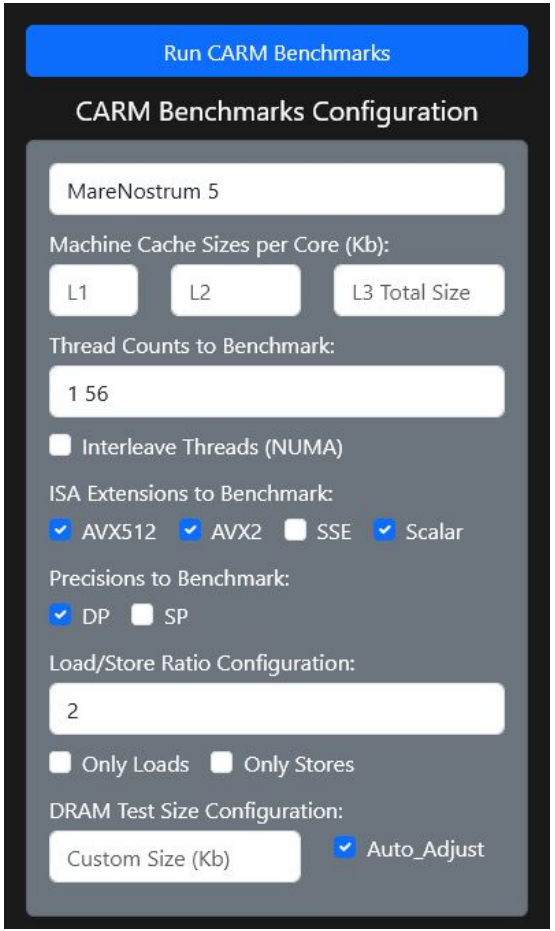
Benchmarking

From the CLI:

```
python run.py
    --isa ...
    --threads ...
    --inst ...
```

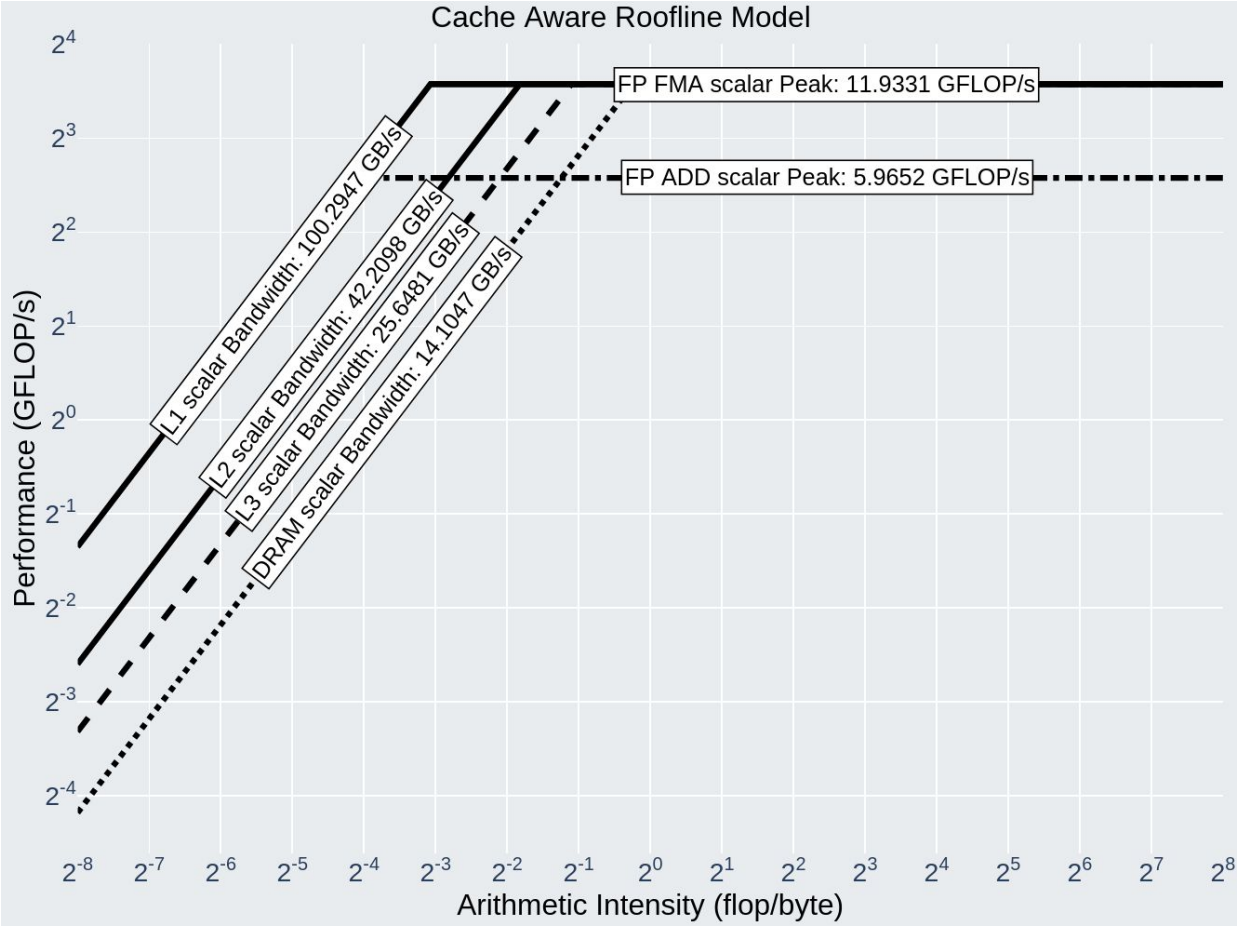


From the web interface



- Automatically detects cache size, frequency, etc.
- Generates and runs:
  - 4 Memory microbenchmarks (L1, L2, L3, DRAM)
  - 2 Arithmetic microbenchmark (add, fused multiply-add)
- Stores the results ready for plotting

MareNostrum 5 – Scalar, single-threaded roof

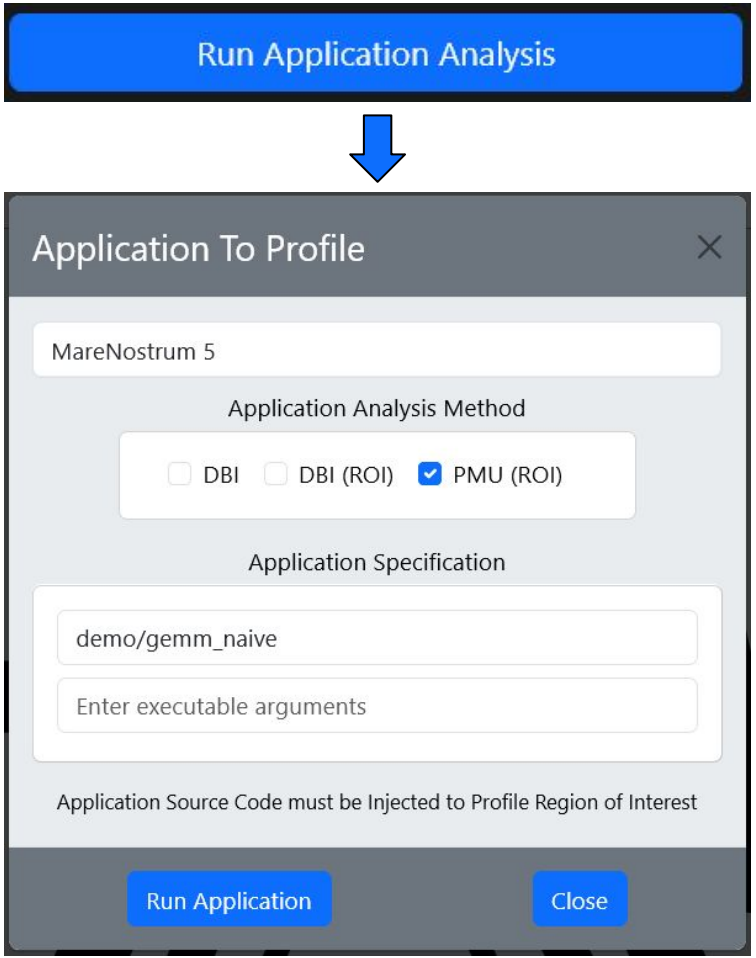


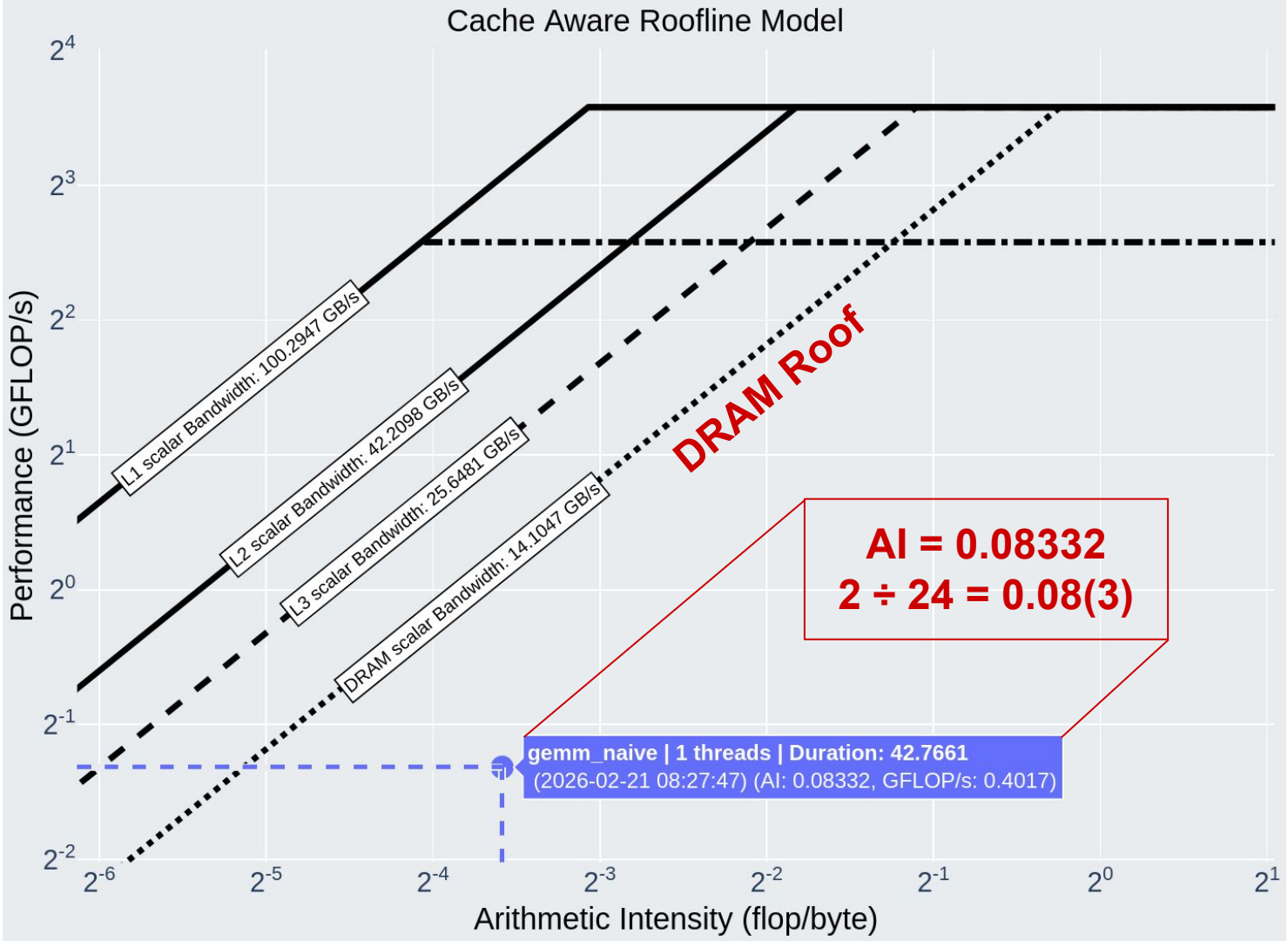
Profiling

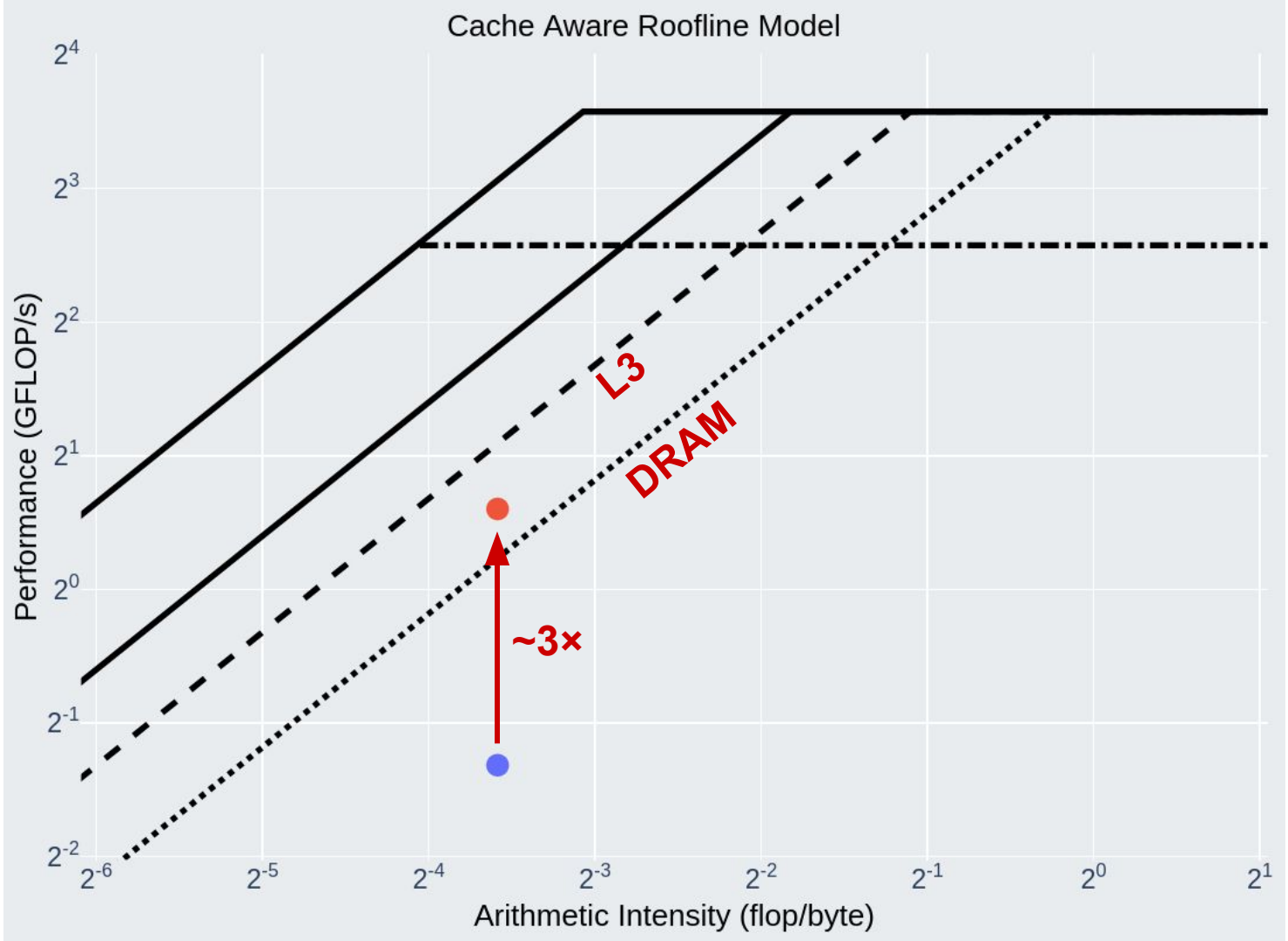
From the CLI:

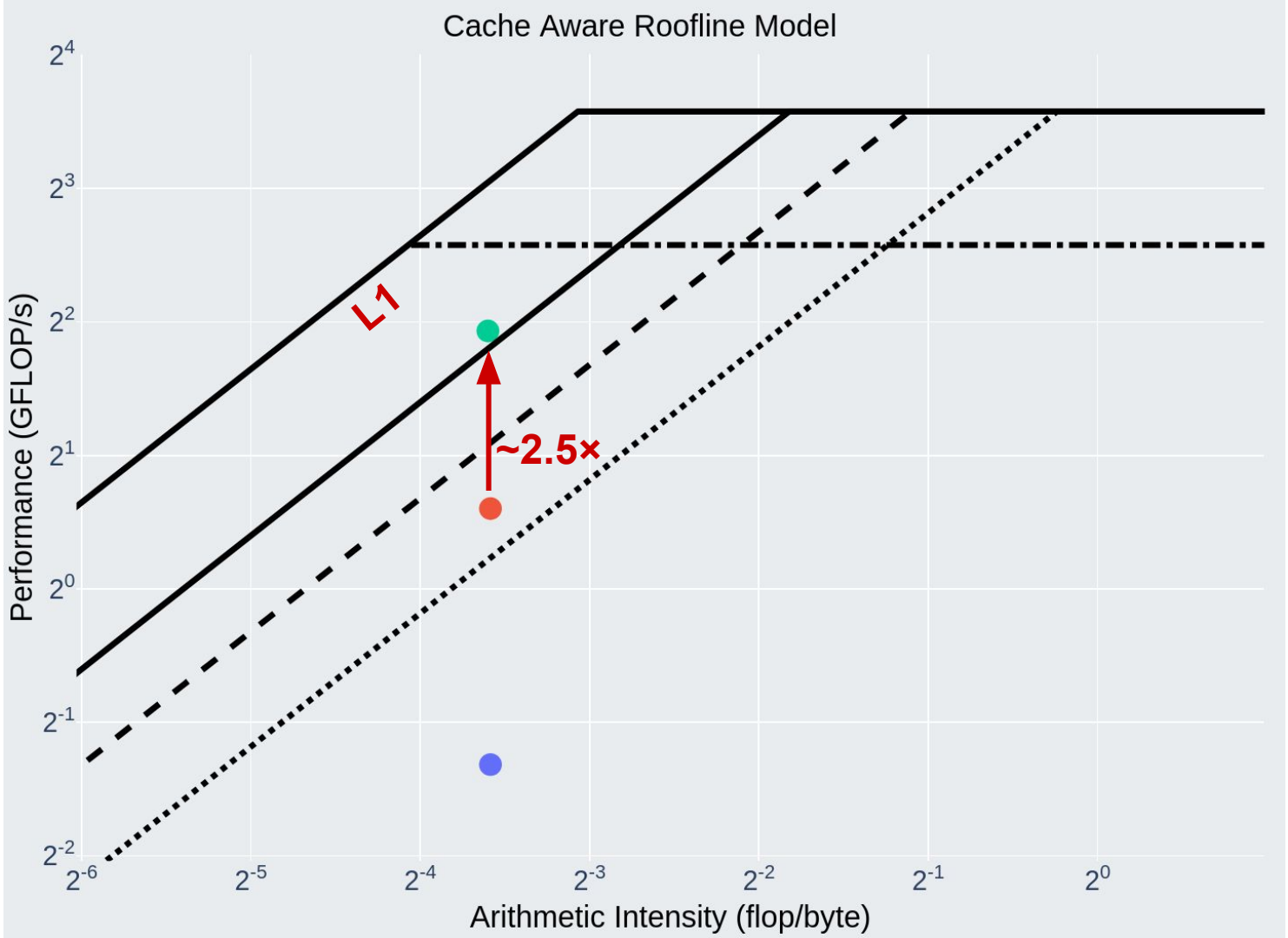
```
python PMU_AI_Calculator.py <binary>
```

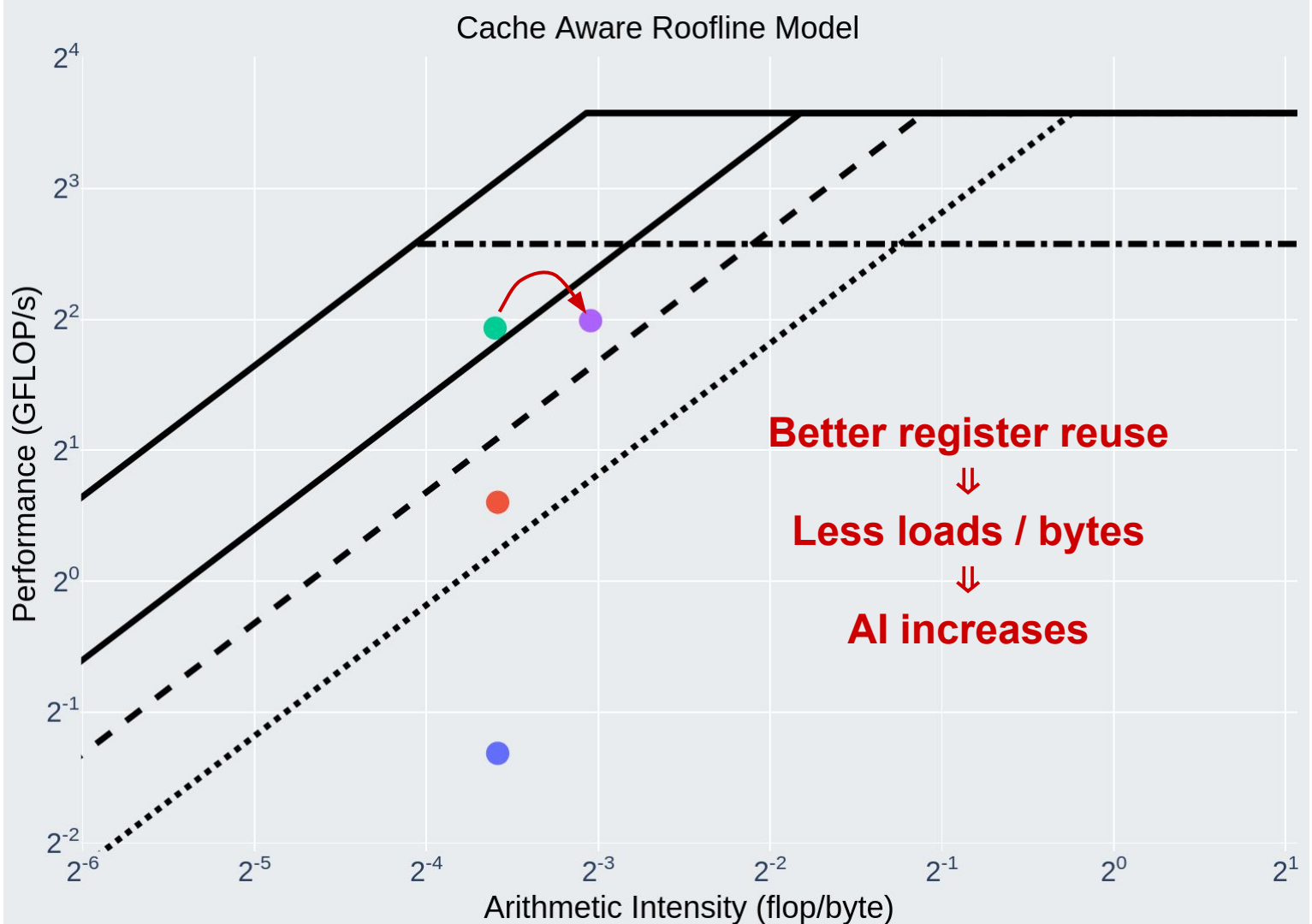
- Captures PAPI output
  - Application must be annotated with PAPI *region of interest* calls
- Processes and stores result for plotting




Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
L1 scalar Bandwidth: 100.2947 GB/s
L2 scalar Bandwidth: 42.2098 GB/s
L3 scalar Bandwidth: 25.6481 GB/s
DRAM scalar Bandwidth: 14.1047 GB/s
DRAM Roof
AI = 0.08332
2 ÷ 24 = 0.08(3)
gemm_naive | 1 threads | Duration: 42.7661
(2026-02-21 08:27:47) (AI: 0.08332, GFLOP/s: 0.4017)

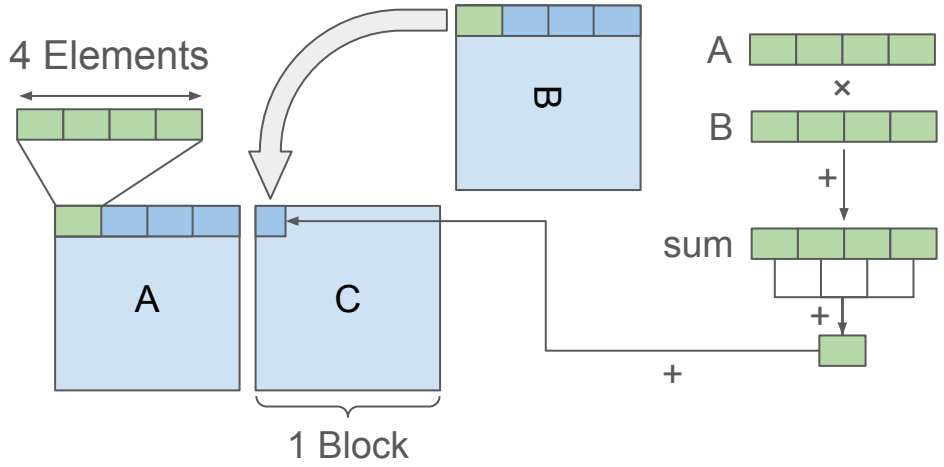

Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
L3
DRAM
~3×


Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
L1
~2.5×
2^4
2^3
2^2
2^1
2^0
2^-1
2^-2
2^-6
2^-5
2^-4
2^-3
2^-2
2^-1
2^0

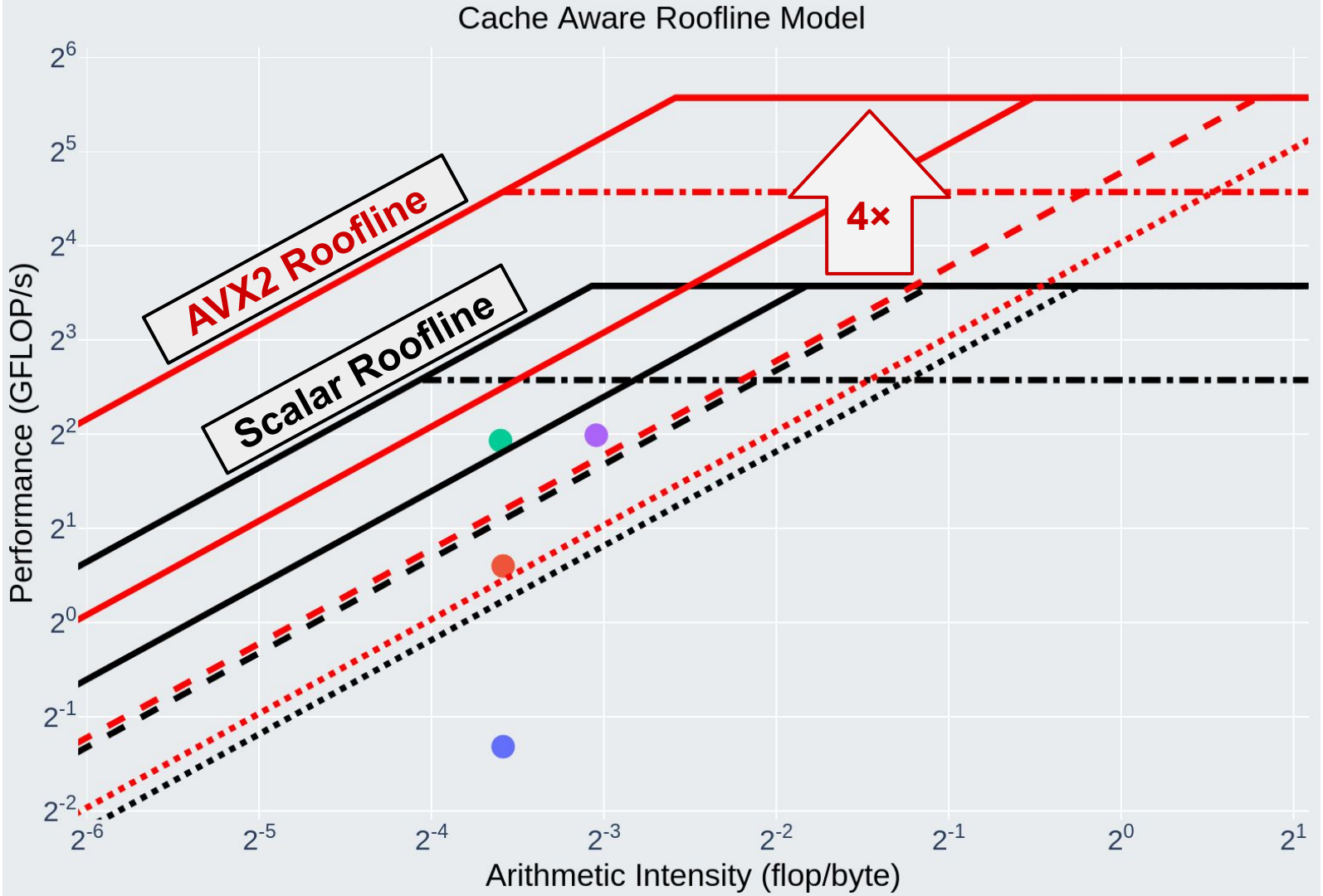

Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
2^{4}
2^{3}
2^{2}
2^{1}
2^{0}
2^{-1}
2^{-2}
2^{-6}
2^{-5}
2^{-4}
2^{-3}
2^{-2}
2^{-1}
2^{0}
2^{1}
Better register reuse
⇓
Less loads / bytes
⇓
AI increases

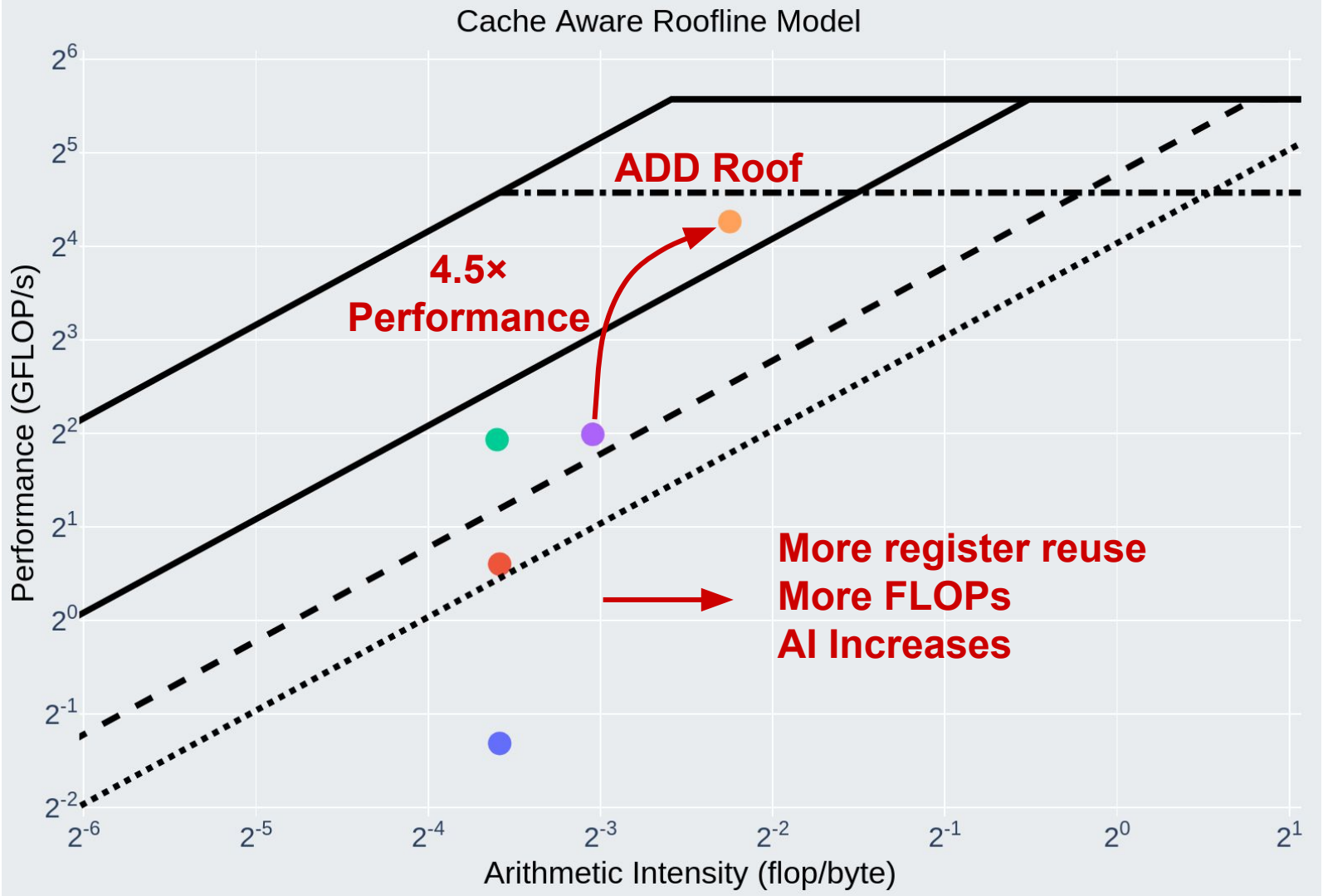
Optimizing Matrix Multiplication

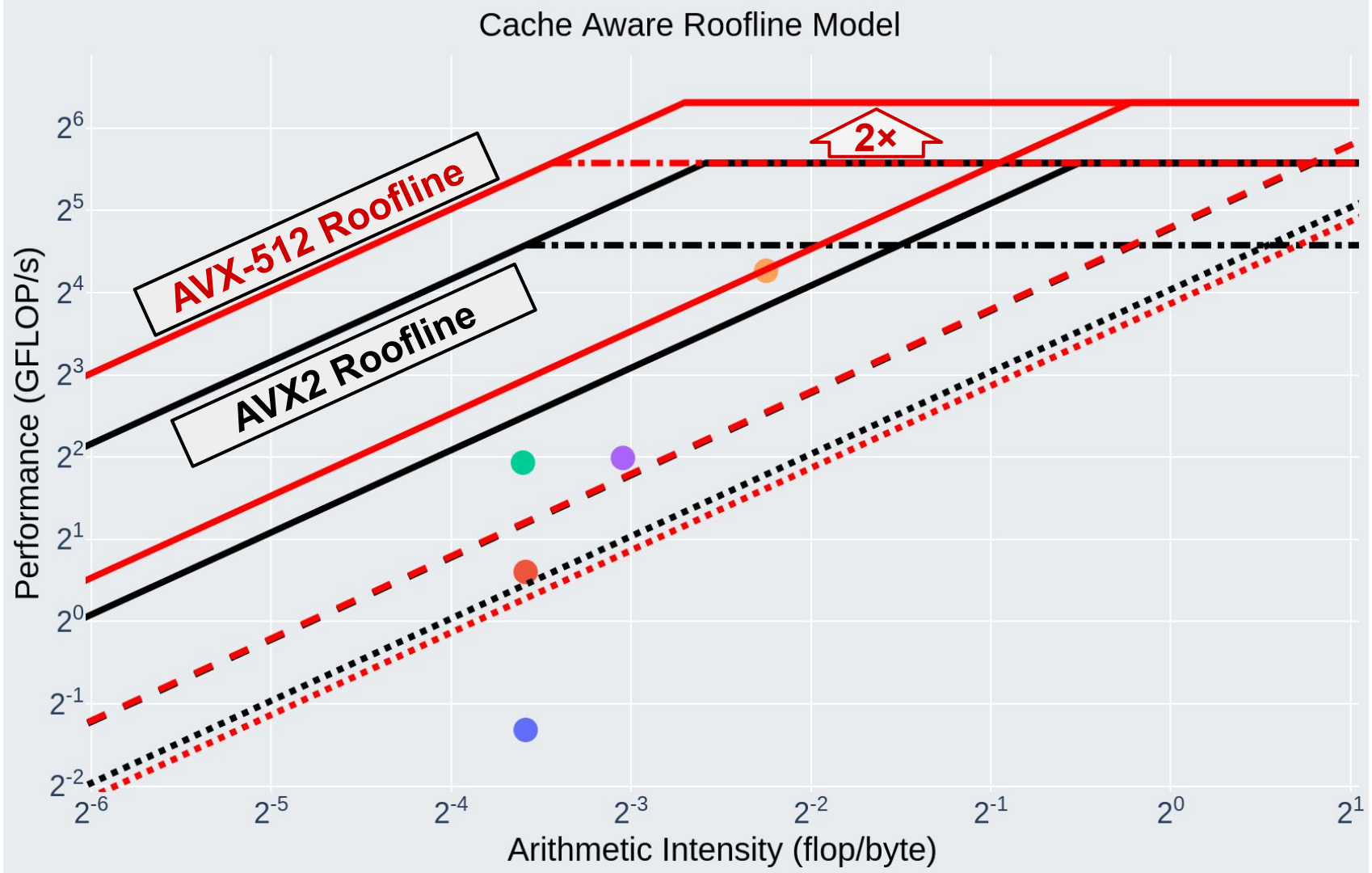
- AVX2 vectorization (256-bit)
  - Load 4-element vectors of A and B
  - Vector fused multiply-add
  - Horizontal reduction (add the 4 elements)
  - Store in C

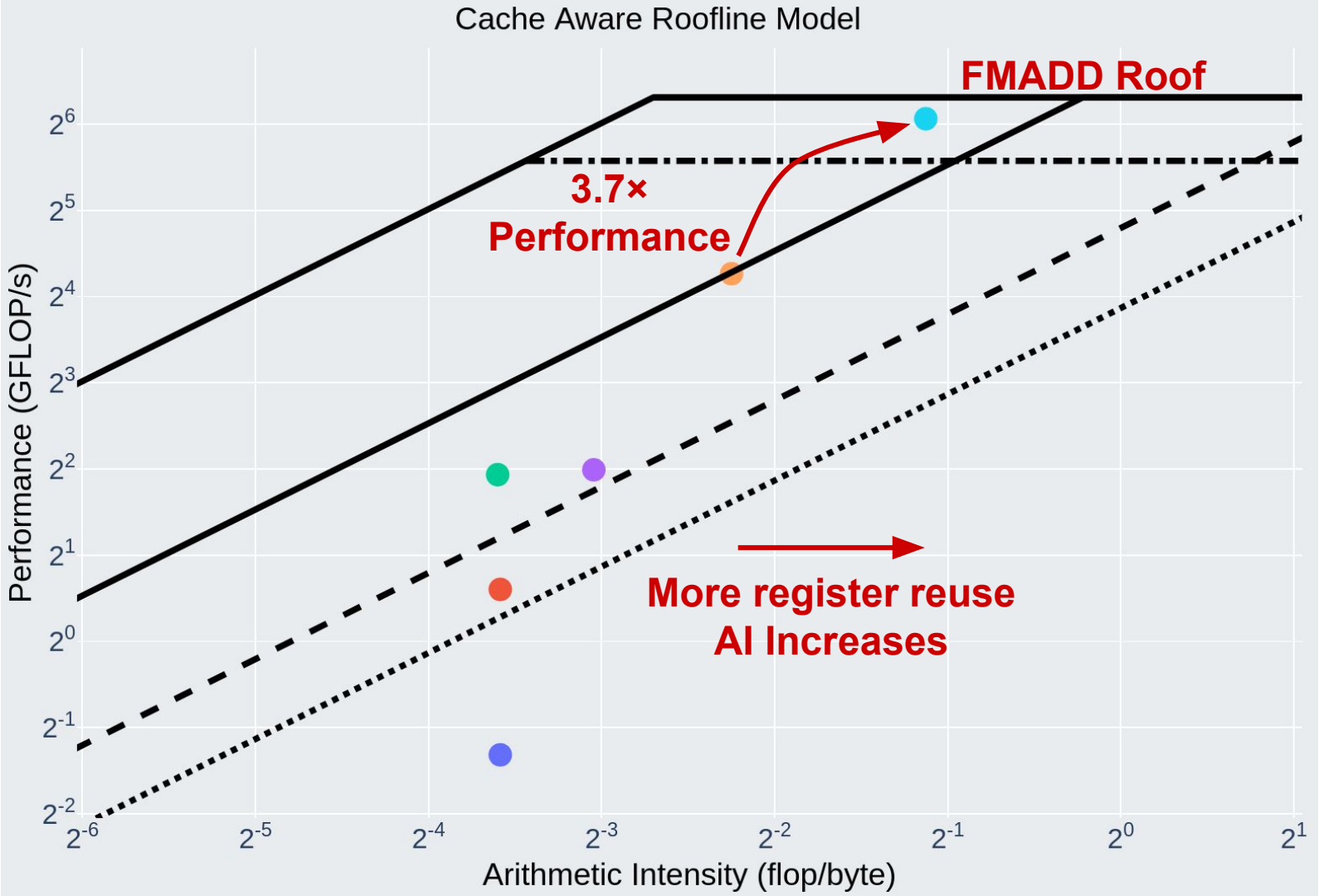


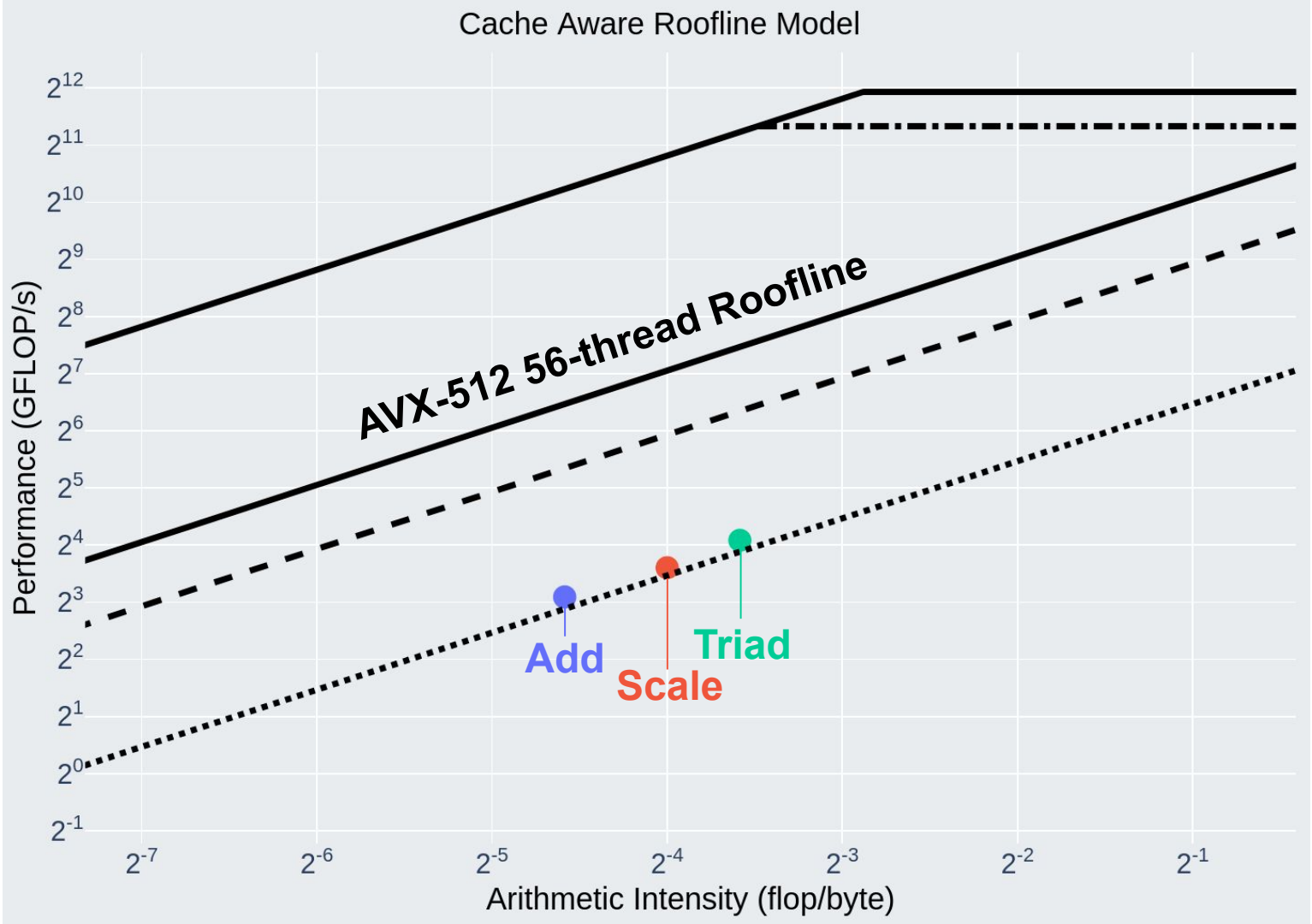

```
1  // abridged
2  void gemm_blocked_restrict_avx2(double *restrict A, ...)
3  {
4      // 3 block loops ...
5      // inner i,j loops ...
6      __m256d vsum = _mm256_setzero_pd();
7
8      int k = k0;
9      for (; k <= k0 + BLOCK_SIZE - 4; k += 4) {
10         __m256d va = _mm256_loadu_pd(&A[i * N + k]);
11         __m256d vb = _mm256_loadu_pd(&B[j * N + k]);
12
13         vsum = _mm256_fmadd_pd(va, vb, vsum);
14     }
15
16     /* horizontal reduction */
17     __m128d low = _mm256_castpd256_pd128(vsum);
18     __m128d high = _mm256_extractf128_pd(vsum, 1);
19     __m128d sum2 = _mm_add_pd(low, high);
20     sum2 = _mm_hadd_pd(sum2, sum2);
21
22     double sum = _mm_cvtsd_f64(sum2);
23
24     C[i * N + j] += sum;
25 }
```

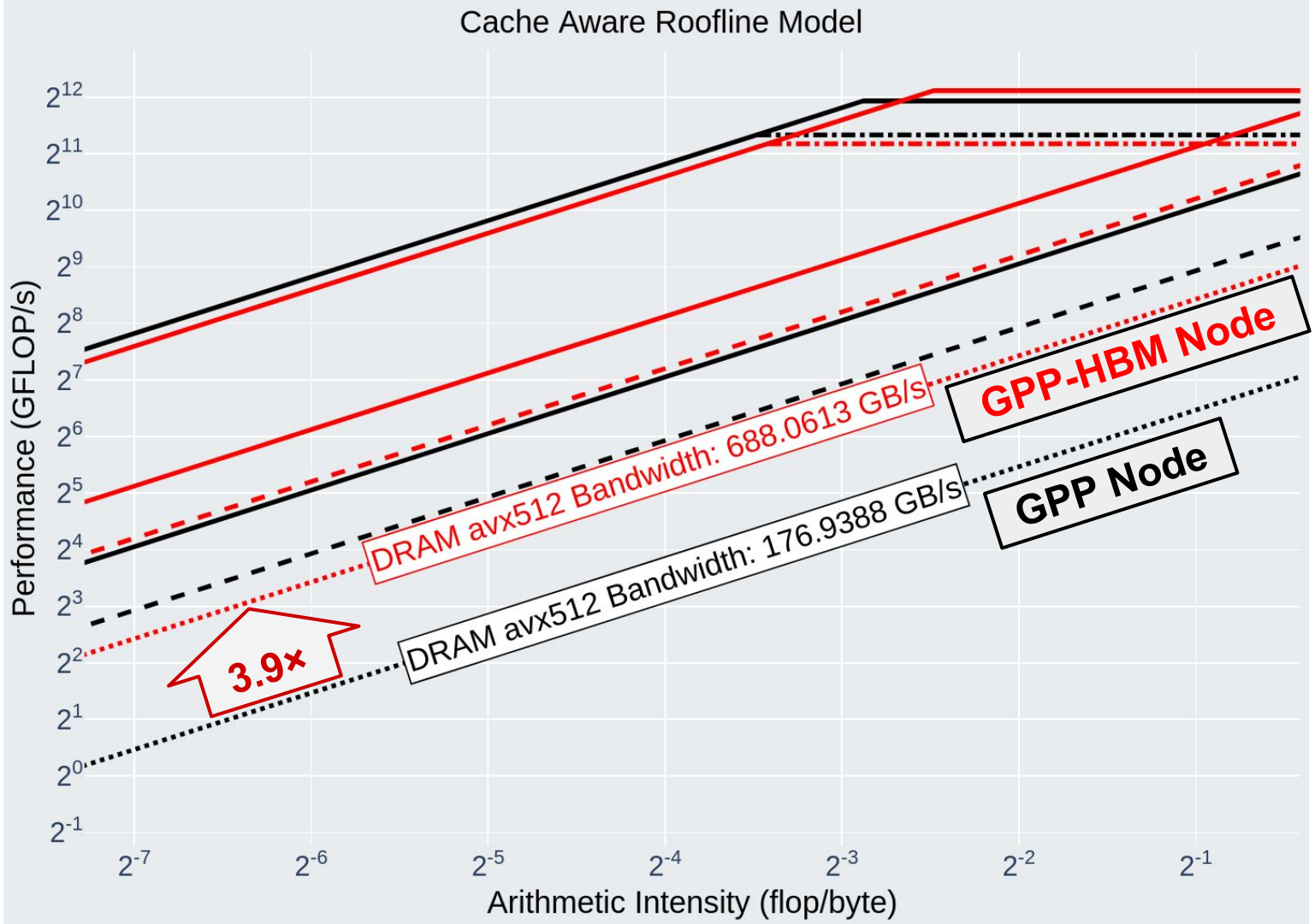

Cache Aware Roofline Model
AVX2 Roofline
Scalar Roofline
4×
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
2^{6}
2^{5}
2^{4}
2^{3}
2^{2}
2^{1}
2^{0}
2^{-1}
2^{-2}
2^{-6}
2^{-5}
2^{-4}
2^{-3}
2^{-2}
2^{-1}
2^{0}
2^{1}

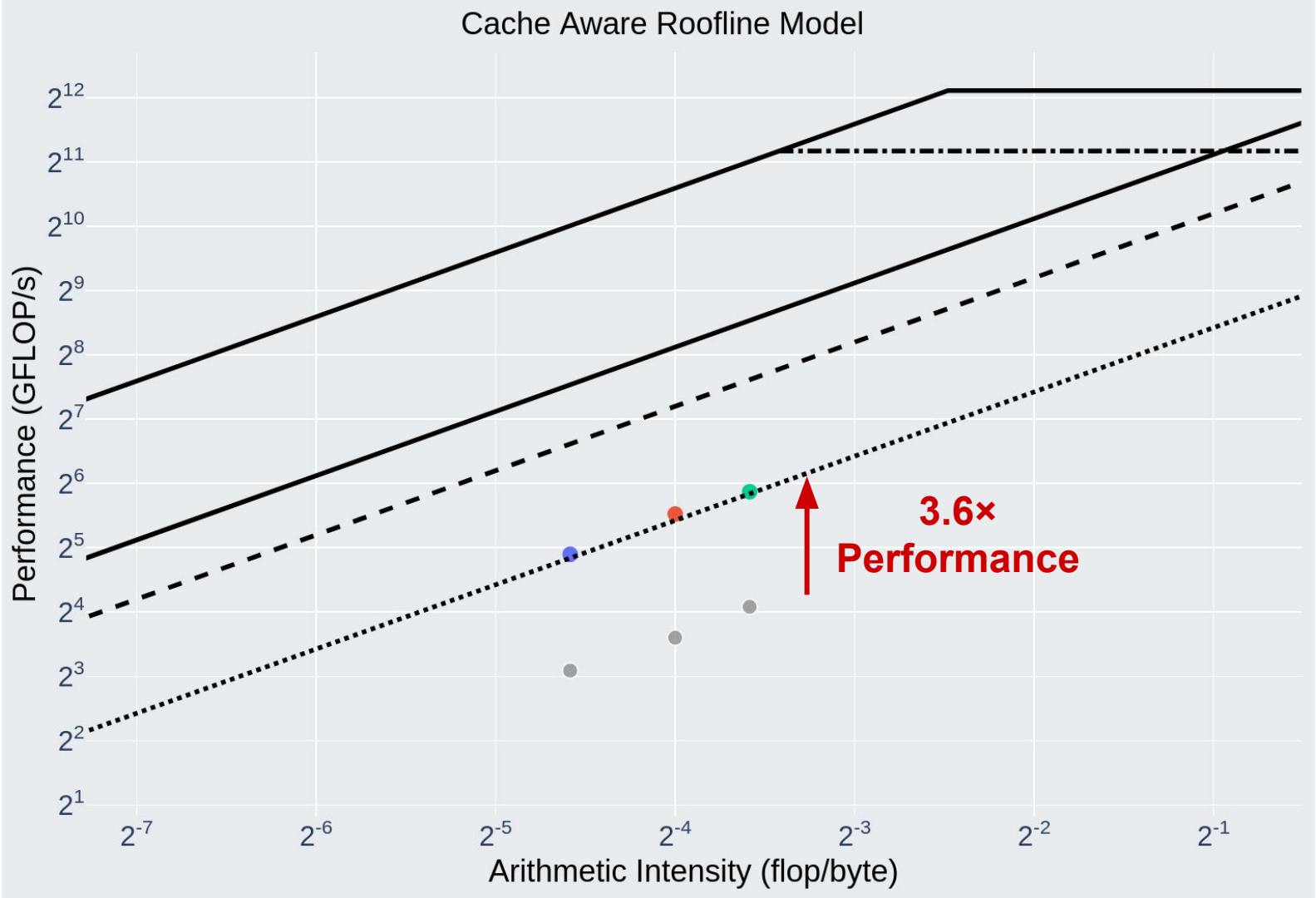

Cache Aware Roofline Model
ADD Roof
4.5×
Performance
More register reuse
More FLOPs
AI Increases
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
2^6
2^5
2^4
2^3
2^2
2^1
2^0
2^-1
2^-2
2^-6
2^-5
2^-4
2^-3
2^-2
2^-1
2^0
2^1


Cache Aware Roofline Model
AVX-512 Roofline
AVX2 Roofline
2×
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
2^6
2^5
2^4
2^3
2^2
2^1
2^0
2^-1
2^-2
2^-6
2^-5
2^-4
2^-3
2^-2
2^-1
2^0
2^1


Cache Aware Roofline Model
FMADD Roof
3.7×
Performance
More register reuse
AI Increases
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
2^{6}
2^{5}
2^{4}
2^{3}
2^{2}
2^{1}
2^{0}
2^{-1}
2^{-2}
2^{-6}
2^{-5}
2^{-4}
2^{-3}
2^{-2}
2^{-1}
2^{0}
2^{1}


Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
AVX-512 56-thread Roofline
Add
Scale
Triad


Cache Aware Roofline Model
Performance (GFLOP/s)
Arithmetic Intensity (flop/byte)
DRAM avx512 Bandwidth: 688.0613 GB/s
DRAM avx512 Bandwidth: 176.9388 GB/s
GPP-HBM Node
GPP Node
3.9×


Cache Aware Roofline Model
Performance (GFLOP/s)
2^{12}
2^{11}
2^{10}
2^{9}
2^{8}
2^{7}
2^{6}
2^{5}
2^{4}
2^{3}
2^{2}
2^{1}
2^{-7}
2^{-6}
2^{-5}
2^{-4}
2^{-3}
2^{-2}
2^{-1}
Arithmetic Intensity (flop/byte)
3.6×
Performance