Lecture 10. The Roofline Model

Introductions to Data Systems and Data Design

Ce Zhang

Introduction

The Fundamental Question

When optimizing code, we need to know:

Is my code limited by compute or by memory bandwidth?

The Fundamental Question

When optimizing code, we need to know:

Is my code limited by compute or by memory bandwidth?

- If **compute bound**: Optimize arithmetic (SIMD, ILP, better algorithms)
- If **memory bound**: Optimize data movement (blocking, prefetching, data layout)

The Fundamental Question

When optimizing code, we need to know:

Is my code limited by compute or by memory bandwidth?

- If **compute bound**: Optimize arithmetic (SIMD, ILP, better algorithms)
- If **memory bound**: Optimize data movement (blocking, prefetching, data layout)

The **Roofline Model** answers this question visually and quantitatively.

The Two Ceilings

Every program faces two fundamental limits:

Compute Ceiling

Peak FLOPS of the processor

π flops/cycle

Memory Ceiling

Peak memory bandwidth

β bytes/cycle

Which ceiling limits you?

Your Program

Platform Parameters

Peak Performance π [flops/cycle or GFLOPS]:

$$\pi = \text{cores} \times \text{SIMD width} \times \text{FMA units} \times \text{ops/FMA}$$

Platform Parameters

Peak Performance π [flops/cycle or GFLOPS]:

$$\pi = \text{cores} \times \text{SIMD width} \times \text{FMA units} \times \text{ops/FMA}$$

Example: Intel Core i7 (Skylake), single core:

- 2 FMA units
- 4-way SIMD (AVX, doubles)
- 2 ops per FMA (multiply + add)

$$\pi = 1 \times 4 \times 2 \times 2 = 16 \text{ flops/cycle}$$

Platform Parameters

Peak Bandwidth β [bytes/cycle or GB/s]:

Theoretical bandwidth is determined by hardware specs:

$$\beta_{\text{theo}} = \text{channels} \times \text{data rate} \times \text{bus width}$$

Platform Parameters

Peak Bandwidth β [bytes/cycle or GB/s]:

Theoretical bandwidth is determined by hardware specs:

$$\beta_{\text{theo}} = \text{channels} \times \text{data rate} \times \text{bus width}$$

Example: DDR4-2400, dual channel:

- 2 channels $\times$ 2400 MT/s $\times$ 8 bytes $=$ 38.4 GB/s

Algorithm Parameters

Work $W(n)$ [flops]: Total floating-point operations

Data Movement $Q(n)$ [bytes]: Bytes transferred between cache and memory

Algorithm Parameters

Work $W(n)$ [flops]: Total floating-point operations

Data Movement $Q(n)$ [bytes]: Bytes transferred between cache and memory

Operational Intensity:

$$I(n) = \frac{W(n)}{Q(n)} \quad \text{[flops/byte]}$$

Algorithm Parameters

Work $W(n)$ [flops]: Total floating-point operations

Data Movement $Q(n)$ [bytes]: Bytes transferred between cache and memory

Operational Intensity:

$$I(n) = \frac{W(n)}{Q(n)} \quad \text{[flops/byte]}$$

Performance:

$$P(n) = \frac{W(n)}{T(n)} \quad \text{[flops/cycle]}$$

where $T(n)$ is runtime in cycles.

Deriving the Roofline

The Two Bounds

Bound 1: Compute Bound

$$P \leq \pi$$

Performance cannot exceed peak compute throughput.

The Two Bounds

Bound 1: Compute Bound

$$P \leq \pi$$

Performance cannot exceed peak compute throughput.

Bound 2: Memory Bound

$$\beta \geq \frac{Q}{T} = \frac{W/T}{W/Q} = \frac{P}{I}$$

Rearranging:

$$P \leq \beta \cdot I$$

Combining the Bounds

$$P \leq \min(\pi, \beta \cdot I)$$

Combining the Bounds

$$P \leq \min(\pi, \beta \cdot I)$$

Taking logarithms:

$$\log_2(P) \leq \min(\log_2(\pi), \log_2(\beta) + \log_2(I))$$

Combining the Bounds

$$P \leq \min(\pi, \beta \cdot I)$$

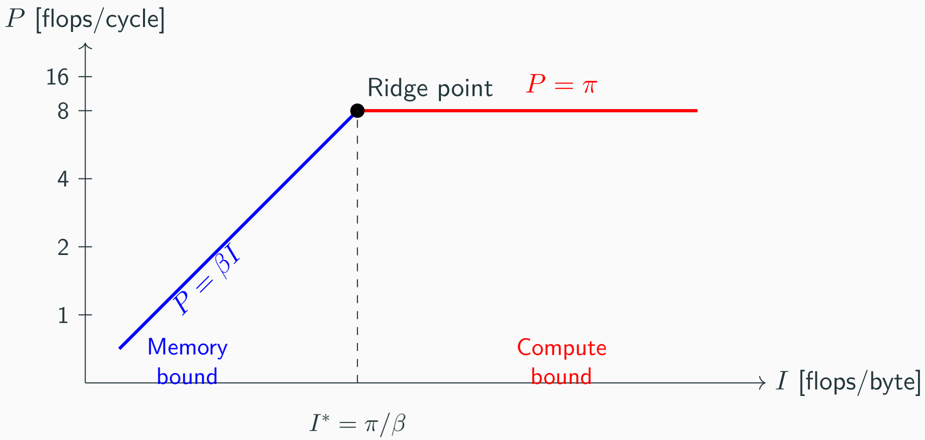
Taking logarithms:

$$\log_2(P) \leq \min(\log_2(\pi), \log_2(\beta) + \log_2(I))$$

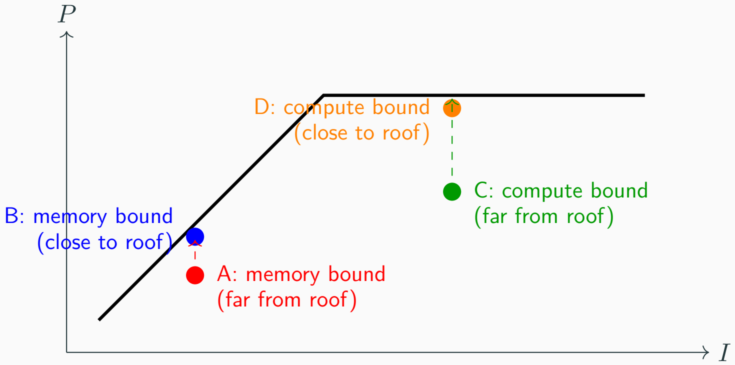
This is a **piecewise linear function** in log-log space:

- For low I: $P = \beta \cdot I$ (diagonal line, slope 1)
- For high I: $P = \pi$ (horizontal line)
- **Ridge point:** $I^* = \pi/\beta$

The Roofline Plot



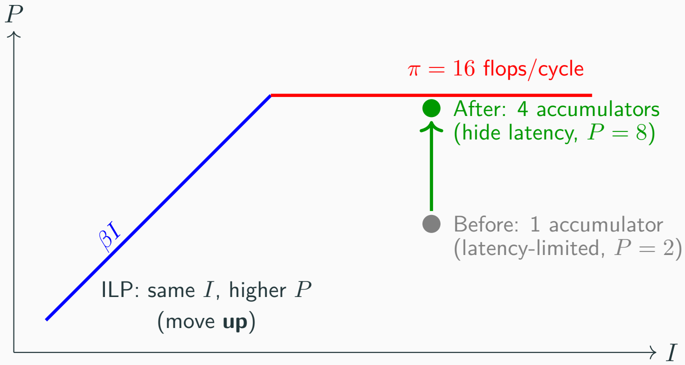
Reading the Roofline



Goal: Move points up toward the roof, or right to increase I.

Optimization 1: ILP (Instruction-Level Parallelism)

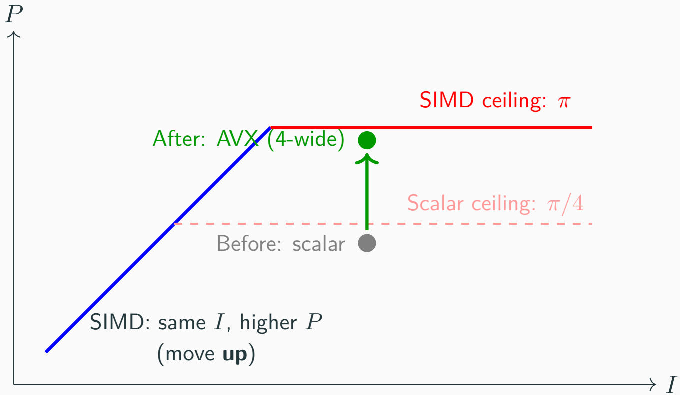
Scenario: Compute-bound code with a single accumulator (latency-limited).



ILP does not change operational intensity – same work, same data.

Optimization 2: SIMD (Vectorization)

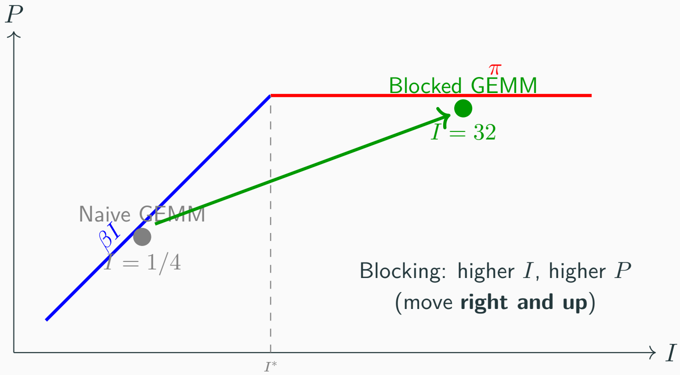
Scenario: Code that processes one element at a time.



SIMD does not change operational intensity – same work, same data.

Optimization 3: Blocking (Tiling)

Scenario: Matrix multiplication – reduce data movement via cache reuse.



Blocking reduces data movement Q, increasing operational intensity $I = W/Q$.

Computing Operational Intensity

Example 1: Vector Addition (DAXPY)

$$y = \alpha x + y$$

```cpp
// daxpy.cpp
for (int i = 0; i < n; i++)
    y[i] = alpha * x[i] + y[i];
```

```
$ cd examples && make run_daxpy
```

Example 1: Vector Addition (DAXPY)

$$y = \alpha x + y$$

```cpp
// daxpy.cpp
for (int i = 0; i < n; i++)
    y[i] = alpha * x[i] + y[i];
```

```
$ cd examples && make run_daxpy
```

Work: $W = 2n$ flops (1 multiply + 1 add per element)

Data movement (cold cache):

- Read x: $8n$ bytes
- Read y: $8n$ bytes
- Write y: $8n$ bytes

$Q = 24n$ bytes

Example 1: Vector Addition (DAXPY)

$$y = \alpha x + y$$

```
// daxpy.cpp
for (int i = 0; i < n; i++)
    y[i] = alpha * x[i] + y[i];
```

```
$ cd examples && make run_daxpy
```

Work: $W = 2n$ flops (1 multiply + 1 add per element)

Data movement (cold cache):

- Read x: $8n$ bytes
- Read y: $8n$ bytes
- Write y: $8n$ bytes

$Q = 24n$ bytes

Operational Intensity: $I = \frac{2n}{24n} = \frac{1}{12} \approx 0.083$ flops/byte

Example 2: Dot Product

$$s = x^T y = \sum_{i=0}^{n-1} x_i \cdot y_i$$

```cpp
// dot.cpp
double s = 0;
for (int i = 0; i < n; i++)
    s += x[i] * y[i];
```

```
$ cd examples && make run_dot
```

Example 2: Dot Product

$$s = x^T y = \sum_{i=0}^{n-1} x_i \cdot y_i$$

```
// dot.cpp
double s = 0;
for (int i = 0; i < n; i++)
    s += x[i] * y[i];
```

```
$ cd examples && make run_dot
```

Work: $W = 2n$ flops

Data movement:

- Read x: $8n$ bytes
- Read y: $8n$ bytes

$Q = 16n$ bytes

Example 2: Dot Product

$$s = x^T y = \sum_{i=0}^{n-1} x_i \cdot y_i$$

```cpp
// dot.cpp
double s = 0;
for (int i = 0; i < n; i++)
    s += x[i] * y[i];
```

```
$ cd examples && make run_dot
```

Work: $W = 2n$ flops

Data movement:

- Read x: $8n$ bytes
- Read y: $8n$ bytes

$Q = 16n$ bytes

Operational Intensity: $I = \frac{2n}{16n} = \frac{1}{8} = 0.125$ flops/byte

Example 3: Matrix-Vector Multiplication

$$y = Ax$$

```cpp
// gemv.cpp
for (int i = 0; i < n; i++) {
    y[i] = 0;
    for (int j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

```
$ cd examples && make run_gemv
```

Example 3: Matrix-Vector Multiplication

$$y = Ax$$

```
// gemv.cpp
for (int i = 0; i < n; i++) {
    y[i] = 0;
    for (int j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

```
$ cd examples && make run_gemv
```

Work: $W = 2n^2$ flops

Data movement:

- Read A: $8n^2$ bytes
- Read x: $8n$ bytes (reused n times if fits in cache)
- Write y: $8n$ bytes

$Q \approx 8n^2$ bytes (dominated by A)

Example 3: Matrix-Vector Multiplication

$$y = Ax$$

```
// gemv.cpp
for (int i = 0; i < n; i++) {
    y[i] = 0;
    for (int j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

```
$ cd examples && make run_gemv
```

Work: $W = 2n^2$ flops

Data movement:

- Read A: $8n^2$ bytes
- Read x: $8n$ bytes (reused n times if fits in cache)
- Write y: $8n$ bytes

$Q \approx 8n^2$ bytes (dominated by A)

Operational Intensity:

$$I = \frac{2n^2}{8n^2} = \frac{1}{4} = 0.25 \text{ flops/byte}$$

Example 4: Matrix-Matrix Multiplication (Naive)

$$C = AB$$

```cpp
// gemm.cpp - compares naive vs blocked
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

```
$ cd examples && make run_gemm
```

Example 4: Matrix-Matrix Multiplication (Naive)

$$C = AB$$

```
// gemm.cpp - compares naive vs blocked
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

```
$ cd examples && make run_gemm
```

Work: $W = 2n^3$ flops

Data movement (naive, cache « matrices):

- Each C_{ij} requires reading row of A and column of B
- Total: $Q \approx 8n^3$ bytes (terrible!)

Example 4: Matrix-Matrix Multiplication (Naive)

$$C = AB$$

```
// gemm.cpp - compares naive vs blocked
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

```
$ cd examples && make run_gemm
```

Work: $W = 2n^3$ flops

Data movement (naive, cache « matrices):

- Each C_{ij} requires reading row of A and column of B
- Total: $Q \approx 8n^3$ bytes (terrible!)

Operational Intensity (naive):

$$I = \frac{2n^3}{8n^3} = \frac{1}{4} \text{ flops/byte}$$

Same as matrix-vector!

Example 4b: Matrix-Matrix Multiplication (Blocked)

With blocking (block size b, $3b^2 \leq$ cache):

Data movement: We divide A, B, C into $b \times b$ blocks. For each block C_{ij}:

$$C_{ij} = \sum_{k=1}^{n/b} A_{ik} B_{kj}$$

Each block multiply loads three $b \times b$ blocks (A_{ik}, B_{kj}, C_{ij}), fitting in cache since $3b^2 \leq \gamma$.

Example 4b: Matrix-Matrix Multiplication (Blocked)

With blocking (block size b, $3b^2 \leq$ cache):

Data movement: We divide A, B, C into $b \times b$ blocks. For each block C_{ij}:

$$C_{ij} = \sum_{k=1}^{n/b} A_{ik} B_{kj}$$

Each block multiply loads three $b \times b$ blocks (A_{ik}, B_{kj}, C_{ij}), fitting in cache since $3b^2 \leq \gamma$.

Total block multiplies: $(n/b)^3 = n^3/b^3$

Bytes per block multiply: $3 \times 8b^2$ (but C_{ij} is reused across k)

Net loads: each of the n^2/b^2 blocks of A is read n/b times, same for B:

$$Q = 2 \times \frac{n^2}{b^2} \times \frac{n}{b} \times 8b^2 = \frac{2n^3}{b} \times 8$$

Example 4b: Matrix-Matrix Multiplication (Blocked)

With blocking (block size b, $3b^2 \leq$ cache):

Data movement: We divide A, B, C into $b \times b$ blocks. For each block C_{ij}:

$$C_{ij} = \sum_{k=1}^{n/b} A_{ik} B_{kj}$$

Each block multiply loads three $b \times b$ blocks (A_{ik}, B_{kj}, C_{ij}), fitting in cache since $3b^2 \leq \gamma$.

Total block multiplies: $(n/b)^3 = n^3/b^3$

Bytes per block multiply: $3 \times 8b^2$ (but C_{ij} is reused across k)

Net loads: each of the n^2/b^2 blocks of A is read n/b times, same for B:

$$Q = 2 \times \frac{n^2}{b^2} \times \frac{n}{b} \times 8b^2 = \frac{2n^3}{b} \times 8$$

Operational Intensity (blocked):

$$I = \frac{W}{Q} = \frac{2n^3}{16n^3/b} = \frac{b}{8} \text{ flops/byte}$$

Example 4b: Matrix-Matrix Multiplication (Blocked)

With blocking (block size b, $3b^2 \leq$ cache):

Data movement: We divide A, B, C into $b \times b$ blocks. For each block C_{ij}:

$$C_{ij} = \sum_{k=1}^{n/b} A_{ik} B_{kj}$$

Each block multiply loads three $b \times b$ blocks (A_{ik}, B_{kj}, C_{ij}), fitting in cache since $3b^2 \leq \gamma$.

Total block multiplies: $(n/b)^3 = n^3/b^3$

Bytes per block multiply: $3 \times 8b^2$ (but C_{ij} is reused across k)

Net loads: each of the n^2/b^2 blocks of A is read n/b times, same for B:

$$Q = 2 \times \frac{n^2}{b^2} \times \frac{n}{b} \times 8b^2 = \frac{2n^3}{b} \times 8$$

Operational Intensity (blocked):

$$I = \frac{W}{Q} = \frac{2n^3}{16n^3/b} = \frac{b}{8} \text{ flops/byte}$$

Summary: Operational Intensity

Operation	W	Q	I
DAXPY: $y = \alpha x + y$	$2n$	$24n$	$1/12$
Dot product: $x^T y$	$2n$	$16n$	$1/8$
GEMV: $y = Ax$	$2n^2$	$8n^2$	$1/4$
GEMM naive: $C = AB$	$2n^3$	$8n^3$	$1/4$
GEMM blocked: $C = AB$	$2n^3$	$16n^3/b$	$b/8 \sim O(\sqrt{\gamma})$

Summary: Operational Intensity

Operation	W	Q	I
DAXPY: $y = \alpha x + y$	$2n$	$24n$	$1/12$
Dot product: $x^T y$	$2n$	$16n$	$1/8$
GEMV: $y = Ax$	$2n^2$	$8n^2$	$1/4$
GEMM naive: $C = AB$	$2n^3$	$8n^3$	$1/4$
GEMM blocked: $C = AB$	$2n^3$	$16n^3/b$	$b/8 \sim O(\sqrt{\gamma})$

Key insight: Only GEMM (with blocking) can achieve high operational intensity!

Roofline Examples

Example Platform: Intel Skylake

Single core specifications:

- Peak: $\pi = 16$ flops/cycle (2 FMA units $\times$ 4-wide AVX $\times$ 2)
- At 3 GHz: $\pi = 48$ GFLOPS

Example Platform: Intel Skylake

Single core specifications:

- Peak: $\pi = 16$ flops/cycle (2 FMA units $\times$ 4-wide AVX $\times$ 2)
- At 3 GHz: $\pi = 48$ GFLOPS

Memory bandwidth:

- Let's assume: $\beta \approx 15$ GB/s $= 5$ bytes/cycle at 3 GHz

Example Platform: Intel Skylake

Single core specifications:

- Peak: $\pi = 16$ flops/cycle (2 FMA units $\times$ 4-wide AVX $\times$ 2)
- At 3 GHz: $\pi = 48$ GFLOPS

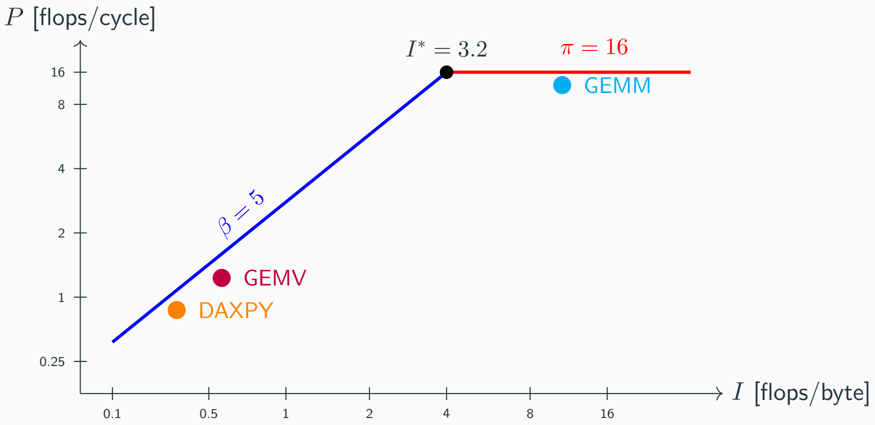
Memory bandwidth:

- Let's assume: $\beta \approx 15$ GB/s $= 5$ bytes/cycle at 3 GHz

Ridge point:

$$I^* = \frac{\pi}{\beta} = \frac{16}{5} = 3.2 \text{ flops/byte}$$

Skylake Roofline



Where Do Common Operations Fall?

Memory Bound	Compute Bound
BLAS Level 1: DAXPY, Dot product, Norms	BLAS Level 3: GEMM, Matrix factorizations
BLAS Level 2: GEMV, Matrix-vector solve	Dense linear algebra
Stencils, SpMV, Graph algorithms	Convolutions, GEMM

Numerical Example: DAXPY

Problem: $y = 2x + y$, vectors of length $n = 10^8$

Platform: 3 GHz Skylake, $\pi = 48$ GFLOPS, $\beta = 15$ GB/s

Numerical Example: DAXPY

Problem: $y = 2x + y$, vectors of length $n = 10^8$

Platform: 3 GHz Skylake, $\pi = 48$ GFLOPS, $\beta = 15$ GB/s

Analysis:

- $W = 2 \times 10^8$ flops
- $Q = 24 \times 10^8$ bytes = 2.4 GB
- $I = 1/12 \approx 0.083$ flops/byte

Numerical Example: DAXPY

Problem: $y = 2x + y$, vectors of length $n = 10^8$

Platform: 3 GHz Skylake, $\pi = 48$ GFLOPS, $\beta = 15$ GB/s

Analysis:

- $W = 2 \times 10^8$ flops
- $Q = 24 \times 10^8$ bytes = 2.4 GB
- $I = 1/12 \approx 0.083$ flops/byte

Maximum achievable performance:

$$P = \min(\pi, \beta \cdot I) = \min(48, 15 \times 0.083) = \min(48, 1.25) = 1.25 \text{ GFLOPS}$$

Numerical Example: DAXPY

Problem: $y = 2x + y$, vectors of length $n = 10^8$

Platform: 3 GHz Skylake, $\pi = 48$ GFLOPS, $\beta = 15$ GB/s

Analysis:

- $W = 2 \times 10^8$ flops
- $Q = 24 \times 10^8$ bytes = 2.4 GB
- $I = 1/12 \approx 0.083$ flops/byte

Maximum achievable performance:

$$P = \min(\pi, \beta \cdot I) = \min(48, 15 \times 0.083) = \min(48, 1.25) = 1.25 \text{ GFLOPS}$$

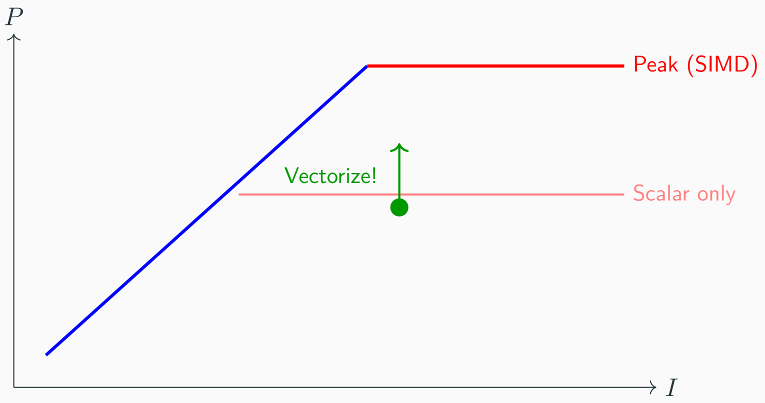
Minimum runtime:

$$T = \frac{W}{P} = \frac{2 \times 10^8}{1.25 \times 10^9} = 0.16 \text{ seconds}$$

Adding More Roofs

Multiple Compute Ceilings

Not all code can achieve peak π:



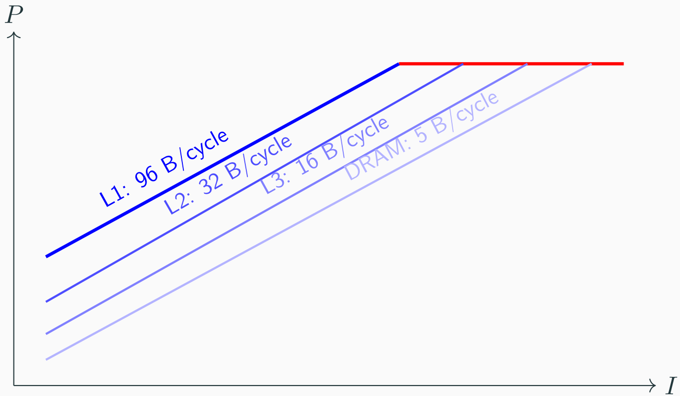
Example: Instruction Mix Ceilings

Your code's ceiling depends on:

- Whether it vectorizes (SIMD)
- Whether compiler generates FMA
- Mix of adds vs multiplies

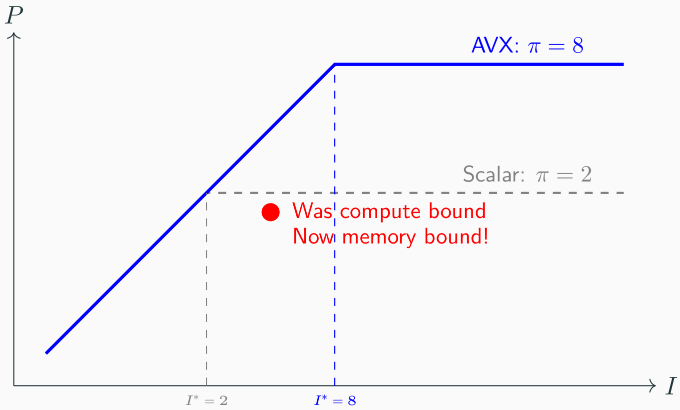
Multiple Memory Ceilings

Different memory levels have different bandwidths:



Implication: Same code, different problem sizes = different rooflines!

Effect of SIMD on Roofline



SIMD increases π, shifting the ridge point right! **

Example: Matrix Multiplication - Batch Size Effect

Forward pass of fully-connected layer: $Y = XW$ where X is batch $\times$ input, W is input $\times$ output

Example: Matrix Multiplication - Batch Size Effect

Forward pass of fully-connected layer: $Y = XW$ where X is batch $\times$ input, W is input $\times$ output

Work: $W = 2 \times$ batch $\times$ input $\times$ output

Data:

- W matrix: $8 \times$ input $\times$ output bytes
- X matrix: $8 \times$ batch $\times$ input bytes
- Y matrix: $8 \times$ batch $\times$ output bytes

Example: Matrix Multiplication - Batch Size Effect

Forward pass of fully-connected layer: $Y = XW$ where X is batch $\times$ input, W is input $\times$ output

Work: $W = 2 \times \text{batch} \times \text{input} \times \text{output}$

Data:

- W matrix: $8 \times \text{input} \times \text{output}$ bytes
- X matrix: $8 \times \text{batch} \times \text{input}$ bytes
- Y matrix: $8 \times \text{batch} \times \text{output}$ bytes

Operational intensity:

$$I \approx \frac{2 \times \text{batch} \times \text{in} \times \text{out}}{8(\text{in} \times \text{out} + \text{batch} \times \text{in} + \text{batch} \times \text{out})}$$

Example: Matrix Multiplication - Batch Size Effect

Forward pass of fully-connected layer: $Y = XW$ where X is batch $\times$ input, W is input $\times$ output

Work: $W = 2 \times \text{batch} \times \text{input} \times \text{output}$

Data:

- W matrix: $8 \times \text{input} \times \text{output}$ bytes
- X matrix: $8 \times \text{batch} \times \text{input}$ bytes
- Y matrix: $8 \times \text{batch} \times \text{output}$ bytes

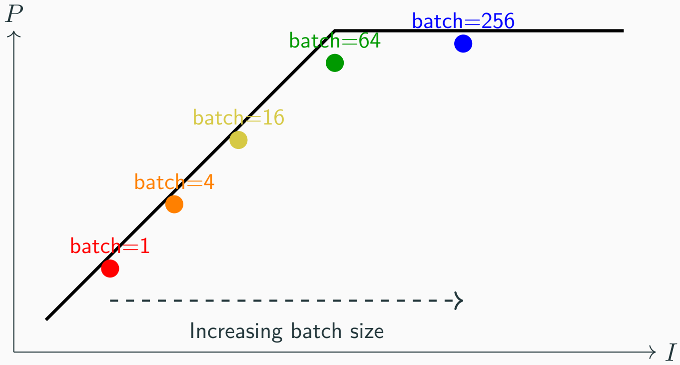
Operational intensity:

$$I \approx \frac{2 \times \text{batch} \times \text{in} \times \text{out}}{8(\text{in} \times \text{out} + \text{batch} \times \text{in} + \text{batch} \times \text{out})}$$

For large batch: $I \to \frac{\text{batch}}{4}$ (weights reused across batch)

For batch = 1: $I \approx 1/4$ (memory bound!)

Batch Size and Roofline

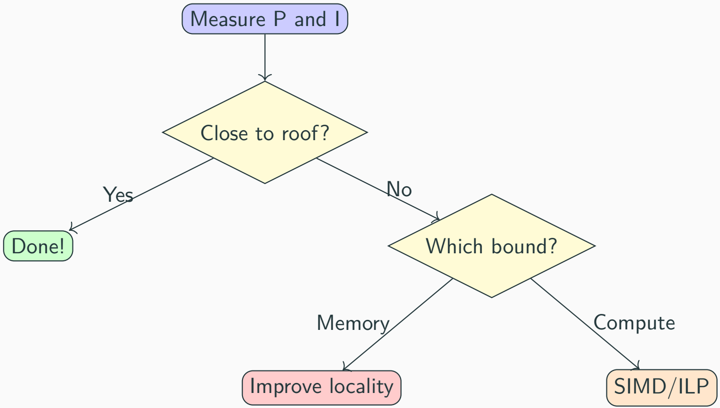


This is why LLM decoding (batch=1) is memory bound.

Batching as an optimization works because a larger batch leads to a higher performance celing (ie doing more makes it more efficient, so we better do more)

Using the Roofline for Optimization

Optimization Strategy



If Memory Bound...

Goal: Increase operational intensity I or get closer to bandwidth roof.

If Memory Bound...

Goal: Increase operational intensity I or get closer to bandwidth roof.

Techniques:

1. **Blocking/Tiling:** Reuse data in cache
2. **Loop fusion:** Combine loops to reuse loaded data
3. **Data layout:** Improve spatial locality
4. **Prefetching:** Hide memory latency
5. **Compression:** Reduce data volume

If Compute Bound...

Goal: Increase performance P toward compute roof.

If Compute Bound...

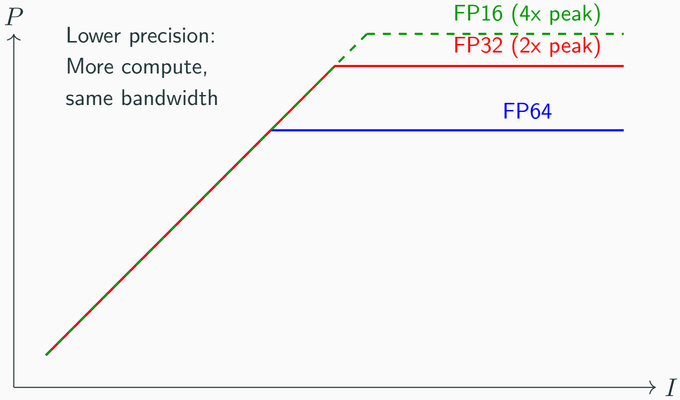
Goal: Increase performance P toward compute roof.

Techniques:

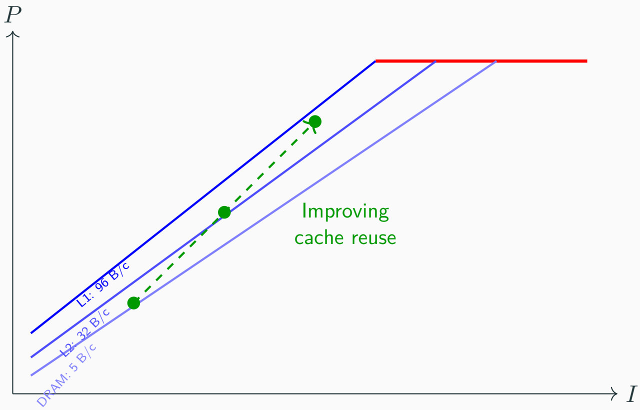
1. **Vectorization (SIMD):** Process multiple elements per instruction
2. **ILP:** Multiple accumulators, loop unrolling
3. **FMA:** Fused multiply-add instructions
4. **Algorithm improvement:** Reduce total work
5. **Parallelization:** Use multiple cores

Roofline Variations

Roofline for Different Data Types



Hierarchical Roofline



Summary

The Roofline Model

$$P \leq \min(\pi, \beta \cdot I)$$

The Roofline Model

$$P \leq \min(\pi, \beta \cdot I)$$

Two bounds:

1. **Compute bound:** $P \leq \pi$ (limited by FLOPS)
2. **Memory bound:** $P \leq \beta I$ (limited by bandwidth)

The Roofline Model

$$P \leq \min(\pi, \beta \cdot I)$$

Two bounds:

1. **Compute bound:** $P \leq \pi$ (limited by FLOPS)
2. **Memory bound:** $P \leq \beta I$ (limited by bandwidth)

Ridge point: $I^* = \pi/\beta$ separates the two regions

The Roofline Model

$$P \leq \min(\pi, \beta \cdot I)$$

Two bounds:

1. **Compute bound:** $P \leq \pi$ (limited by FLOPS)
2. **Memory bound:** $P \leq \beta I$ (limited by bandwidth)

Ridge point: $I^* = \pi/\beta$ separates the two regions

Key insight: Operational intensity $I = W/Q$ determines which bound applies!

Key Takeaways

1. **Calculate I before optimizing** – know your bottleneck!
2. **Memory bound** $(I < I^*)$: Improve locality, increase reuse
3. **Compute bound** $(I > I^*)$: SIMD, ILP, better algorithms
4. **Multiple roofs** exist for different:
   - Instruction mixes (FMA vs non-FMA)
   - Cache levels (L1 vs L2 vs DRAM)
   - Data types (FP64 vs FP32 vs FP16)
5. **Measure** with perf counters to validate analysis