“Simple” performance modeling: The Roofline Model

Loop-based performance modeling: Execution vs. data transfer
A simple performance model for loops

Simplistic view of the hardware:

- Execution units
  - max. performance $P_{peak}$
  - Data path, bandwidth $b_S$
  - Unit: byte/s

Simplistic view of the software:

```plaintext
! may be multiple levels
do i = 1,<sufficient>
  <complicated stuff doing
    N flops causing
    V bytes of data transfer>
endo
```

Computational intensity $I = \frac{N}{V}$
- Unit: flop/byte
Prelude: Modeling customer dispatch in a bank

How fast can tasks be processed? $P$ [flop/s]

The bottleneck is either

- The execution of work: $P_{\text{peak}}$ [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{\text{peak}}, I \cdot b_S)$$

This is the “Naïve Roofline Model”

- High intensity: $P$ limited by execution
- Low intensity: $P$ limited by data transfer
- “Knee” at $P_{\text{max}} = I \cdot b_S$:
  Best use of resources

- Roofline is an “optimistic” model ("light speed")

(c) RRZE 2019

Roofline Model
The Roofline Model in computing – Basics

Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: \( P_{\text{peak}} \left[ \frac{F}{s} \right] \)
- Machine parameter #2: Memory bandwidth: \( b_s \left[ \frac{B}{s} \right] \)
- Code characteristic: Computational Intensity: \( I \left[ \frac{F}{B} \right] \)

Machine properties:

\[
P_{\text{peak}} = 4 \frac{\text{GF}}{s}
\]

\[
b_s = 10 \frac{\text{GB}}{s}
\]

Application property: \( I \)

\[
I = \frac{2F}{8B} = 0.25 \frac{F}{B}
\]

double s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];
}

(c) RRZE 2019 Roofline Model
Prerequisites for the Roofline Model

- The roofline formalism is based on some (crucial) prerequisites:
  - There is a clear concept of “work” vs. “traffic”
    - “work” = flops, updates, iterations…
    - “traffic” = required data to do “work”

- Attainable bandwidth of code = input parameter! Determine effective saturated bandwidth of the chip via simple streaming benchmarks to model more complex kernels and applications

Assumptions behind the model:
- Data transfer and core execution overlap perfectly!
  - Either the limit is core execution or it is data transfer
- Slowest limiting factor “wins”; all others are assumed to have no impact
- Latency effects are ignored, i.e. perfect streaming mode
- “Steady state” code execution (no wind-up/-down effects)
The Roofline Model in computing – Basics

Compare capabilities of different machines:

- Roofline always provides upper bound – but is it realistic?
- If code is not able to reach this limit (e.g., contains add operations only) machine parameters need to be redefined (e.g., $P_{peak} \rightarrow \frac{P_{peak}}{2}$)

Assuming double precision – for single precision: $P_{peak} \rightarrow 2 \cdot P_{peak}$
A refined Roofline Model

1. \( P_{\text{max}} = \) Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily \( P_{\text{peak}} \))
   \[ \Rightarrow \text{e.g., } P_{\text{max}} = 176 \text{ GFlop/s} \]

2. \( I = \) Computational intensity ("work" per byte transferred) over the slowest data path utilized (code balance \( B_C = I^{-1} \))
   \[ \Rightarrow \text{e.g., } I = 0.167 \text{ Flop/Byte} \Rightarrow B_C = 6 \text{ Byte/Flop} \]

3. \( b_S = \) Applicable (saturated) peak bandwidth of the slowest data path utilized
   \[ \Rightarrow \text{e.g., } b_S = 56 \text{ GByte/s} \]

Expected performance:

\[ P = \min(P_{\text{max}}, I \cdot b_S) = \min\left(P_{\text{max}}, \frac{b_S}{B_C}\right) \]


Factors to consider in the Roofline Model

Bandwidth-bound (simple case)
- Accurate traffic calculation (write-allocate, strided access, …)
- Practical ≠ theoretical BW limits
- Saturation effects → consider full socket only

Core-bound (may be complex)
- Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
- Limit is linear in # of cores
Multiple ceilings may apply

- Different $P_{\text{max}}$  
  $\rightarrow$ different flat ceilings

- Different bandwidths / data paths  
  $\rightarrow$ different inclined ceilings
Complexities of in-core execution ($P_{\text{max}}$)

Multiple bottlenecks:

- Decode/retirement throughput
- Port contention (direct or indirect)
- Arithmetic pipeline stalls (dependencies)
- Overall pipeline stalls (branching)
- L1 Dcache bandwidth (LD/ST throughput)
- Scalar vs. SIMD execution
- L1 Icache (LD/ST) bandwidth
- Alignment issues
- …
Estimating per-core $P_{\text{max}}$ on a given architecture

Haswell port scheduler model:

![Diagram of Haswell port scheduler model](image)

(c) RRZE 2019
Zeroth order throughput limits on Haswell

- **Per cycle with AVX, SSE, or scalar**
  - 2 LOAD instructions AND 1 STORE instruction
  - 2 instructions selected from the following five:
    - 2 FMA (fused multiply-add)
    - 2 MULT
    - 1 ADD
  - **Overall maximum of 4 instructions per cycle**
    - In practice, 3 is more realistic
    - µ-ops may be a better indicator for short loops

- **Remember: one AVX instruction handles**
  - 4 DP operands or
  - 8 SP operands

- **First order correction**
  - Typically only two LD/ST instructions per cycle due to one AGU handling “simple” addresses only
  - See SIMD chapter for more about memory addresses
Example: Estimate $P_{\text{max}}$ of vector triad on Haswell

for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

Assembly code (AVX2+FMA, no additional unrolling):

..B2.9:

  vmovupd (%rdx,%rax,8), %ymm2  # LOAD
  vmovupd (%r12,%rax,8), %ymm1  # LOAD
  vfmadd213pd (%rbx,%rax,8), %ymm1, %ymm2  # LOAD+FMA
  vmovupd %ymm2, (%rdi,%rax,8)  # STORE
  addq $4, %rax
  cmpq %r11, %rax
  jb ..B2.9
  # remainder loop omitted

Iterations are independent → throughput assumption justified!

Best-case execution time?
Example: Estimate $P_{\text{max}}$ of vector triad on Haswell

for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

Minimum number of cycles to process one AVX-vectorized iteration (equivalent to 4 scalar iterations) on one core?

→ Assuming full throughput:

Cycle 1: LOAD + LOAD + STORE
Cycle 2: LOAD + LOAD + FMA + FMA
Cycle 3: LOAD + LOAD + STORE

Answer: 1.5 cycles
Example: Estimate $P_{\text{max}}$ of vector triad on Haswell (2.3 GHz)

for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

What is the performance in GFlops/s per core and the bandwidth in GBytes/s?

One AVX iteration (1.5 cycles) does $4 \times 2 = 8$ flops:

$$2.3 \cdot 10^9 \text{cy/s} \cdot \frac{8 \text{flops}}{1.5 \text{cy}} = 12.27 \frac{\text{Gflops}}{\text{s}}$$

$$12.27 \frac{\text{Gflops}}{\text{s}} \cdot 16 \frac{\text{bytes}}{\text{flop}} = 196 \frac{\text{Gbyte}}{\text{s}}$$

(c) RRZE 2019
Roofline Model
$P_{\text{max}}$ + bandwidth limitations: The vector triad

Vector triad $A(:) = B(:) + C(:) \times D(:)$ on a 2.3 GHz 14-core Haswell chip

Consider full chip (14 cores):

Memory bandwidth: $b_S = 50 \text{ GB/s}$

Code balance (incl. write allocate):

$B_c = (4+1) \text{ Words / 2 Flops} = 20 \text{ B/F} \rightarrow I = 0.05 \text{ F/B}$

$\rightarrow I \cdot b_S = 2.5 \text{ GF/s} \text{ (0.5% of peak performance)}$

$P_{\text{peak}} / \text{core} = 36.8 \text{ Gflop/s} \text{ ((8+8) Flops/cy x 2.3 GHz)}$

$P_{\text{max}} / \text{core} = 12.27 \text{ Gflop/s} \text{ (see prev. slide)}$

$\rightarrow P_{\text{max}} = 14 \times 12.27 \text{ Gflop/s} = 172 \text{ Gflop/s} \text{ (33% peak)}$

$P = \min(P_{\text{max}}, I \cdot b_S) = \min(172, 2.5) \text{ GFlop/s} = 2.5 \text{ GFlop/s}$
Code balance: more examples

```c
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i] + b[i];
}
```

\[ B_C = 24B / 1F = 24 \text{ B/F} \]
\[ I = 0.042 \text{ F/B} \]

```c
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i] + s * b[i];
}
```

\[ B_C = 24B / 2F = 12 \text{ B/F} \]
\[ I = 0.083 \text{ F/B} \]

Scalar – can be kept in register

```c
float s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];
}
```

\[ B_C = 4B/2F = 2 \text{ B/F} \]
\[ I = 0.5 \text{ F/B} \]

Scalar – can be kept in register

```c
float s=0, a[], b[];
for(i=0; i<N; ++i) {
    s = s + a[i] * b[i];
}
```

\[ B_C = 8B / 2F = 4 \text{ B/F} \]
\[ I = 0.25 \text{ F/B} \]

Scalar – can be kept in register

(c) RRZE 2019 Roofline Model
Machine balance for hardware characterization

- For quick comparisons the concept of **machine balance** is useful

\[ B_m = \frac{b_s}{P_{\text{peak}}} \]

- Machine Balance = How much input data can be delivered for each FP operation? (“Memory Gap characterization”)
  - Assuming balanced MULT/ADD
- Rough estimate: \( B_m \ll B_c \rightarrow \) strongly memory-bound code
- Typical values (main memory):

  Intel Haswell 14-core 2.3 GHz
  \[ B_m = \frac{60 \text{ GB/s}}{(14 \times 2.3 \times 16) \text{ GF/s}} \approx 0.12 \text{ B/F} \]

  Intel Skylake 24-core 2.3 GHz
  \[ \approx 0.06 \text{ B/F} \]

  Nvidia P100
  \[ \approx 0.10 \text{ B/F} \]

  Nvidia V100
  \[ \approx 0.10 \text{ B/F} \]

  Intel Xeon Phi Knights Landing (MCDRAM)
  \[ \approx 0.18 \text{ B/F} \]
Machine balance over time

Higher is better! → more balanced

Year


Machine balance $B_m$ [byte/flop]

0.1 1.0

NHL Nehalem
SNB Sandy Bridge
HSW Haswell
KNC Knights Corner
KNL Knights Landing

SX-8 (4 B/F)
SX-9 (2.56 B/F)
SX-ACE
Pentium D
NHL
SNB
KNC
V100
HSL
SKylake SP
KNL (main mem.)
KNL (HBM)
K20
P100

NEC Aurora


(c) RRZE 2019

Roofline Model
A not so simple Roofline example

Example:  
\[ \text{do } i=1,N; \ s=s+a(i); \ \text{enddo} \]
in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

\[ P = \min(P_{\text{max}}, I \cdot b_s) \]

- Machine peak (ADD+MULT) Out of reach for this code
- ADD peak (best possible code)
- no SIMD
- 3-cycle latency per ADD if not unrolled

\[ P \] (better loop code)

\[ I = 1 \text{ flop / 4 byte (SP!)} \]

\[ b_s = 40 \text{ GB/s} \]

\[ 282 \text{ GF/s} \]

\[ 141 \text{ GF/s} \]

\[ 17.6 \text{ GF/s} \]

\[ 5.9 \text{ GF/s} \]

See architecture intro
... on the example of

```
do i=1,N; s=s+a(i); enddo
```

... in single precision

Input to the roofline model

**Code analysis:**
1 ADD + 1 LOAD

**Throughput:** 1 ADD + 1 LD/cy
**Pipeline depth:** 3 cy (ADD)
**8-way SIMD, 8 cores**

Minimum memory bandwidth 40 GB/s

**Worst code:** $P = 5.9$ GF/s (core bound)
**Better code:** $P = 10$ GF/s (memory bound)

(c) RRZE 2019

Roofline Model
Shortcomings of the roofline model

- **Saturation effects in multicore chips are not explained**
  - Reason: “saturation assumption”
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - It is not sufficient to measure single-core STREAM to make it work
  - Only increased “pressure” on the memory interface can saturate the bus
    → need more cores!

- **In-cache performance is not correctly predicted**

- **The ECM performance model gives more insight:**
  
Exercise: Dense matrix-vector multiplication in DP

\[
\text{do } c = 1 \text{ , } \text{NC}
\]
\[
\text{do } r = 1 \text{ , } \text{NR}
\]
\[
y(r) = y(r) + A(r,c) \times x(c)
\]
\enddo
\enddo

- Assume \( \text{NC} = \text{NR} \approx 10,000 \)
- Applicable peak performance?
- Relevant data path?
- Computational Intensity?

\[
\text{do } c = 1 \text{ , } \text{NC}
\]
\[
tmp = x(c)
\]
\[
\text{do } r = 1 \text{ , } \text{NR}
\]
\[
y(r) = y(r) + A(r,c) \times tmp
\]
\enddo
\enddo
Dense MVM Roofline discussion for Haswell
14 cores, 2.3 GHz

- $P_{\text{max}}$
  - AVX kernel: 2 LOADs, 1 STORE, 1 FMA
    → 1 cy per AVX iteration → per-core $P_{\text{max}} = 8$ F/cy = $\frac{1}{2} P_{\text{peak}}$
  - Full socket: $P_{\text{max}} = (14 \times 2.3 \times 8)$ F/s = 258 GF/s

- $B_c$
  - $\mathbf{x}(\cdot)$ does not cause significant data traffic
  - $R=10000$ → memory for $\mathbf{y}(\cdot) = 80$ kB → fits into L2 cache of each core
    → only $\mathbf{A}(\cdot,\cdot)$ causes traffic from main memory of 8 bytes for 2 Flops
  → $B_c = 4$ B/F

- $b_S = 60$ GB/sec (saturated read-only memory BW measurement)

- Roofline model:
  $$P = \min(P_{\text{max}}, I \cdot b_S) = \min(258, 15) \text{ GFlop/s} = 15 \text{ GFlop/s}$$
1. Hit the BW bottleneck by good serial code (e.g., Perl → Fortran)

2. Increase intensity to make better use of BW bottleneck (e.g., loop blocking [see later])

3. Increase intensity and go from memory-bound to core-bound (e.g., temporal blocking)

4. Hit the core bottleneck by good serial code (e.g., -fno-alias [see later])

5. Shift $P_{\text{max}}$ by accessing additional hardware features or using a different algorithm/implementation (e.g., scalar → SIMD)