Introduction to the Roofline Model

HPC

Assoc.Prof. Dr. Sascha Hunold

TU Wien

Learning Goals and Objectives of this Lecture

1. **Understand** the **Need** for **Performance Models**
2. **Understand** the **Building Blocks** of the Roofline Model
3. **Create** a **Roofline Model** for a Parallel Machine
4. **Apply** the **Roofline Model** for Performance Analysis

Central Question

Is my Program fast (enough)?

Analyzing Programs

- Algorithms often **analyzed theoretically** (Big O notation)
  - $\mathcal{O}(n^2)$ better than $\mathcal{O}(n^3)$ (asymptotically)
- Evaluation in **practice** through **experiments**
  - Measuring the **running time**,
  - Measuring the **throughput**,
  - ...

Analyzing Programs

- Algorithms often **analyzed theoretically** (Big O notation)
  - $\mathcal{O}(n^2)$ better than $\mathcal{O}(n^3)$ (asymptotically)
- Evaluation in **practice** through **experiments**
  - Measuring the **running time**,
  - Measuring the **throughput**,
  - ...

We could ask

How fast can our **code possibly run** on a given machine?

Analyzing Programs

- Algorithms often **analyzed theoretically** (Big O notation)
  - $\mathcal{O}(n^2)$ better than $\mathcal{O}(n^3)$ (asymptotically)
- Evaluation in **practice** through **experiments**
  - Measuring the **running time**,
  - Measuring the **throughput**,
  - $\ldots$

We could ask

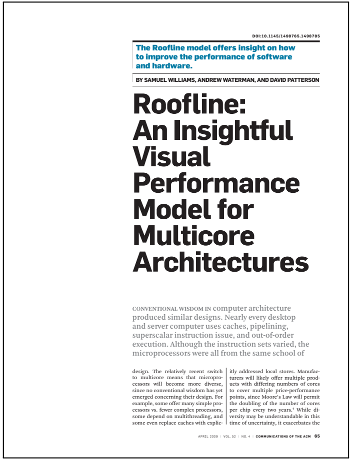
How fast can our **code possibly run** on a given machine?

- Use **performance models** for answering this question

Performance Models – Many Use Cases

- **Identify** performance **bottlenecks**
- **Motivate** software **optimizations**
- **Motivate** need for **algorithmic** changes
- **Determine when programs can hardly be further optimized**
- **Predict performance** on future/different architectures

The Roofline Model
A visual performance model



DOI:10.1145/1498765.1498785

The Roofline model offers insight on how to improve the performance of software and hardware.

BY SAMUEL WILLIAMS, ANDREW WATERMAN, AND DAVID PATTERSON

Roofline: An Insightful Visual Performance Model for Multicore Architectures

CONVENTIONAL WISDOM IN computer architecture produced similar designs. Nearly every desktop and server computer uses caches, pipelining, superscalar instruction issue, and out-of-order execution. Although the instruction sets varied, the microprocessors were all from the same school of design. The relatively recent switch to multicore means that microprocessors will become more diverse, since no conventional wisdom has yet emerged concerning their design. For example, some offer many simple processors vs. fewer complex processors, some depend on multithreading, and some even replace caches with explicitly addressed local stores. Manufacturers will likely offer multiple products with differing numbers of cores to cover multiple price-performance points, since Moore's Law will permit the doubling of the number of cores per chip every two years.[4] While diversity may be understandable in this time of uncertainty, it exacerbates the

APRIL 2009 | VOL. 52 | NO. 4 | COMMUNICATIONS OF THE ACM 65

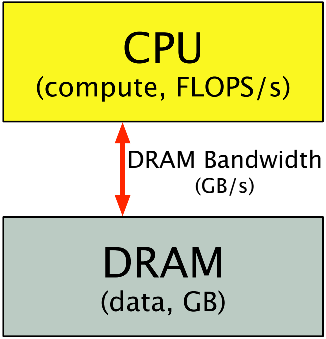
S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), p. 65

Assumptions

Basic Architectural Model

- architectural model consists of
  - **computational elements** (CPUs, cores)
  - **memory elements** (DRAM)
- **computation**
  - computation consists of **floating-point operations** (multiply, add, compare, etc.)
  - **processors** have a certain **computational rate** (e.g., 2 GFLOPs/s)
- **communication**
  - **movement of data is bounded** by processor-memory interconnect
  - links have a **maximum memory bandwidth** (e.g., $10\,$MB/s)



The Arithmetic Intensity of Computational Kernels

- programs can be thought of "sequence of computational kernels"
  - `kernel1 (long burst)...some file I/O / message passing...kernel2 (long burst)...`
  - kernels: matrix-vector operations, FFT, Stencil
- **Goal: analyze performance of computational kernels**

Arithmetic Intensity (AI)

The **Arithmetic Intensity** (AI) is a kernel's ratio of **computation to traffic**.
It is measured in **FLOPs/Bytes**.

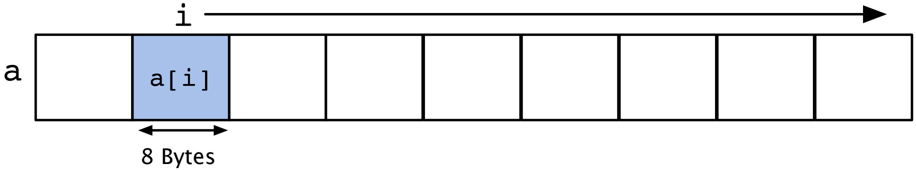
- The **AI is central** to the Roofline model!

Examples of Arithmetic Intensity

Arithmetic Intensity: Example 1

```
temp = 0.0;   // this stays in a register
for(i=0; i<N; i++) {
  /* a[i] is a double */
  temp += a[i] * a[i];
}
magnitude = sqrt(temp);
```

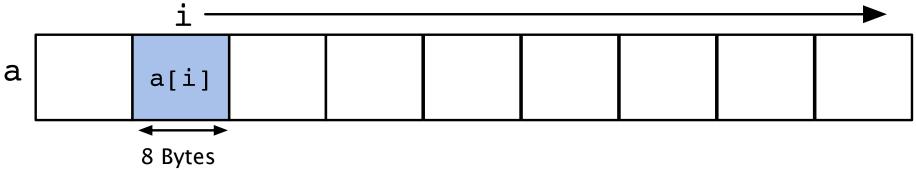
- analyze loop



Arithmetic Intensity: Example 1

```
temp = 0.0;   // this stays in a register
for(i=0; i<N; i++) {
  /* a[i] is a double */
  temp += a[i] * a[i];
}
magnitude = sqrt(temp);
```

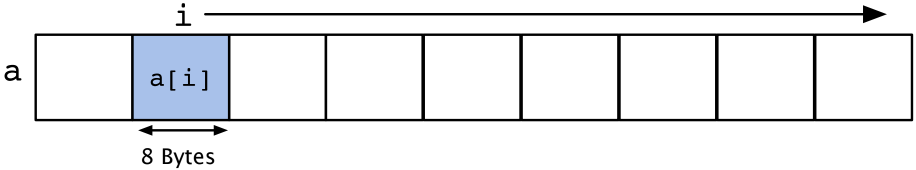
- analyze loop
- **2 FLOPs** (1 ADD, 1 MUL)



Arithmetic Intensity: Example 1

```
temp = 0.0;   // this stays in a register
for(i=0; i<N; i++) {
  /* a[i] is a double */
  temp += a[i] * a[i];
}
magnitude = sqrt(temp);
```

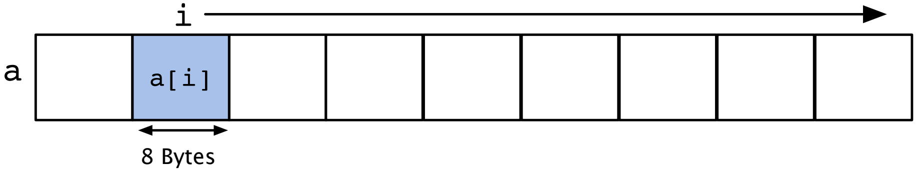
- analyze loop
- **2 FLOPs** (1 ADD, 1 MUL)
- **1 FLOAT** (`a[i]`, double precision) needs to be read from DRAM (8 Bytes)



Arithmetic Intensity: Example 1

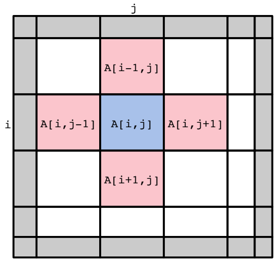
```
temp = 0.0;   // this stays in a register
for(i=0; i<N; i++) {
  /* a[i] is a double */
  temp += a[i] * a[i];
}
magnitude = sqrt(temp);
```

- analyze loop
- **2 FLOPs** (1 ADD, 1 MUL)
- **1 FLOAT** (`a[i]`, double precision) needs to be read from DRAM (8 Bytes)
- AI = 2N FLOPS / 8N Bytes
  AI = **1/4 FLOPS/Byte**



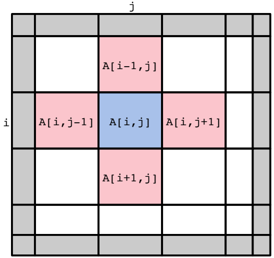
Arithmetic Intensity: Example 2

```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    C[i][j] = a *  A[i][j] +
              b * (A[i][j-1] +
                   A[i-1][j] +
                   A[i+1][j] +
                   A[i][j+1]);
  }
}
```



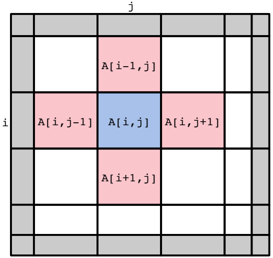
Arithmetic Intensity: Example 2

```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    C[i][j] = a *  A[i][j] +
              b * (A[i][j-1] +
                   A[i-1][j] +
                   A[i+1][j] +
                   A[i][j+1]);
  }
}
```



- **6 FLOPs** per iteration (2 MULs, 4 ADDs)

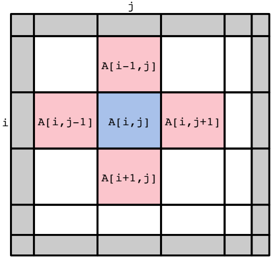
```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    C[i][j] = a *  A[i][j] +
              b * (A[i][j-1] +
                   A[i-1][j] +
                   A[i+1][j] +
                   A[i][j+1]);
  }
}
```



- **6 FLOPs** per iteration (2 MULs, 4 ADDs)
- N^2 reads for `A[i][j]`
- N^2 writes for `C[i][j]`
- AND: N^2 reads for `C[i][j]` (write-allocate)
- Overall comm.: $3N^2 \cdot 8$ Bytes

Arithmetic Intensity: Example 2

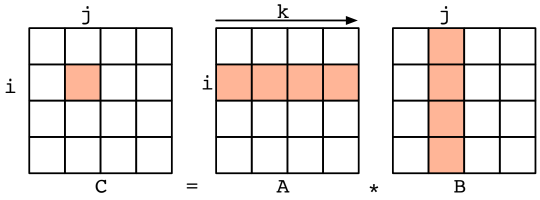
```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    C[i][j] = a *  A[i][j] +
              b * (A[i][j-1] +
                   A[i-1][j] +
                   A[i+1][j] +
                   A[i][j+1]);
  }
}
```



- **6 FLOPs** per iteration (2 MULs, 4 ADDs)
- N^2 reads for `A[i][j]`
- N^2 writes for `C[i][j]`
- AND: N^2 reads for `C[i][j]` (write-allocate)
- Overall comm.: $3N^2 \cdot 8$ Bytes
- AI $= (6N^2$FLOPs$)/(24N^2$Bytes$)$
  AI $= 1/4$ **FLOPs/Byte**

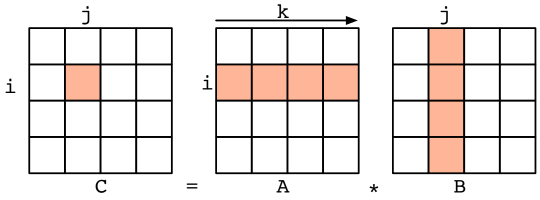
Arithmetic Intensity: Example 3

```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    for(k=0; k<N; k++) {
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
}
```



Arithmetic Intensity: Example 3

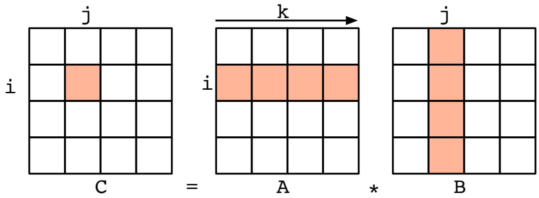
```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    for(k=0; k<N; k++) {
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
}
```



- $2N^3$ **FLOPs**

Arithmetic Intensity: Example 3

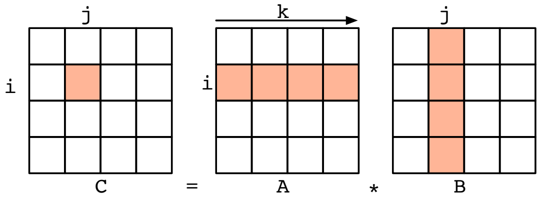
```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    for(k=0; k<N; k++) {
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
}
```



- $2N^3$ **FLOPs**
- $3N^2$ reads for `A`, `B`, `C`
- $1N^2$ writes for `C`

Arithmetic Intensity: Example 3

```
for(i=0; i<N; i++) {
  for(j=0; j<N; j++) {
    for(k=0; k<N; k++) {
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
}
```



- $2N^3$ **FLOPs**
- $3N^2$ reads for `A`, `B`, `C`
- $1N^2$ writes for `C`
- AI $= 2N^3$ FLOPs / $32N^2$ Bytes
  AI $= N/16$ **FLOPs/Byte**
- note: simplistic analysis (in practice: cache lines will be evicted)

Arithmetic Intensity: Example 4

It's your turn: What is the AI of the following kernel?

```
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

Arithmetic Intensity: Example 4

It's your turn: What is the AI of the following kernel?

```
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

- **2 FLOPs**

Arithmetic Intensity: Example 4

It's your turn: What is the AI of the following kernel?

```
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

- **2 FLOPs**
- read 2 doubles, write 1 double: **24 Bytes**

Arithmetic Intensity: Example 4

It's your turn: What is the AI of the following kernel?

```
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

- **2 FLOPs**
- read 2 doubles, write 1 double: **24 Bytes**
- AI $= 2/24 = 1/12$ FLOPs/Byte

Arithmetic Intensity: Overview

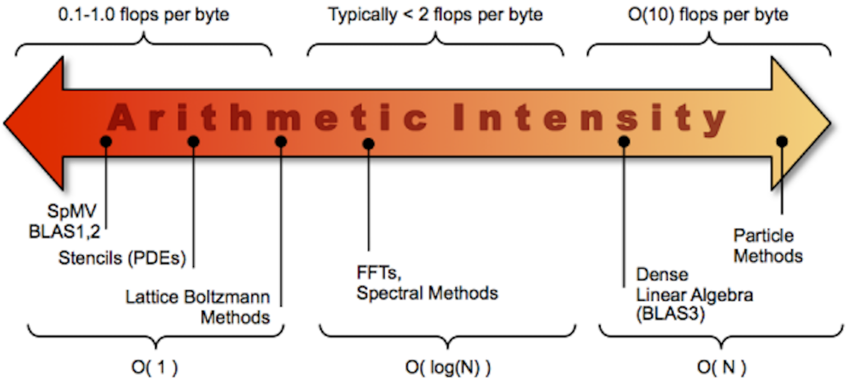


image source:

https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/roofline/introduction/

Determine Arithmetic Intensity in Practice

AI in Practice I

- usually **code** far too **complex** for analytical estimation of AI
  - e.g. libraries
  - e.g. nested pieces of code
  - e.g. late binding, command-line parameters (conditionals)
- solution: use **hardware performance counters** to **determine**
  - total **number of FLOPs**
  - total **communication volume**
- several **tools** available, e.g.,
  - `LIKWID` (Uni Erlangen), `https://github.com/RRZE-HPC/likwid`
  - `PAPI` (UTK), `http://icl.utk.edu/papi/`
  - Linux `perf`

AI in Practice II

```
temp = 0.0;
for(i=0; i<N; i++) {
  temp += a[i] * a[i];
}
magnitude = sqrt(temp);
```

- analytical analysis revealed an **AI of 0.25 FLOPs/Byte**

AI in Practice III

```
hunold@mars:~/exp$ /opt/likwid/bin/likwid-perfctr -m -C 0
  -g MEM ./src/roofline_ex1_dot 100000000
-----------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E7- 8850  @ 2.00GHz
CPU type:       Intel Westmere EX processor
CPU clock:      2.00 GHz
-----------------------------------------------------------
+------------------------------------+-----------+
|               Metric               |  Core 0   |
+------------------------------------+-----------+
|        Runtime (RDTSC) [s]         |    0.7397 |
|        Runtime unhalted [s]        |    0.7359 |
|            Clock [MHz]             | 2000.0634 |
|                CPI                 |    0.7359 |
|  Memory read data volume [GBytes]  |    0.8019 |
| Memory write data volume [GBytes]  |    0.0252 |
|    Memory data volume [GBytes]     |    0.8271 |
+------------------------------------+-----------+
```

- **measure communication volume** with LIKWID
- measured total **communication volume**: 0.83 GBytes

AI in Practice IV

```
hunold@mars:~$ /opt/likwid/bin/likwid-perfctr -m -C 0
  -g FLOPS_DP ./src/roofline_ex1_dot 100000000
-------------------------------------------------------------
CPU name:    Intel(R) Xeon(R) CPU E7- 8850  @ 2.00GHz
CPU type:    Intel Westmere EX processor
+-----------------------+-----------+
|        Metric         |   Core 0  |
+-----------------------+-----------+
|  Runtime (RDTSC) [s]  |    0.7468 |
|      DP MFLOP/s       |  267.8271 |
|    Packed MUOPS/s     |         0 |
|    Scalar MUOPS/s     |  267.8271 |
|      SP MUOPS/s       |         0 |
|      DP MUOPS/s       |  267.8271 |
+-----------------------+-----------+
```

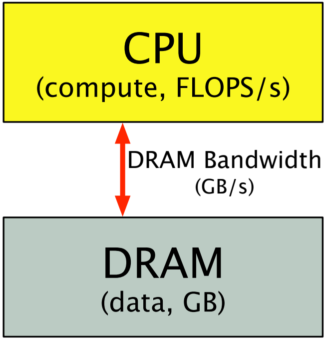
- **measure number of executed FLOPs**
- 267.8 MFLOPs/s $\cdot$ 0.75 s ≈ 200 MFLOPs $= 0.2$ GFLOPs
- **overall AI**

  $$\textbf{AI} = \frac{0.2\,\textbf{GFLOPs}}{0.82\,\textbf{GBytes}} = 0.24\ \textbf{FLOPs/Byte}$$

Eventually
The Roofline Model

- **hope** to always **attain peak performance (FLOPs/s)**
- BUT **performance limited** by
  - **finite reuse** of data
  - **maximum bandwidth** available
- (minimum) **run-time of kernel** is bounded by

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$



Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

- divide by #FLOPs

$$\frac{Time}{\#FLOPs} = \max \begin{cases} \frac{1}{PeakGFlop/s} \\ \frac{\#Bytes/\#FLOPs}{PeakGB/s} \end{cases}$$

Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

- divide by #FLOPs

$$\frac{Time}{\#FLOPs} = \max \begin{cases} \frac{1}{PeakGFlop/s} \\ \frac{\#Bytes/\#FLOPs}{PeakGB/s} \end{cases}$$

- reciprocating

$$\frac{\#FLOPs}{Time} = \min \begin{cases} PeakGFlop/s \\ \frac{\#FLOPs}{\#Bytes} \cdot PeakGB/s \end{cases}$$

Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \dfrac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \dfrac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

- divide by #FLOPs

$$\frac{Time}{\#FLOPs} = \max \begin{cases} \dfrac{1}{PeakGFlop/s} \\ \dfrac{\#Bytes/\#FLOPs}{PeakGB/s} \end{cases}$$

- reciprocating

$$\underbrace{\frac{\#FLOPs}{Time}}_{\text{GFLOPs/s}} = \min \begin{cases} PeakGFlop/s \\ \underbrace{\dfrac{\#FLOPs}{\#Bytes}}_{\text{Arithmetic Intensity}} \cdot PeakGB/s \end{cases}$$

Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

- divide by #FLOPs

$$\frac{Time}{\#FLOPs} = \max \begin{cases} \frac{1}{PeakGFlop/s} \\ \frac{\#Bytes/\#FLOPs}{PeakGB/s} \end{cases}$$

- reciprocating

$$\textbf{Attainable GFLOPs/s} = \min \begin{cases} PeakGFlop/s \\ \textbf{Arithm. Intensity} \cdot PeakGB/s \end{cases}$$

Roofline Model: The Hunt for GFLOPs/s

- we have

$$Time = \max \begin{cases} \frac{\#FLOPs \text{ (program)}}{PeakGFlop/s \text{ (machine)}} \\ \frac{\#Bytes \text{ (program)}}{PeakGB/s \text{ (machine)}} \end{cases}$$

- divide by #FLOPs

$$\frac{Time}{\#FLOPs} = \max \begin{cases} \frac{1}{PeakGFlop/s} \\ \frac{\#Bytes/\#FLOPs}{PeakGB/s} \end{cases}$$

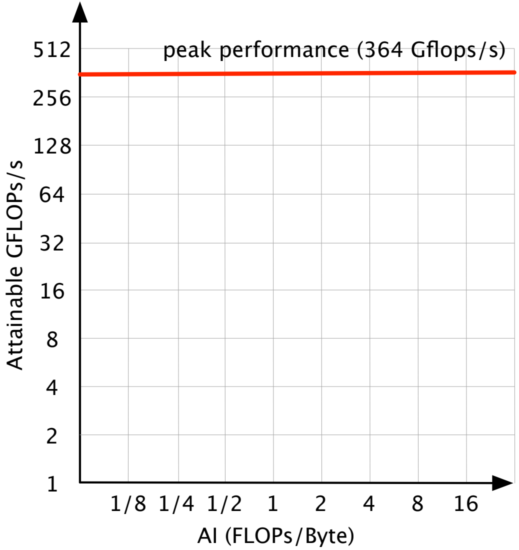
- reciprocating

$$\textbf{Attainable GFLOPs/s} = \min \begin{cases} PeakGFlop/s \\ \textbf{Arithm. Intensity} \cdot PeakGB/s \end{cases}$$

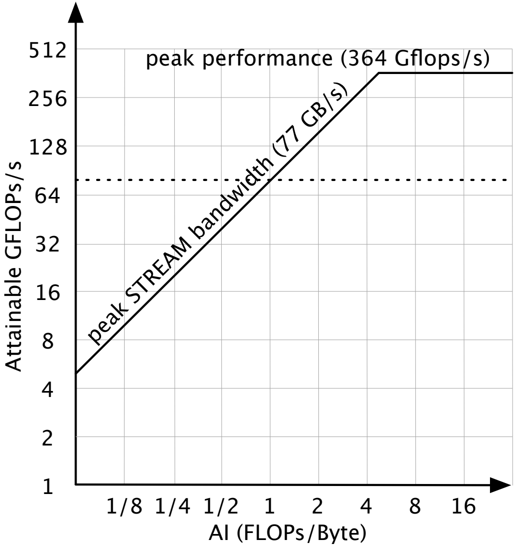
Target Performance Metric

The Roof

Roofline Model: Basics

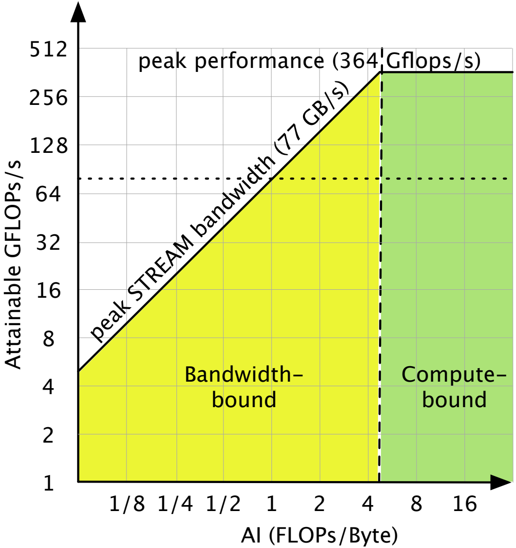


- **saturn**: 4 x Opteron 6168@1.9 GHz, 12 cores each
- **peak compute** speed: 364.8 GFLOPs/s (computed from specs, will be discussed in a minute)



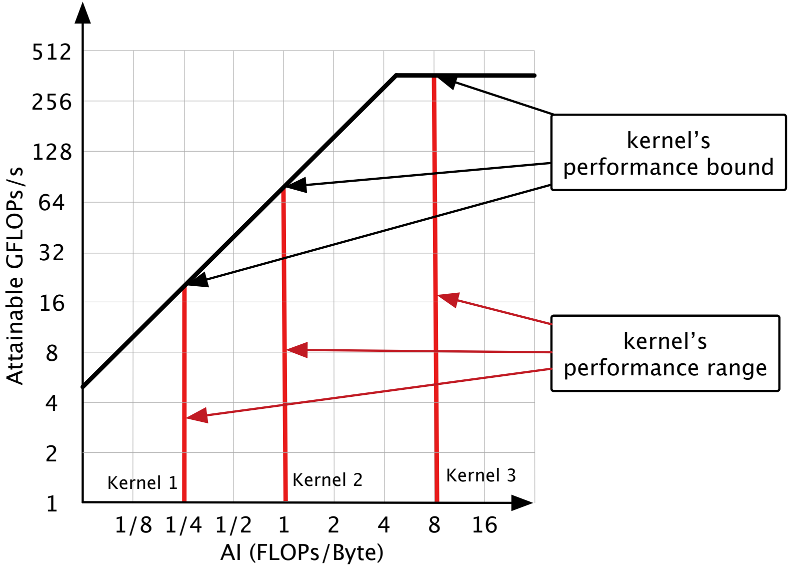
- **saturn**: 4 x Opteron 6168@1.9 GHz, 12 cores each
- **peak compute** speed: 364.8 GFLOPs/s (computed from specs, will be discussed in a minute)
- **peak DRAM bandwidth**: 77 GB/s
- **attainable GFLOPs/s** is **bounded by**
  - **AI · Peak GB/s**
  - AI=1/2 FLOPs/Byte $\rightarrow$ 1/2 FLOPS/Byte · 77 GB/s = 38.5 GFLOPs/s

Roofline Model: Basics



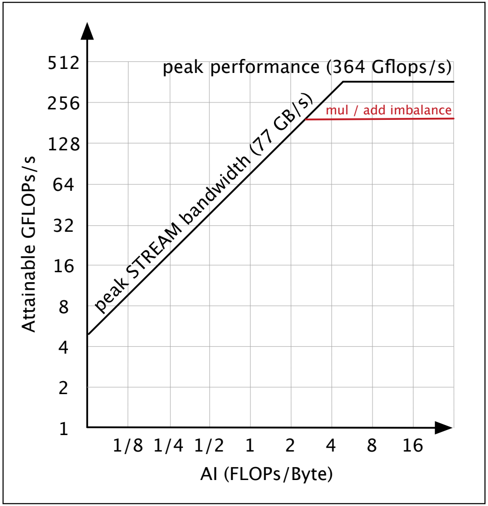
Key Facts

- **log log** scale
- **ridgepoint**: where roof meets diagonal
- roofline has to be created only once
  - **machine-dependent**
  - independent of kernel


512
256
128
64
32
16
8
4
2
1
Attainable GFLOPs/s
1/8
1/4
1/2
1
2
4
8
16
AI (FLOPs/Byte)
Kernel 1
Kernel 2
Kernel 3
kernel's performance bound
kernel's performance range

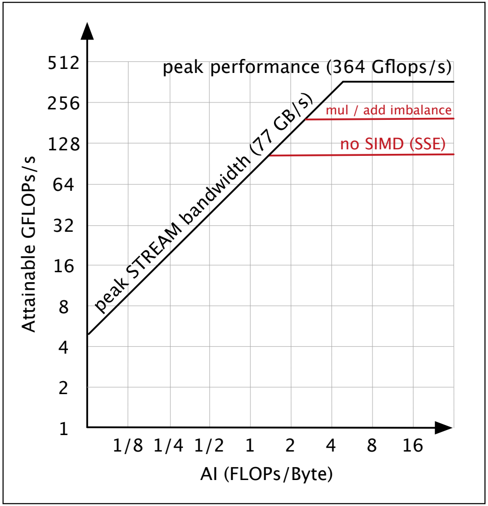
Ceilings in the Roofline Model

Reality: most codes are **unable to attain peak performance**

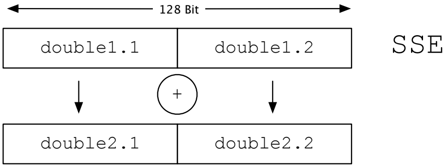


- **imbalance of ADDs and MULs**
  - fused-multiply-add (FMA) not applicable
    FMA3 (Intel): `a = a * c + b`
  - or processor has distinct execution units for ADD and MUL
  - result: halves attainable performance

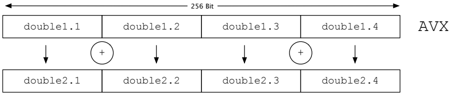
Reality: most codes are **unable to attain peak performance**



- code **not vectorized**
  - SSE (128 bit, 2 double FP per cycle): performance / 2

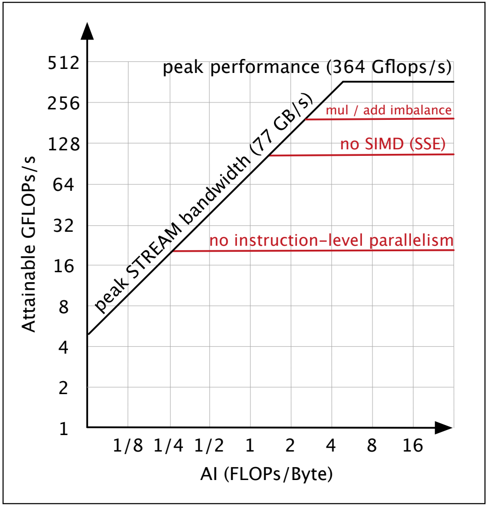


  - AVX (256 bit, 4 double FP per cycle): performance / 4



Ceilings in the Roofline Model

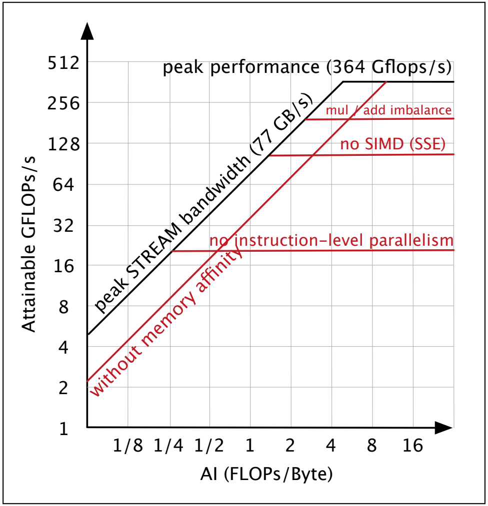
Reality: most codes are **unable to attain peak performance**



- no **instruction level parallelism**
  - instructions have **specific latency** on architecture
  - e.g. 4 cycle latency for ADD and MUL on AMD Magny Cours
  - result: cannot hide latency of operations (performance / 4)
  - **thread-level parallelism** (TLP) only
  - 48 cores: 1.9 GHz / 4 cycles $\cdot$ 48 $=$ 22.8 GFLOPs/s
  - solution: **unroll loops**

Ceilings in the Roofline Model

Reality: most codes are **unable to attain peak performance**



- code **not NUMA-aware**
  - physical addresses placed to remote memory controllers
  - requires cross-traffic between NUMA-nodes (e.g. sockets)
  - result: less bandwidth

Obtaining a Roofline Model

1) **Compute Peak FLOP Rate**
2) **Measure Peak Memory Bandwidth**

Know your Processor!

tesla: 2 x AMD EPYC 7262 I

Detecting Processor Model

```
hunold@tesla:~$ lscpu  |grep Model
Model:                           49
Model name:                      AMD EPYC 7262 8-Core Processor

hunold@tesla:~$ lscpu  |grep Socket
Socket(s):                       2
```

tesla: 2 x AMD EPYC 7262 II

AMD EPYC™ 7262

General Specifications

Platform: Server	**Product Family:** AMD EPYC™	**Product Line:** AMD EPYC™ 7002 Series
# of CPU Cores: 8	**# of Threads:** 16	**Max. Boost Clock:** Up to 3.4GHz
Base Clock: 3.2GHz	**L3 Cache:** 128MB	**1kU Pricing:** 505 USD
Default TDP: 155W	**CPU Socket:** SP3	**Socket Count:** 1P/2P

Connectivity

PCI Express® Version: x128	**System Memory Type:** DDR4	**Memory Channels:** 8
System Memory Specification: Up to 3200MHz	**Per Socket Mem BW:** 204.8 GB/s	

https://www.amd.com/en/products/cpu/amd-epyc-7262

- AMD EPYC 7002 Series
- Codename: "Rome"
- Microarchitecture: Zen 2

Determining Peak FLOPS

AMD EPYC 7002

- 256-bit AVX2 instructions (supports FMA)
- 2 × 256-bit FP units per CPU core
  - 2 × 8 FLOPs (see `https://en.wikichip.org/wiki/flops`)
- in total per core: 16 double-precision FLOPS per cycle

tesla

- our machine has 2 × 8-core EPYC 7002 series processors
- max FLOPS per cycle
  - 16 × 16 double-precision FLOPS per cycle
  - 256 double-precision FLOPS per cycle
- theoretical peak performance: 256 FLOPS/cycle × 3.2 GHz = 819.2 GFLOPS/s

Determining Peak DRAM Bandwidth I

- measuring bandwidth using **STREAM benchmark** (John McCalpin aka "Dr. Bandwidth")
- `http://www.cs.virginia.edu/stream/`

Determining Peak DRAM Bandwidth II

```
hunold@tesla:~/tmp/stream$ gcc -DSTREAM_ARRAY_SIZE=1000000000 -Ofast -march=native -mcmodel=medium -fopenmp
      stream.c -o  stream.epyc
hunold@tesla:~/tmp/stream$ ./stream.epyc
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 1000000000 (elements), Offset = 0 (elements)
Memory per array = 7629.4 MiB (= 7.5 GiB).
Total memory required = 22888.2 MiB (= 22.4 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 32
Number of Threads counted = 32
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           78814.7     0.208977     0.203008     0.214503
Scale:          49384.4     0.332616     0.323989     0.344185
Add:            54657.5     0.450318     0.439098     0.458451
Triad:          53882.3     0.448303     0.445415     0.452282
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Determining Peak DRAM Bandwidth III

- hence, we get a bandwidth of roughly $80\,\mathrm{GB\,s^{-1}}$
- didn't AMD promise us $204\,\mathrm{GB\,s^{-1}}$ per socket, how come?
- $204\,\mathrm{GB\,s^{-1}}$ is the theoretical bandwidth if all 8 memory channels can be used

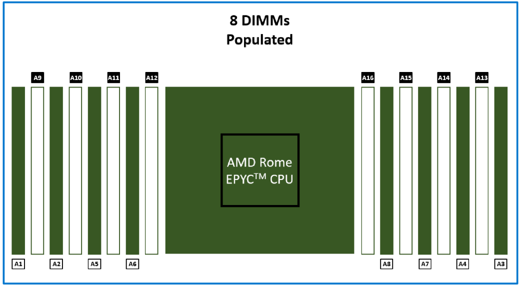


Figure 8: Eight DIMMs are populated in a balanced configuration, producing the highest memory bandwidth while at a lower capacity than sixteen

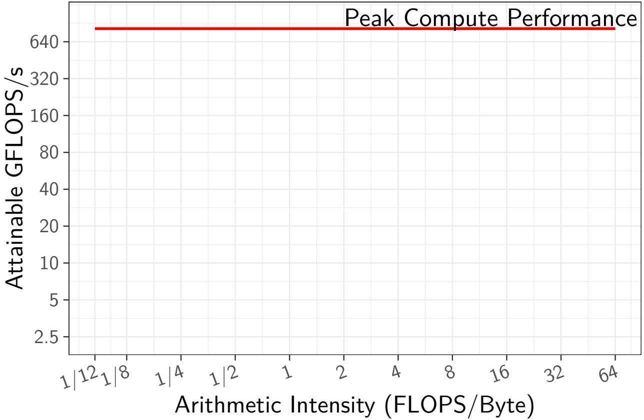
`https://dl.dell.com/manuals/common/dellemc-balanced-memory-2ndgen-amd-epyc-poweredge.pdf`

- we only have 2 DIMMs per socket, i.e., 2 memory channels with 1 DIMM (single rank)
- thus, we can only leverage 2/8 memory channels, and so expect a bandwidth of 25% of the max. peak
- per socket: $204\,\mathrm{GB\,s^{-1}} \times 0.25 \approx 50\,\mathrm{GB\,s^{-1}}$
- for 2 sockets: we expect about $100\,\mathrm{GB\,s^{-1}}$ (max.)
- we got about $80\,\mathrm{GB\,s^{-1}}$

Putting it all together:

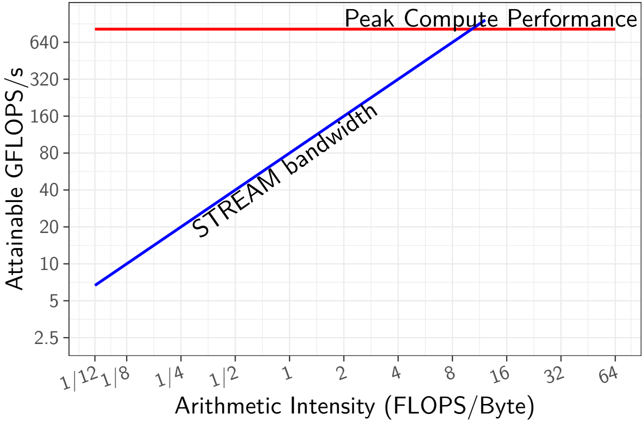
The Roofline Model for `tesla`

Roofline Model for `tesla`



- compute peak: **819 GFLOPs/s**

Roofline Model for `tesla`

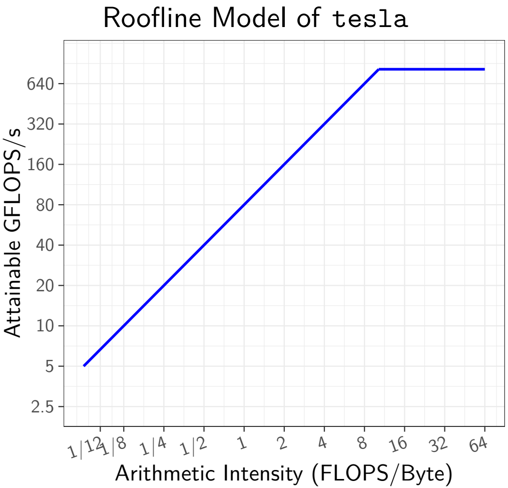


- compute peak: **819 GFLOPs/s**
- bandwidth: **80 GB/s**

Roofline Example: Vector Computation

```
#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

- **2 FLOPs**
- read 2 doubles, write 1 double: **24 Bytes**
- AI = 2/24 = 1/12 FLOPS/Byte

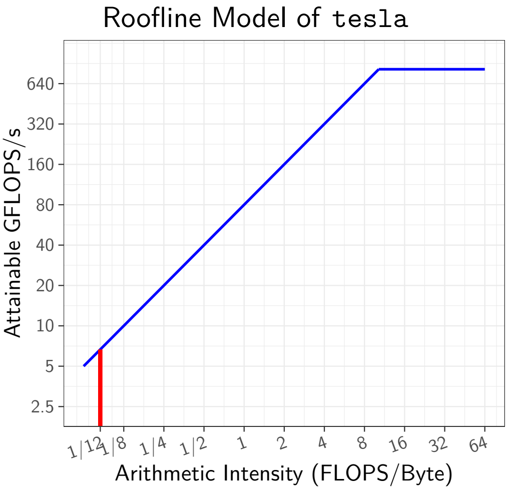


Roofline Example: Vector Computation

```
#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

- **2 FLOPs**
- read 2 doubles, write 1 double: **24 Bytes**
- AI $= 2/24 = 1/12$ FLOPS/Byte

- **bandwidth-bound**
- $1/12$ FLOPs/Byte $\cdot$ 80 GB/s $= 6.6$ GFLOPs/s attainable



Vector Computation: Version 1

```
for(long i=0; i<N; i++) {
  a[i] = i%1000;
  b[i] = i%1000;
}

#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

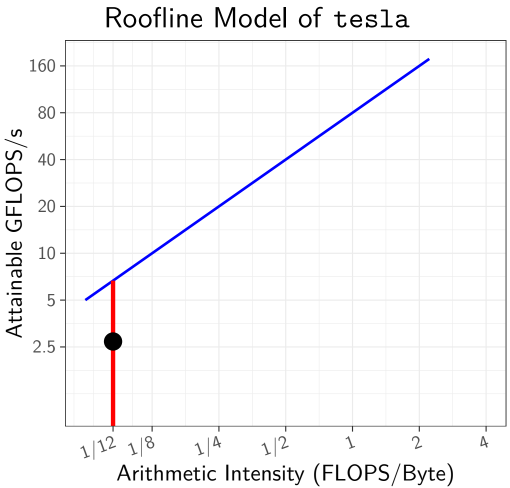
Vector Computation: Version 1

```
for(long i=0; i<N; i++) {
  a[i] = i%1000;
  b[i] = i%1000;
}

#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] += a[i]*b[i];
}
```

```
gcc -fopenmp -O0 -o roofline_dot_opt0
    roofline_dot_opt0.c

hunold@tesla:~$ ./roofline_dot_opt0 400000000
N=400000000
time: 0.294417
GFLOPS/s: 2.717239
```



Vector Computation: Version 2

- **NUMA-aware** initialization (first-touch policy)

```
// parallel initialization is IMPORTANT
#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] = i%1000;
  b[i] = i%1000;
}

#pragma omp parallel for
for(long i=0; i<N; i++)
  a[i] += a[i]*b[i];
```

```
gcc -fopenmp -O0 -o roofline_dot_opt1
     roofline_dot_opt1.c
```

Vector Computation: Version 2

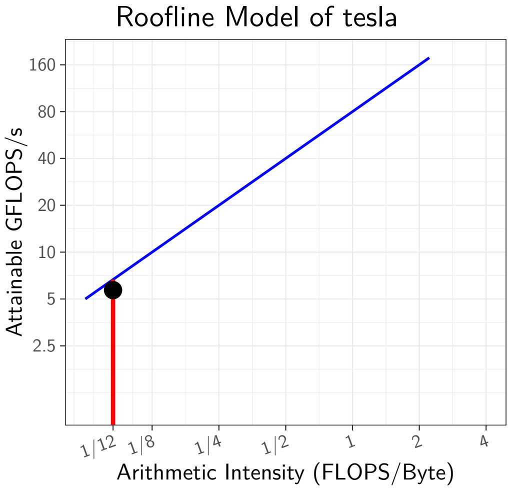
- **NUMA-aware** initialization (first-touch policy)

```
// parallel initialization is IMPORTANT
#pragma omp parallel for
for(long i=0; i<N; i++) {
  a[i] = i%1000;
  b[i] = i%1000;
}

#pragma omp parallel for
for(long i=0; i<N; i++)
  a[i] += a[i]*b[i];
```

```
gcc -fopenmp -O0 -o roofline_dot_opt1
     roofline_dot_opt1.c
```

```
hunold@tesla:~$ ./roofline_dot_opt1 400000000
N=400000000
time: 0.145033
GFLOPS/s: 5.515976
```



Vector Computation: Version 3

- same code as version 2
- `-march=native` enables **AVX (Advanced Vector Extensions)**
- e.g. vfmadd132pd: Fused Multiply-Add of Packed Double-Precision Floating-Point Values

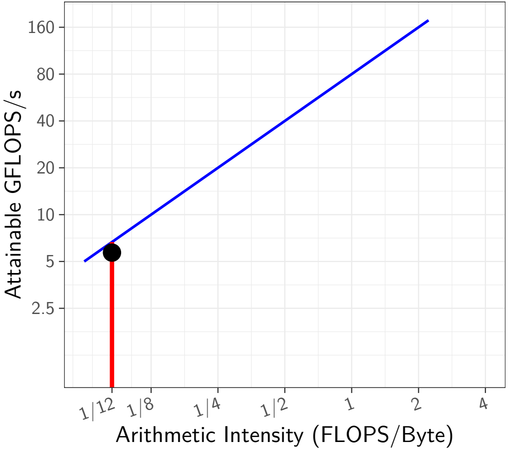
```
$ gcc -fopenmp -O3 -march=native -S roofline_dot_opt2.c
$ cat roofline_dot_opt2.s |grep fma
        vfmadd132pd     (%r9,%rcx), %ymm0, %ymm0
        vfmadd132pd     (%r8,%rax), %xmm0, %xmm0
        vfmadd132sd     (%r8,%rdx), %xmm0, %xmm0
        vfmadd132sd     (%r8,%rdx,8), %xmm0, %xmm0
```

Vector Computation: Version 3

- same code as version 2
- `-march=native` enables **AVX (Advanced Vector Extensions)**
- e.g. vfmadd132pd: Fused Multiply-Add of Packed Double-Precision Floating-Point Values

```
$ gcc -fopenmp -O3 -march=native -S roofline_dot_opt2.c
$ cat roofline_dot_opt2.s |grep fma
        vfmadd132pd     (%r9,%rcx), %ymm0, %ymm0
        vfmadd132pd     (%r8,%rax), %xmm0, %xmm0
        vfmadd132sd     (%r8,%rdx), %xmm0, %xmm0
        vfmadd132sd     (%r8,%rdx,8), %xmm0, %xmm0
```

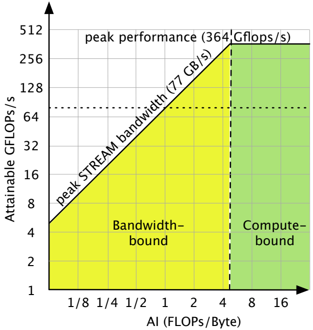
```
hunold@tesla:~$ ./roofline_dot_opt2 400000000
N=400000000
time: 0.137945
GFLOPS/s: 5.799422
```



- no improvement (or very tiny, code was already saturating bandwidth)

Take Home Message

- Roofline model helps to **estimate performance potential** of computational **kernels** on **multi-core machines**
- visual (graphical) **2D performance model** (log-log scale)
  - **y axis**: attainable GFLOPs/s
    - could also be any other metric: INTs/s
  - **x axis**: arithmetic intensity of a computational kernel
    - identify kernels that are **bandwidth-** or **compute-bound**
  - **independent** of computational **kernel**
  - **dependent** on **machine**
- Roofline models
  - can also be generated for **GPUs** (other accelerators)
  - can also consider **cache levels** (see Intel Advisor)



References

[Tho11] M. Thomadakis. "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms". In: *JFE Technical Report* (Mar. 2011).

[WWP09] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), p. 65.

Increase ILP

```
for(i=..) {
  sum += a*b[i];
}
```

- **loop unrolling**

```
for(i=..) {
  sum += a*b[i];
  sum += a*b[i+1];
  sum += a*b[i+2];
  sum += a*b[i+3];
}
```

- adds to `sum` will be serialized
- does not increase ILP
- reduces loop overhead

```
for(i=..) {
  sum0 += a*b[i];
  sum1 += a*b[i+1];
  sum2 += a*b[i+2];
  sum3 += a*b[i+3];
}
sum += sum0 + sum1 ..
```

- increases ILP
- should outperform version on the left

Cache Architecture: Write-allocate Policy

Cache

- sits **between CPU and main memory** (MM)
- holds **copies** of chunks of **data from MM** (cache lines/blocks)

General Scenario: **Cache-miss**

- **data** that should be read or written from/to the cache is **missing**

Cache Architecture: Write-allocate Policy

Cache

- sits **between CPU and main memory** (MM)
- holds **copies** of chunks of **data from MM** (cache lines/blocks)

General Scenario: **Cache-miss**

- **data** that should be read or written from/to the cache is **missing**

Specific Scenario: **Write-miss** (data not in cache) and **Write-allocate** policy (processor)

1. **allocate** cache line and **bring it into** the **cache**
2. write (write-back) to cache (hope that subsequent writes go to cache)