Peak Performance, Memory Wall, Machine Balance, Roofline Diagrams, ...

December 13, 2023

Role-Playing Game

You are the CNRS and you have just had a delivery

- 1528 nodes
- 2 $\times$ Xeon Gold 6248
  - “Cascade Lake”
  - 20 cores @ 2.5 Ghz
- 192 Go RAM/node
- Omnipath 100Gbit/s

Legitimate question

How much faster than the previous machine is this going to be?

Peak performance

Definition

Maximal number of FLOPS that the hardware can do (in theory)

- # Nodes
- SMP ?
- CPU frequency
- # Cores
- SIMD units? (vector width ?)
- Instruction-level parallelism (multiple ALUs?)
- Fused Multiply-Add

Easy Case

Turing

Easy Case
Turing

- Nodes: 6144
- $1\times$ PowerPC A2 @ 1.6Ghz
- Cores: 16
- SIMD units: 256-bit (4 $\times$ double)
- Fused Multiply-Add : yes
- 1 instruction/cycle maximum

Result

$6144 \times 1.6e9 \times 16 \times 4 \times 2 = 1144$ teraFLOPS (with `double`)

Hard Case

Jean-Zay

Hard Case

Jean-Zay

- Nodes: 1528
- $2\times$ Xeon Gold 6248 @ [it depends] GHz
- Cores : 20
- SIMD units: 512-bits (8 $\times$ double)
- Fused Multiply-Add : yes
- instruction/cycle: 2 $\times$ FMA
  - Some (cheap) Xeons: only one
  - Some (expensive) Xeons: 2

CPU Frequency Scaling

Thermal Envelope Limitations

The frequency at which a core runs depends on:

- The kind of instructions it executes
- What the other cores are doing
- The quality of its hardware components

For the Intel Xeon Gold 6248 (in GHz):

Mode	Base	Turbo with *x* active cores					
		1-2	3-4	5-8	9-12	13-16	17-20
normal	2.5	3.9	3.7	3.6	3.6	3.4	3.2
AVX2	1.9	3.8	3.6	3.5	3.4	3.0	2.8
AVX512	1.6	3.8	3.6	3.5	3.0	2.7	2.5

Conclusion (with `double`)

$$\text{scalar} \xrightarrow{\times 2} \text{SSE} \xrightarrow{\times 1.75} \text{AVX-2} \xrightarrow{\times 1.8} \text{AVX-512}$$

Hard Case

Jean-Zay

- Nodes: 1528
- $2\times$ Xeon Gold 6248 @ [it depends] GHz
- Cores : 20
- SIMD units: 512-bits (8 $\times$ double)
- Fused Multiply-Add : yes
- instruction/cycle: 2 $\times$ FMA
  - Some (cheap) Xeons: only one
  - Some (expensive) Xeons: 2

Result

$1528 \times 2.5e9 \times 40 \times 8 \times 2 \times 2 =$4890 teraFLOPS (with `double`)

Public enemy #1
of HPC programming

Public enemy #1
of HPC programming

```
double x = A[i];
```

The “*Memory Wall*”

- **Computing Power** increases
  - Quick increase in FLOP/s
- Speed of memory **does not follow** at the same pace
  - *Less quick* increase in GB/s

Can distinguish

- Compute-bound (or CPU-bound) algorithms
  - limited by peak FLOP/s
- Memory-bound algorithms
  - limited by peak RAM bandwidth (GB/s)

The "*Memory Wall*"

FLOPS $\div$ `[memory bandwidth]`

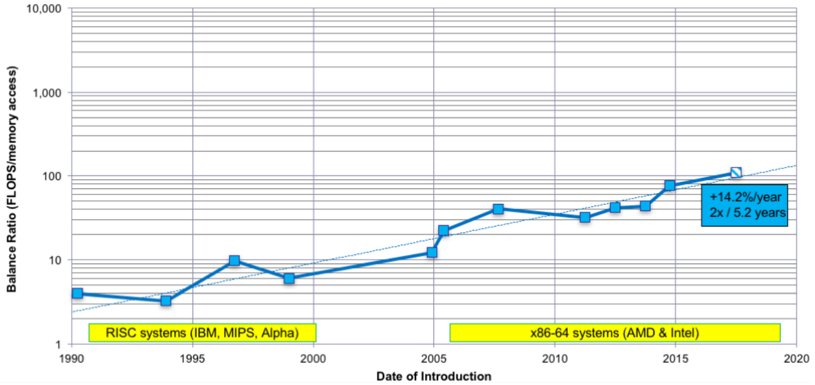


image: John McCalpin

The "*Memory Wall*"

FLOPS ÷ `[memory latency]`

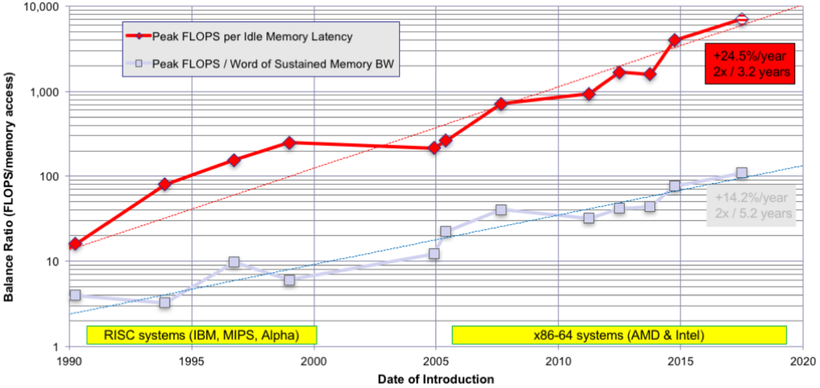


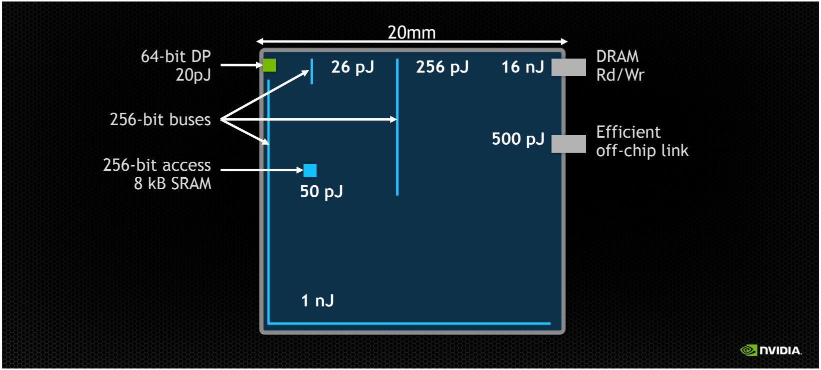
image: John McCalpin

Multicore Horror

STREAM benchmark

Machine	Threads	GB/s	Speedup
Laptop	1	10.9	-
	2	10.9	1
Raspberry 3B+	1	1.8	-
	2	2.3	1.3
	4	2.0	1.1
BlueGene/Q	1	7.5	-
	2	15	2
	4	26.8	3.6
	8	27.9	3.7
	16	28.0	3.7
Cluster node	1	12.7	-
	2	24.6	1.9
	4	47.8	3.7
	8	67.6	5.3
	16	73.4	5.7

Energy Cost of Data Transfers



(image : Bill Dally, NVIDIA, “ the path to exascale”)

On a usual CPU

- Read RAM = $10\times$ FP64 multiplication

Non-contiguous Accesses

T = array of N random integers in $[0; N)$

```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```

In theory

Complexity independent of N

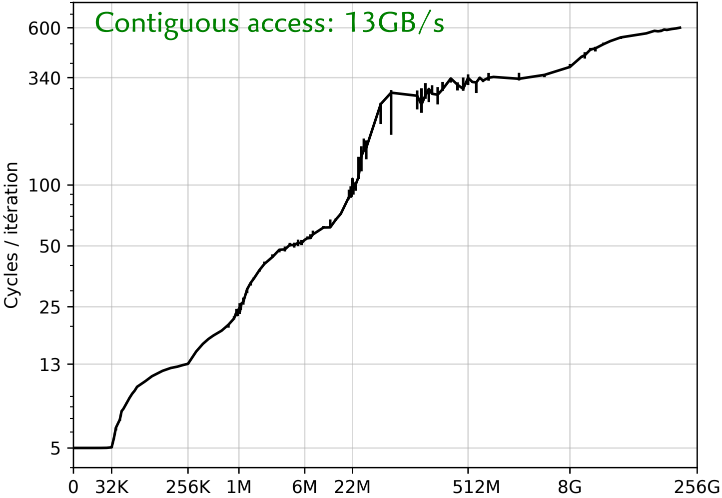
In practice

Exposes memory **latency**

Non-contiguous Accesses

$T =$ array of N random integers in $[0; N)$

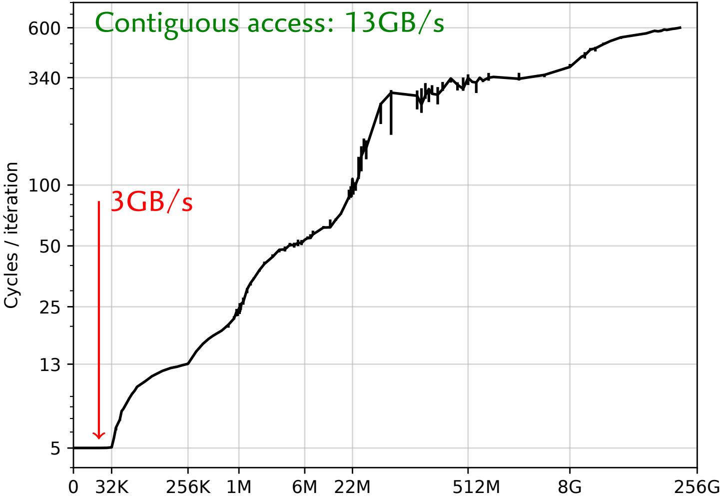
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Non-contiguous Accesses

T = array of N random integers in $[0; N)$

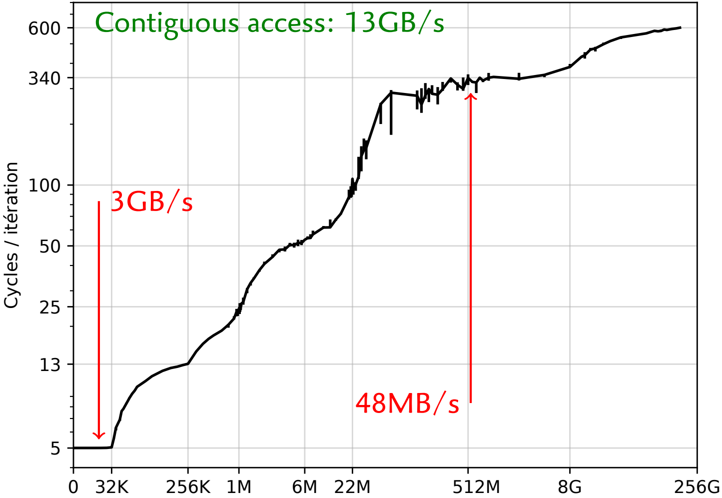
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Non-contiguous Accesses

T = array of N random integers in $[0; N)$

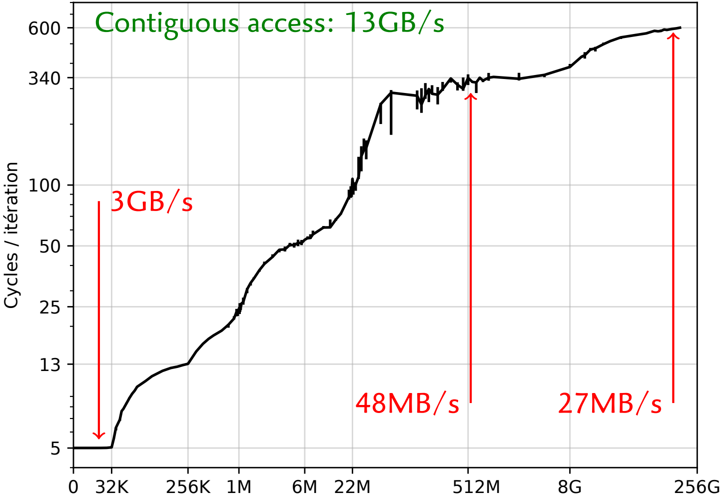
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Non-contiguous Accesses

$T =$ array of N random integers in $[0; N)$

```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



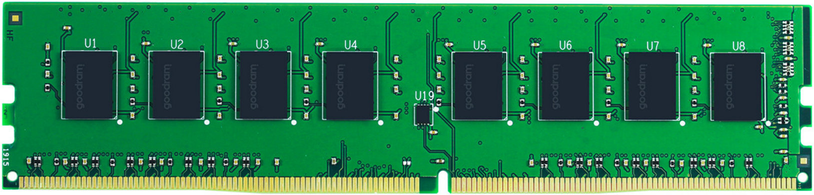
Roadmap

1. The hardware
2. Memory hierarchy (caches)
3. Improving data locality
4. (bonus) Paging-related issues

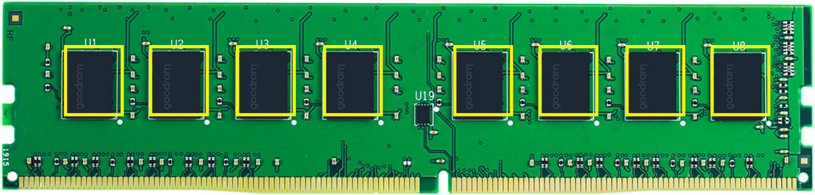
Roadmap

1. The hardware
2. Memory hierarchy (caches)
3. Improving data locality
4. (bonus) Paging-related issues

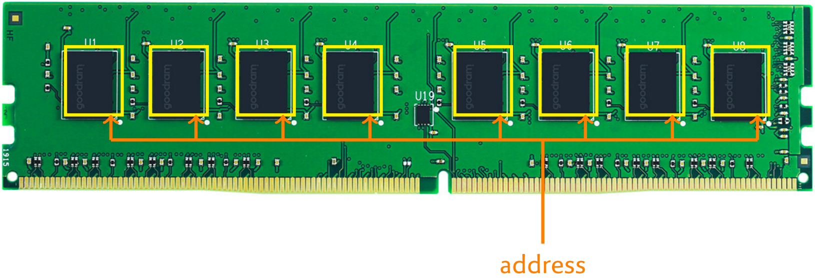
A *DIMM* (a “ memory stick”)



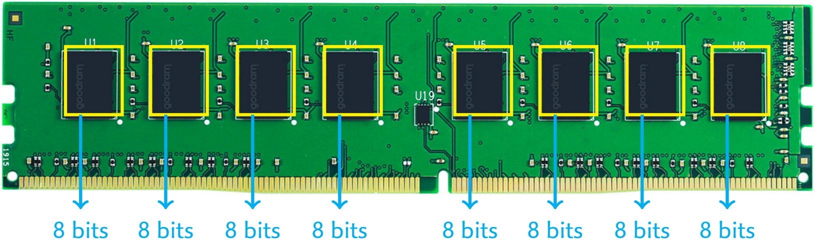
A *DIMM* (a “ memory stick”)



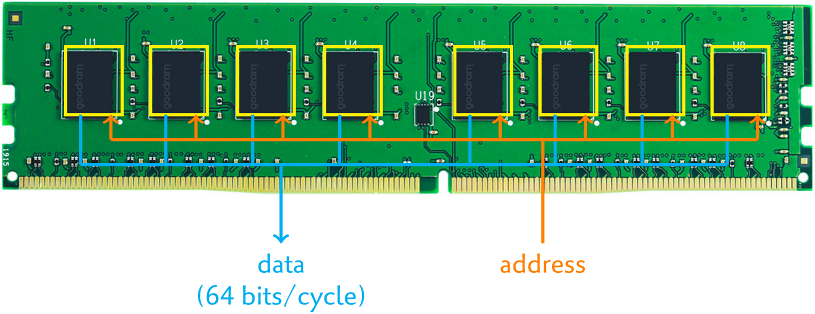
A *DIMM* (a “ memory stick”)



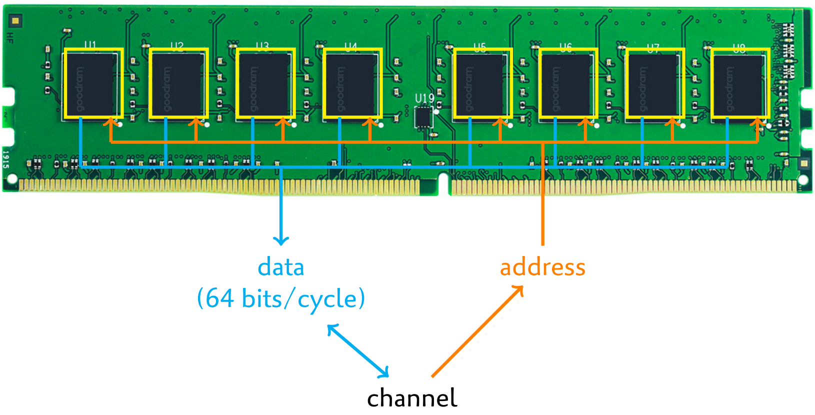
A *DIMM* (a “ memory stick”)



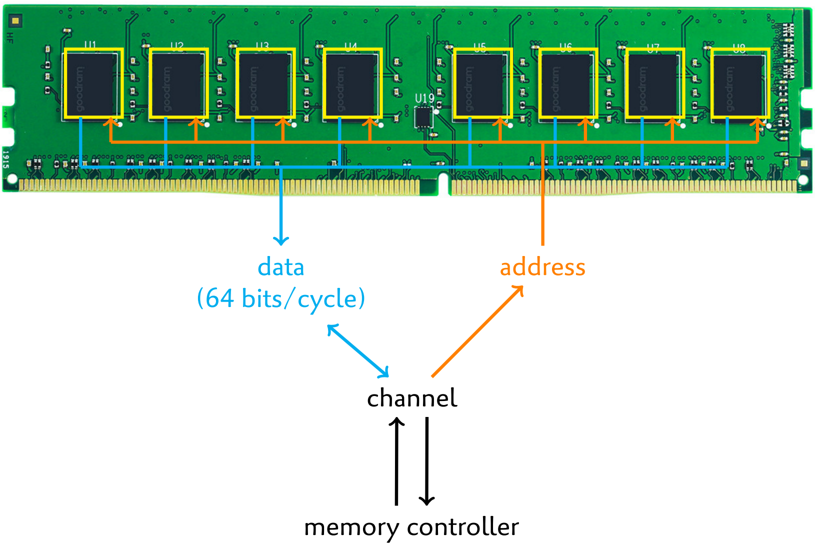
A *DIMM* (a " memory stick")



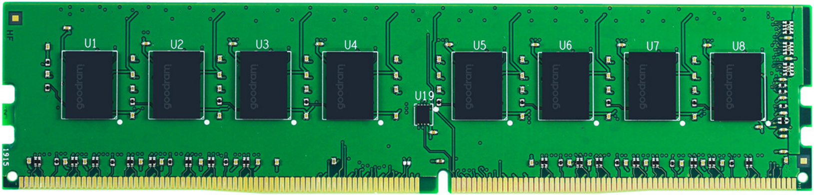
A *DIMM* (a " memory stick")



A *DIMM* (a “ memory stick”)



A *DIMM* (a “ memory stick”)

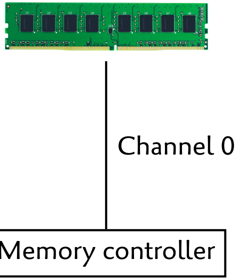


Generation	Year	Mhz	Prefetch	GB/s
DDR	2000	100–200	16	1.6–3.2
DDR2	2003	100–266	32	3.2–8.5
DDR3	2007	100–266	64	6.4–17
DDR4	2014	200–400	64	12.8–25.6
DDR5	2020	200–450	64	25.6–57.6

Memory Channels

More parallelism, more bandwidth

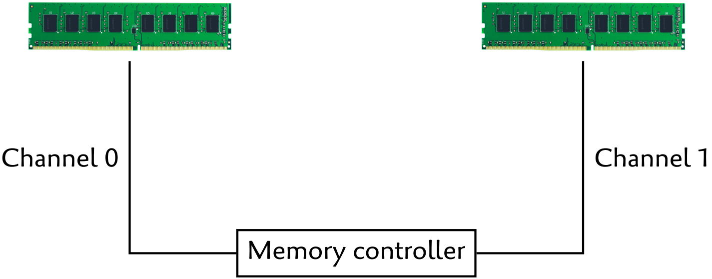
≤ 2000



Memory Channels

More parallelism, more bandwidth

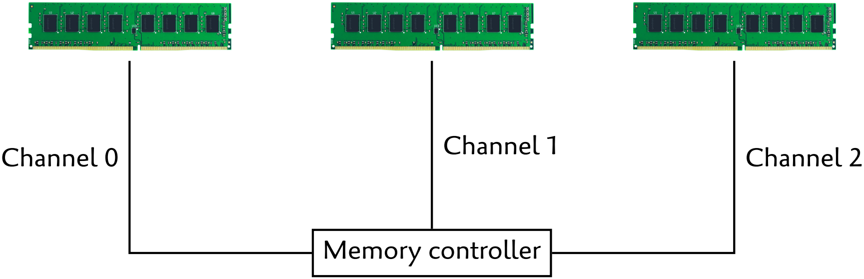
$\geq$ 2000, almost all "consummer" CPUs (Core i3, i5, ...)



Memory Channels

More parallelism, more bandwidth

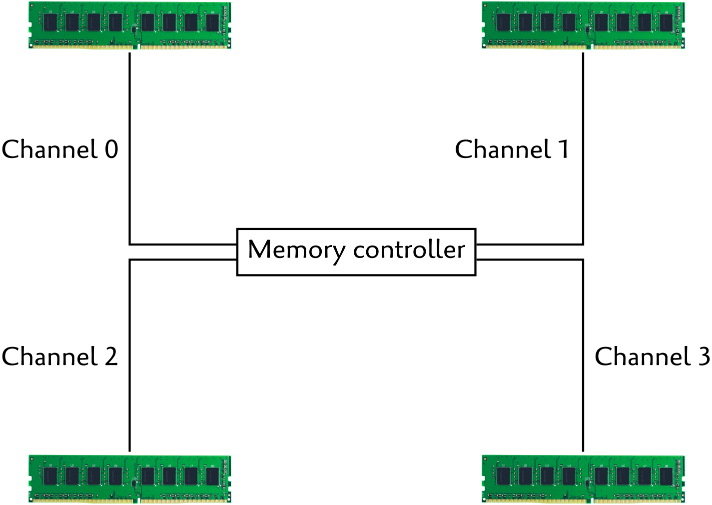
2008, Core i7 920 “ Bloomfield”



Memory Channels

More parallelism, more bandwidth

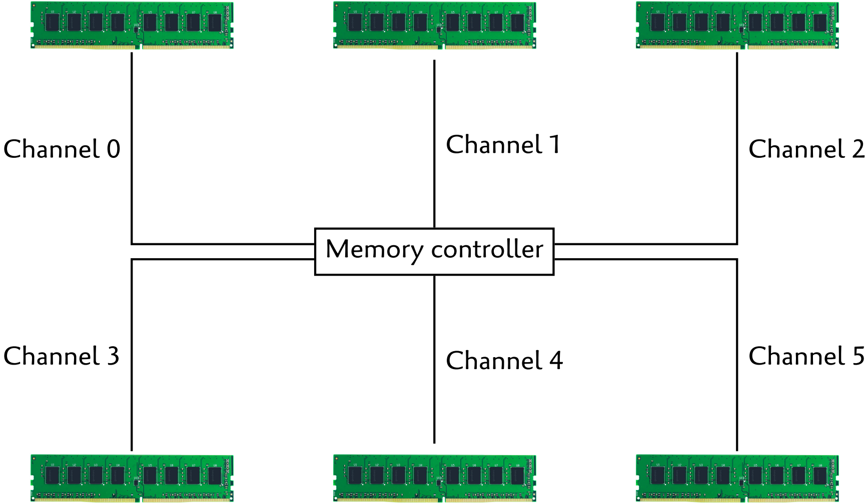
2010, AMD Opteron 6100, AMD Ryzen, Core i7/i9 “X series”



Memory Channels

More parallelism, more bandwidth

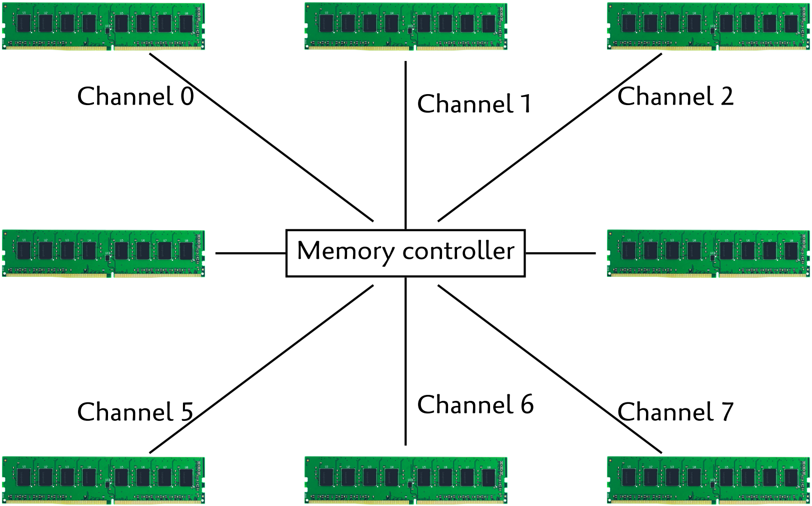
2017, Xeon Scalable (“ Skylake”)



Memory Channels

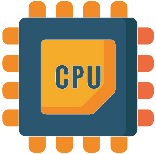
More parallelism, more bandwidth

2019, AMD Epyc, IBM Power9

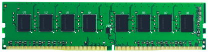


Closer, Faster

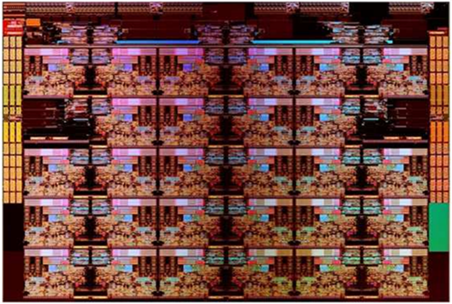
≤ 2008 : controller on the motherboard (" chipset, north bridge")



memory controller

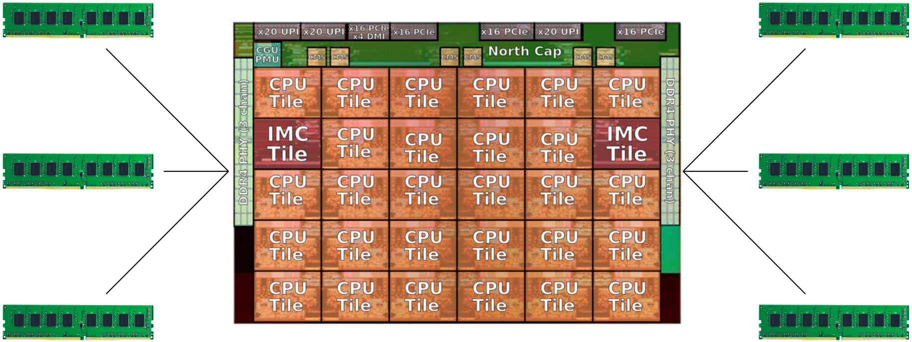


Closer, Faster



Closer, Faster

$\geq$ 2008, controller on the CPU



Roadmap

1. The hardware
2. Memory hierarchy (caches)
3. Improving data locality
4. (bonus) Paging-related issues

Memory Hierarchy

RAM

$\updownarrow$

L4 cache (rare, IBM z15: 960Mo)

$\updownarrow$

L3 cache ($\leq$ 64Mo)

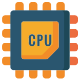
$\updownarrow$

L2 cache (256Ko–1Mo)

$\updownarrow$

L1 cache ($\approx$ 32Ko)

$\updownarrow$



Memory Hierarchy

RAM

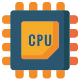
L4 cache (rare, IBM z15: 960Mo)

L3 cache ($\leq$ 64Mo)

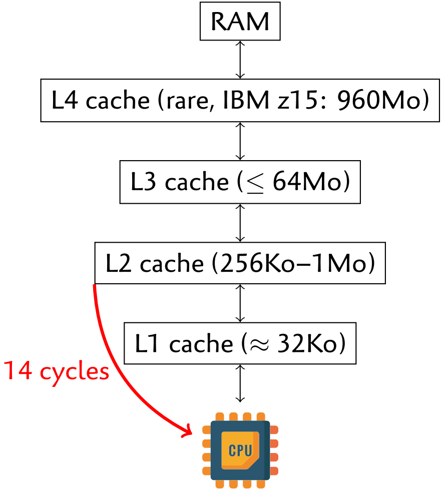
L2 cache (256Ko–1Mo)

L1 cache ($\approx$ 32Ko)

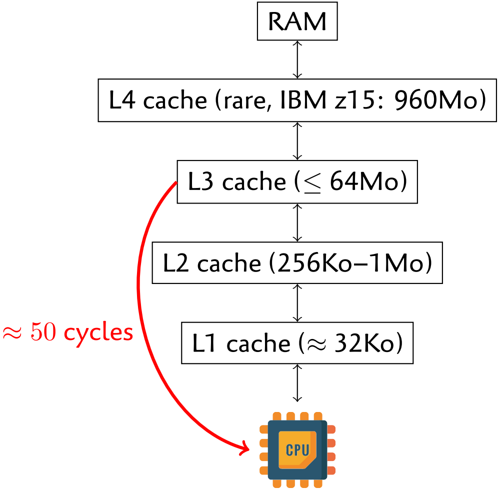
5 cycles



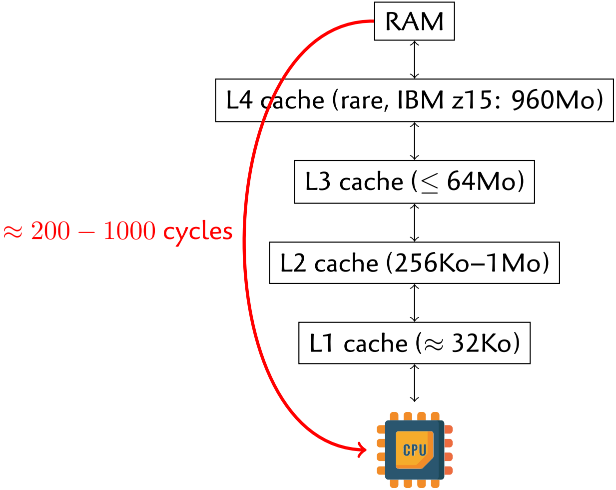
Memory Hierarchy



Memory Hierarchy



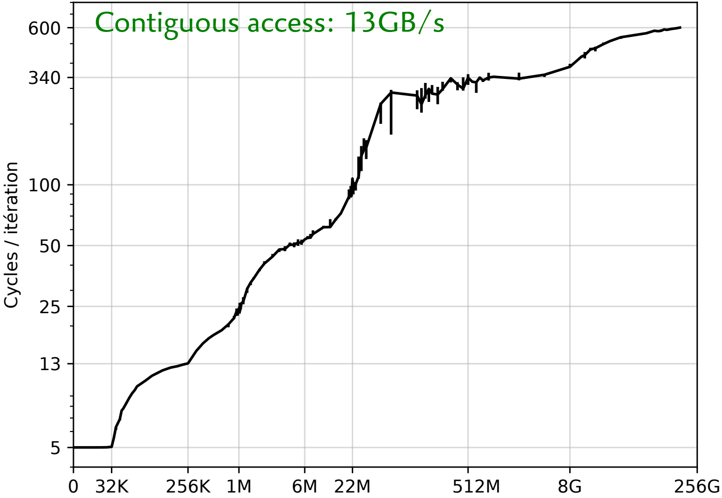
Memory Hierarchy



Effect of the Memory Hierarchy

$T =$ array of N random integers in $[0; N)$

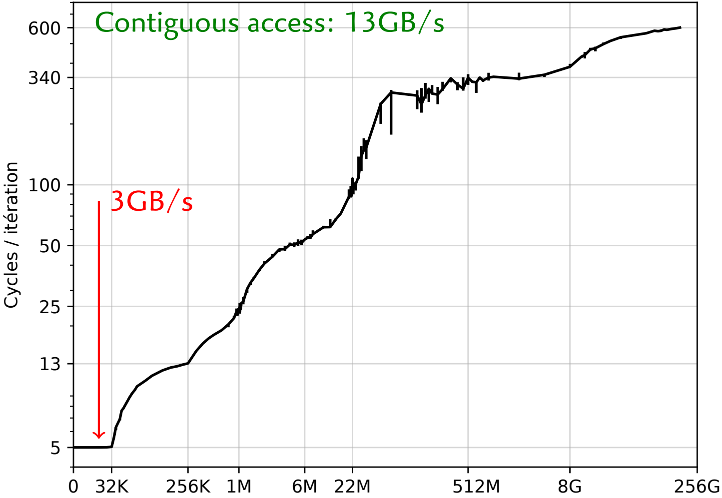
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Effect of the Memory Hierarchy

$T =$ array of N random integers in $[0; N)$

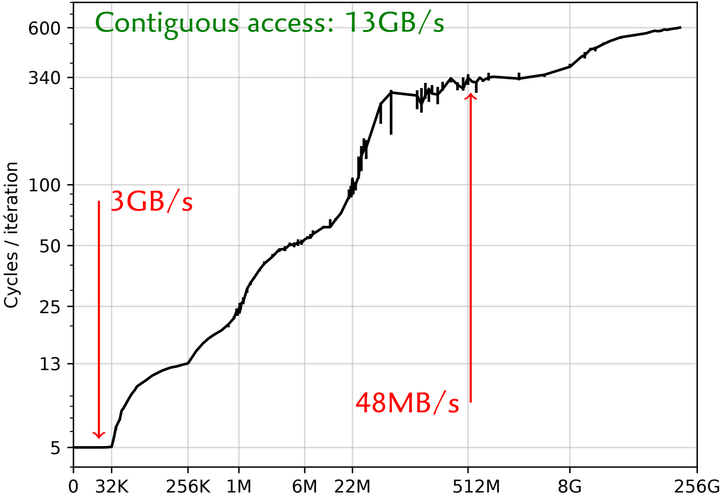
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Effect of the Memory Hierarchy

$T =$ array of N random integers in $[0; N)$

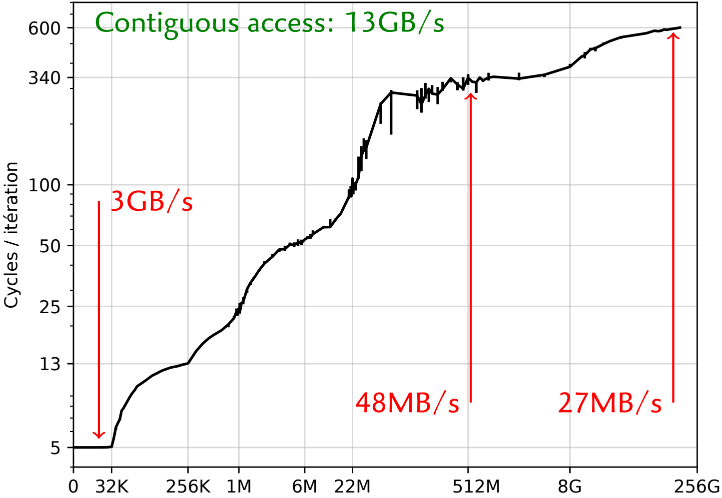
```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



Effect of the Memory Hierarchy

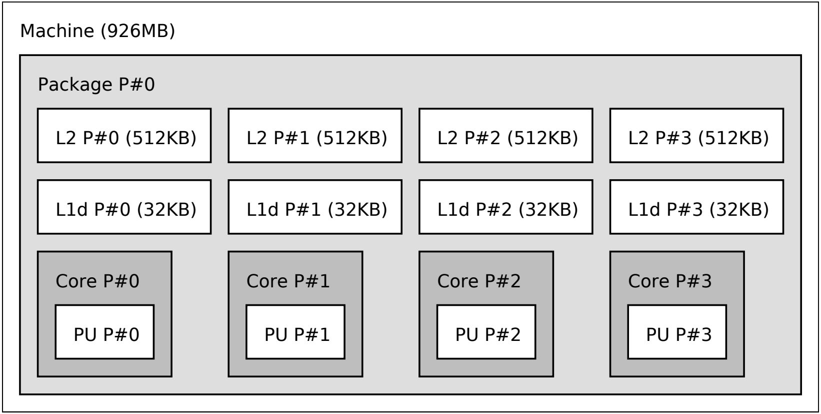
$T =$ array of N random integers in $[0; N)$

```
for (int i=0, x=0; i < 1000000000; i++) x = T[x];
```



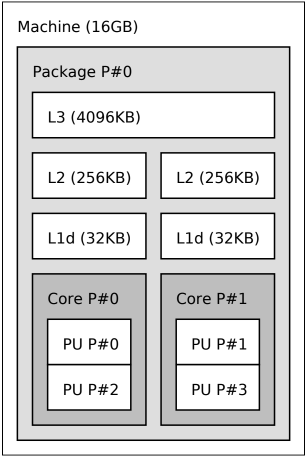
Caches: Wide Variety

Raspberry Pi 3B+ (ARM Cortex A53)



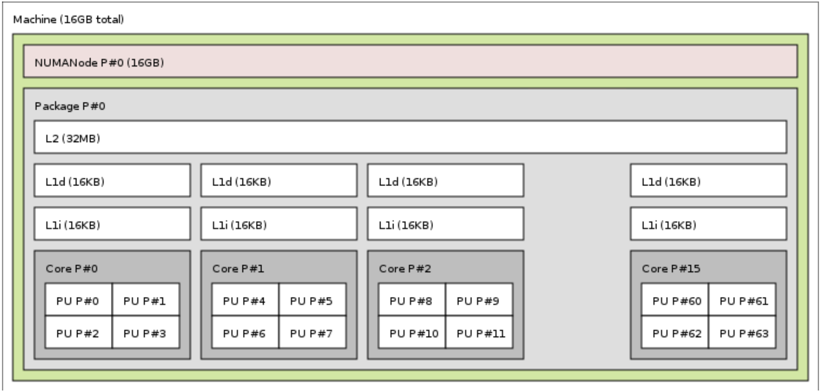
Caches: Wide Variety

My laptop (Intel Core i7 6600U)



Caches: Wide Variety

IBM BlueGene/Q (PowerPC A2)

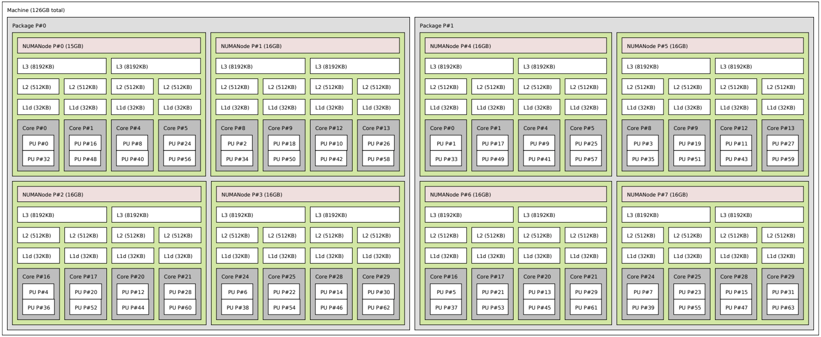


Caches: Wide Variety

Recent Cluster Node (2 $\times$ Intel Xeon Gold 6130)

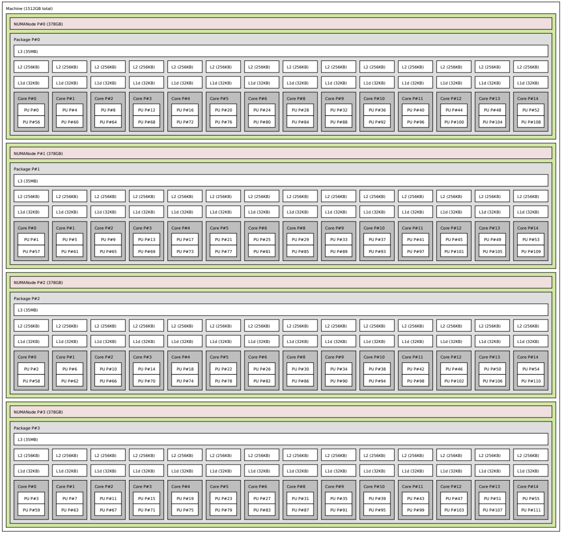
Caches: Wide Variety

Nodes from Another Less Recent Cluster (2 $\times$ AMD EPYC 7301)



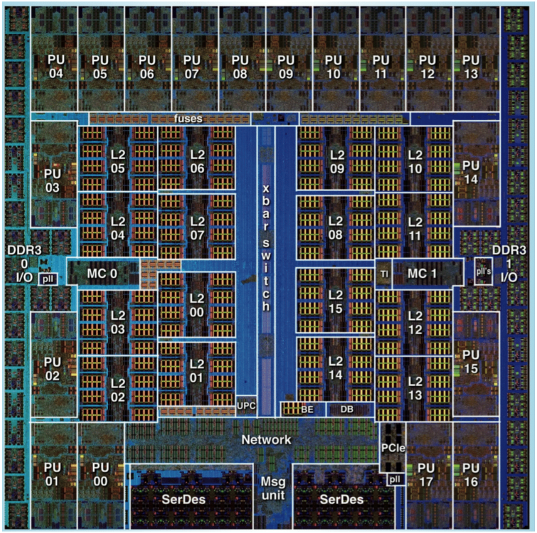
Caches: Wide Variety

Fat Node (4 × Intel Xeon E7-4850 v3) + 1.5TB of RAM



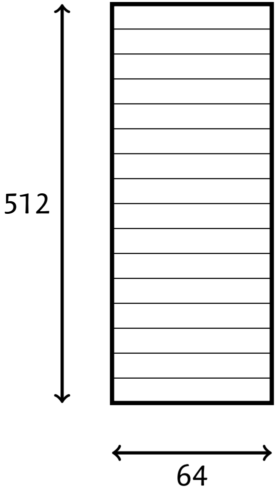
Caches: it Takes Up a Lot of Space!

PowerPC A2



Cache Organization

Typical L1 Cache

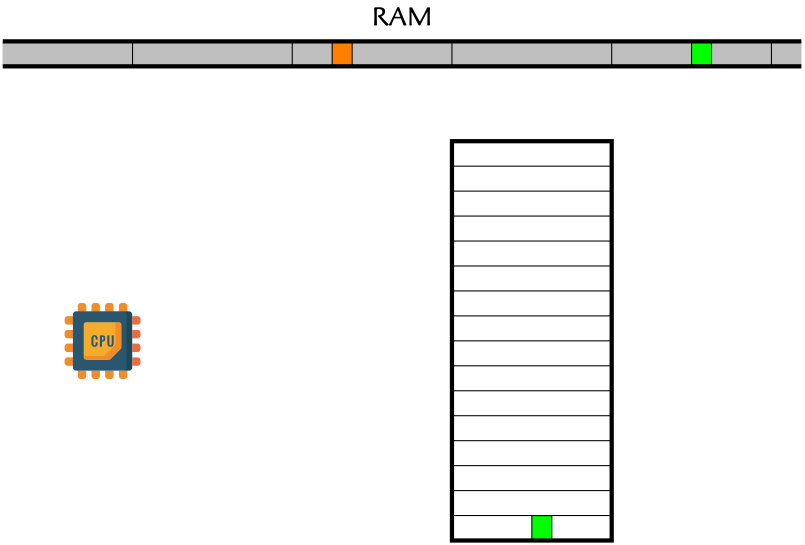


Cache line

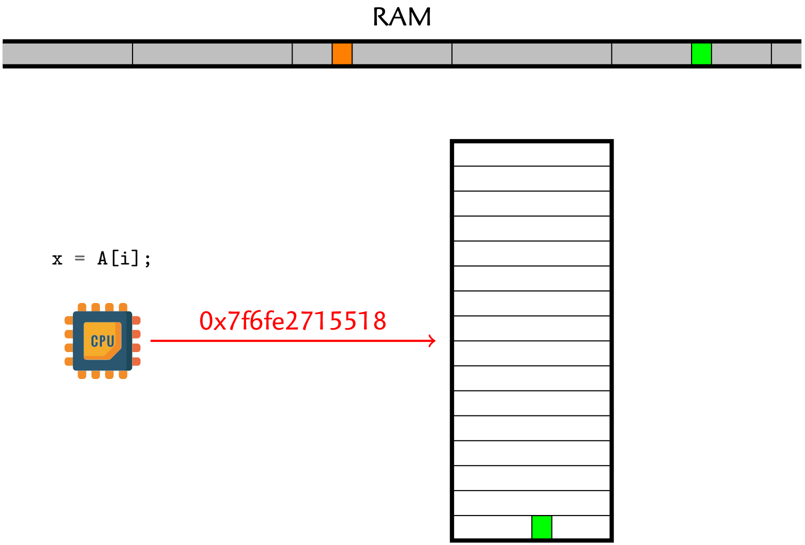
= 64 aligned bytes

(address is a multiple of 64)

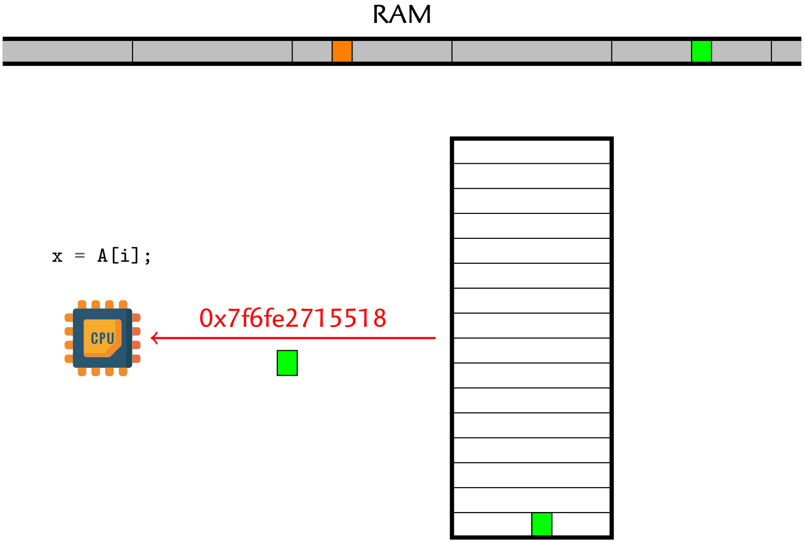
Cache Misses



Cache Misses

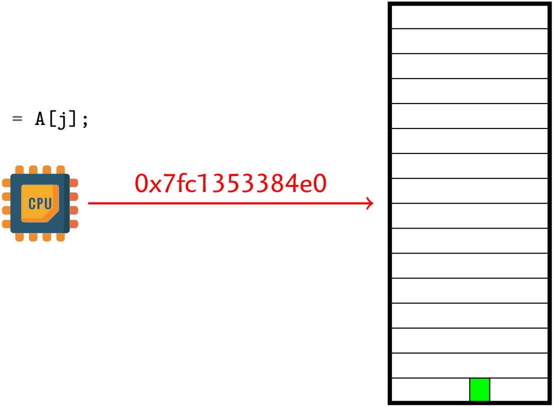


Cache Misses

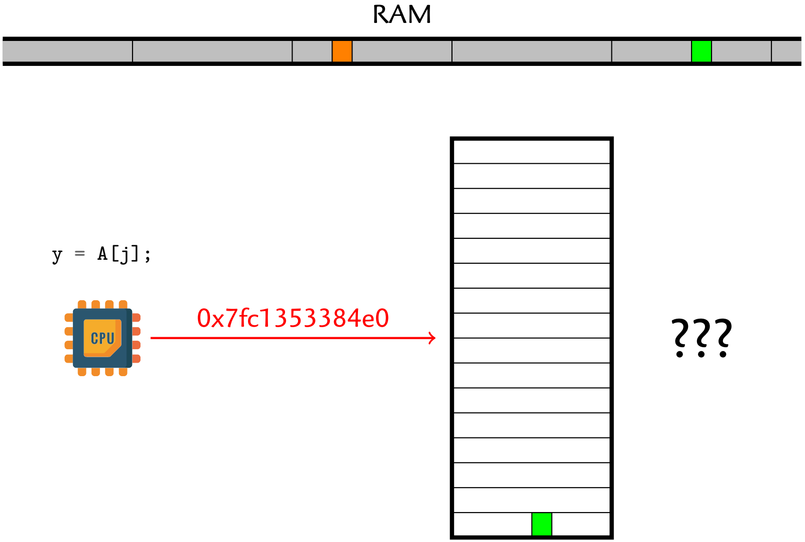


Cache Misses

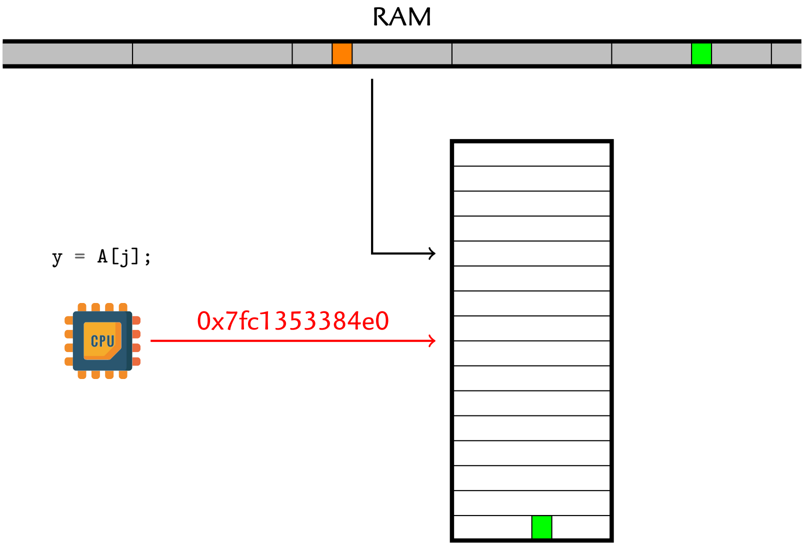

```
y = A[j];
```



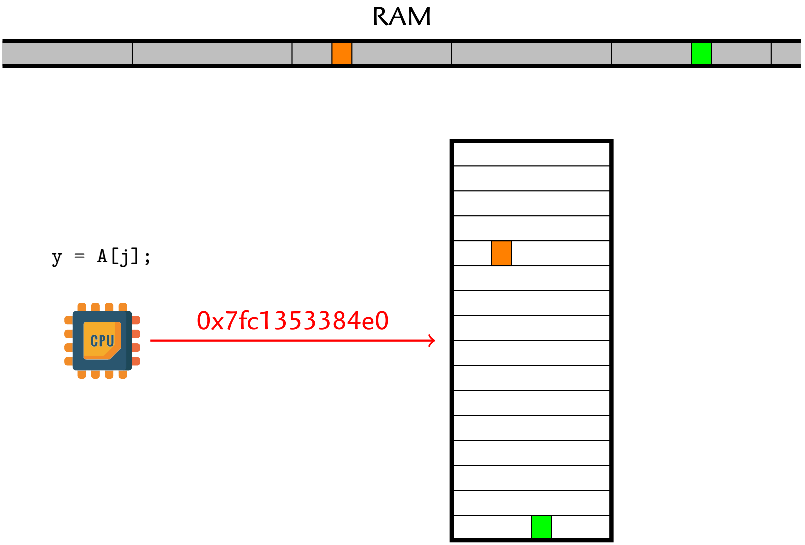
Cache Misses



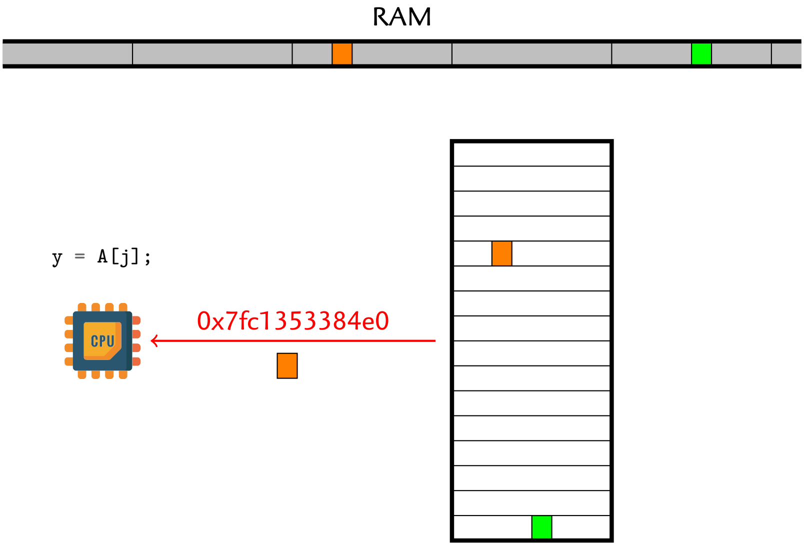
Cache Misses



Cache Misses



Cache Misses

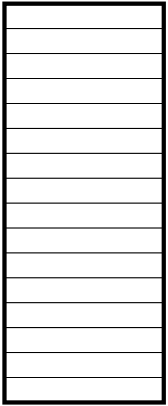


Cache Misses (continued)

11111100010011100010101010100011000

set index

position inside the cache line



In case of cache miss

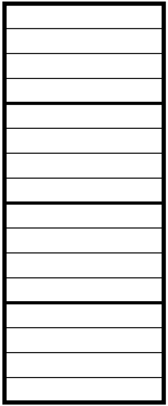
- Evict which cache line?
  - LRU, PLRU
- What about writes?
  - Write-through, write-back
  - False-sharing
- What possible locations for a given line
  - Associativity

Cache Misses (continued)

11111100010011100010101010100011000

set index

position inside the cache line



In case of cache miss

- Evict which cache line?
  - LRU, PLRU
- What about writes?
  - Write-through, write-back
  - False-sharing
- What possible locations for a given line
  - Associativity

Small Examples

2D array copy

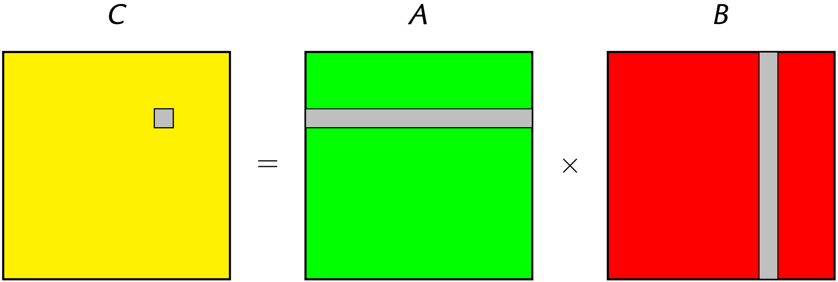
```
/* Bad */
for (int i = 0; i < N; i++)
    for (int j = 0; j <N; j++)
        dst[j][i] = src[j][i];

/* Good */
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        dst[i][j] = src[i][j];
```

Small Examples

Naive GEMM (Matrix-Matrix Product)

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



Small Examples

Naive GEMM (Matrix-Matrix Product)

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

Trick #1

- Swap the loops over j and k

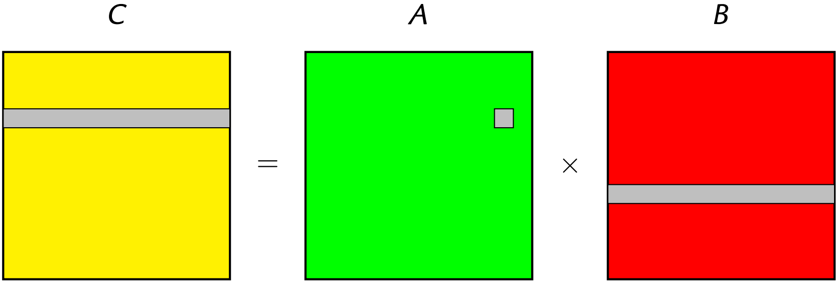
$\Rightarrow$ Contiguous accesses (spatial locality)

- Bonus: opens vectorization possibilities

Small Examples

Naive GEMM (Matrix-Matrix Product)

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



Small Examples

Naive GEMM (Matrix-Matrix Product)

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

Trick #2

- Pre-transpose *B*.
- Bonus: opens vectorization possibilities

Small Examples

Naive GEMM (Matrix-Matrix Product)

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

Summary

- Small matrices: not profitable
  - Overhead too high
- Large matrices: clear gain
  - $N = 3200 : 177s \rightsquigarrow 63s.$

Small Examples

Naive GEMM (Matrix-Matrix Product)

Trick #3

- Product by blocks
- Con: naturally recursive instead of iterative
- Pro: small blocks fit in cache
- (3 matrices 32×32 fit)

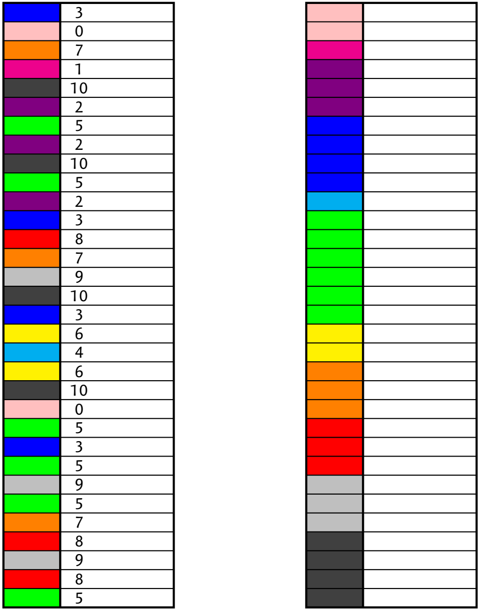
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



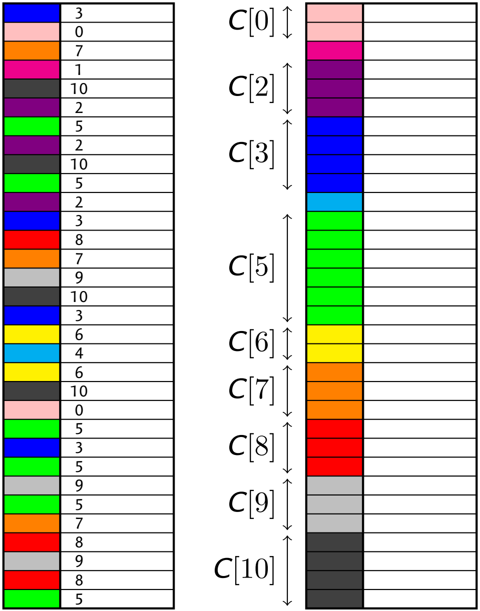
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



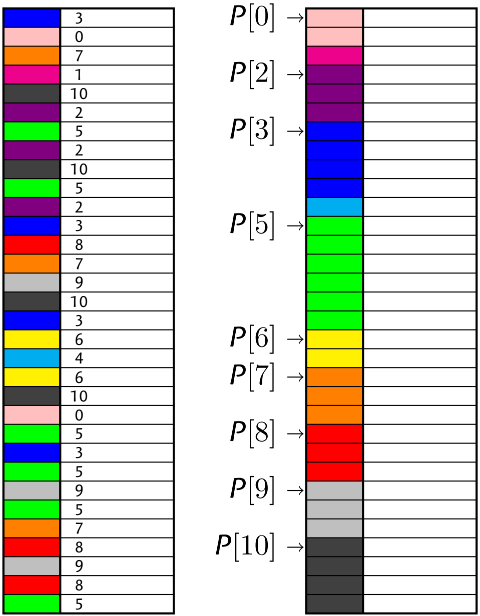
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



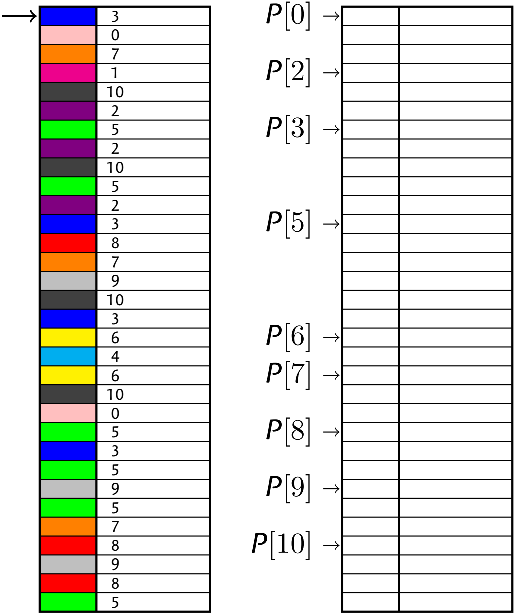
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



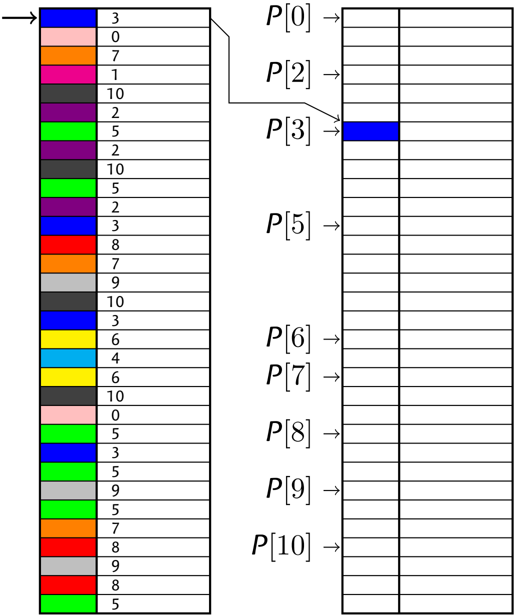
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



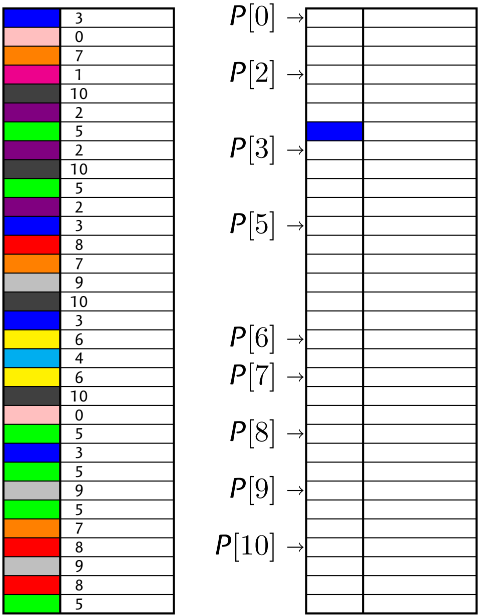
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



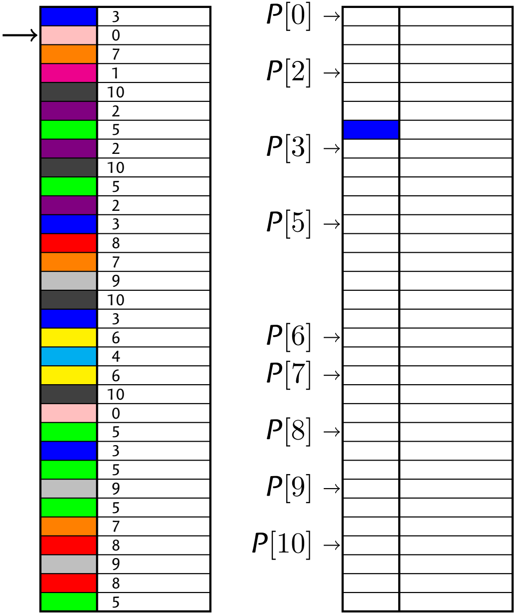
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



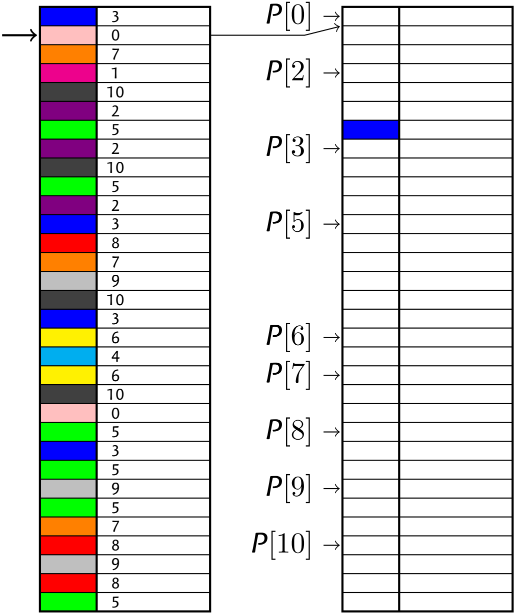
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



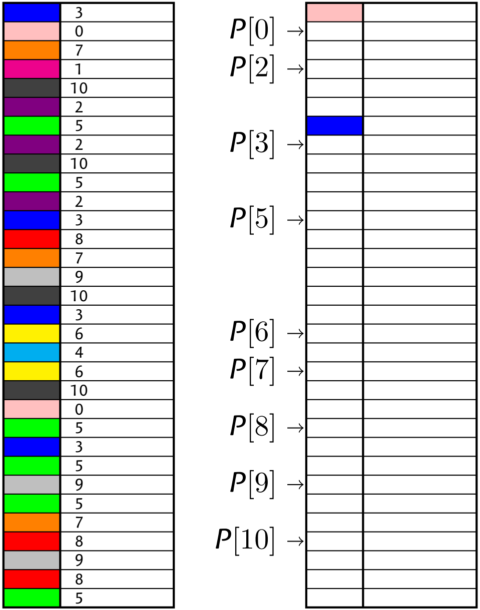
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



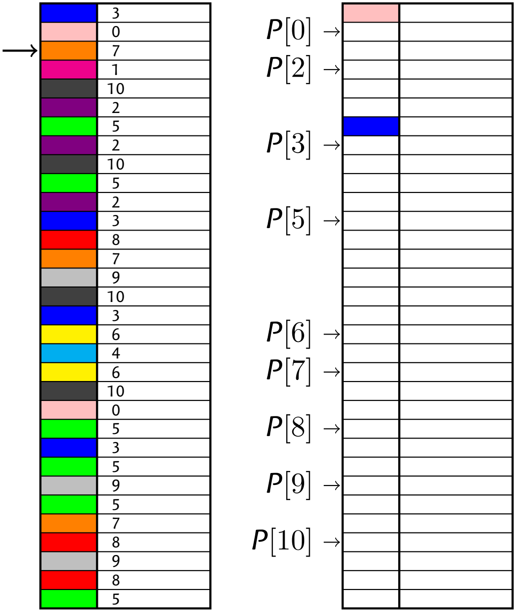
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



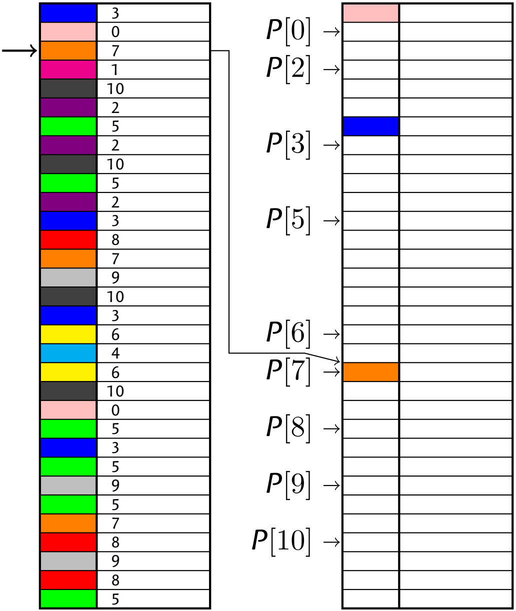
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



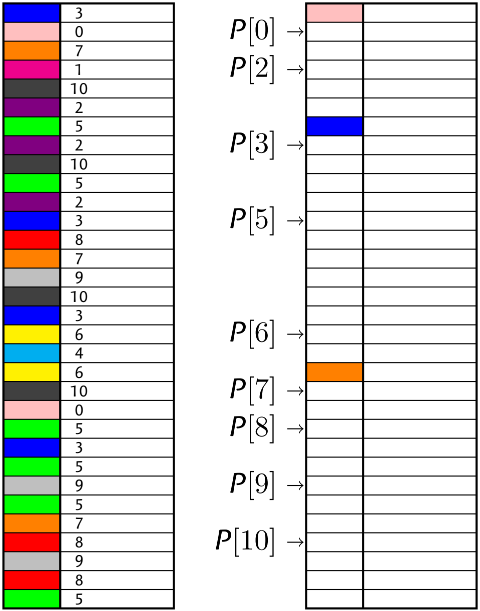
Example: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



Bucket Sort: Analyze

Histogram phase

- Does C fit in cache?
- # buckets $\times$ `sizeof(int)` $\leq$ 32KB ?
- # buckets $\leq$ 8192 😛

Dispatching Phase

- Write into #buckets (far apart) addresses
- One bucket $\leftrightarrow$ one cache line
- # buckets $\leq$ 512 😛
- Requires C + target in cache

Can We Observe All These Phenomena?

Yes!

- Execution: " events" (cache miss, etc.)
- These events have **names**...
  - ... that vary from one system / mechanism to another
- Hardware counter $\rightsquigarrow$ mesure
- Not very easy to access and not very portable (OS / CPU-specific)

Under Linux, with `perf`

- Available event list: `perf list`
  - `cpu-cycles`, `instructions`, `L1-dcache-load-misses`, ...
- `perf stat -e [list evt] ./prog`
- `perf record -e [list evt] ./prog` then `perf report`

+ profiling with `score-p`

Can We Observe All These Phenomena?

Manual Instrumentation with the `PAPI` library

- Performance API (Application Programming Interface)
- Documentation USED TO BE unreadable
- New version 7.0
- New high-level API
- Command-line tool `papi_avail` list events
  - `PAPI_L1_DCM : Level 1 data cache misses`
- Then instrument your code...
- **Do** read “*Redesigning PAPI’s High-Level API*”, Frank Winkler

PAPI in Action

```c
#include <err.h>
#include <papi.h>

int main()
{
  int retval;
  retval = PAPI_hl_region_begin("computation");
  if (retval != PAPI_OK)
      errx(1, "something went wrong");

  // HERE: observed code

  retval = PAPI_hl_region_end ("computation");
  if (retval != PAPI_OK)
     ....
```

- Control using environment variables (`PAPI_EVENTS`)
- Write result in a file
- Can also use to low-level API to get results inside the code

Operational Intensity

Goal

Predict/understand the performance of some code on a given machine

Definition

The operational intensity of an algorithm is the number of arithmetic operations performed per byte transferred from the memory

$$OI = \frac{\text{FLOP}}{\text{Bytes} \leftrightarrow \text{RAM}}$$

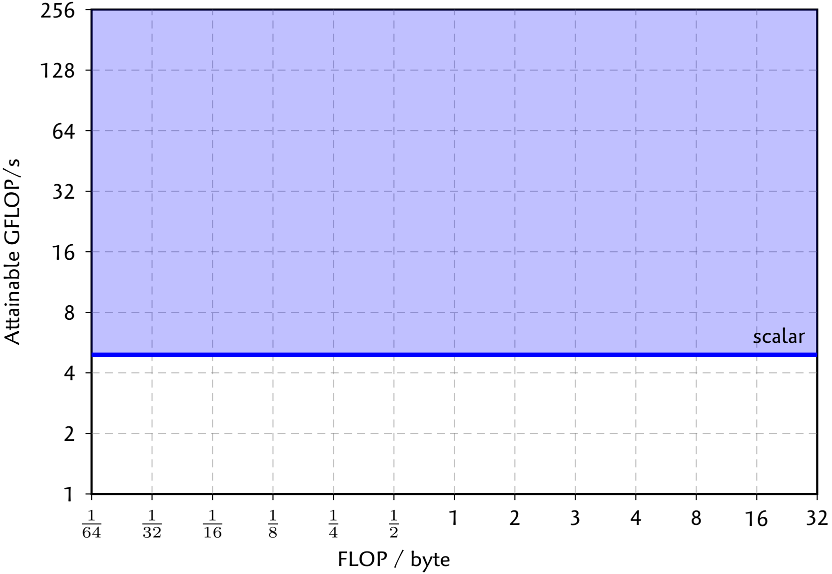
(see also: arithmetic intensity)

Generally Speaking

$$
\begin{aligned}
[\text{Time}] &\leq [\text{Local Computation}] + [\text{Memory Transfer}] \\
&\leq \frac{[\#\text{FLOP}]}{[\text{CPU speed}]} + \frac{[\#\text{ byte transferred}]}{[\text{DRAM bandwidth}]} \\
&\leq \frac{[\#\text{FLOP}]}{[\text{CPU speed}]} \left(1 + \frac{[\#\text{ byte transferred}]}{[\text{DRAM bandwidth}]} \cdot \frac{[\text{CPU speed}]}{[\#\text{FLOP}]}\right) \\
&\leq \frac{[\#\text{FLOP}]}{[\text{CPU speed}]} \left(1 + \frac{[\#\text{ byte transferred}]}{[\#\text{FLOP}]} \cdot \frac{[\text{CPU speed}]}{[\text{DRAM bw}]}\right) \\
&\leq \frac{[\#\text{FLOP}]}{[\text{CPU speed}]} \left(1 + \frac{[\text{Machine Balance}]}{[\text{Operational Intensity}]}\right)
\end{aligned}
$$

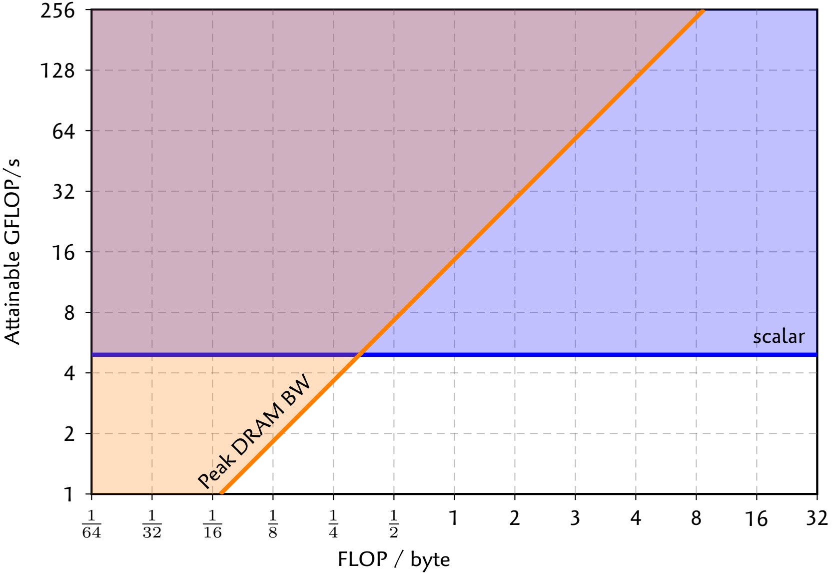
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$\big)$



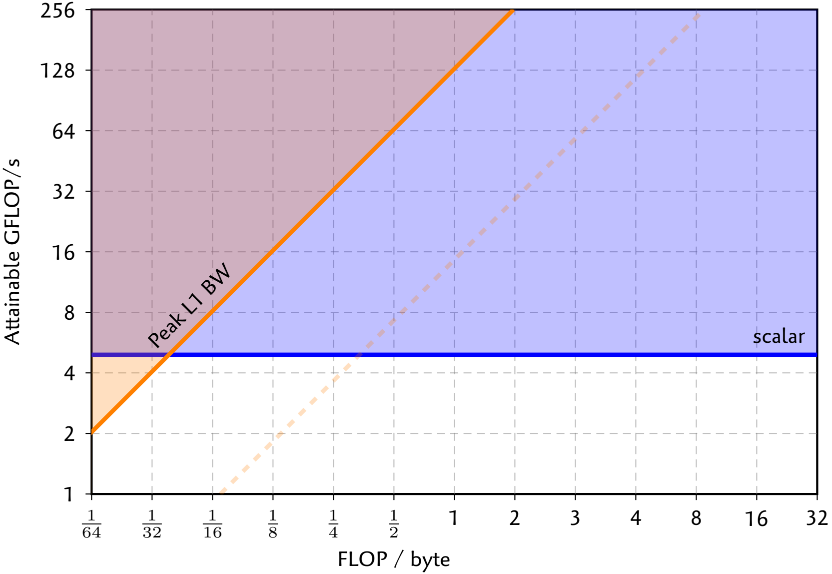
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$)$



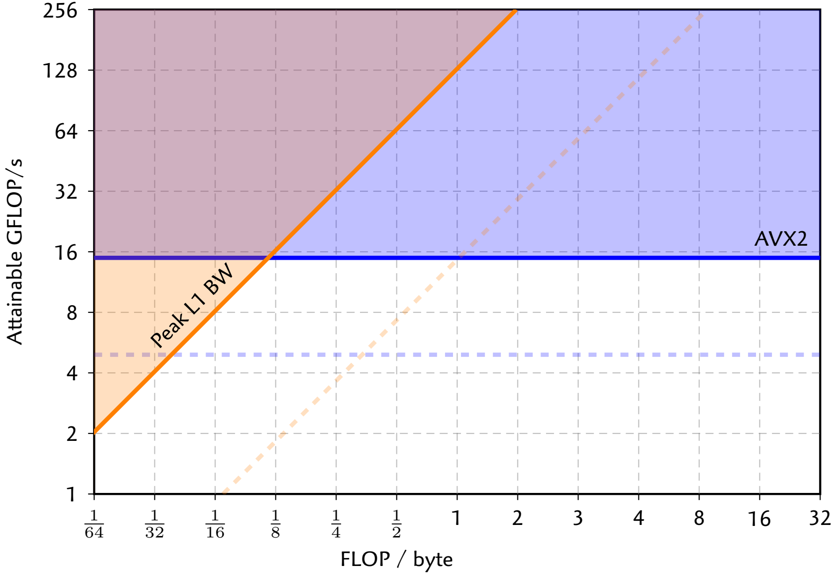
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$\big)$



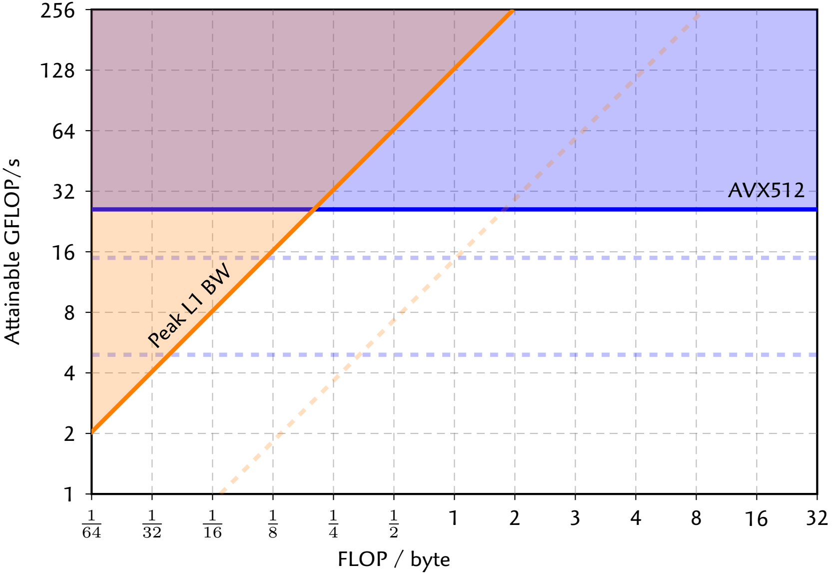
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max([\text{peak FLOPS}], [OI] \times [\text{peak RAM BW}])$



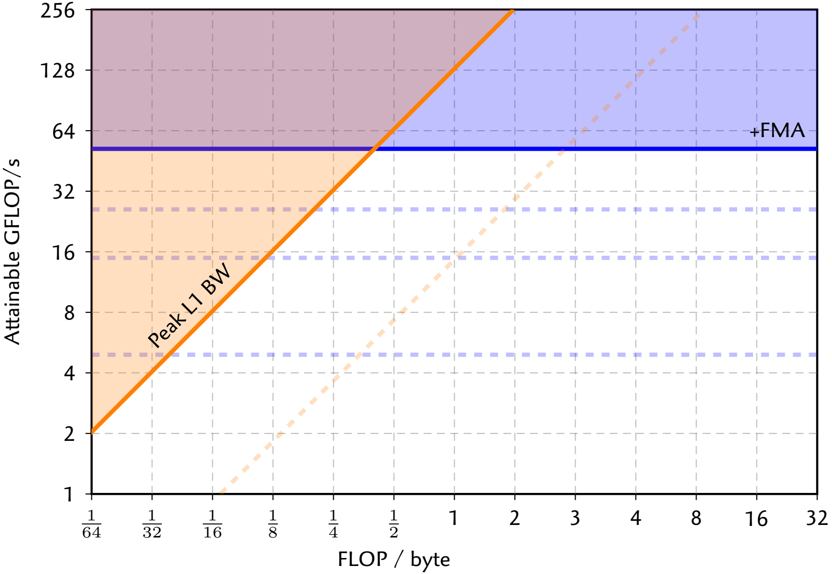
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$\big)$



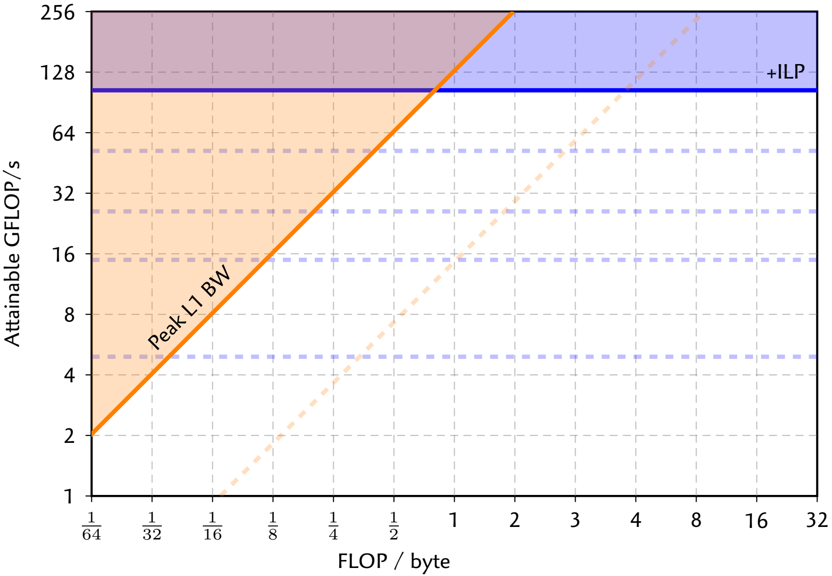
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$)$



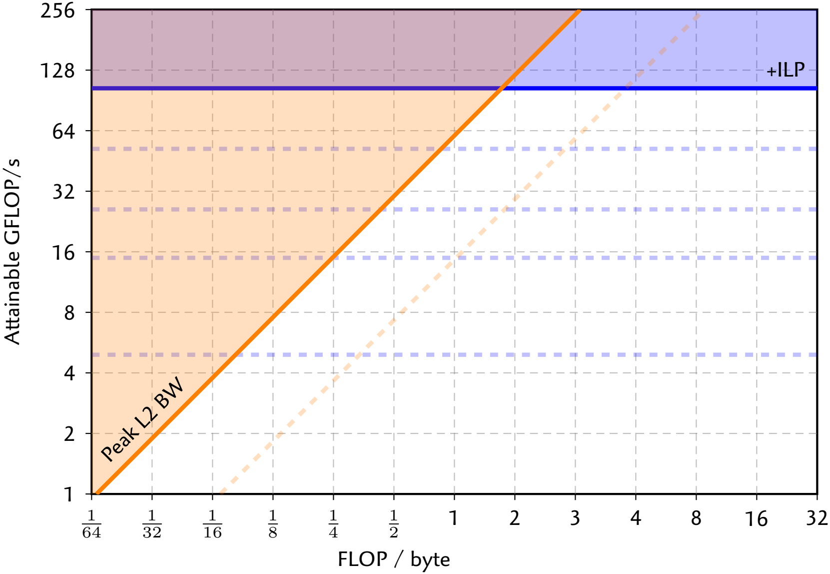
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$\big)$



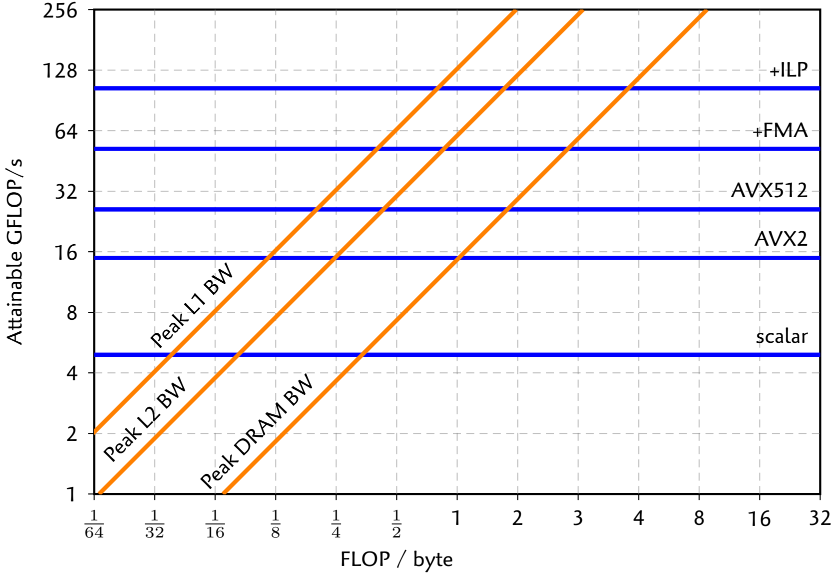
Roofline Diagram — 1 Core

Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*OI*] $\times$ [peak RAM BW]$\big)$

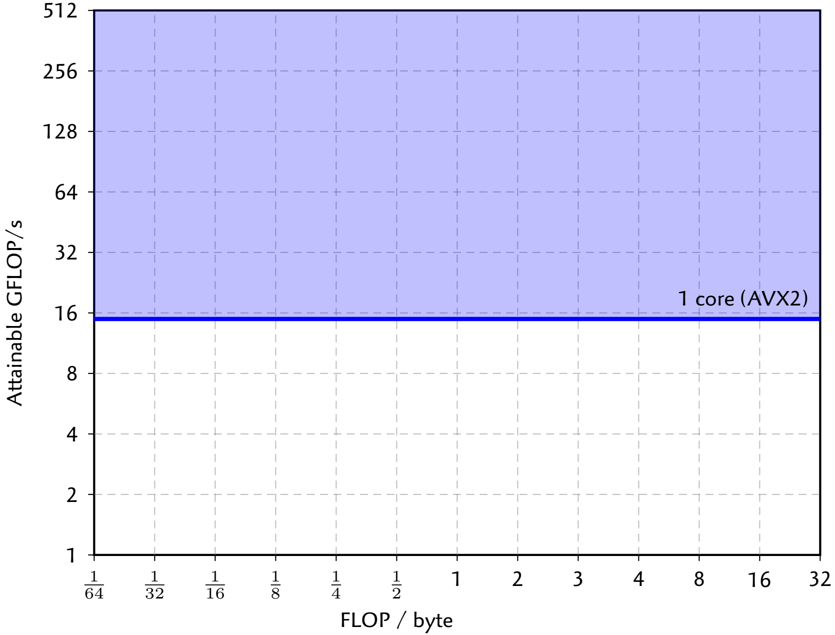


Roofline Diagram — 1 Core (Summary)

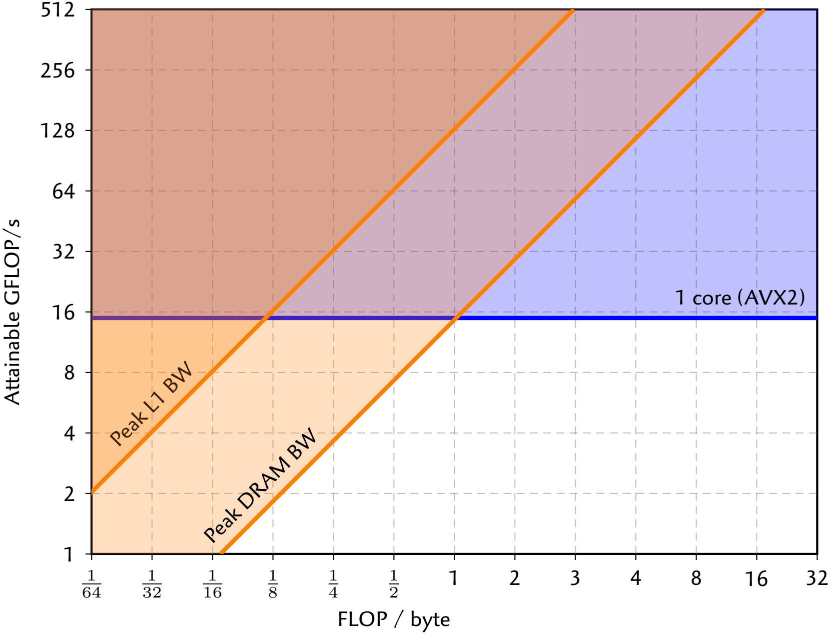
Main Point: [FLOPS] $\leq \max\big($[peak FLOPS], [*IO*] $\times$ [peak RAM BW]$\big)$



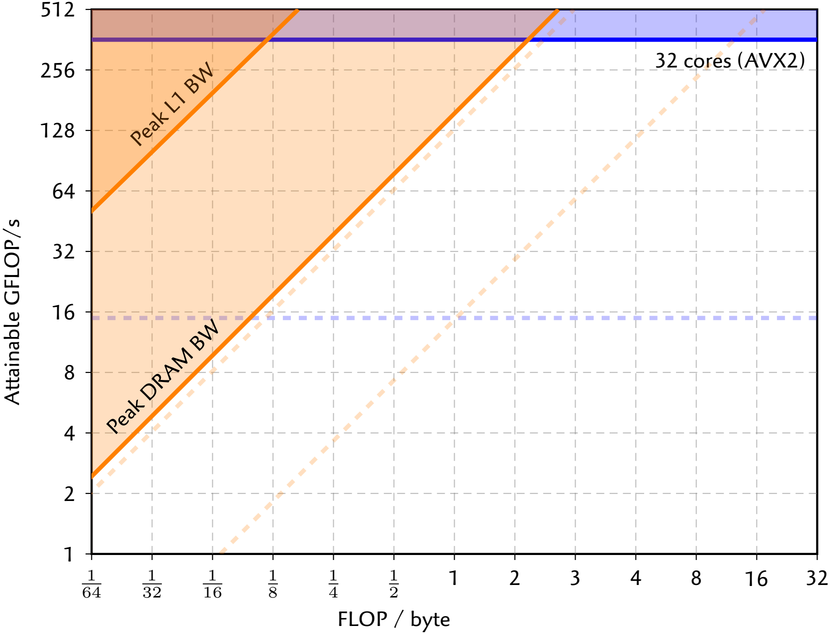
roofline Diagram — 1 Full Node



roofline Diagram — 1 Full Node



roofline Diagram — 1 Full Node



Operational Intensity

Small Examples

Dot Product (and all Level-1 BLAS)

```
double res = 0;
for (int j = 0; j < N; j++)
    res += A[i] * B[i];
```

$$OI = 2\text{FLOP} \;/\; 16 \text{ bytes} = 1/8$$

Operational Intensity

Small Examples

Matrix-vector product (and all Level-2 BLAS)

```
for (int i = 0; i < N; i++) {
    y[i] = 0.0;
    for (int j = 0; j < N; j++)
        y[i] += A[i*N + j] * x[j];
}
```

$$IO = \begin{cases} \text{2 FLOP / 8 bytes} = 1/4 & \text{if } x \text{ fits in cache} \\ \text{2 FLOP / 16 bytes} = 1/8 & \text{otherwise} \end{cases}$$

Operational Intensity

Small Examples

Sparse matrix × dense vector

```
for (int k = 0; i < NNZ; i++) {
    int i = Ai[k];
    int j = Aj[k];
    y[i] += Ax[k] * x[j];
  }
}
```

$$IO = \begin{cases} \text{2 FLOPs / 16 bytes} = 1/8 & \text{best-case — } x \text{ and } y \text{ fits in cache} \\ \text{2 FLOP / 32 bytes} = 1/16 & \text{worst-case} \end{cases}$$

Improving the Operational Intensity?

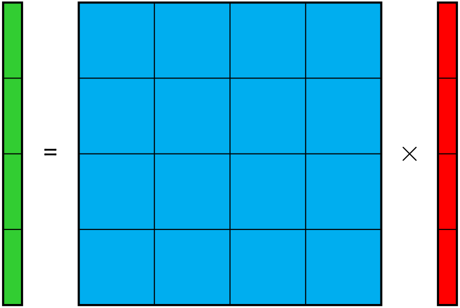
- Algorithms often have to be modified (*blocking*...)
- Some generic optimizations (*loop fusion*)

```c
/* Bad */
for (int i = 0; i < N; i++)
    A[i] += B[i] * C[i];
for (int i = 0; i < N; i++)
    D[i] += B[i] + E[i];
```

```c
/* Better */
for (int i = 0; i < N; i++) {
    double Bi = B[i];
    A[i] += Bi * C[i];
    D[i] += Bi + E[i];
}
```

Improving the Operational Intensity?

- Algorithms often have to be modified (*blocking*...)



- Load a chunk of x in cache
- Load a chunk of y in cache
- Do the product with the corresponding block of A
  - A is read from the RAM
- Rinse, repeat

GEMV (matrix-vector product)

Direct version

```
/* y += A*x */
void gemv(int n, int m, double * A, int ldA, double * x, double * y)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            y[i] += A[i * ldA + j] * x[j];
}
```

- OI = $1/16$ (for large x)
- Reads A at 7GB/s on my laptop
- 50% of peak memory bandwidth
  - Bandwidth shared between A and x...

GEMV (matrix-vector product)

Blocked version

```
static const int nb = 8;
void gemvb(int n, int m, int double *A, int ldA, double *x, double * y)
{
    int nhi = (n / nb) * nb;
    int mhi = (m / nb) * nb;
    int nextra = n - nhi;
    int mextra = m - mhi;
    for (int i = 0; i < mhi; i += nb) {
        for (int j = 0; j < nhi; j += nb)
            gemv(nb, nb, &A[i*ldA + j], ldA, &x[j], &y[i]);
        gemv(nb, nextra, &A[i*ldA + nhi], ldA, &x[nhi], &y[i]);
    }
    for (int j = 0; j < nhi; j += nb)
        gemv(nb, mextra, &A[mhi*ldA + j], ldA, &x[j], &y[mhi]);
    gemv(mextra, nextra, &A[mhi*ldA + nhi], ldA, &x[nhi], &y[mhi]);
}
```

- Reads A at 15GB/s on my laptop
- 100% of peak memory bandwidth
- $2\times$ faster than direct algorithm (reading x "comes for free")

Sparse Matrix × Dense Vector (SpMV)

The Cursed Operation

```
for (int k = 0; i < NNZ; i++) {
    int i = Ai[k];
    int j = Aj[k];
    y[i] += Ax[k] * x[j];
  }
}
```

- Sort `Ai` ⇝ random access to x (= cache misses)
- Sort `Aj` ⇝ random access to y (= cache misses)
- **In all cases**: low operational intensity (1/16 worst-case)

Improving the Operational Intensity?

- Algorithms often have to be modified...
- ... and data structures modified

Sparse matrix × dense vector

```
for (int k = 0; i < NNZ; i++) {
    int i = Ai[k];
    int j = Aj[k];
    y[i] += Ax[k] * x[j];
  }
}
```

- Sort triplets by increasing `Ai[...]`

⇒ (often) read the same *i* as the previous iteration

⇝ Idea: store only positions where *i* changes

Memory Representation of Sparse Matrices

$$A = \begin{pmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{pmatrix}$$

COOrdinate format ("list of triplets")

```
int i[]    = { 2,   1,   3,   0,   1,   3,   3,   1,   0,   2   };
int j[]    = { 2,   0,   3,   2,   1,   0,   1,   3,   0,   1   };
double x[] = { 3.0, 3.1, 1.0, 3.2, 2.9, 3.5, 0.4, 0.9, 4.5, 1.7 };
```

- Size: $\mathtt{nnz} \times (2 \times \mathtt{int} + \mathtt{double})$

Compressed Sparse Row format

```
int p[]    = { 0,       2,            5,        7,            10 };
int j[]    = { 0,   2,   0,   1,   3,   1,   2,   0,   1,   3   };
double x[] = { 4.5, 3.2, 3.1, 2.9, 0.9, 1.7, 3.0, 3.5, 0.4, 1.0 };
```

- Size: $\mathtt{nnz} \times (\mathtt{int} + \mathtt{double}) + (n + 1) \times \mathtt{int}$

Memory Representation of Sparse Matrices

COOrdinate format ("list of triplets")

- Triplets are **not sorted**
- Practical for I/O, free transposition
- Only possible operation: matrix-vector product

Compressed Sparse Row format

- Triplets are **sorted** (requires conversion from COO)
- Possible to iterate over a row

```
for (int i = 0; i < n; i++)
    for (int k = Ap[i]; k < Ap[i + 1]; k++) {
        int j = Aj[k];
        y[i] += Ax[k] * x[j];
    }
```

- More compact
- OI = 2 FLOP / 28 bytes = 1/14 (worst-case)

Common Operations

- $z \leftarrow x \cdot y$ and $y \leftarrow y + Ax$
  - $OI = \mathcal{O}(1)$
  - Almost always memory-bound
- $C \leftarrow FFT(x)$
  - $OI = \mathcal{O}(\log n)$
  - Usually only $\approx 1\%$ of peak performance
- $C \leftarrow C + AB$
  - $OI = \mathcal{O}(n)$
  - 50-80% of Peak performance
  - Use Level-3 BLAS when possible