MAE 5032 High Performance Computing: Methods and Applications

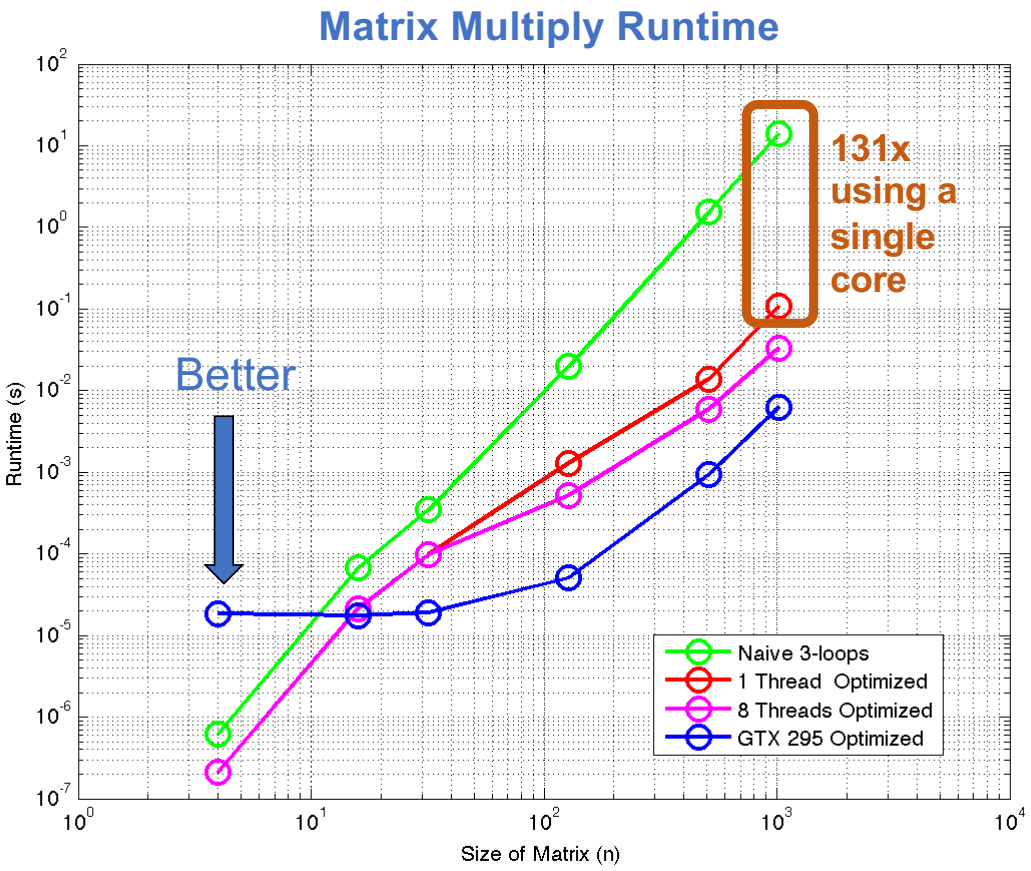
Lecture 4: Single-processor Computer Architecture

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

Motivation

- Memory is too slow to heep up with the processor
- Most applications run at less than 10 percent of the peark performance
- Much of the performance is lost on a single processor
- We need to look under the hood of modern computers
- Our findings may guide our programming styles.

Motivation



Note: Both x and y axes are log-scale

- Parallelizing slow serial code only produces slow parallel code.
- Linear speedup is not necessarily fast code
- Matrix multiply is extreme, but 10x from sequential optimizations is common.

Possible conclusions

“I want to optimize my code right now!”

DO NOT DO THAT!

Optimization can be done by compilers.

Early optimization is the root of all evil.

1. Memory hierarchies
   - von Neumann bottleneck
   - register, cache, and main memory
   - cache details

2. Parallelism within single processors
   - Pipelining
   - SIMD
   - Special instructions (FMA)

3. Case study: Matrix multiplication

4. Roofline model

Latency and bandwidth

Assumption:

Requesting an item of data incurs **an initial delay**;

if this item was the first in a stream of data, usually a consecutive range of memory addresses, the remainder of the stream will arrive with no further delay **at a regular amount per time period**.

Latency and bandwitdth

Two most important concepts related to performance for memory subsystems and for networks:

- **Latency** is the delay between the processor issuing a request for a memory item, and the actual arrival of the item.
  - Various latencies: memory to cache, cache to register, or just memory to processor.
  - Units are usually nano-seconds or clock periods (CP).

- **Bandwidth** is the rate at which data arrives at its destination.
  - Units are [G,M,K]Bytes/Sec or [G,M,K]Bytes/clock cycle

Latency and bandwidth

Bandwidth β

≈ data throughput

Low Bandwidth

High Bandwidth

Latency α

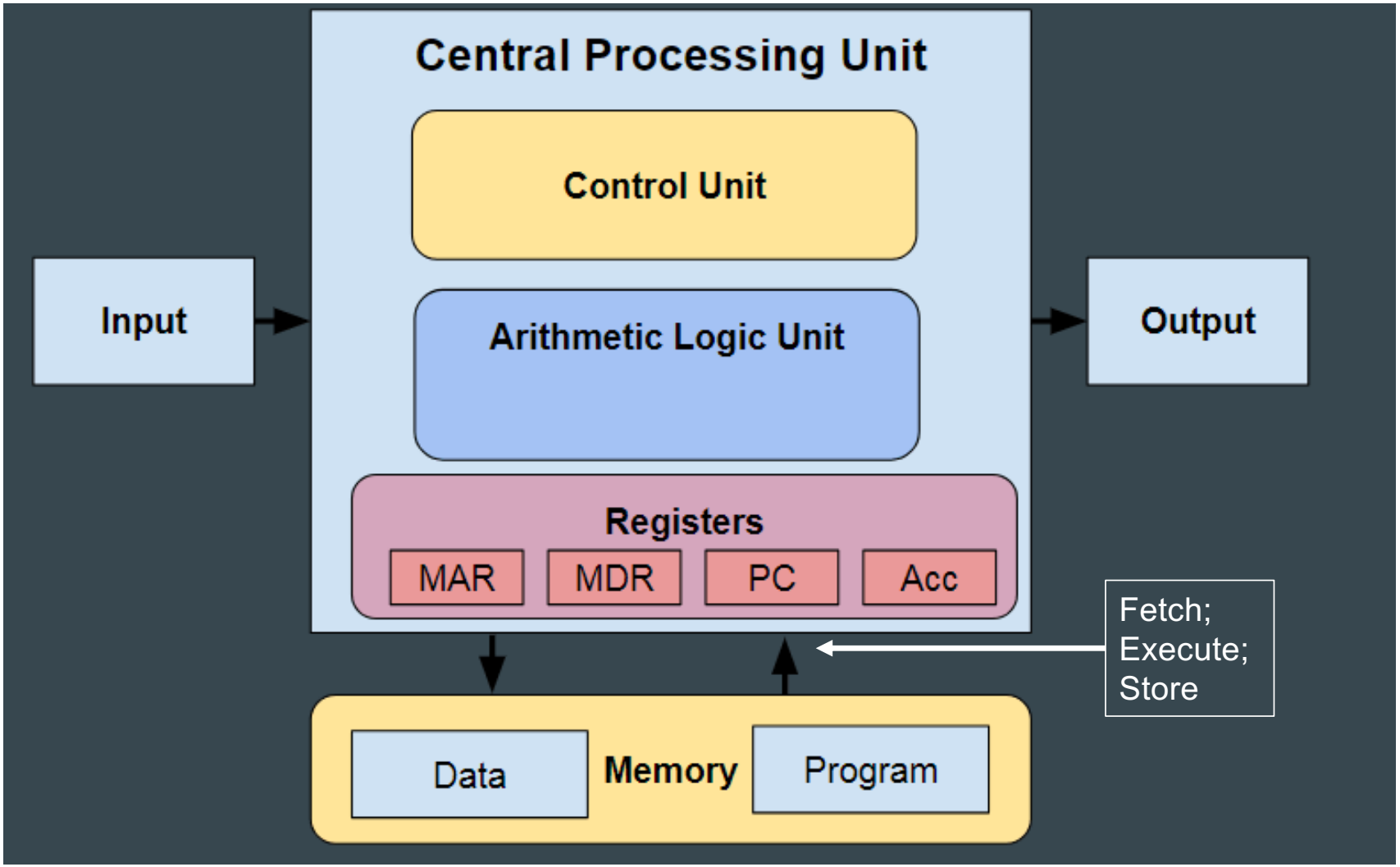
≈ delay due data travel time

Low Latency

High Latency

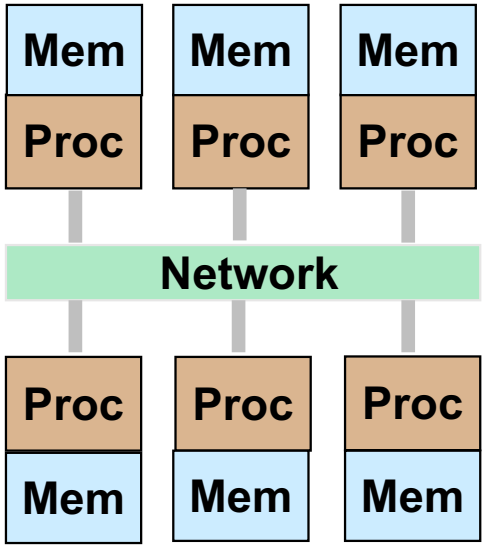
$$T(n) = \alpha + \frac{n}{\beta}$$

von Neumann architecture



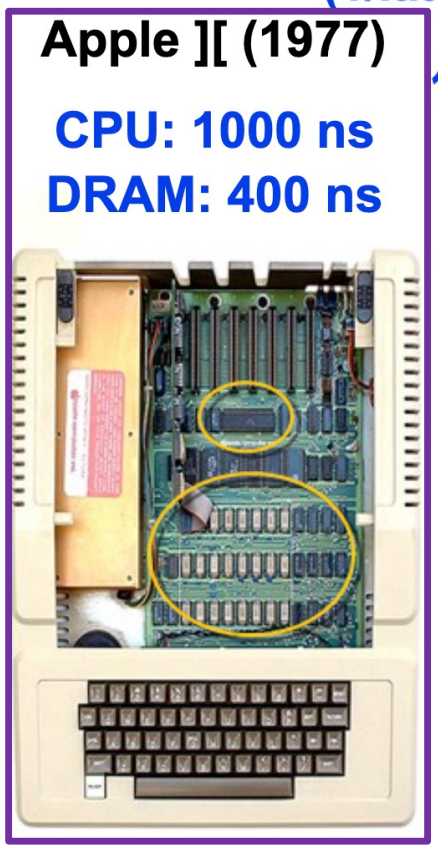
Memory latency gap

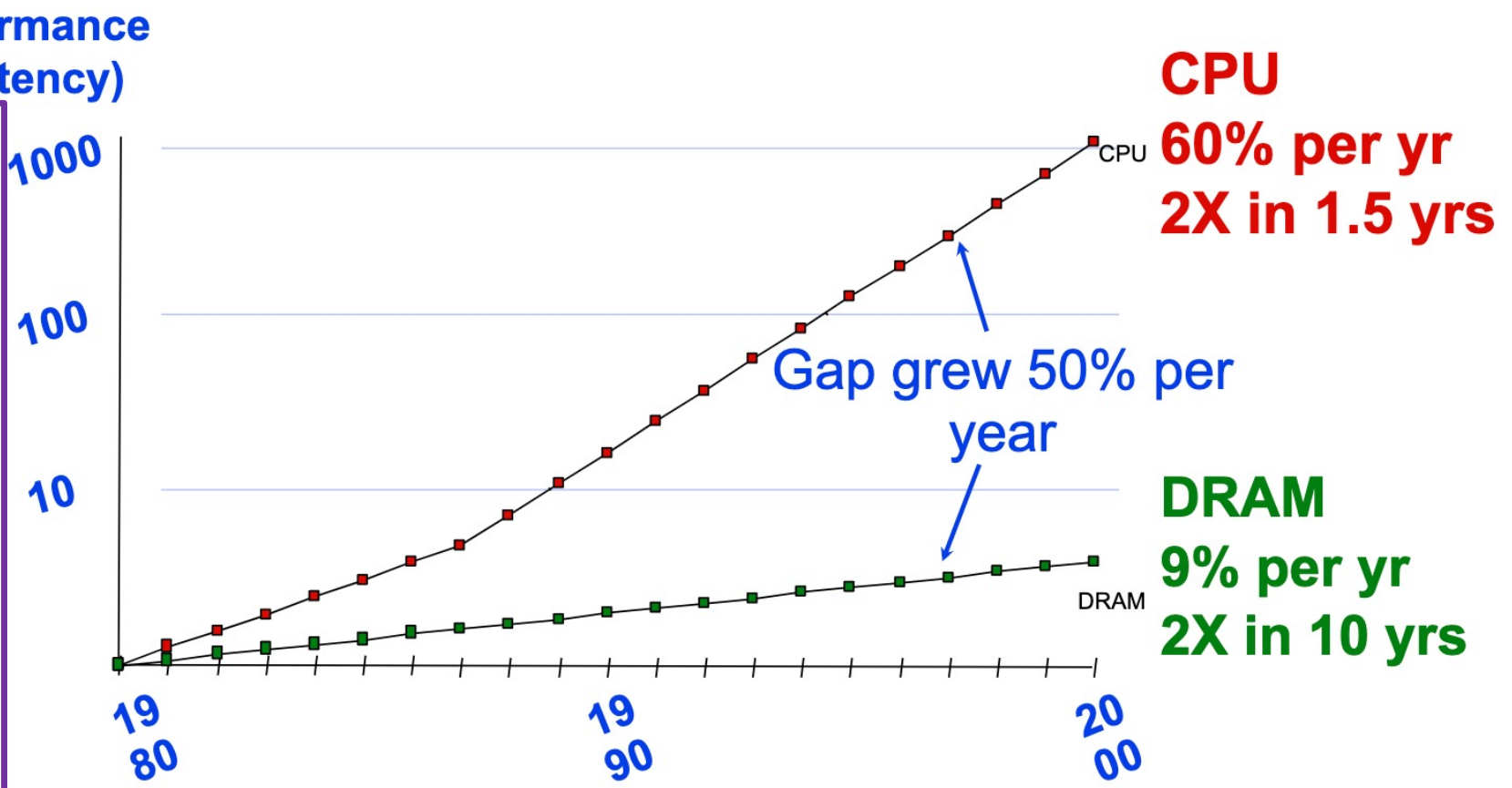
1. Peak FLOPS per socket increases at **50% to 60% per year**
2. Memory bandwidth increases at about **23% per year**
3. Memory latency increases at about **4% per year**
4. Inner-connect bandwidth increases at about **20% per year**
5. Inner-connect latency increases at about **20% per year**



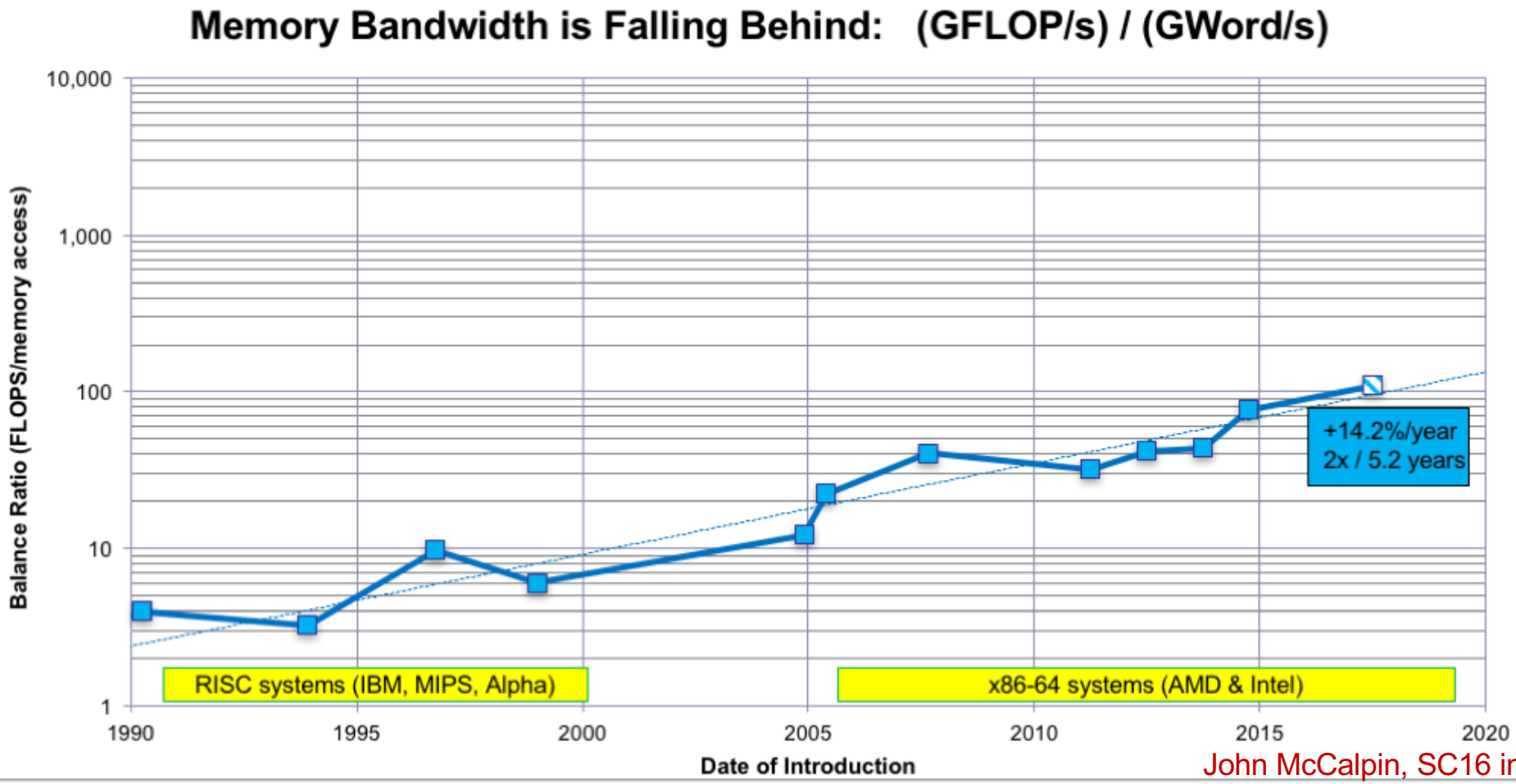
The numbers suggest that processors are being increasingly imbalanced with respect to data motion.

Memory latency gap



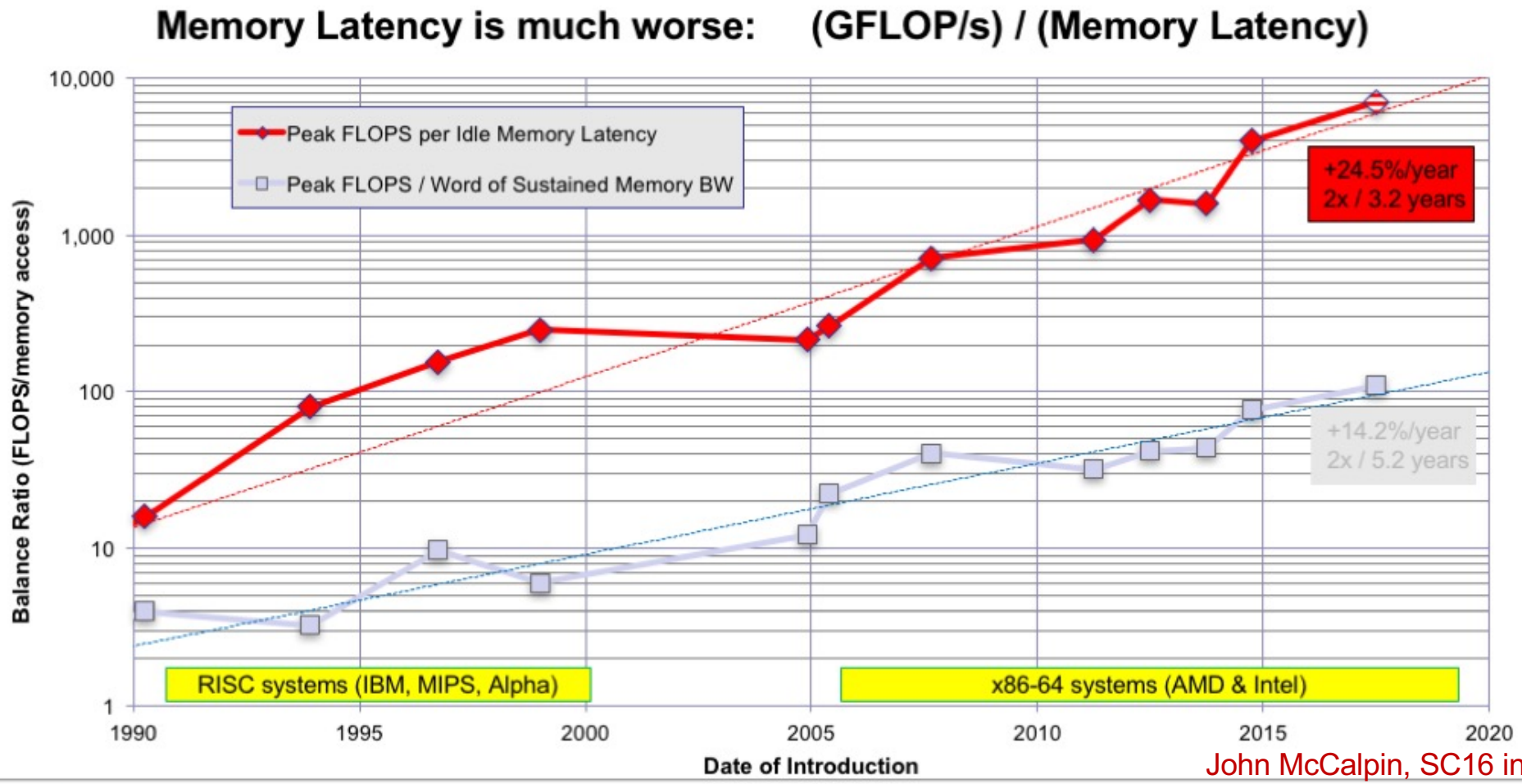


Memory latency gap

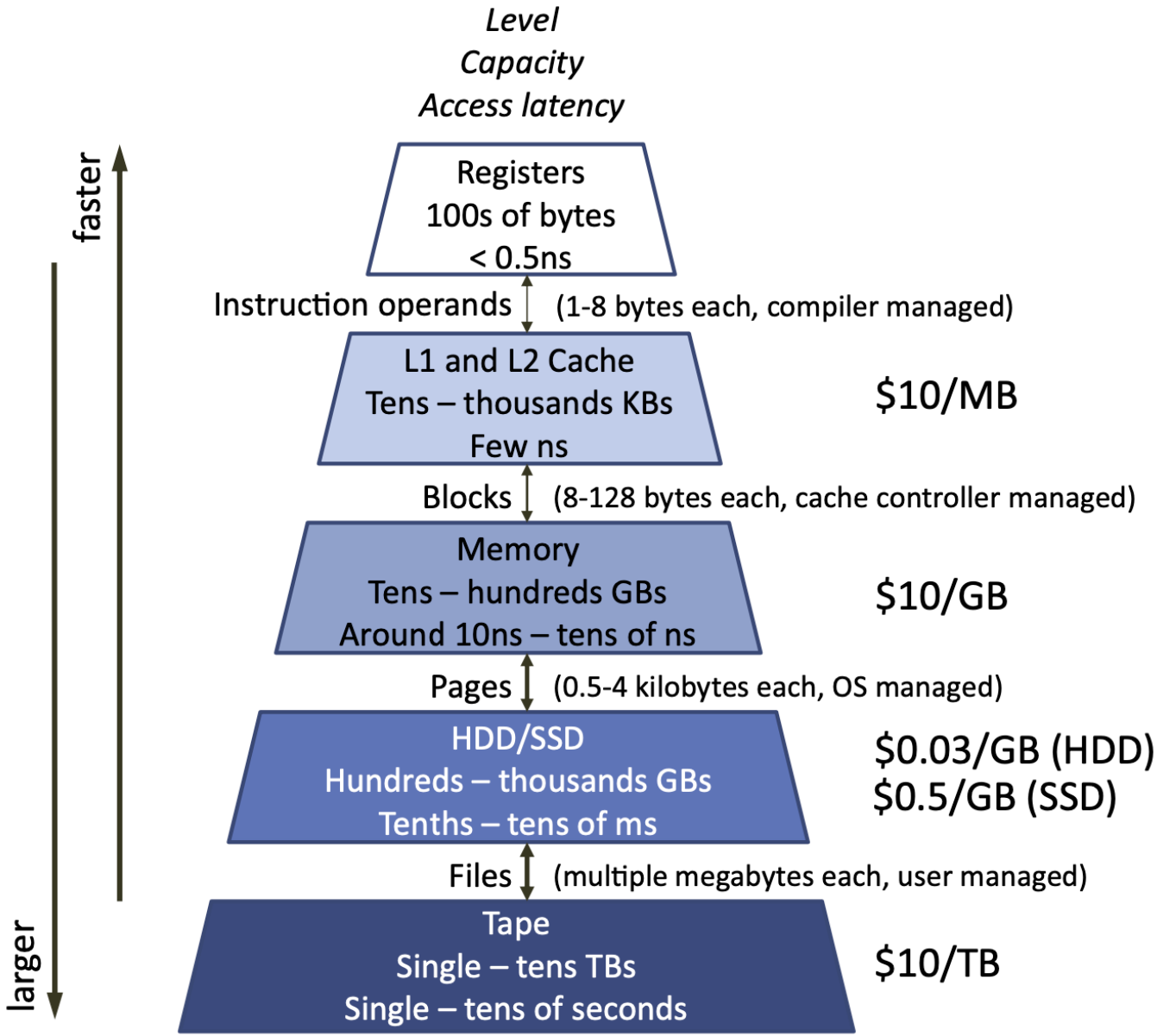


John McCalpin, SC16 invited talk

Memory latency gap



John McCalpin, SC16 invited talk


Level
Capacity
Access latency
faster
larger
Registers
100s of bytes
< 0.5ns
Instruction operands
(1-8 bytes each, compiler managed)
L1 and L2 Cache
Tens – thousands KBs
Few ns
$10/MB
Blocks
(8-128 bytes each, cache controller managed)
Memory
Tens – hundreds GBs
Around 10ns – tens of ns
$10/GB
Pages
(0.5-4 kilobytes each, OS managed)
HDD/SSD
Hundreds – thousands GBs
Tenths – tens of ms
$0.03/GB (HDD)
$0.5/GB (SSD)
Files
(multiple megabytes each, user managed)
Tape
Single – tens TBs
Single – tens of seconds
$10/TB

Memory hierarchies

- Memory is too slow to keep up with the processor (von Neumann bottleneck)
- At considerable cost, it is possible to build faster memory
- Cache is small amount of fast memory
- Memory is divided into different levels
  - Registers
  - Caches
  - Main memory

- Memory is accessed through the hierarchy
  - Registers when possible
  - Caches then
  - Main memory then
  - …

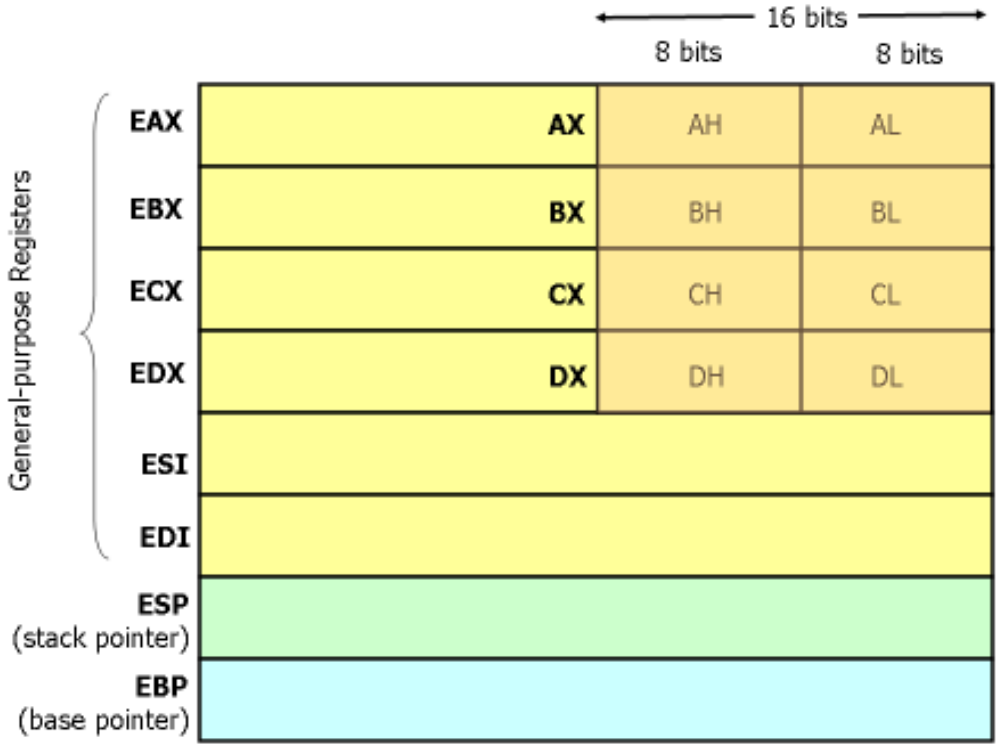
Approaches to handling memory latency

If a processor executes instructions in the order they are found in the assembly code, then execution will often *stall* while data is being fetched from memory; this is also called **memory stall** . For this reason, a low latency is very important.

- Eliminate memory operations by saving values in small, fast memory (cache or register) and reusing them.
  - need temporal locality in program
- Take advantage of better bandwidth by getting a chunk of memory into cache (register) and using whole chunk
  - need spatial locality in program

Registers

- **Highest bandwidth, lowest latency memory that a modern processor can access**
  - **built into the CPU**
  - **often a scarce resource**
  - **data movement in it can be viewed as instantaneous**
  - **Typically a processor has 16 or 32 floating point registers**
  - **Intel Itanium has 128 floating point registers!**



https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Registers

- **Processor instructions operate on registers directly.**
  - **Have assembly language names like: eax, ebx, ecx, etc.**
  - **Sample instruction:**

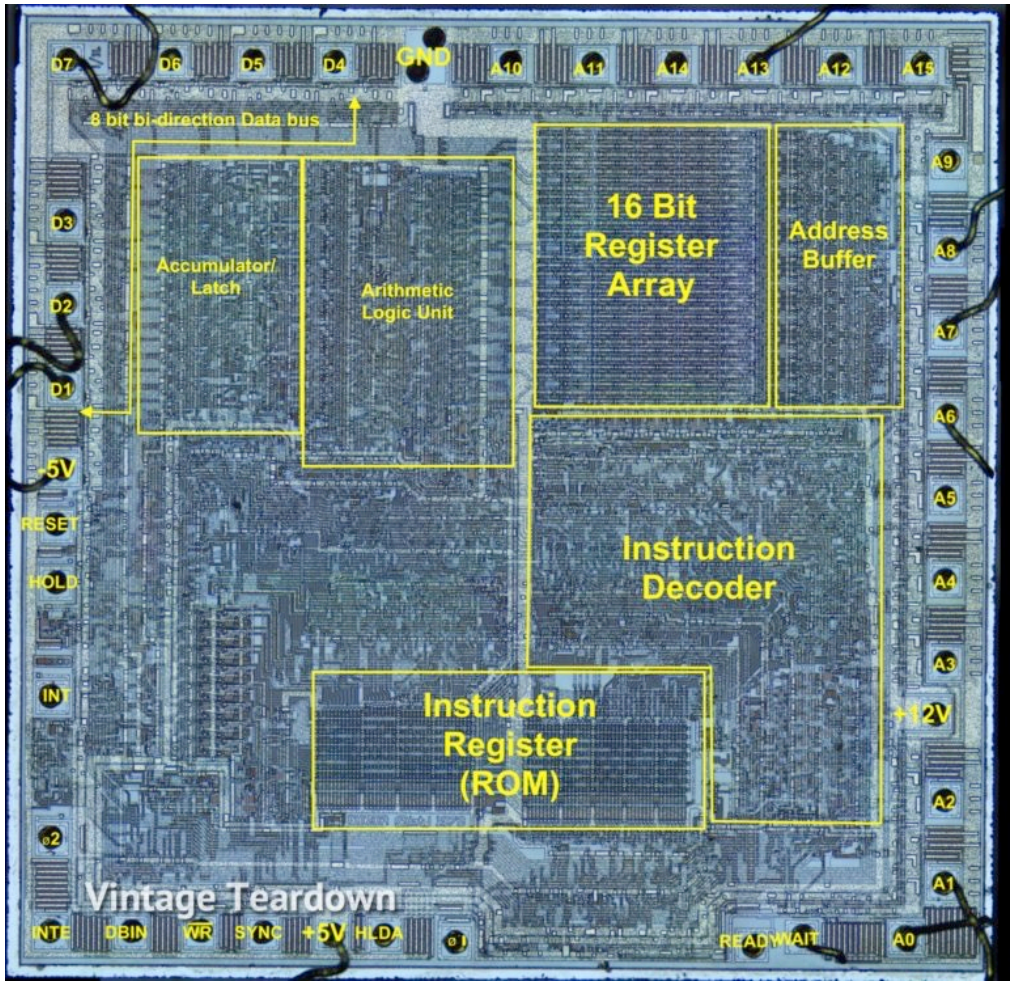
    addl %eax, %edx

Example:

a = b + c

is implemented as

- **load the value of b from mem to register**
- **load the value of c from mem to register**
- **compute the sum and write into another register**
- **write the sum value back to the mem location of a**



Registers

- Moving data from memory is relatively expensive.
- A simple optimization idea is to leave data in register when possible.

Example:

```
a = b + c;          d = a+e;
```

the computed value of a could be left in register. This can be performed by **compiler optimization***:* the compiler will not issue an instruction for storing and reloading a.

Example:

```
t1 = sin(a) * x + cos(a) * y;          t2 = -cos(a) * x + sin(a)*y;
```

The sin(a) and cos(a) will probably be kept in register. You can help the compiler explicity by

```
s = sin(a); c = cos(a); t1 = s * x + c * y; t2 = -c * x + s * y;
```

Note: keeping too many in register is called register spill and lowers the performance!

Registers

- Keeping a variable in register is important if that variable appears in an inner loop.

Example:

```
for i = …
     a[ i ] = b[ i ] * c
```

The variable c will probably be kept in register by the compiler.

In the following,

```
for k=1,n
     for i =1,l
          a[ i, k ]=b[ i, k ] * c[ k ]
```

it can be a good idea to introduce a temp varialbe to hold c[k].

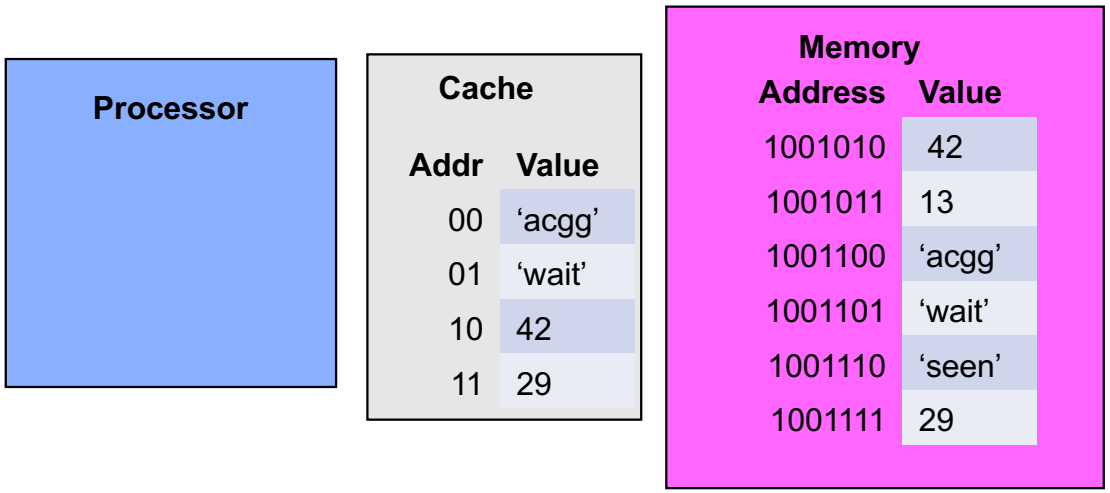
- In C, you can give a hint to the compiler to keep a variable in register by “register”.

```
register double tt;
```

Cache

- **Cache** is fast (expensive) memory which keeps copy of data.
- Cache is between the CPU register and the main memory.

Example: data at memory address xxxxxxx10 goes to location 10 in cache.



Cache: multiple levels

- **L1 Cache:** data cache closest to register. Typically 16～64KB.

- **L2 Cache:** secondary data cache, stores both data and instructions.
  - Data from L2 has to go through L1 to registers
  - L2 is 10 to 100 times larger than L1
  - Some systems have L3 cache (off chip), about 10 times larger than L2.

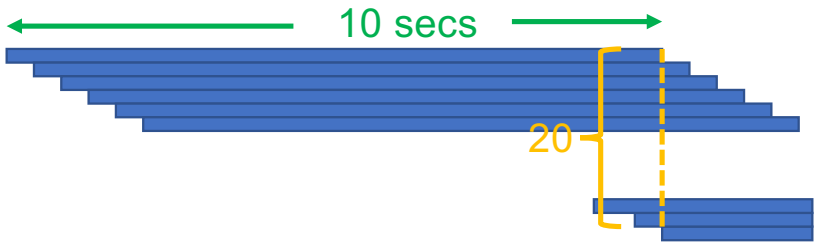
- **Cache line:** the smallest unit of data transferred between main memory and the caches.
  - If you request one word on a cache line, you get the whole line.
  - It can be beneficial to use other items, since you have paid for them in the bandwidth.

Example: you move data across shelves, you move in the unit of book rather than book pages.

Cache concurrency

- Little's Law from queuing theory says:
  concurrency = *latency* x *bandwidth*

- Example:
  latency = 10 sec
  bandwidth = 2 Bytes/sec

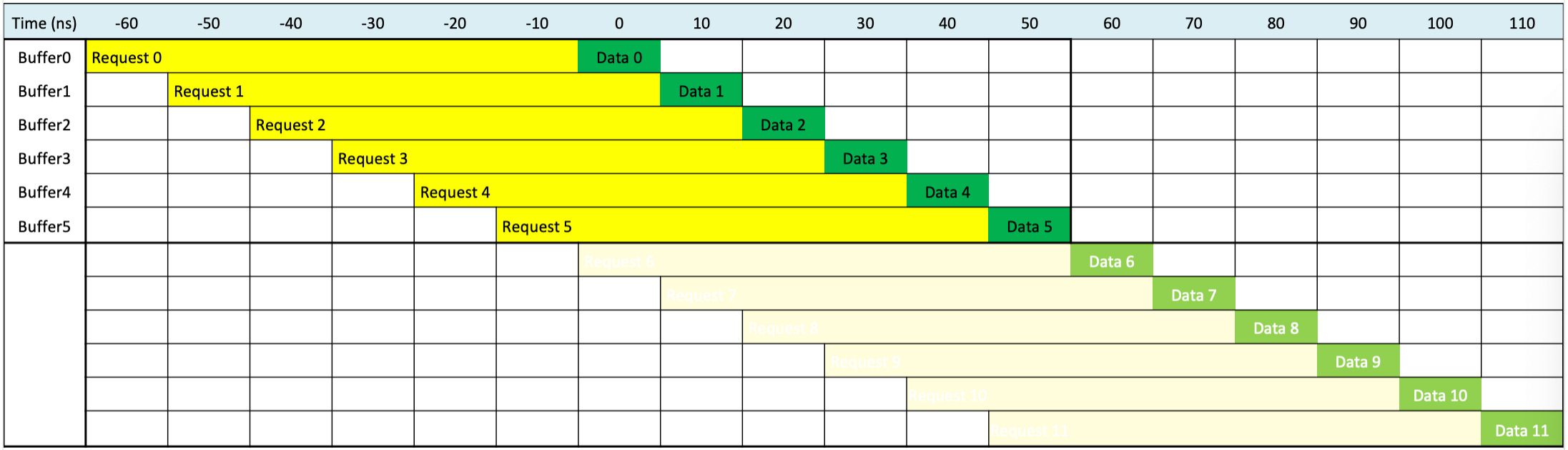


- Requires 20 bytes in flight to hit bandwidth speeds

- That means finding 20 independent things to issue

Cache concurrency

60 ns latency, 6.4 GB/s (=10ns per 64B cache line)



- 60 ns * 6.4 GB/s = 384 Bytes = 6 cache lines
- To keep the pipeline full, there must always be 6 cache lines “in flight”
- Each request must be launched at least 60 ns before the data is needed

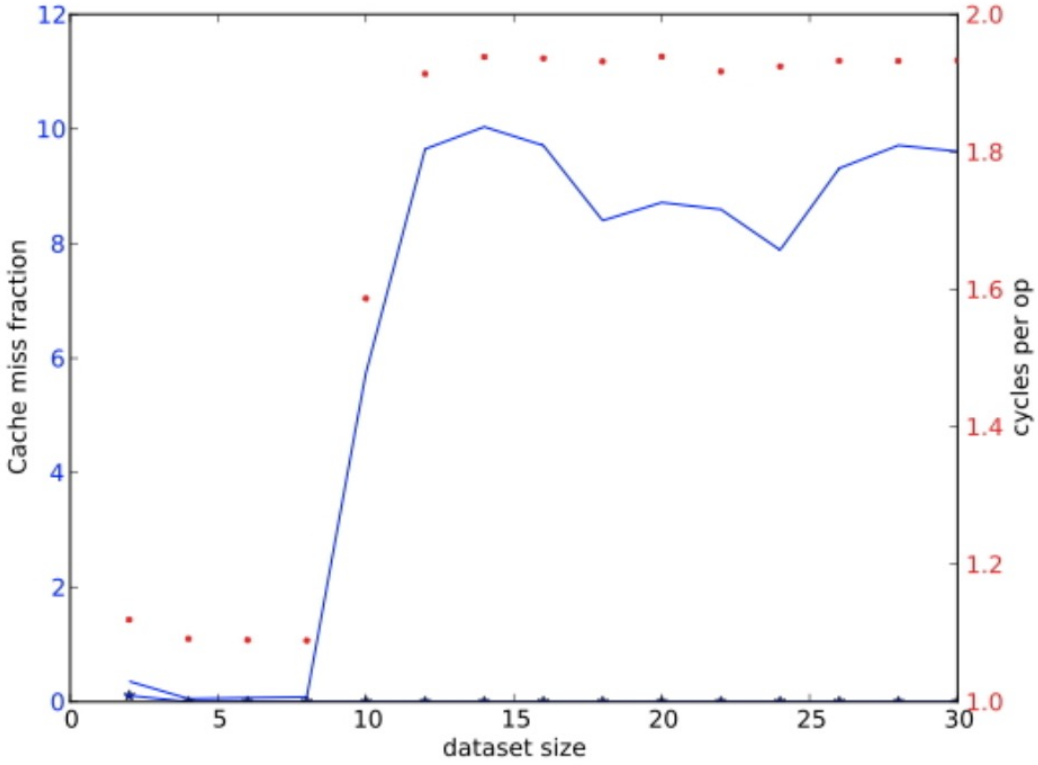
Cache hits, misses, trashing

- **Cache hit**
  - location referenced is found in the cache
- **Cache miss**
  - location referenced is not found in the cache
  - triggers access to the next level slower cache or memory
- **Cache trashing**
  - a trashed cache line must be repeatedly recalled in the process of assessing its elements
  - caused when other cache lines, assigned to the same location are simultaneous accessing data/instructions that replace the TCL with their content

Cache size

```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```

- If the data fits in L1 cache, the transfer is fast
- If there is more data, transfer speed from L2 dominates



Cache size

```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```

```
for (i=0; i<NRUNS; i++) {
   blockstart = 0;
   for (b=0; b<size/l1size; b++)
     for (j=0; j<l1size; j++)
       array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
 }
```

- Data can be arranged to fit in cache

Cache access

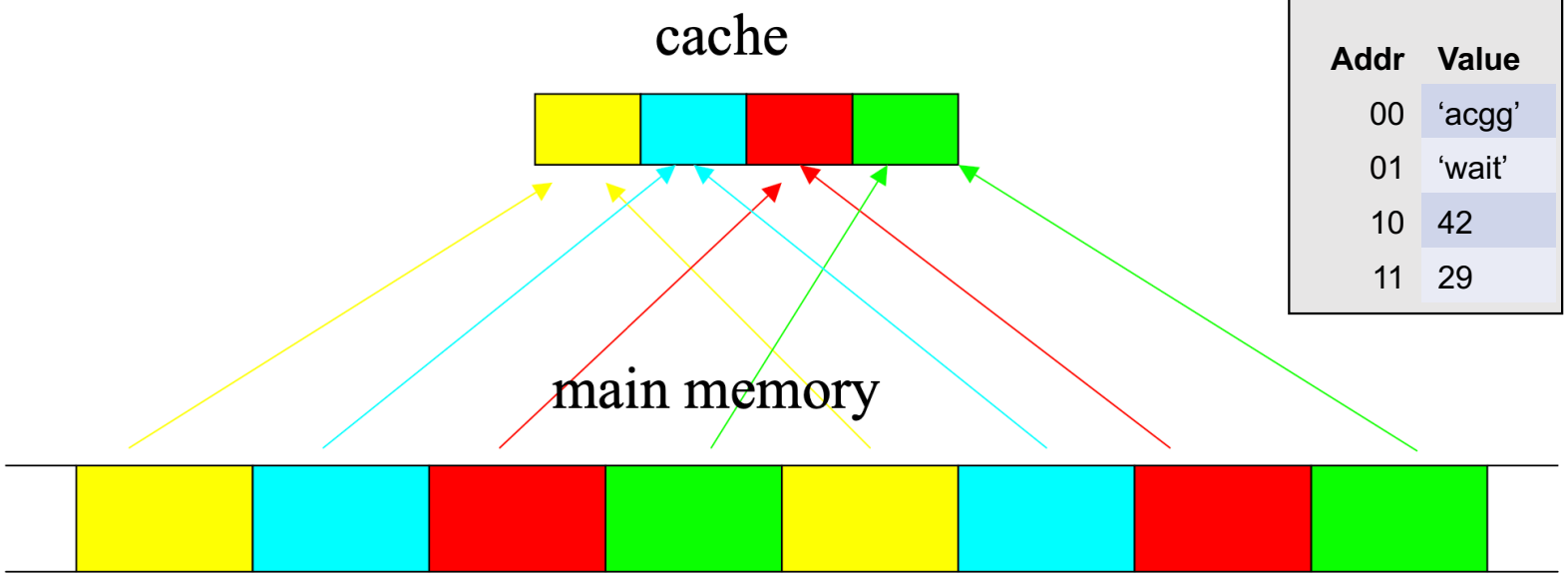
- Access is (almost) **transparent** to the programmer
  - data is in a register or in cache or in memory
  - loaded from the highest level where it is found
  - processor/cache hides cache access from programmer
- But you can influence it
  - Access x (that puts it in L1), access 100k of data, access x again: it will probably be gone from cache
  - If you use an element twice, do not wait too long
  - If you loop over data, try to take chunk of less than cache size.

Cache mapping

- Because each memory subsystem is smaller than the next closer level, data must be mapped.
- Types of mapping
  - Direct
  - Set associative
  - Fully associative

Direct Mapped Caches

Direct mapped cache: A block from main memory can go in exactly one place in the cache. This is called direct mapped because thee is a direct mapping from any block address in memory to a single location in the cache.



Cache

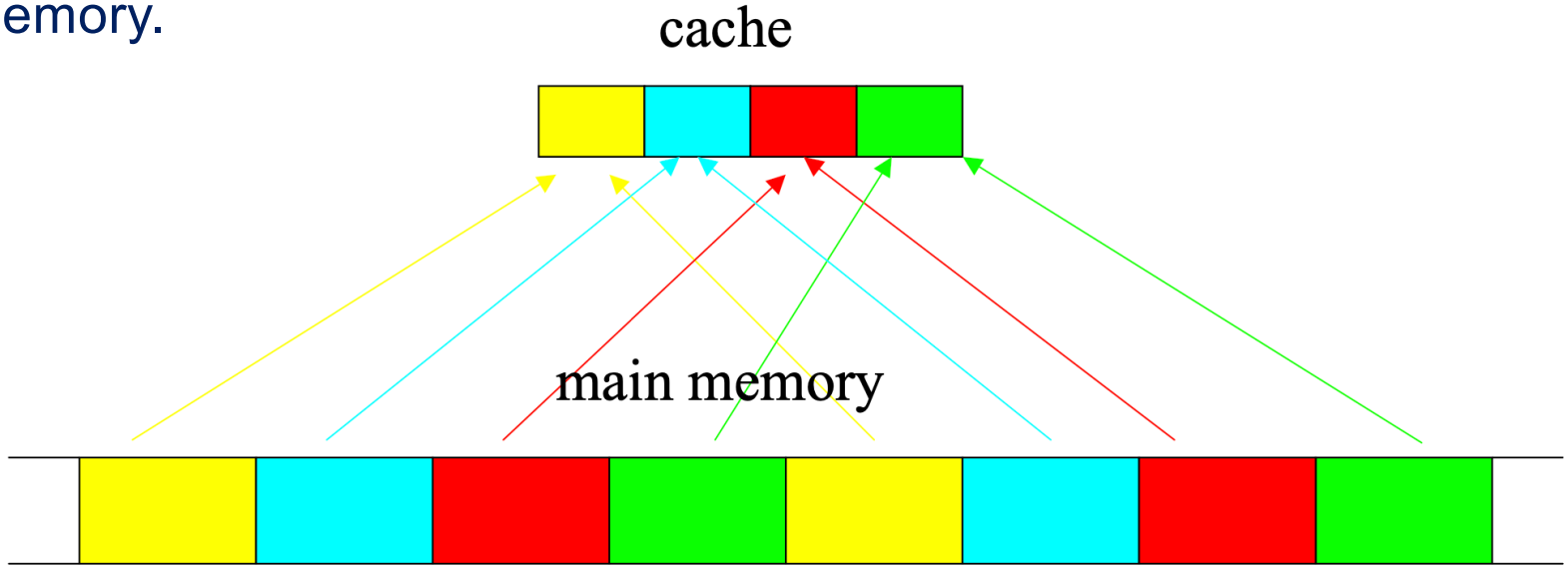
Addr	Value
00	‘acgg’
01	‘wait’
10	42
11	29

Memory

Address	Value
1001010	42
1001011	13
1001100	‘acgg’
1001101	‘wait’
1001110	‘seen’
1001111	29

Direct Mapped Caches

- If the cache size is M and it is divided into k lines, then each cache line is M/k in size (64 bytes typically)
- If the main memory size is N, memory is then divided into N/(M/k) blockes that are mapped into each of the k cache lines.
- Means that each cache line is associated with particular regions of memory.



Direct Mapped Caches

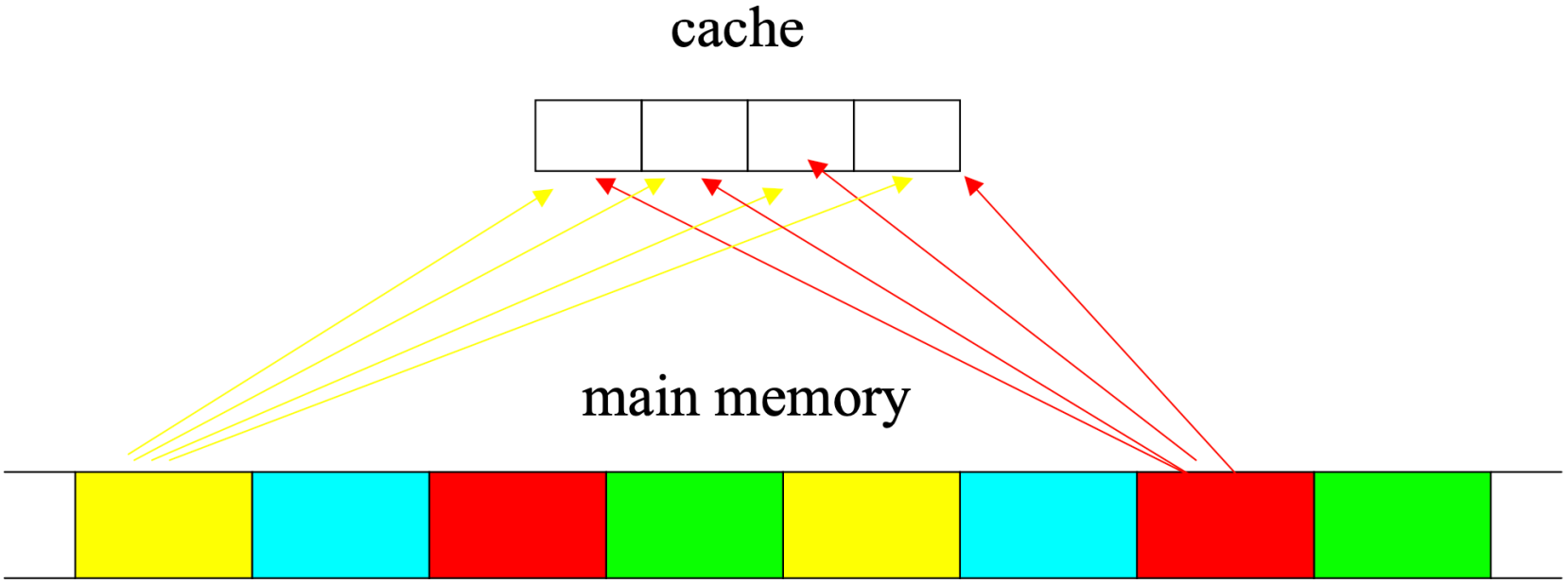
```
double a[8192],b[8192];
for (i=0; i<n; i++) {
  a[i] = b[i]
}
```

- Cache size is 64k = 2^{16} bytes
- a[0] and b[0] are mapped to the same cache location
- Cache line is 32 bytes
- 1 double takes 8 bytes

- Trashing:
  - b[0]…b[3] loaded to cache and register
  - a[0]…a[3] loaded, kicks b[0] … b[3] out of cache
  - b[1] requested, so b[0] ... b[3] loaded again
  - a[1] requested, loaded, kickes b[0] … b[3] out again

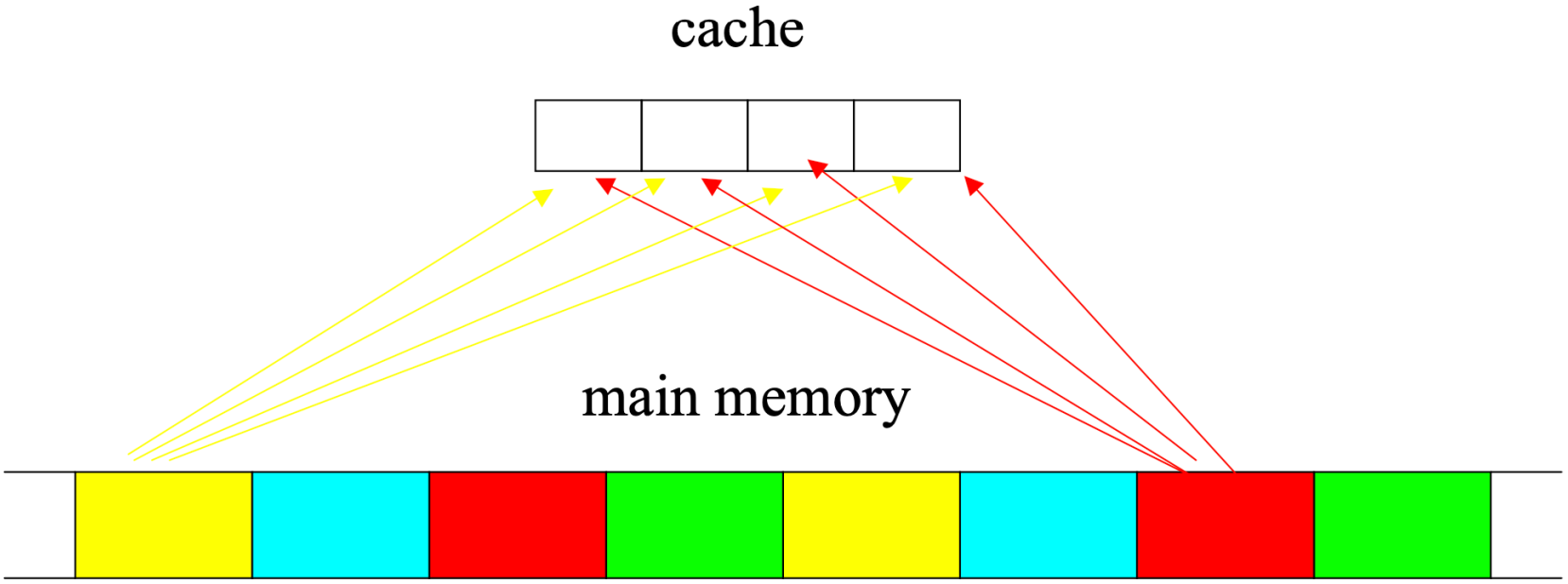
Fully Associative Caches

Fully associative cache: a block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache.



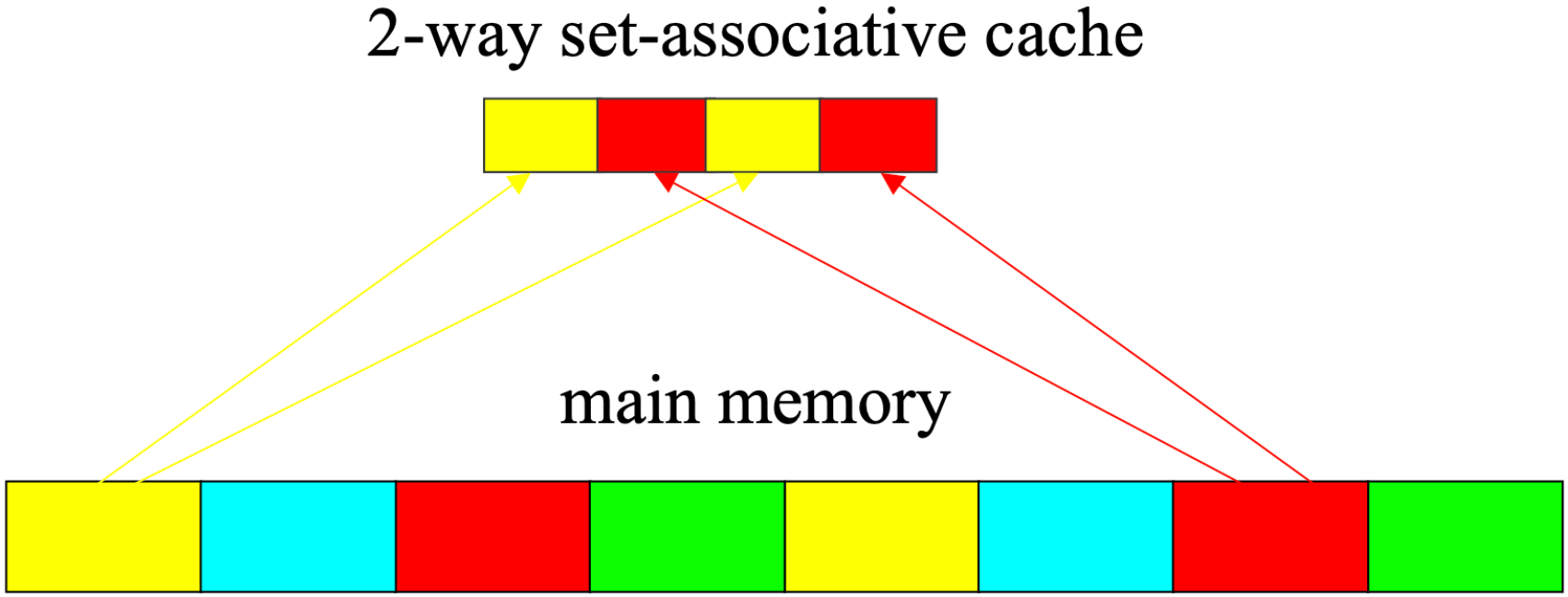
Fully Associative Caches

- Ideal situation
- Any memory location can be associated with any cache line
- Cost prohibitive



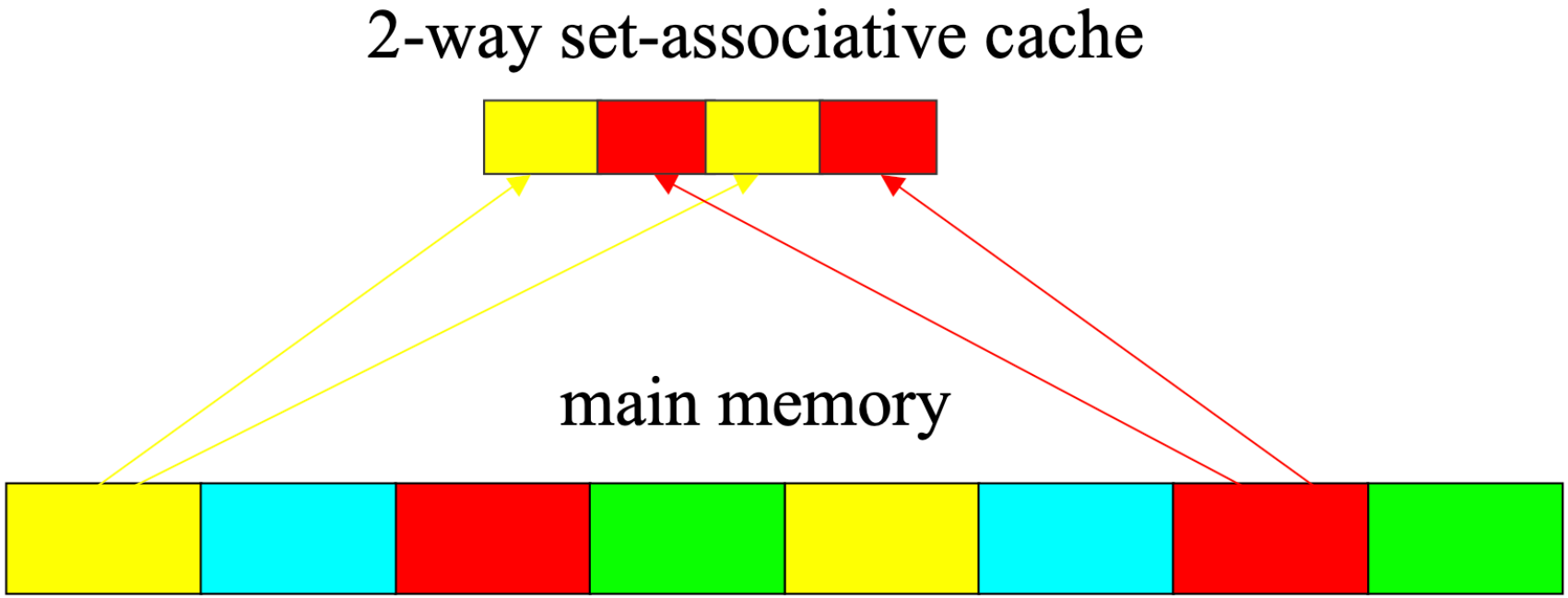
Set Associative Caches

Set associative cache: The middle range of designs between direct mapped cache and fully associative cache. In a *n*-way set associative cache, a block for main memroy can go into *n* locations in the cache.



Set Associative Caches

- Direct mapped cache is 1-way set-associative cache
- For a *n*-way set-associative cache, each memory region can be associated with n cache lines.



Main memory

- **Cheapest for of RAM**
- **Also the slowest**
  - **lowest bandwidth**
  - **highest latency**

Crucial DDR DIMM

- **Unfortunately, most of our data lives here**

Memory hierarchy	*Access latency (cycle)*	*Access latency (ns)*
L1 CACHE	~4 cycles	1.25 ns
L2 CACHE	~10 cycles	3.125 ns
L3 CACHE	~30-50 cycles	9.375-15.625 ns
Local Dram	*~200-300 cycles* (60ns)	62.5-93.75 ns
remote DRAM	*>300 cycles*	*> 90.75 ns*

Multicore Architecture

“太乙”高性能集群系统

系统峰值性能：2.5PFlops

实测持续运算性能：1.687PFlops

登陆节点：4台

配置：2个Xeon Gold 6140 CPU(2.3GHz/18c), 384 GB 内存

计算节点：815台

配置：2个Xeon Gold 6148 CPU(2.4GHz/20c), 192 GB内存

大内存节点：2台

配置：8个Xeon Platinum 8160 CPU (2.1GHz/24c)，6 TB内存

GPU节点：4台

配置：2个Xeon Gold 6148 CPU(2.4GHz/20c)，384 GB内存，2张NVIDIA V100 GPU 卡

存储：5.5 PB

配置：GPFS并行文件系统，实测读写带宽均超过40 GB/s

Multicore Architecture

启明高性能集群系统

系统峰值性能：0.3PFlops

实测持续运算性能：0.191PFlops

登陆节点：4台

配置：2个Intel Xeon E5-2690v3 CPU (2.6GHz/12c)，128 GB内存

刀片节点：230台

配置：2个Intel Xeon E5-2690v3 CPU (2.6GHz/12c)，64 GB内存

大内存节点：7台

配置：8个Intel Xeon E7-8880v3 CPU (2.6GHz/12c)，6 TB内存

GPU节点：6台

配置：2个Intel Xeon E5-2690v3 CPU (2.6GHz/12c)，128 GB内存，4张NVIDIA Tesla K80 GPU 卡

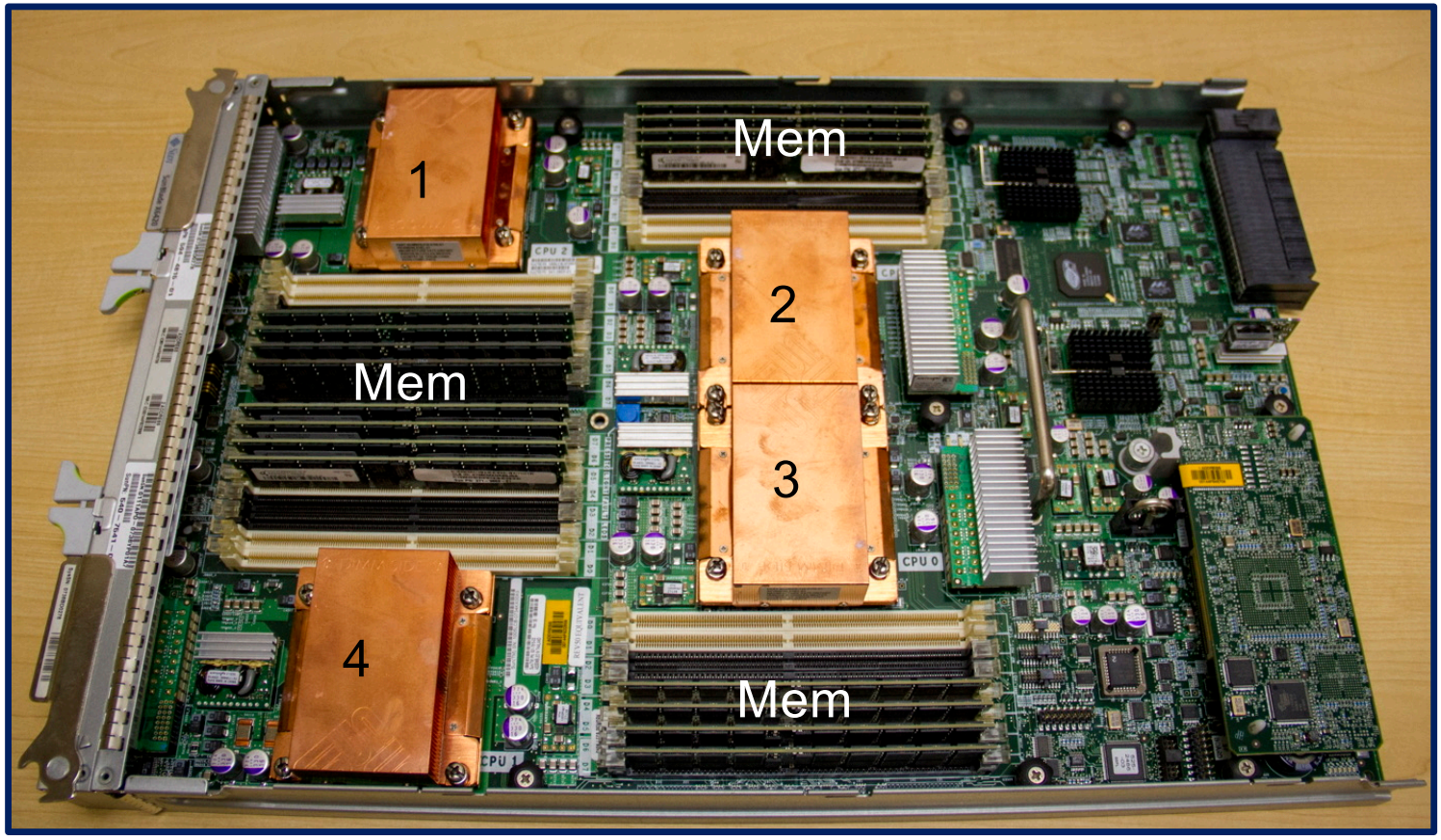
存储：528 TB

配置：Lustre并行文件系统，实测读写带宽超过6 GB/s

Multicore Architecture

Single node

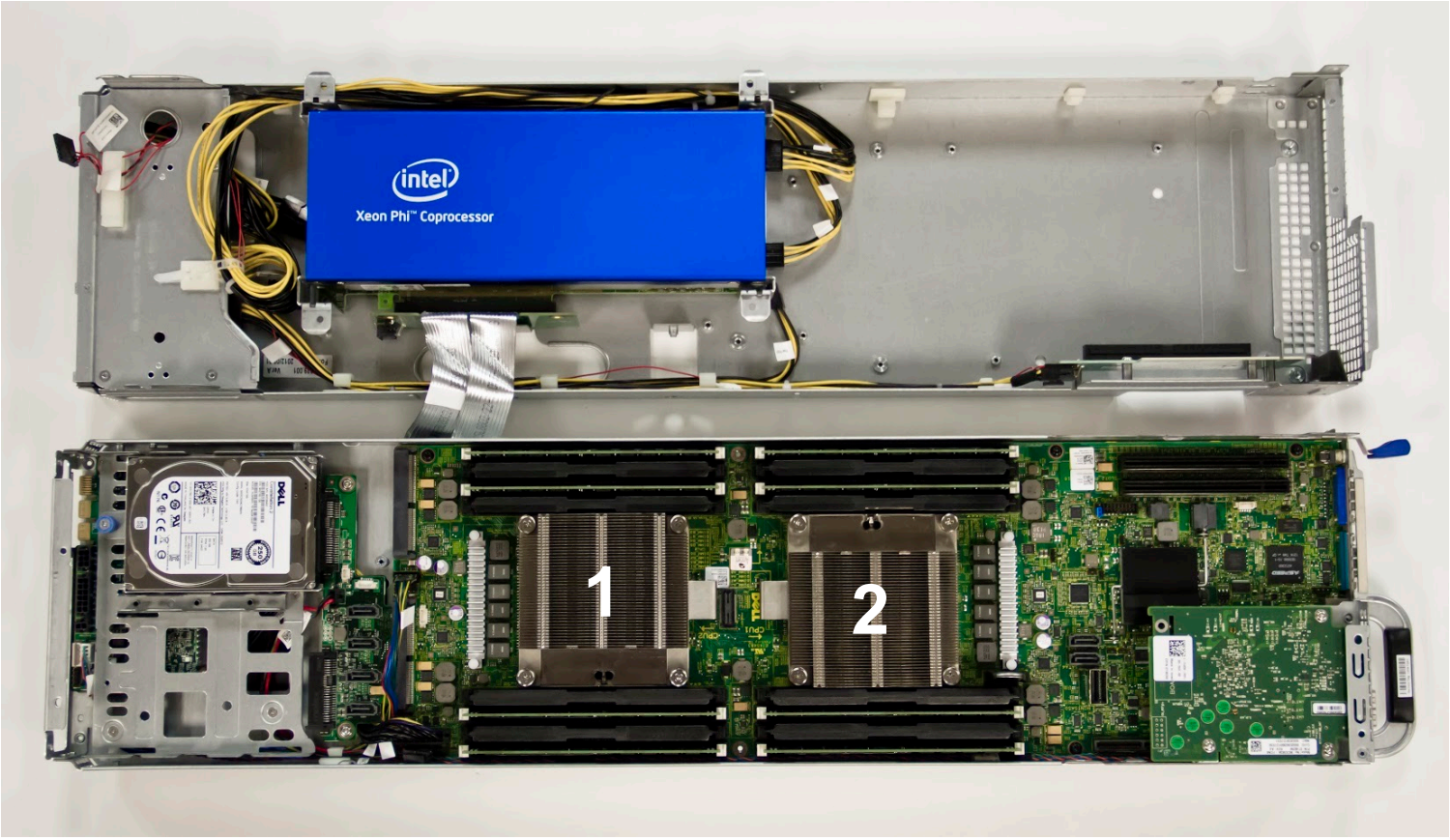
Four sockets



Multicore Architecture

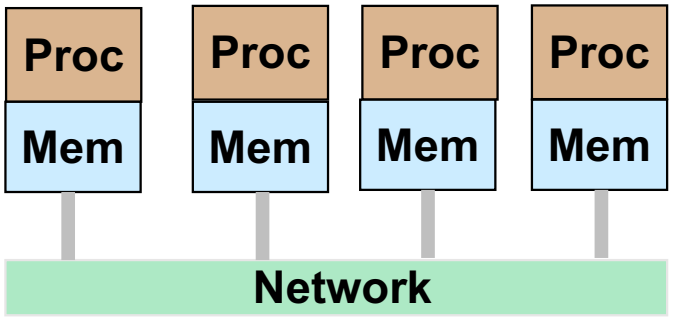
Single node

Two sockets with a coprocessor



Blade server

Non-Uniform Memory Access (NUMA)



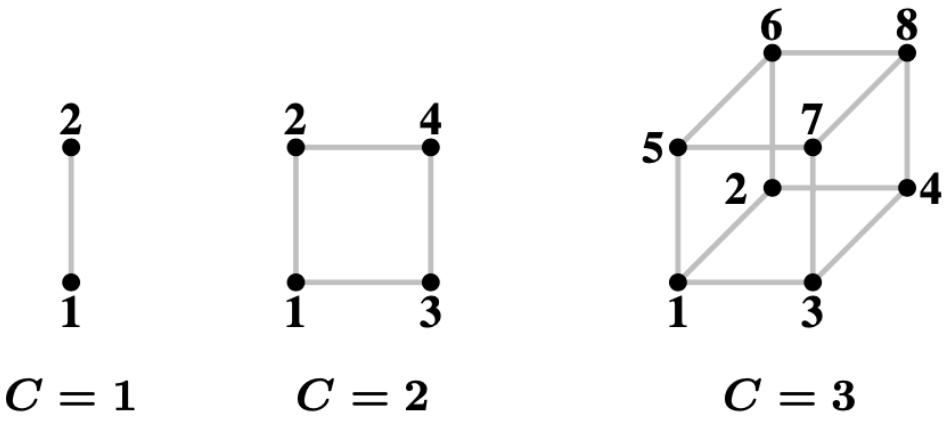
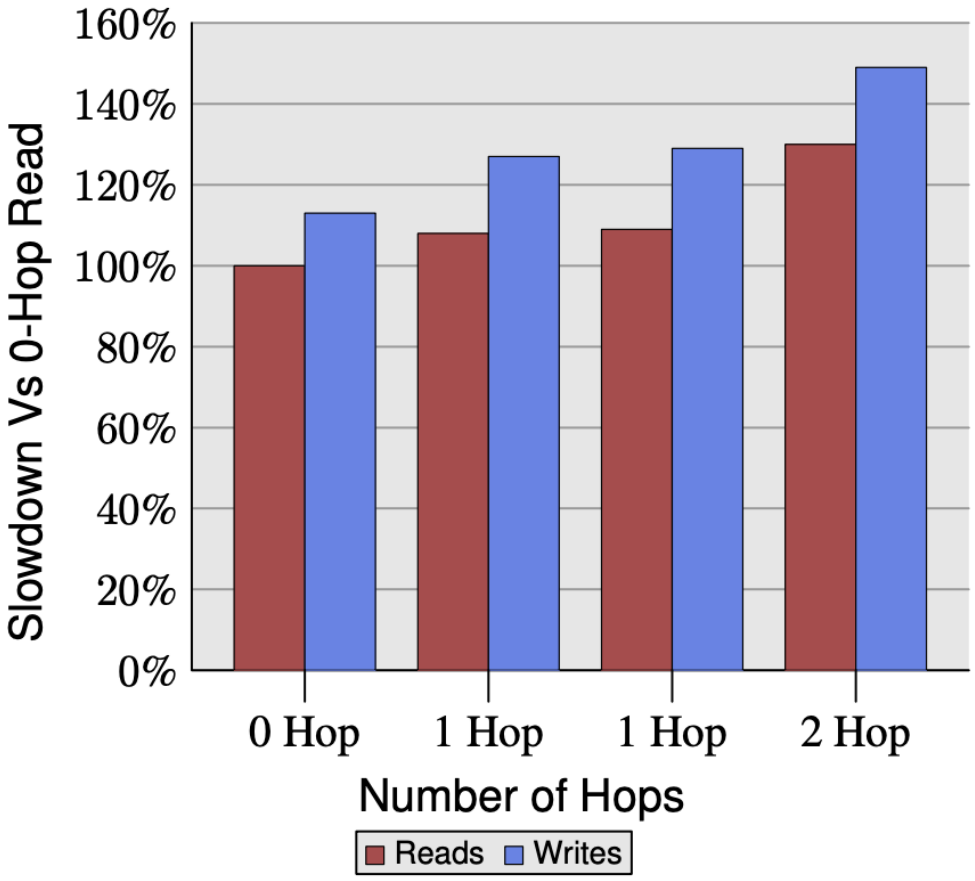


Figure 5.1: Hypercubes

- NUMA is very common, especially in big machines
- A resolution of the bottoleneck between the northbridge and CPU
- Interconnection mechanism provides access to other RAMs
- OS takes care of the distributed nature of the RAMs
- The RAMs collectively serve as a memory pool for all CPUs
- The access time is non-uniform

Non-Uniform Memory Access (NUMA)



- NUMA is very common, especially in big machines
- A resolution of the bottleneck between the northbridge and CPU
- Interconnection mechanism provides access to other RAMs
- OS takes care of the distributed nature of the RAMs
- The RAMs collectively serve as a memory pool for all CPUs
- The access time is non-uniform

Locality

- **Temporal locality** means the current data or instruction that is being fetched may be needed soon.

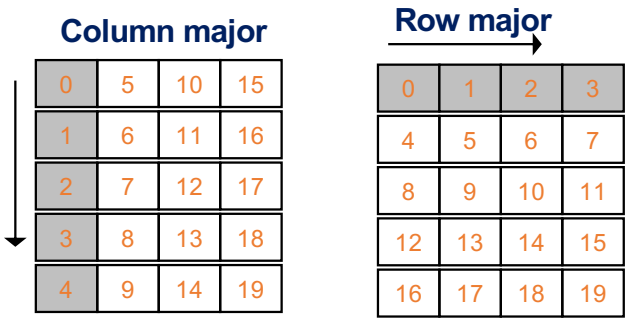
If a variable is used repeatedly or frequently, storing it in a very high-speed memory device very close to the processor will deliver good performance.

- **Spatial locality** means instruction or data near the current memory location that is being fetched may be needed soon in the future.

High spatial locality suggests that the probability of a variable being accessed is higher if one of its adjacent or neighboring variables has been recently accessed.

Locality

- Reuse values in fast memory (bandwidth filtering)
  - need temporal locality in program
- Move larger chunks (achieve higher bandwidth)
  - need spatial locality in program




```
for (int j = 0; j < N; ++j)
{
  for (int i = 0; i < N; ++i)
  {
    A[i][j] = 1.0;
  }
}
```

```
for (int i = 0; i < N; ++i)
{
  for (int j = 0; j < N; ++j)
  {
    A[i][j] = 1.0;
  }
}
```

Locality

- Reuse values in fast memory (bandwidth filtering)
  - need temporal locality in program

- Move larger chunks (achieve higher bandwidth)
  - need spatial locality in program

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

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

Check the machine info

`/sys/devices/system/cpu/cpu*/cache`

		type	level	shared_cpu_map
cpu0	index0	Data	1	00000001
	index1	Instruction	1	00000001
	index2	Unified	2	00000003
cpu1	index0	Data	1	00000002
	index1	Instruction	1	00000002
	index2	Unified	2	00000003
cpu2	index0	Data	1	00000004
	index1	Instruction	1	00000004
	index2	Unified	2	0000000c
cpu3	index0	Data	1	00000008
	index1	Instruction	1	00000008
	index2	Unified	2	0000000c

- **Each core has three caches**
- **L2 cache is unified in storing both instruction and data**
- **shared_cpu_map is a hexadecimal number**

  00000001 → 00000001 : cpu 0
  00000002 → 00000010 : cpu 1
  00000003 → 00000011 : cpu 0 & 1
  00000004 → 00000100 : cpu 3
  00000008 → 00001000 : cpu 4
  0000000c → 00001100 : cpu 3 & 4

Check the machine info

`/sys/devices/system/cpu/cpu*/cache`

		type	level	shared_cpu_map
cpu0	index0	Data	1	00000001
	index1	Instruction	1	00000001
	index2	Unified	2	00000003
cpu1	index0	Data	1	00000002
	index1	Instruction	1	00000002
	index2	Unified	2	00000003
cpu2	index0	Data	1	00000004
	index1	Instruction	1	00000004
	index2	Unified	2	0000000c
cpu3	index0	Data	1	00000008
	index1	Instruction	1	00000008
	index2	Unified	2	0000000c

- Each core has three caches
- L2 cache is unified in storing both instruction and data
- shared_cpu_map is a hexadecimal number
- L1 cache are private
- L2 cache are shared

		type	level	shared_cpu_map
cpu0	index0	Data	1	00000001
	index1	Instruction	1	00000001
	index2	Unified	2	00000001
cpu1	index0	Data	1	00000002
	index1	Instruction	1	00000002
	index2	Unified	2	00000002
cpu2	index0	Data	1	00000004
	index1	Instruction	1	00000004
	index2	Unified	2	00000004
cpu3	index0	Data	1	00000008
	index1	Instruction	1	00000008
	index2	Unified	2	00000008
cpu4	index0	Data	1	00000010
	index1	Instruction	1	00000010
	index2	Unified	2	00000010
cpu5	index0	Data	1	00000020
	index1	Instruction	1	00000020
	index2	Unified	2	00000020
cpu6	index0	Data	1	00000040
	index1	Instruction	1	00000040
	index2	Unified	2	00000040
cpu7	index0	Data	1	00000080
	index1	Instruction	1	00000080
	index2	Unified	2	00000080

achine info

`ı/cpu/cpu*/cache`

- Each core has three caches
- L2 cache is unified in storing both instruction and data
- shared_cpu_map is a hexadecimal number
- L1 cache are private
- L2 cache are private also

Check the machine info

`/sys/devices/system/cpu/cpu*/node`

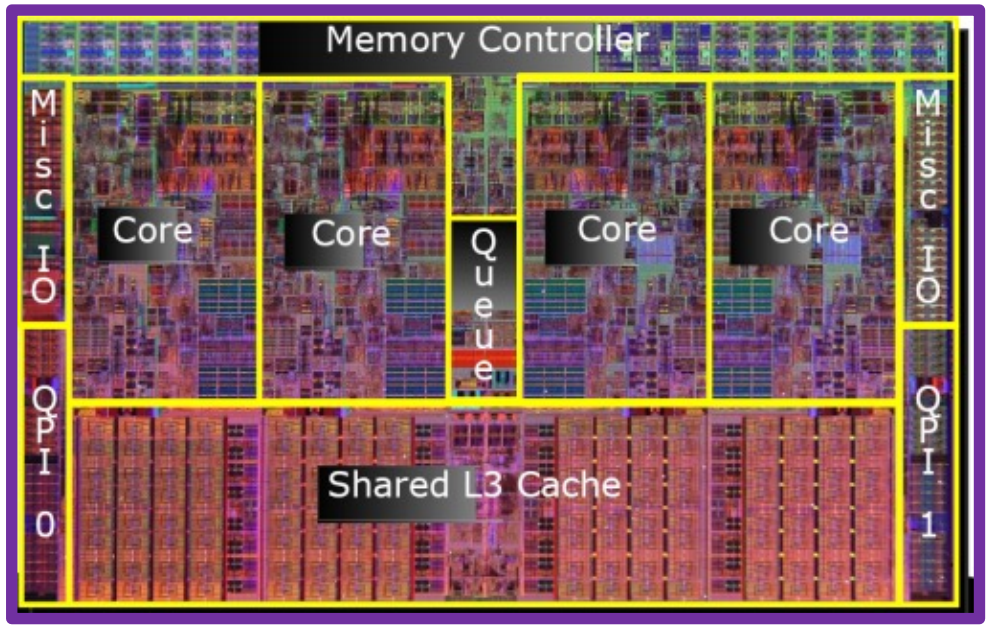
	cpumap	distance
node0	00000003	10 20 20 20
node1	0000000c	20 10 20 20
node2	00000030	20 20 10 20
node3	000000c0	20 20 20 10

- **NUMA node information**
- **cpumap lists which CPUs belong to a given NUMA node**
  convert 00000003 to
  0000 0000 0000 0011
  meaning CPUs 0&1 belong to node 0

  0000000c → 0000 0000 0000 1100
  00000030 → 0000 0000 0011 0000
  000000c0 → 0000 0000 1100 0000
- **distance values indicate the relative latency of memory access**

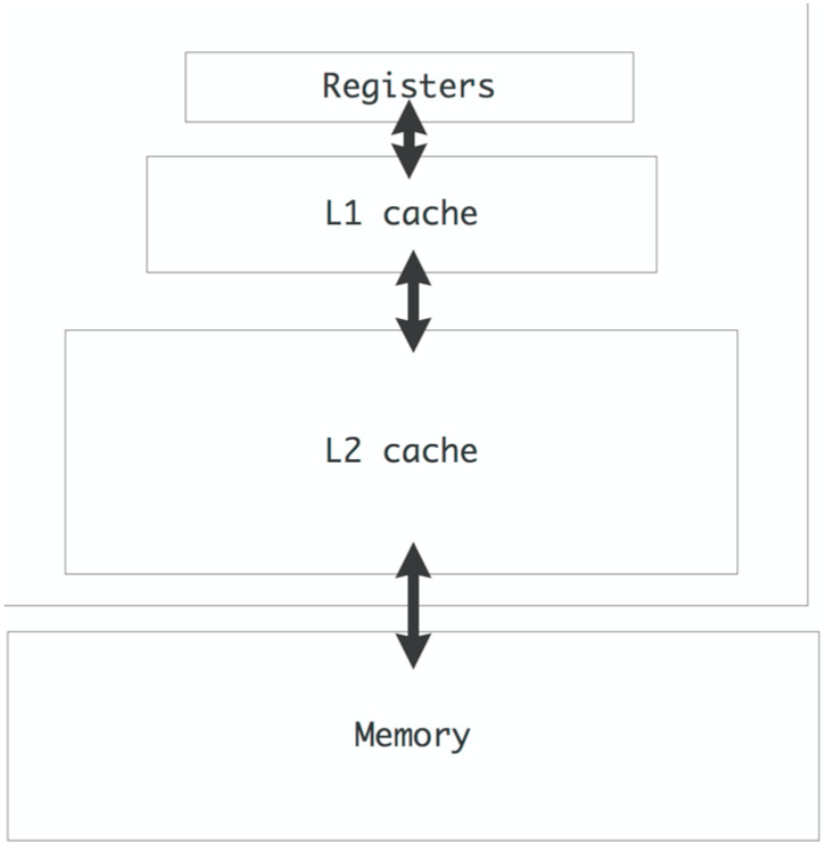
Multicore Architecture

- One of the ways of getting higher utilization out of a single processor chip is then to move from further sophistication of a single processor to a division of the chip into multiple processing **cores**.
- The cores collaborate on a common task at a higher overall efficiency.
- Two cores at lower frequency can have the same throughput as a single processr at higher frequency.

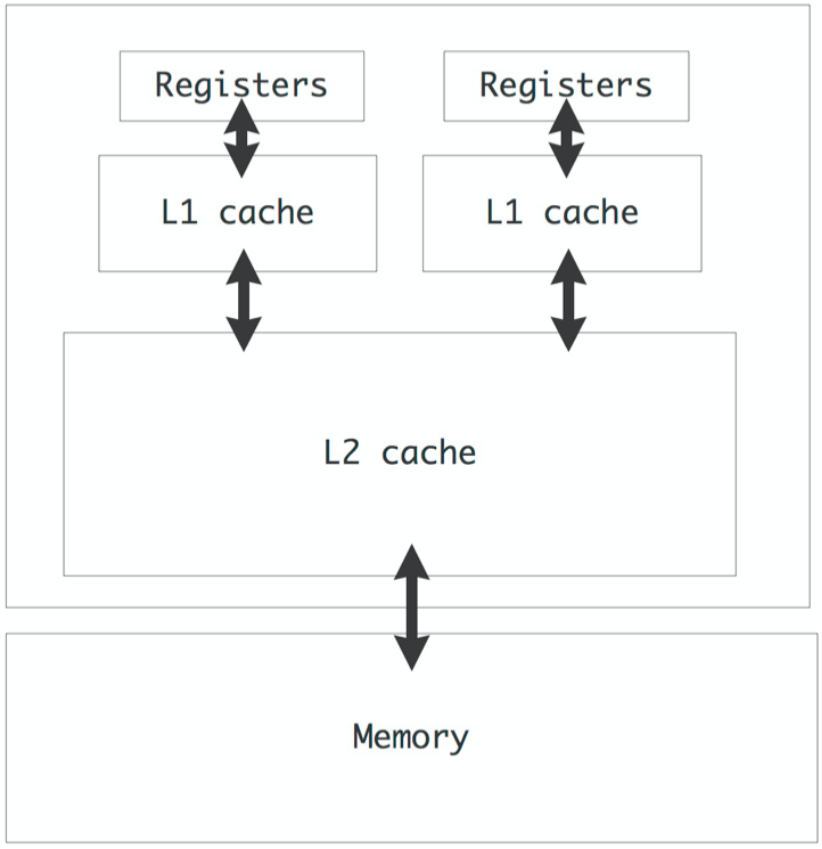


Multicore Architecture

Single-core chip



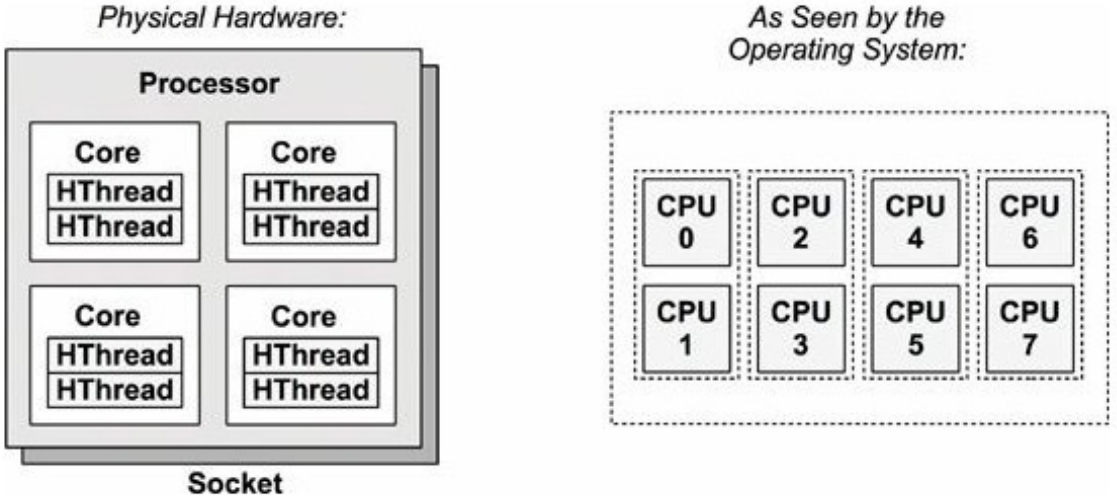
Dual-core chip



Multicore Architecture

The terminology `processor' is ambiguous nowadays. To distinguish them we have the following.

- **Core**: the core have their own private L1 cache.
- **Socket**: on one socket, there is often a **shared L2 or L3 cache**, which is shared memory for the cores.



Check the machine info

`/sys/devices/system/cpu/cpu*/topology`

	physical_ package_id	core_id	core_ siblings	thread_ siblings
cpu0	0	0	00000003	00000001
cpu1		1	00000003	00000002
cpu2	1	0	0000000c	00000004
cpu3		1	0000000c	00000008
cpu4	2	0	00000030	00000010
cpu5		1	00000030	00000020
cpu6	3	0	000000c0	00000040
cpu7		1	000000c0	00000080

- physical_package_id : physical socket
- Each socket has two cores
- core_siblings is a hexadecimal number
  - dual core in each processor
  - e.g. 00000030 means CPU 4&5 in package 2
- thread_siblings show the logical thread of the physical core
  - there is no hyperthreading

Multicore Architecture

The terminology `processor' is ambiguous nowadays. To distinguish them we have the following.

- **Core**: the core have their own private L1 cache.
- **Socket**: on one socket, there is often a **shared L2 or L3 cache**, which is shared memory for the cores.
- **Node**: there can be multiple sockets on a single `node' or motherboard, accessing the same shared memory.
- **Network**: Distributed memory programming need this for node communications.

1. Memory hierarchies
   - von Neumann bottleneck
   - register, cache, and main memory
   - cache details

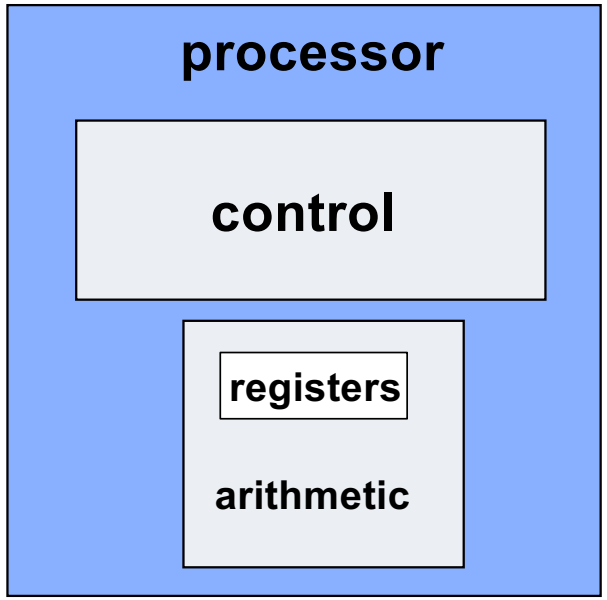
2. Parallelism within single processors
   - Pipelining
   - SIMD
   - Special instructions (FMA)

3. Case study: Matrix multiplication

4. Roofline model

Uniprocessor model

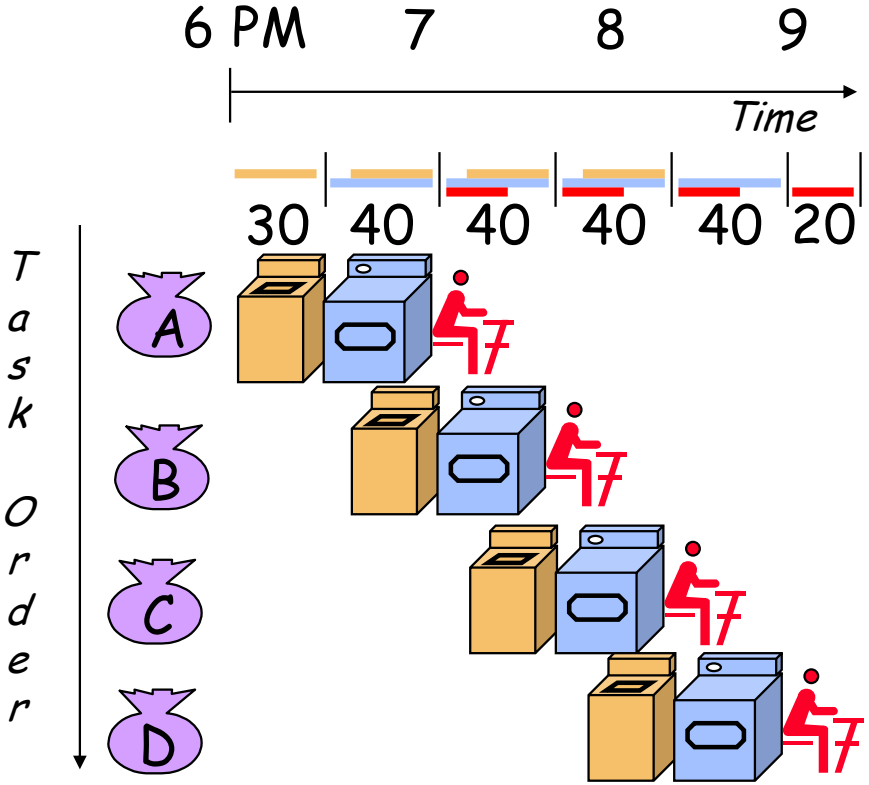
- **Processor names variables:**
  - Integers, doubles, pointers, arrays, classes, etc.
  - They are really words, e.g. 64-bit doubles, 32-bit ints, etc.
- **Processor performs operations:**
  - Arithmetic, logical operations, etc.
  - They are only performed on values in registers.
- **Processor control the order of operations:**
  - Branches, loops, functions, etc.
- **Each operation cost approximately the same.**



Pipelining

Dave Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min



In this example:

- Sequential execution takes 4 * 90min = 6 hours
- Pipelined execution takes 30+4*40+20 = 3.5 hours

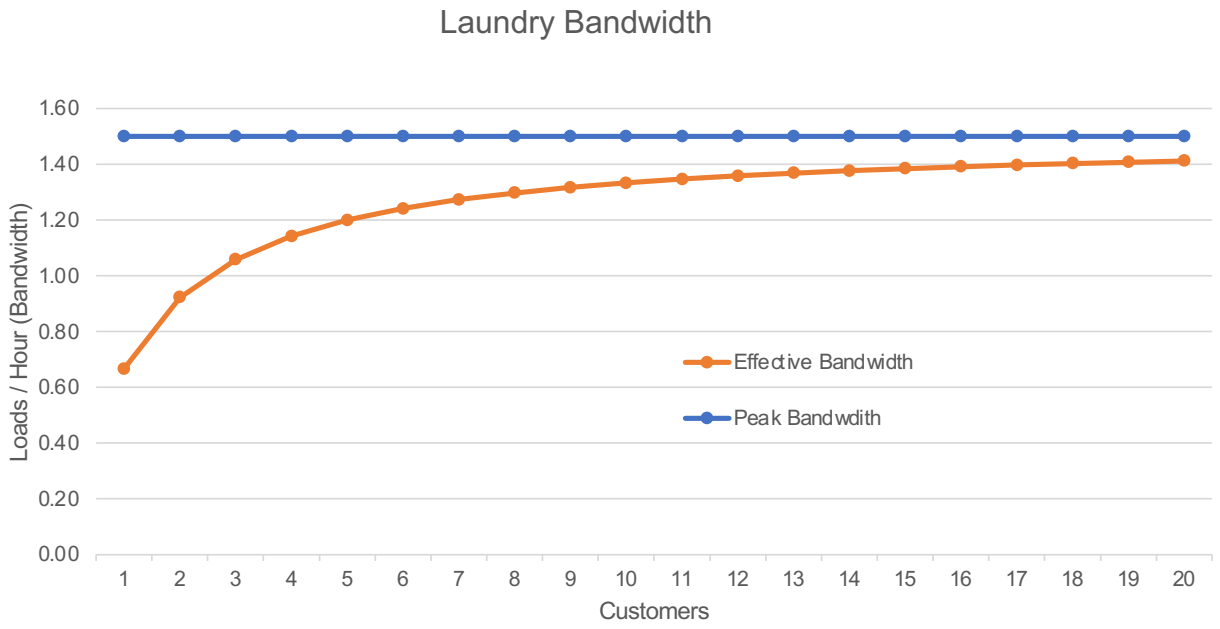
loads/hour

- 4/6 l/h w/o pipelining
- 4/3.5 l/h w pipelining

Speedup = 6/3.5 <= # of stages=3

Laundry peak performance

- The system bandwidth is 1 load per 40 minutes (1.5 Loads / hour)
- How much of that you see depends on available “customers” ready to do laundry



Uniprocessor model

An **addition** instruction includes the following.

- Decoding the instruction, including finding the locations of the operands.
- Copying the operands into registers, known as data fetch.
- Aligning the exponents, e.g. 0.35 e-1 + 0.6 e-2 becomes

  0.35 e-1 + 0.06 e-1

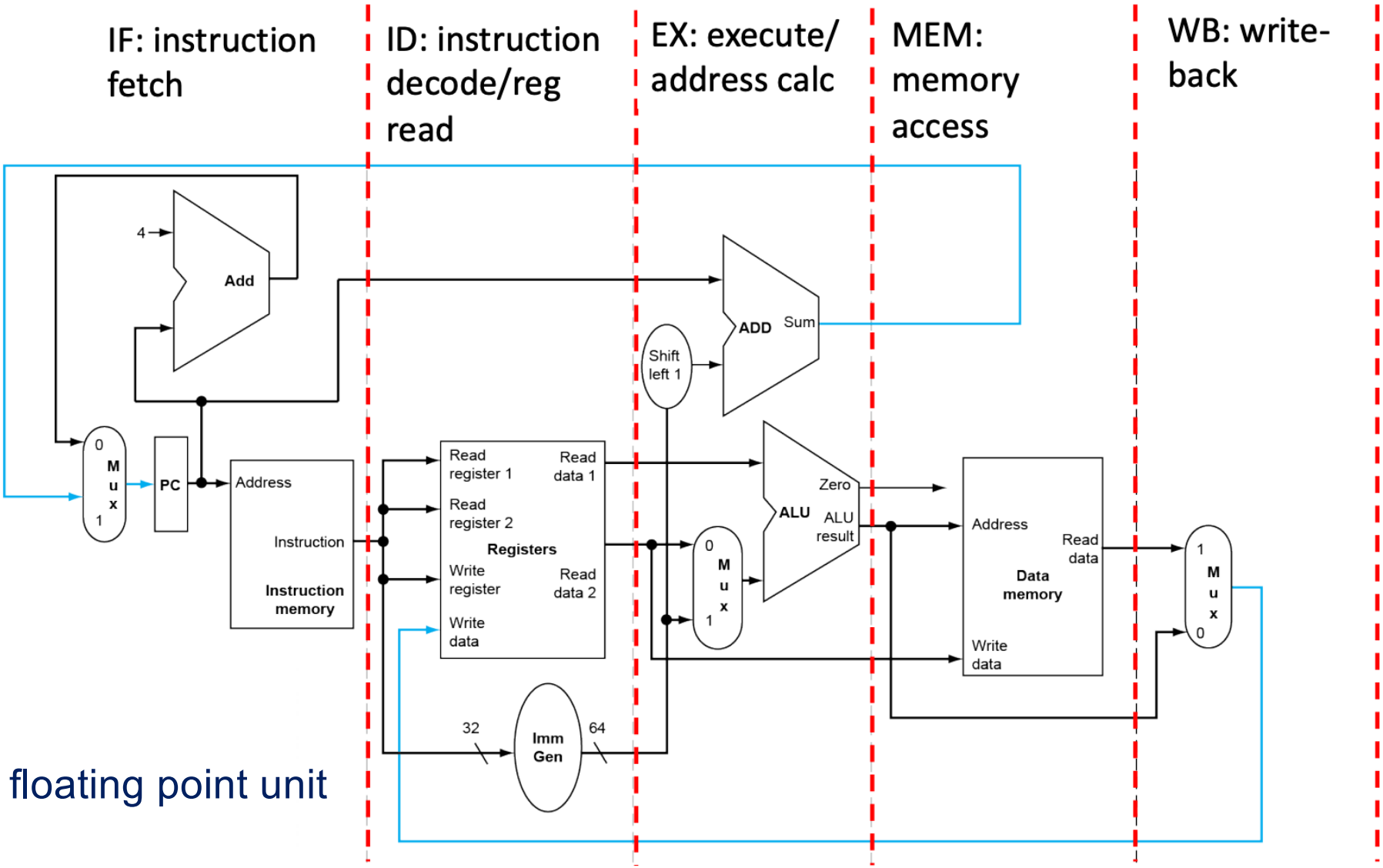
- Executing the addition, in this case 0.41 e-1
- Storing the result.

The above are often called the **stages** or **segments** of the **pipeline**.

$$t(n) = nl\tau$$

number of stages

clock cycle time per stage



Pipelined floating point unit

Pipelining

An **addition** instruction includes the following.

- Decoding the instruction, including finding the locations of the operands.
- Copying the operands into registers, known as data fetch.
- Aligning the exponents, e.g. 0.35 e-1 + 0.6 e-2 becomes

  0.35 e-1 + 0.06 e-2

- Executing the addition, in this case 0.41 e-1
- Storing the result.

The above are often called the **stages** or **segments** of the **pipeline**.

$$t(n) = [s + n + l - 1]\tau$$

clock cycle time per stage

setup cost

number of stages

Pipelining

Without pipelining, traditional floating point processing unit takes $\boldsymbol{t(n) = nl\tau}$ to do *n* additions.

With pipelining, FPU takes $\boldsymbol{t(n) = [s + n + l - 1]\tau}$ for *n* additions.

Addition performance rate $\boldsymbol{r_n := \frac{n}{t(n)}}$.

Traditional: $\boldsymbol{r_\infty = \frac{1}{l\tau}}$ Pipelined: $\boldsymbol{r_\infty = \frac{1}{\tau}}$

Ideally, speedup by the number of stages l

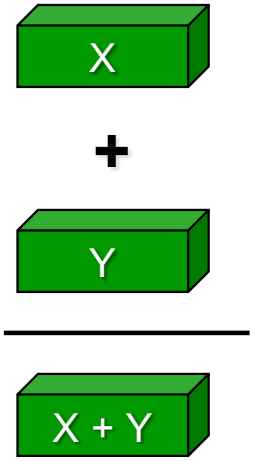
Pipelining

- Pipelining is the key technique to make today's CPUs fast
- All processors since about 1985 use pipelining to improve their performance
- It is based on the fact that to execute one instruction, multiple clocks are required (an instruction is usually decomposed into multiple micro-instructions)
- Therefore, the pipelining technique reduces the number of clock cycles per instruction.
- Speed is limited by overhead

SIMD: Single Instruction Multiple Data

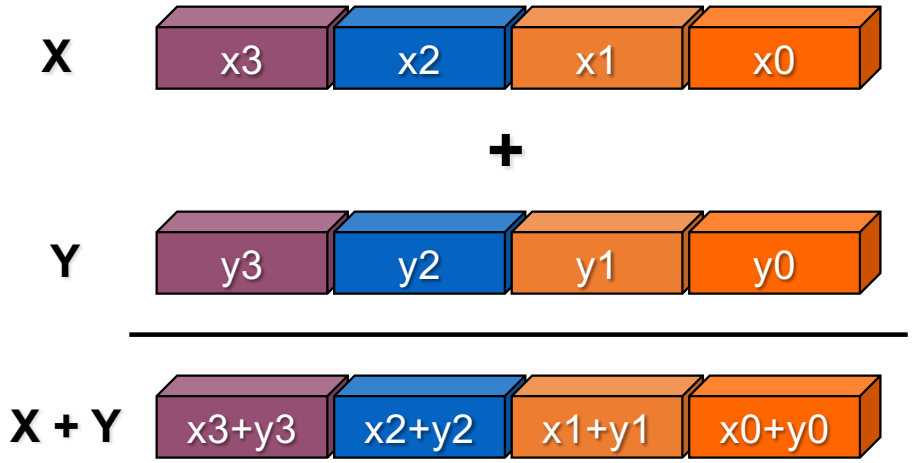
Scalar processing

- traditional mode
- one operation produces one result



SIMD processing: vectors

- Pentium: SSE2 (128 bit)
- Sandy Bridge: AVX (256 bit)
- Haswell: AVX2 (256 bit w/ FMA)
- KNL: AVX-512 (512 bit)

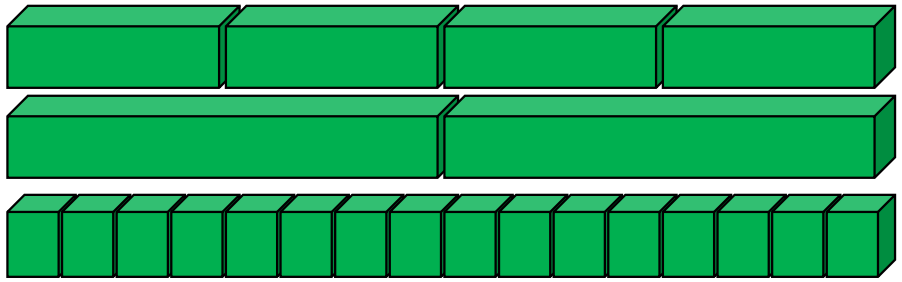


https://www.nersc.gov/assets/Uploads/Vectorization.pdf

SIMD: Single Instruction Multiple Data

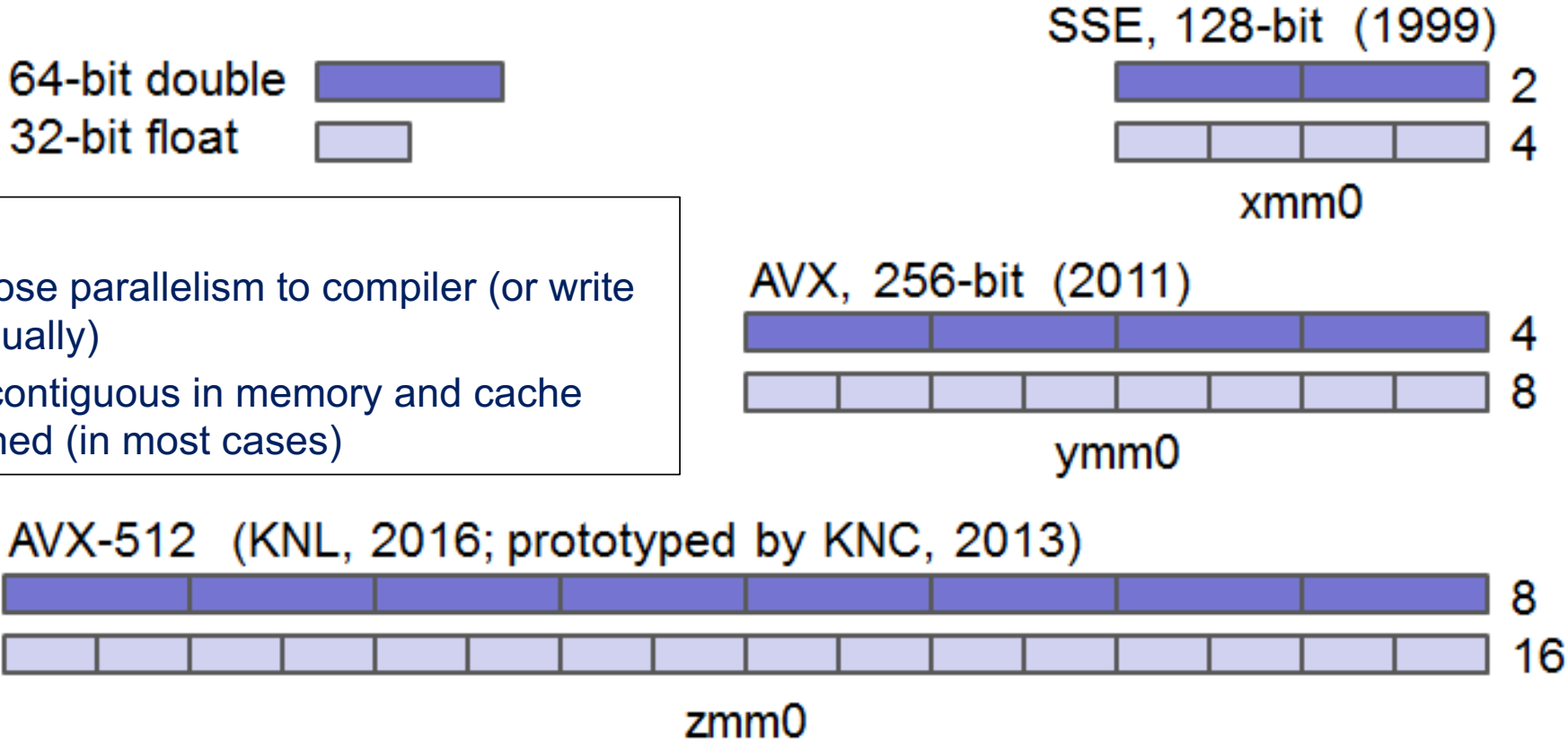
SIMD: Single Instruction Multiple Data

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and cache aligned
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

SIMD: Single Instruction Multiple Data



Need to:

- Expose parallelism to compiler (or write manually)
- Be contiguous in memory and cache aligned (in most cases)

SIMD: Single Instruction Multiple Data

Parallelism can get the wrong answer if instructions execute out of order

Types of dependencies

- RAW: Read-After-Write
  X = A ; B = X;
- WAR: Write-After-Read
  A = X ; X = B;
- WAW: Write-After-Write
  X = A ; X = B;
- No problem / dependence for RAR: Read-After-Read

Hardware can't break these

Compiler can't either

SIMD: Single Instruction Multiple Data

Intel compiler start vectorizing at optimization level –O2

to inline functions: -ipo

to tune the vectors to the host machine: -xHost

to compile for AVX-512F instructions: -xCOMMON-AVX512

GCC compiler start vectoriztion at optimization level –O3

to tune the vectors to the host machine –march-native

On TaiYi, it has been recommended to use –xSKYLAKE-AVX512

SIMD: loop unrolling

```
for ii = 1 , 6
    for jj = 1 , n
       a(ii, jj) = a( ii, jj ) * 2 + 1.0
```

```
for jj = 1 , n
    for ii = 1 , 6
       a(ii, jj) = a( ii, jj ) * 2 + 1.0
```

```
for jj = 1 , n
    a(1, jj) = a( 1, jj ) * 2 + 1.0
    a(2, jj) = a( 2, jj ) * 2 + 1.0
    a(3, jj) = a( 3, jj ) * 2 + 1.0
    a(4, jj) = a( 4, jj ) * 2 + 1.0
    a(5, jj) = a( 5, jj ) * 2 + 1.0
    a(6, jj) = a( 6, jj ) * 2 + 1.0
```

If the inner loop is small, we may consider unroll the inner loop. SIMD and FMA makes it really fast. Compilers can do this for you (will talk about this).

SIMD: Single Instruction Multiple Data

Basic requirements of vectorizable loops:

- Number of iterations is known on entry
  - No conditional termination
- Single control flow; no if or switch statement
- Must be the innermost loop if nested
  - Compiler may reorder loops as an optimization.
- No functions calls but basic math: pow(), sqrt(), sin(), etc.
  - Compiler may inline functions as an optimization.
- All loop iterations must be independent of each other

Instruction-level parallelism

- **SIMD**: instructions that are independent can be started at the same time (is generalized to multi-threading)
- **pipelining**: arithmetic units can deal with multiple operations in various stages of completion
- **out-of-order execution**: instruction can be rearranged if they are not dependent on each other
- **prefetching**: data can be speculatively requested before any instruction needing it is actually encountered. (predictable code is good!)

```
for (int i=0; i<1024; i++) {
    array1[i] = 2 * array1[i];
}
```

```
for (int i=0; i<1024; i++) {
    prefetch (array1 [i + k]);
    array1[i] = 2 * array1[i];
}
```

Fused Multiply-Add instructions

- Arithmetic instructions don't all have the same cost
- Multiply followed by add is very common on programs

  x = y + c * z

- Useful in matrix multiplication
- Fused Multiply-Add (**FMA**) instructions:
  - Performs multiply/add, at the same rate as + or * alone
- **Division** operation (/) is NOT nearly as much optimized in a modern CPU as the additions and multiplications are. It takes 10 to 20 cycles.

Fused Multiply-Add instructions

- **Division** operation (/) is NOT nearly as much optimized in a modern CPU as the additions and multiplications are. It takes 10 to 20 cycles.

Example:

```
for ii =1,…,n
      a[ii] = b[ii] +
a[ii] / c


double inv_c = 1.0 / c;
for ii = 1, … n
      a[ii] = a[ii] *
inv_c + b[ii]
```

- More: use x*x rather than std::pow(x,2);
  use tmp = exp(t); tmp – tmp*tmp rather than exp(t) – exp(2*t)
  … …

What does this mean?

- In theory, the compiler understands all of this
  - It will rearrange instructions to maximizes parallelism, uses FMAs and SIMD
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Links to optimized libraries written by people who know all the tricks
  - Using special functions (“intrinsics”) or write in assembly

1. Memory hierarchies
    - von Neumann bottleneck
    - register, cache, and main memory
    - cache details

2. Parallelism within single processors
    - Pipelining
    - SIMD
    - Special instructions (FMA)

3. Case study: Matrix multiplication

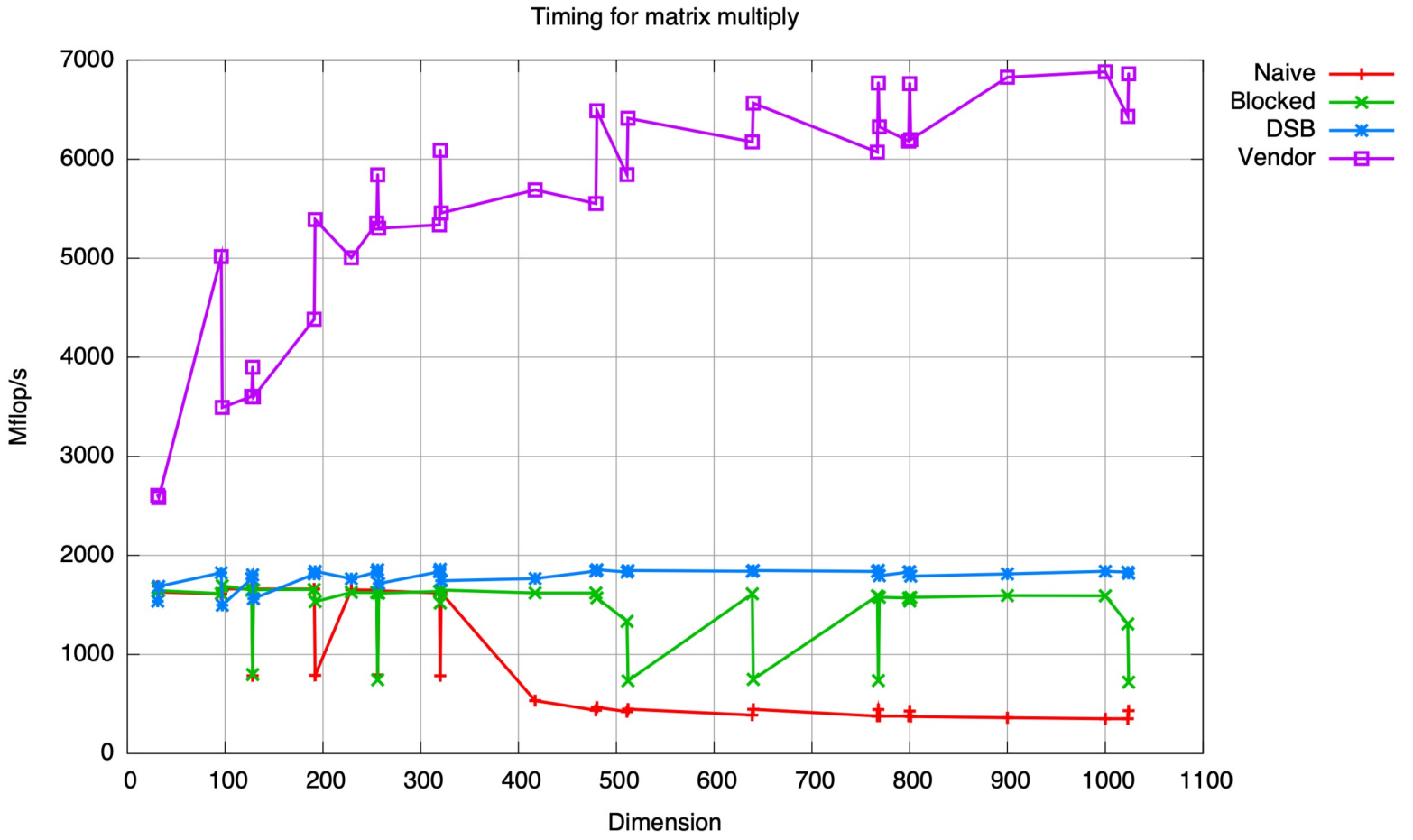
4. Roofline model

Case study: matrix multiplication

- **An important kernel in many problems**
  - **Appears in many linear algebra algorithms**

    Bottleneck for dense linear algebra, including Top500
  - **One of the motifs of parallel computing**
  - **Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall**
  - **And dominates training time in deep learning (CNNs)**
- **Optimization ideas can be used in other problems**
- **The most-studied algorithm in high performance computing**

Commerical and CSE applications in common



Matrix storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout

  by column, or "column major" (Fortran default); A(i,j) at A+i+j*n

  by row, or "row major" (C default) A(i,j) at A+i*n+j

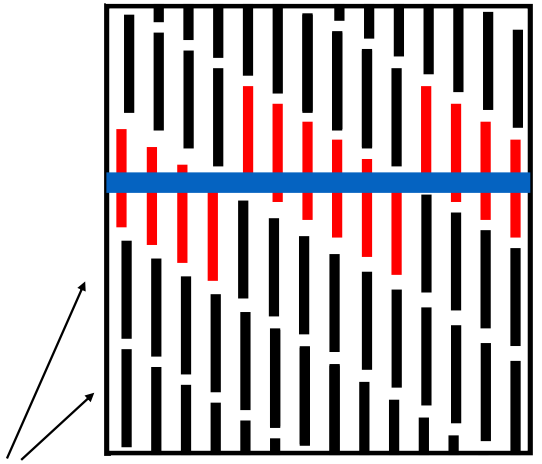
Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Column major matrix in memory



cache lines

Blue row of matrix is stored in red cache lines

A simple model for memory analysis

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory

  m = number of memory elements moved between fast and slow memory

  t_m = time per slow memory operation

  f = number of arithmetic operations

  t_f = time per arithmetic operation << t_m

  q = f / m average number of flops per slow memory access

Computational intensity: key to algorithm efficiency

- Minimum possible time = f * t_f when all data in fast memory
- Actual time

  f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)

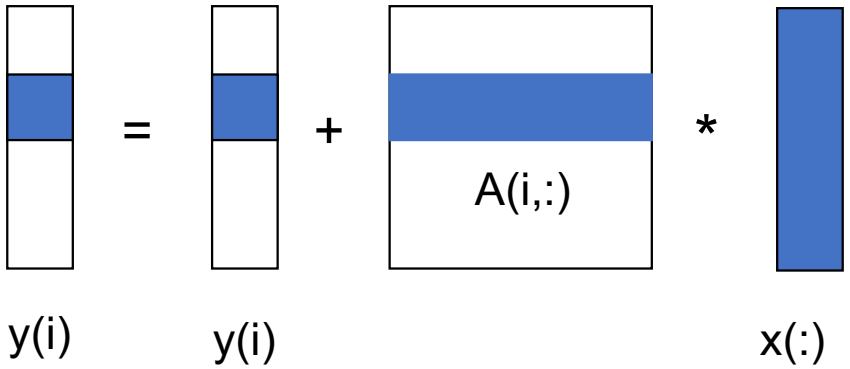
Key to machine efficiency

- Larger q means time closer to minimum f * t_f

  $q \geq t_m/t_f$ needed to get at least half of peak speed

Matrix-vector multiplication

```
{implements y = y + A*x}
        for i = 1:n
            for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Matrix-vector multiplication

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
        {read row i of A into fast memory}
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- m = number of slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- q = f / m ≈ 2 (Low Computational Intensity)

- Matrix-vector multiplication limited by slow memory speed

Matrix-vector multiplication

- Compute time for *nxn* = 1000x1000 matrix
- Time

  $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$

  $= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$
- For t_f and t_m, using data from R. Vuduc's PhD (pp 351-3)

  http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

  - m = number of slow memory refs = $3n + n^2$
  - f = number of arithmetic operations = $2n^2$
  - q = f / m ≈ 2 (Low Computational Intensity)

  - Matrix-vector multiplication limited by slow memory speed

Matrix-vector multiplication

Compute time for n x n = 1000 x 1000 matrix

Time

$$f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$$
$$= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$$

For t_f and t_m, using data from R. Vuduc's PhD (pp 351-3)

http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

For t_m use minimum-memory-latency / words-per-cache-line

t_f = 1 / Peak Mflop/s

	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

***machine balance* (q must be at least this for ½ peak speed)**

Matrix-vector multiplication

Compute time for nxn = 1000x1000 matrix

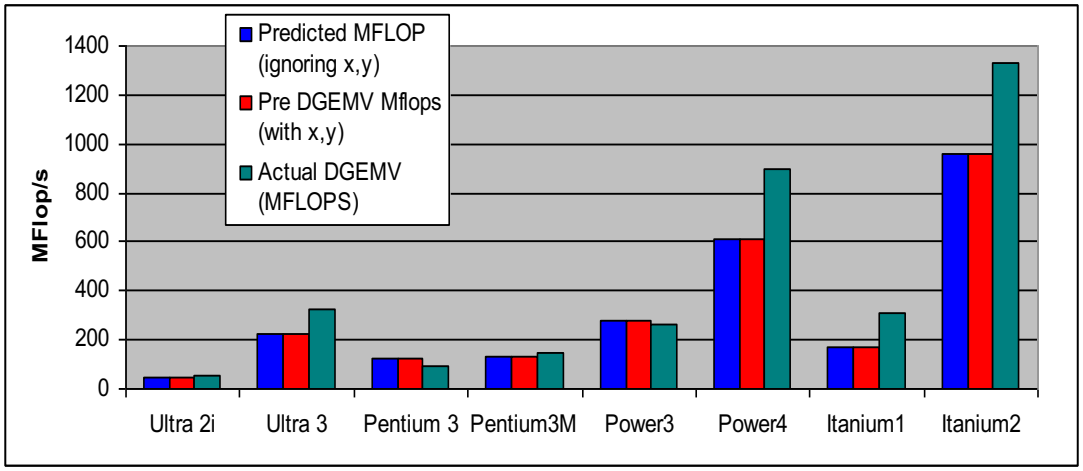
Time

$$f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$$
$$= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$$

For t_f and t_m, using data from R. Vuduc's PhD (pp 351-3)

http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

For t_m use minimum-memory-latency / words-per-cache-line



- **Model sufficient to compare across machines**
- **Under-prediction due to latency estimate**

Matrix-vector multiplication

- **What simplifying assumptions did we make in this analysis?**
  - **Ignored parallelism in processor between memory and arithmetic within the processor**

    Sometimes drop arithmetic term in this type of analysis
  - **Assumed fast memory was large enough to hold three vectors**

    Reasonable if we are talking about any level of cache

    Not if we are talking about registers (~256 bytes)
  - **Assumed the cost of a fast memory access is 0**

    Reasonable if we are talking about registers

    Not necessarily if we are talking about cache (1-2 cycles for L1)
  - **Memory latency is constant**
- **Could simplify even further by ignoring memory operations in X and Y vectors**

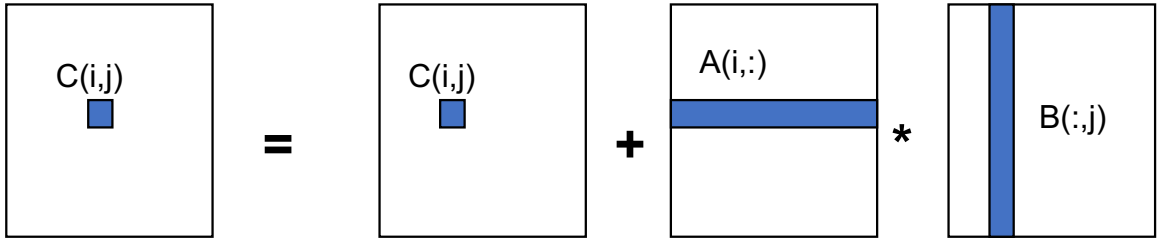
  time to read matrix may dominate

Matrix multiplication

```
{implements C = C + A*B}
        for i = 1 to n
                for j = 1 to n
                        for k = 1 to n
                                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

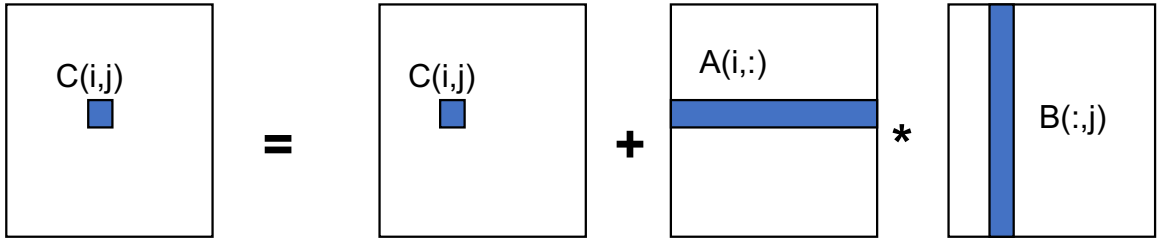
Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

Computational intensity (q) *potentially* as large as $2*n^3 / 3*n^2 = O(n)$



Matrix multiplication

```
{implements C = C + A*B}
        for i = 1 to n
         {read row i of A into fast memory}
                for j = 1 to n
                 {read C(i,j) into fast memory}
                 {read column j of B into fast memory}
                        for k = 1 to n
                                C(i,j) = C(i,j) + A(i,k) * B(k,j)
                 {write C(i,j) into fast memory}
```



Matrix multiplication

Number of slow memory references on unblocked matrix multiply

$m = n^3$ to read each column of B nxn times

$+ n^2$ to read each row of A once

$+ 2n^2$ to read and write each element of C once

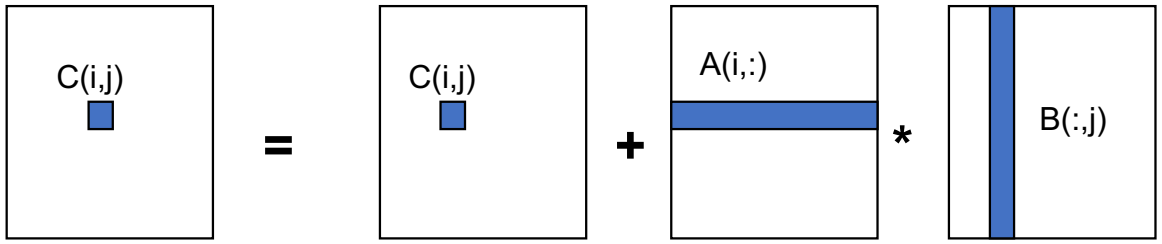
$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$ computational intensity

≈ 2 for large n, no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row i of A times B

Similar for any other order of 3 loops



Matrix multiplication

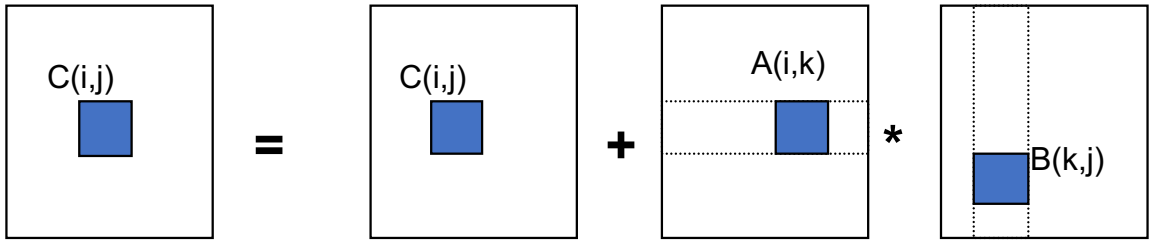
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
for i = 1 to N
        for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
             C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
         {write block C(i,j) back to slow memory}
```

cache does this automatically

3 nested loops inside

block size = loop bounds



Tiling for registers or caches

Matrix multiplication

Recall:

- m is the amount of memory traffic between slow and fast memory
- matrix has nxn elements, and NxN blocks each of size bxb
- f is number of floating-point operations, $2n^3$ for this problem
- q = f / m is our measure of computational intensity

So:

m = $N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

+ $N*n^2$ read each block of A N^3 times

+ $2n^2$ read and write each block of C once ($2N^2 * b^2 = 2n^2$)

= $(2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\approx n / N = b$ for large n

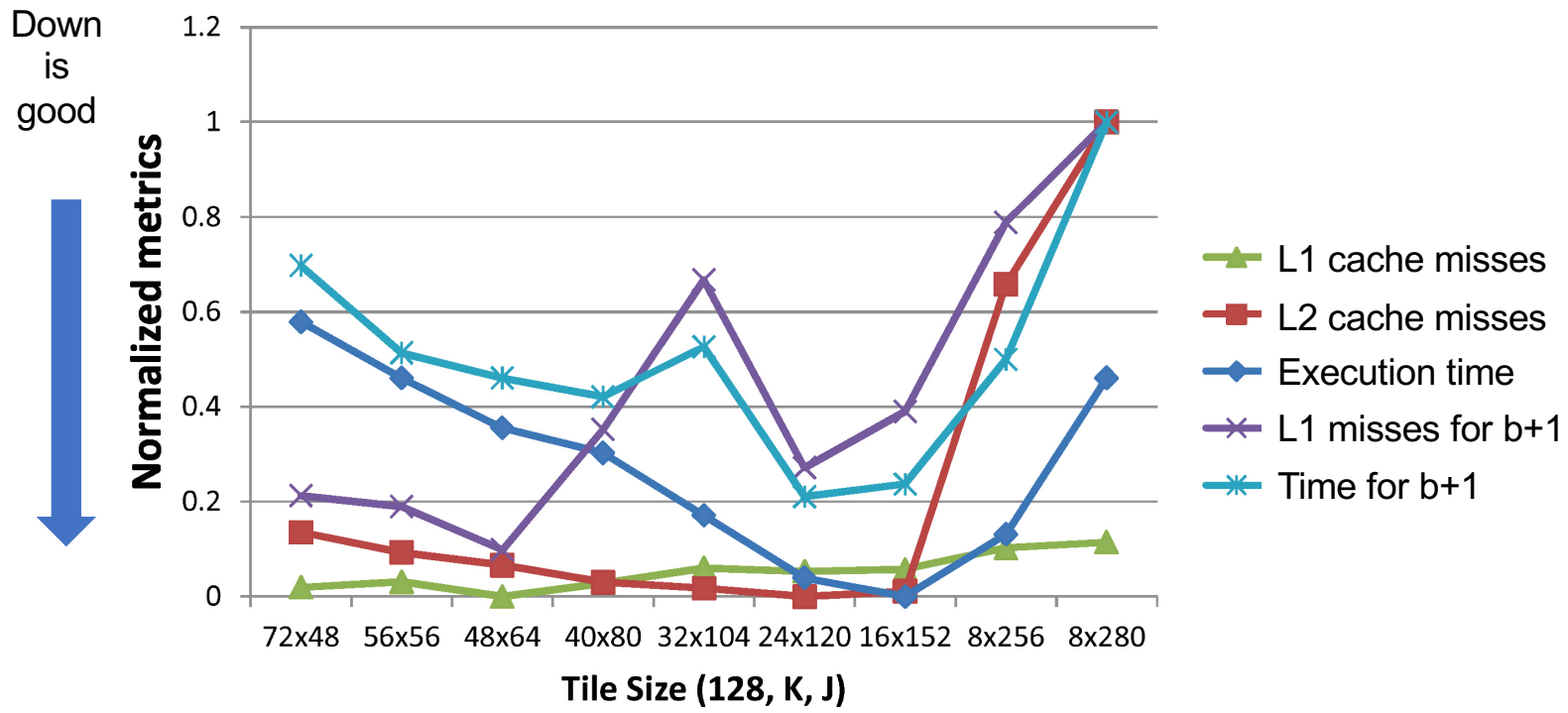
So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply (q=2)

Matrix multiplication

Maximize b, but small enough to fit in cache (or in registers)

Not necessarily square block (row / column accesses different)



BLAS

BLAS: Basic Linear Algebra Subprograms

- Standard API for dense vector and matrix operations
- Official reference implementation
  - original Fortran version: https://www.netlib.org/blas
  - C wrapper: https://www.netlib.org/clapack
- Building block of many libraries:
  - LAPACK, LINPACK
    - LINPACK benchmarks: used to rank top500 supercomputers
  - Matlab, GNU Octave

BLAS1: 1970s

- 15 operations
- vector ops: dot product, saxpy ($y=\alpha*x+y$), root-sum-squared, etc.
- **Computational Intensity**:
- m=2n, f=2n
- CI = ~1 or less

BLAS2: Mid-1980s

- 25 operations
- matrix-vector ops: matvec, etc.
- **Computational Intensity**:
- $m=n^2$, $f=2*n^2$
- CI~2 and less overhead

BLAS3: Late-1980s

- 9 operations
- matrix-matrix ops: matmul, etc.
- **Computational Intensity**:
- $m <= 3n^2$, $f=O(n^3)$,
- CI as large as n

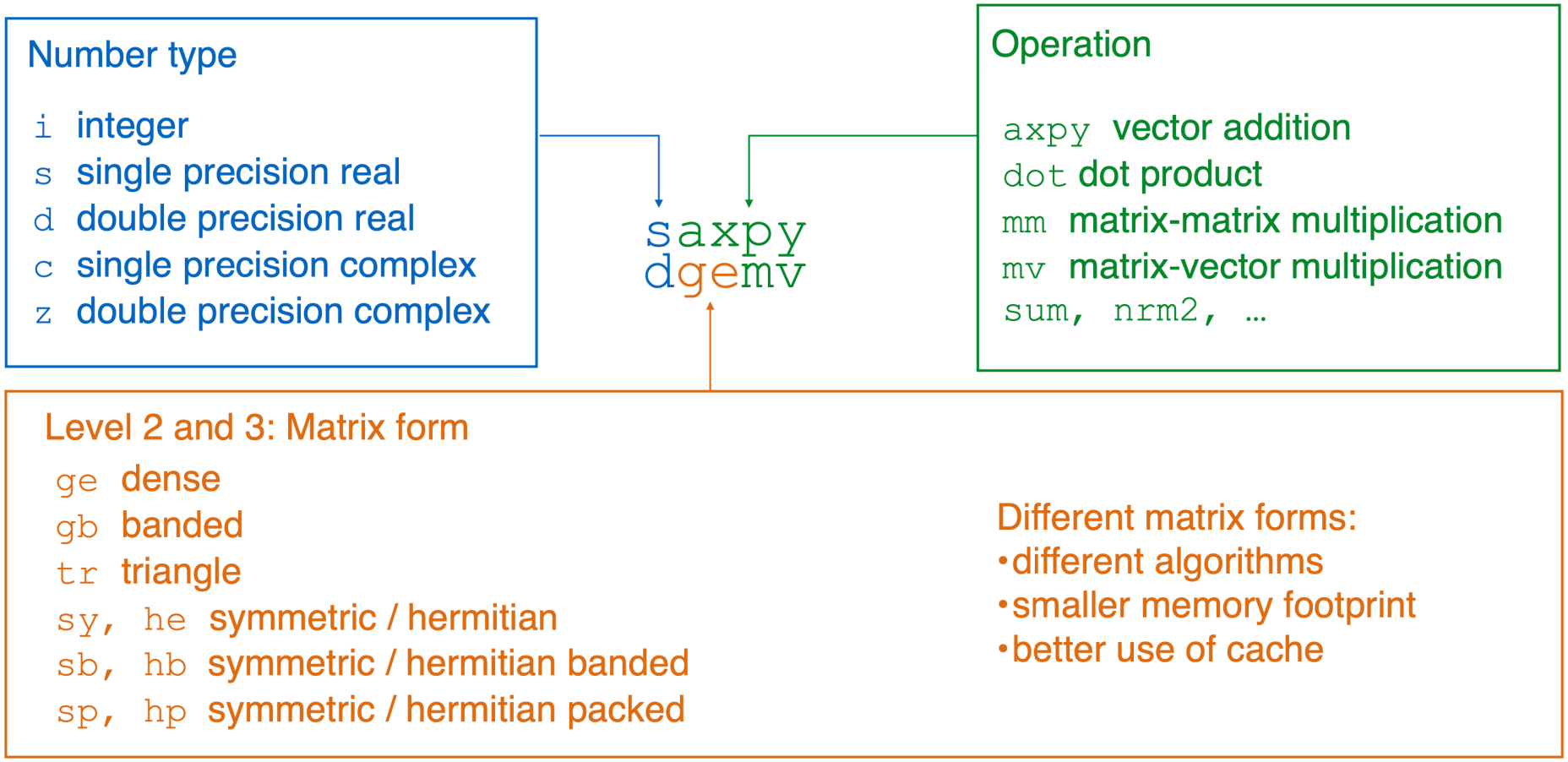
Basic linear algebra subroutines (BLAS)

- Industry standard interface (evolving)
  - www.netlib.org/blas, www.netlib.org/blas/blast--forum
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s): 15 different operations
    - vector operations: dot product, saxpy (y=α*x+y), root-sum-squared, etc
    - m=2*n, f=2*n, q = f/m = computational intensity ~1 or less
  - BLAS2 (mid 1980s): 25 different operations
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s): 9 different operations
    - matrix-matrix operations: matrix matrix multiply, etc
    - m <= 3n^2, f=O(n^3), so q=f/m can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK & ScaLAPACK)
  - See `www.netlib.org/{lapack,scalapack}`
  - More later in the course

Basic linear algebra subroutines (BLAS)

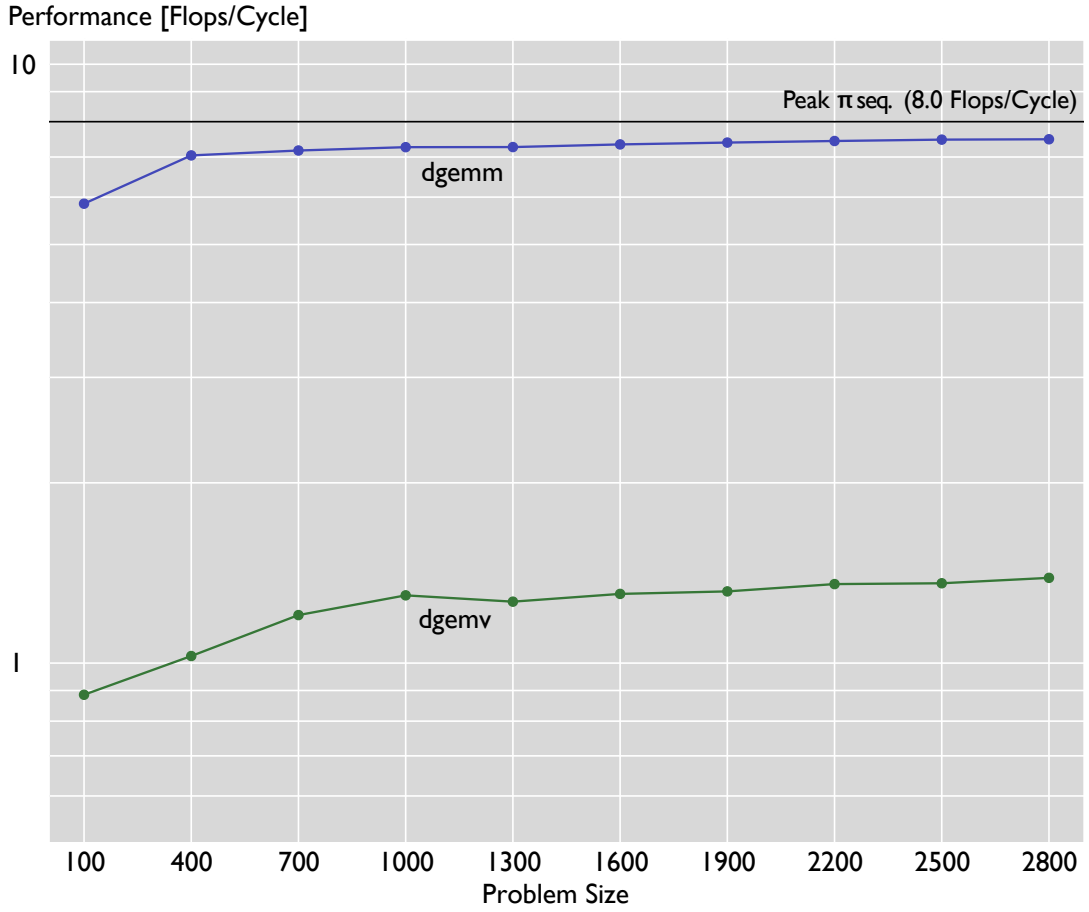
- **Tiling for registers**
  - loop unrolling, use of named “register” variables
- **Tiling for multiple levels of cache**
- **Exploiting fine-grained parallelism in processor**
  - superscalar; pipelining; SIMD
- **Hard to do by hand**
- **Automatic optimization an active research area**

BLAS rotines: naming conventions



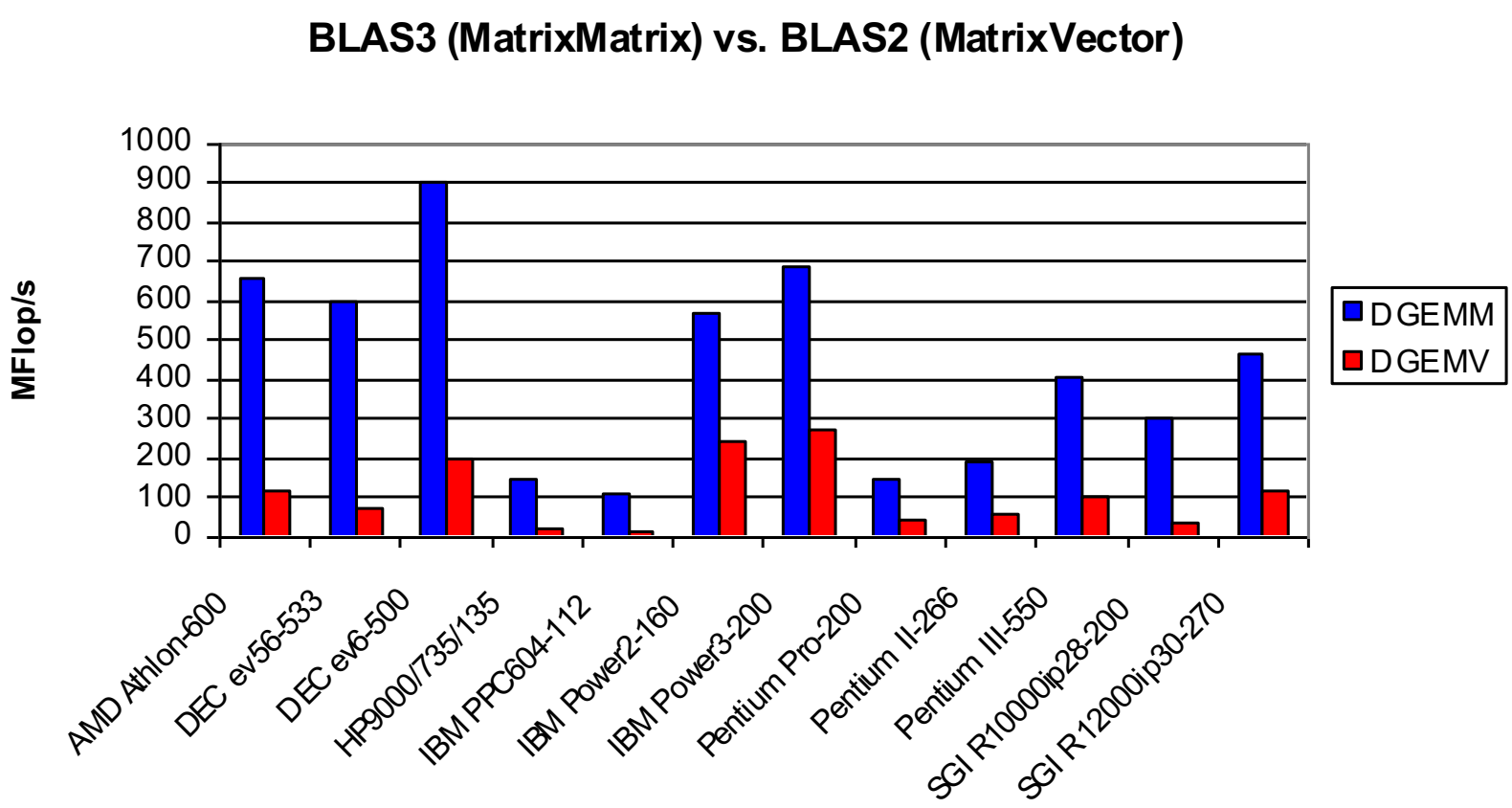
Matrix multiplication

Measuring Performance — Flops/Cycle

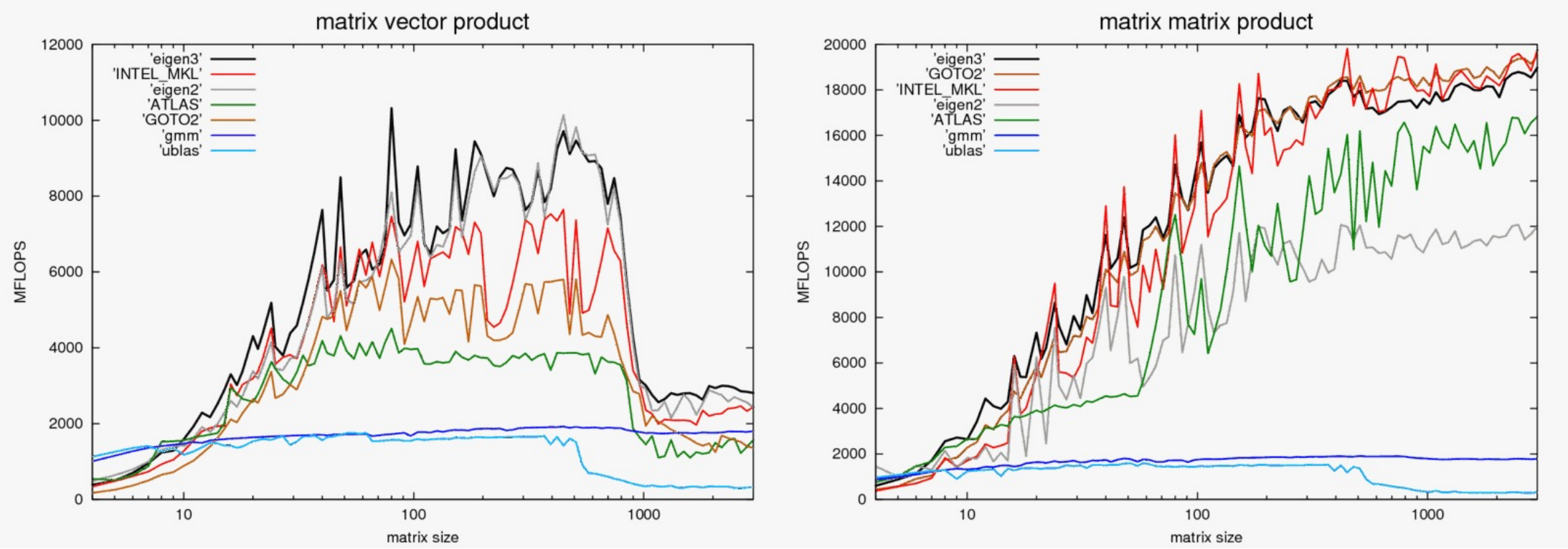


Dense Linear Algebra

BLAS2 and BLAS3 have very different computational intensity, and therefore different performance



Matrix multiplication



http://eigen.tuxfamily.org/index.php?title=Benchmark

Summary

- **Matrix matrix multiplication**
  - $2n^3$ flops on $3n^2$ data, so Computational Intensity up to O(n)

- **Tiling matrix multiplication (cache aware)**
  - Can increase block size to b if bxb blocks fit in fast memory
  - b = sqrt(M/3), the fast memory size M
  - Tiling (aka blocking) “cache-aware”
  - Other issues: cache alignment, SIMD, loop unrooling, etc.

- **Optimized libraries (BLAS, Eigen, etc.) exist**

References

- "Performance Optimization of Numerically Intensive Codes", by Stefan Goedecker and Adolfy Hoisie, SIAM 2001.
- Libraries:
  - BLAS (Basic Linear Algebra Subroutines), Reference for (unoptimized) implementations of the BLAS, with documentation.
  - LAPACK (Linear Algebra PACKage), a standard linear algebra library optimized to use the BLAS effectively on uniprocessors and shared memory machines (software, documentation and reports)
  - ScaLAPACK (Scalable LAPACK), a parallel version of LAPACK for distributed memory machines (software, documentation and reports)
- Tuning Strassen's Matrix Multiplication for Memory Efficiency Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck in Proceedings of Supercomputing '98, November 1998 postscript
- Recursive Array Layouts and Fast Parallel Matrix Multiplication” by Chatterjee et al. IEEE TPDS November 2002.

1. Memory hierarchies
   - von Neumann bottleneck
   - register, cache, and main memory
   - cache details

2. Parallelism within single processors
   - Pipelining
   - SIMD
   - Special instructions (FMA)

3. Case study: Matrix multiplication

4. Roofline model

Roofline

Idea: applications are limited by either compute peak or memory bandwidth:

- **Bandwidth bound (matvec)**
- **Compute bound (matmul)**

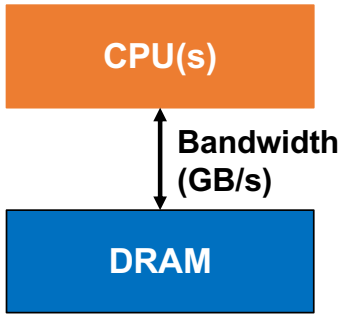
Three pieces: 2 for machine and 1 for application

- Arithmetic performance (flops/sec)
  Clock Speed and Parallelism (ILP, SIMD, Multicore)
- Memory bandwidth (bytes /sec)
  Latency not included (looking at best case)
- Computational (Arithmetic) Intensity
  Application balances (flops/word or flops/byte)

DRAM Roofline

Which takes longer?

- **Data movement**
- **Compute**

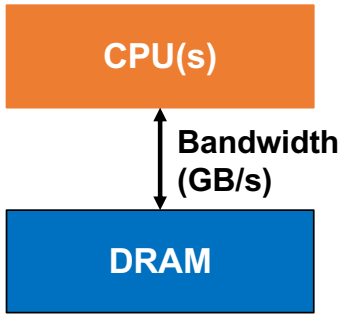


$$\text{Time} = \max \begin{cases} \#\text{FP ops} / \text{Peak GFLOP/s} \\ \#\text{Bytes} / \text{Peak GB/s} \end{cases}$$

DRAM Roofline

Which takes longer?

- **Data movement**
- **Compute**

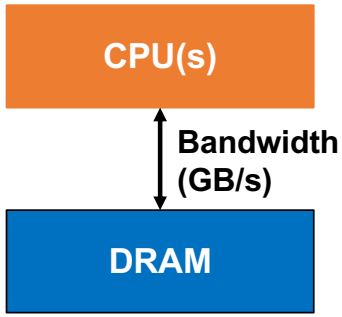


$$\frac{\#\text{FP ops}}{\text{Time}} = \min \begin{cases} \text{Peak GFLOP/s} \\ (\#\text{FP ops}/\ \#\text{Bytes}) \times \text{Peak GB/s} \end{cases}$$

DRAM Roofline

Which takes longer?

- **Data movement**
- **Compute**

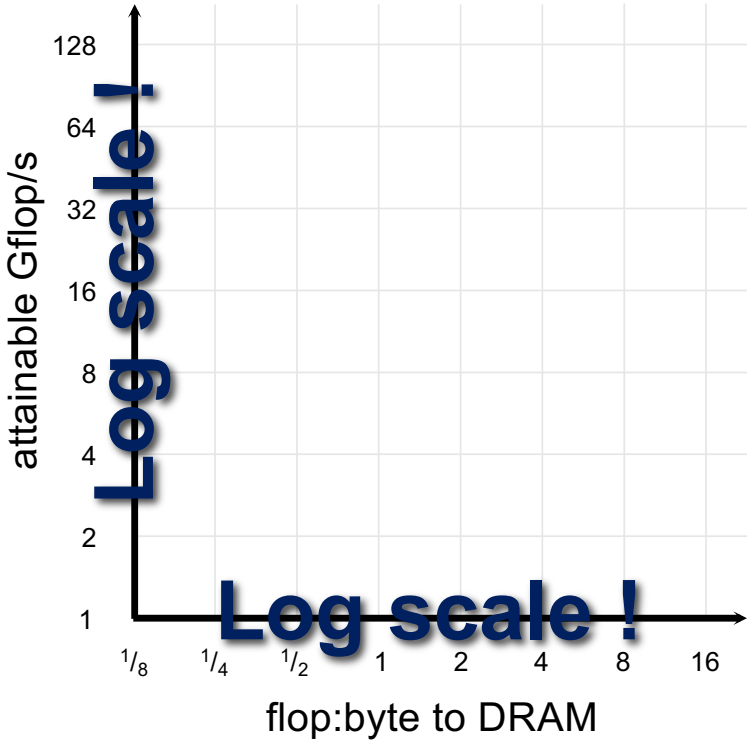


$$\text{Gflop/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ q \times \text{Peak GB/s} \end{cases}$$

q = #FP ops/ #Bytes algorithm intensity
or computational intensity

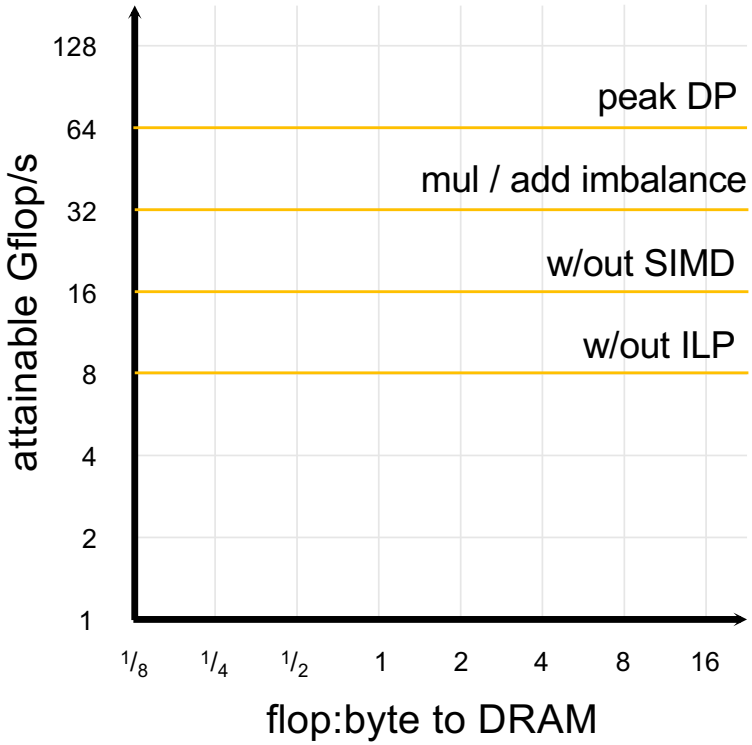
Measure of data locality (data reuse)

Roofline



$$\text{Gflop/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ q \times \text{Peak GB/s} \end{cases}$$

Roofline



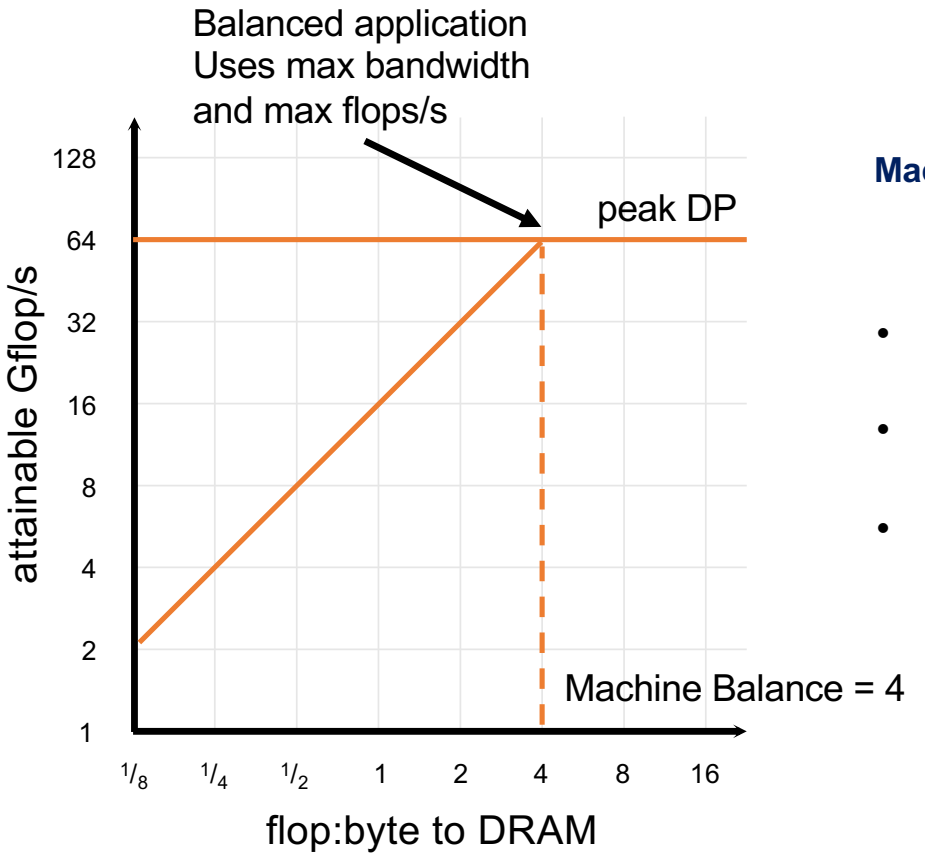
$$\text{Gflop/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{q x Peak GB/s} \end{cases}$$

- Top of the roof is the peak compute rate
- No FMA, no SIMD, no ILP will lower what is attainable

Roofline

Top of the roof is the peak compute rate

Corner of roof depends on memory bandwidth (log-log plot)



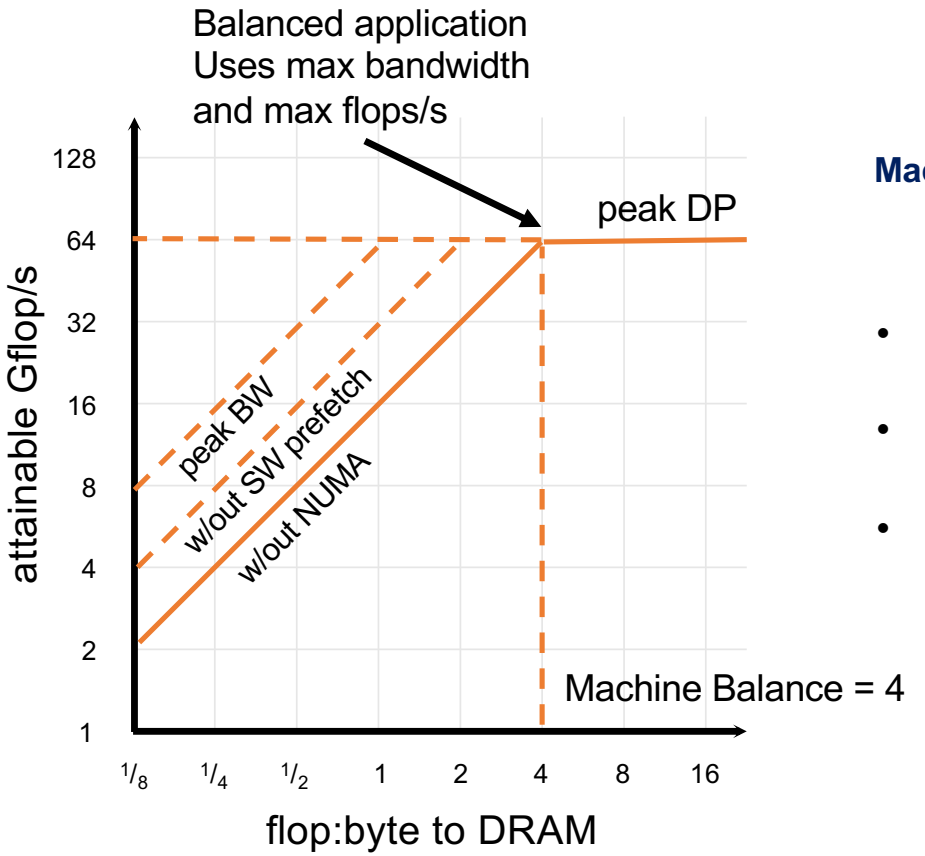
$$\text{Machine Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

- Typically 5-10 Flops / Byte
- 40 – 80 Flops / Double
- Unlikely to improve in the near future

Roofline

Top of the roof is the peak compute rate

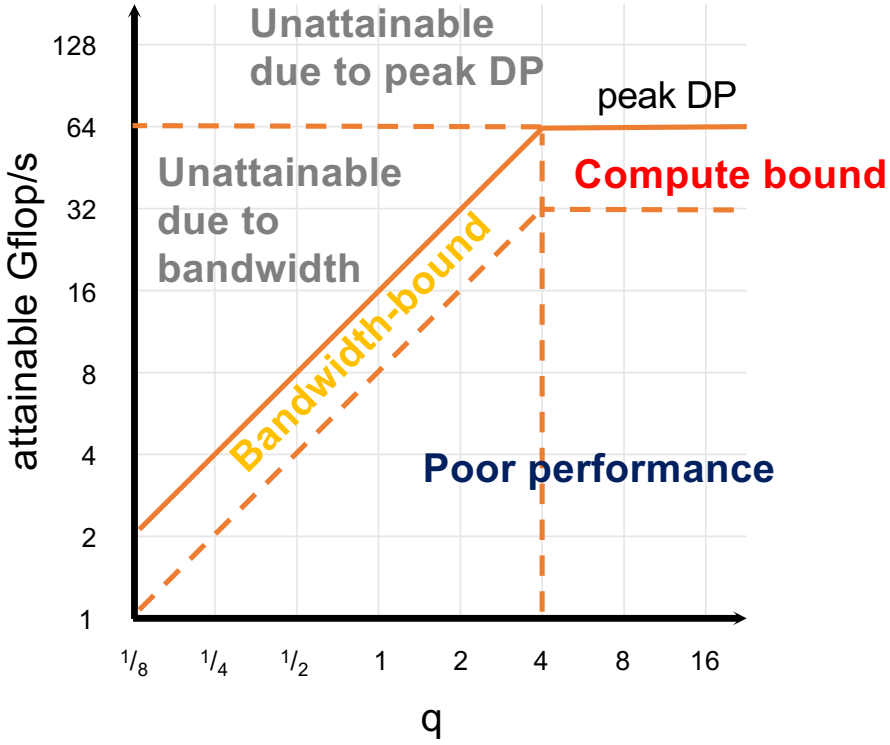
Corner of roof depends on memory bandwidth (log-log plot)



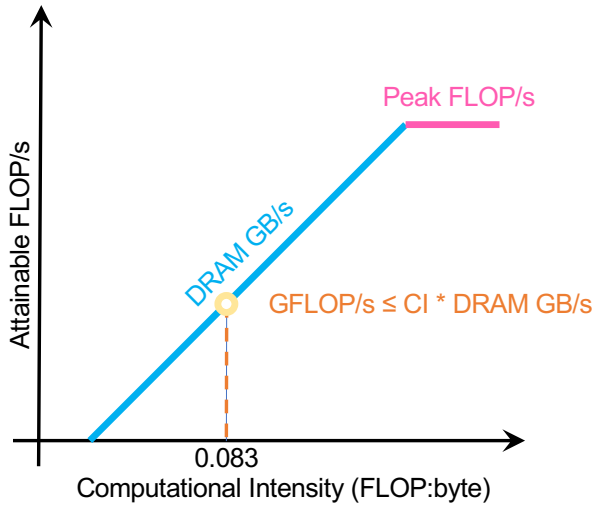
$$\text{Machine Balance} = \frac{\text{Peak DP FLOP/s}}{\text{Peak Bandwidth}}$$

- Typically 5-10 Flops / Byte
- 40 – 80 Flops / Double
- Unlikely to improve in the near future

Roofline


Unattainable
due to peak DP
peak DP
Unattainable
due to
bandwidth
Compute bound
Bandwidth-bound
Poor performance
attainable Gflop/s
128
64
32
16
8
4
2
1
1/8
1/4
1/2
1
2
4
8
16
q

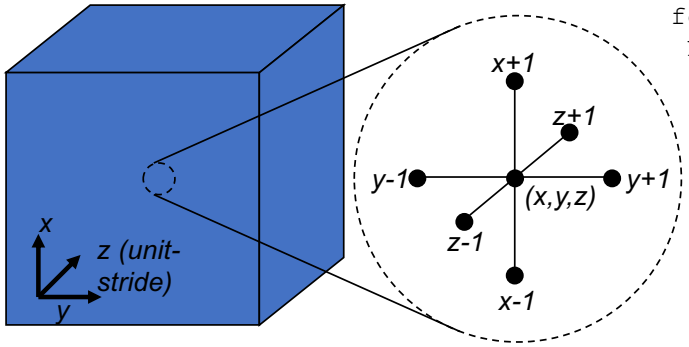
Roofline example 1




```
#pragma omp parallel for
for(i=0;i<N;i++)
{
  Z[i] = X[i] + alpha*Y[i];
}
```

- **2 FLOPs per iteration**
- **Transfer 24 bytes per iteration**
- **q = 0.083 FLOPs / Byte**

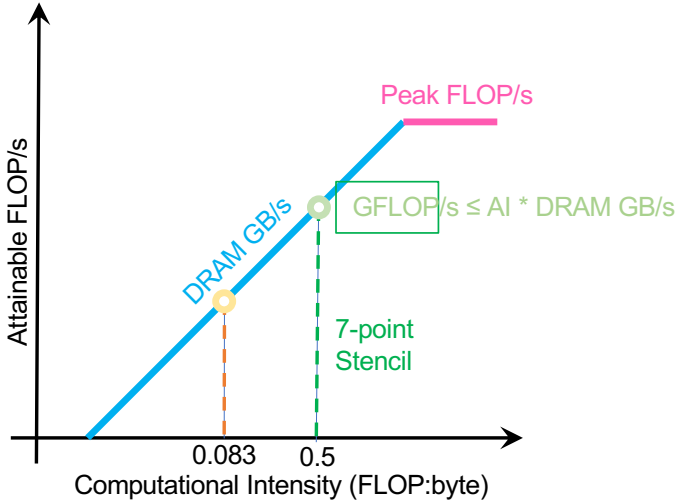
Roofline example 2



3D 7-point stencil

```
for x,y,z in 0 to n-1
 Next[x,y,z]=
   C0 * Current[x,y,z]+
   C1 *(Current[x-1,y,z]+
        Current[x+1,y,z]+
        Current[x,y-1,z]+
        Current[x,y+1,z]+
        Current[x,y,z-1]+
        Current[x,y,z+1]);
```

Inner Loop Pseudocode



A 7-point constant coefficient stencil…

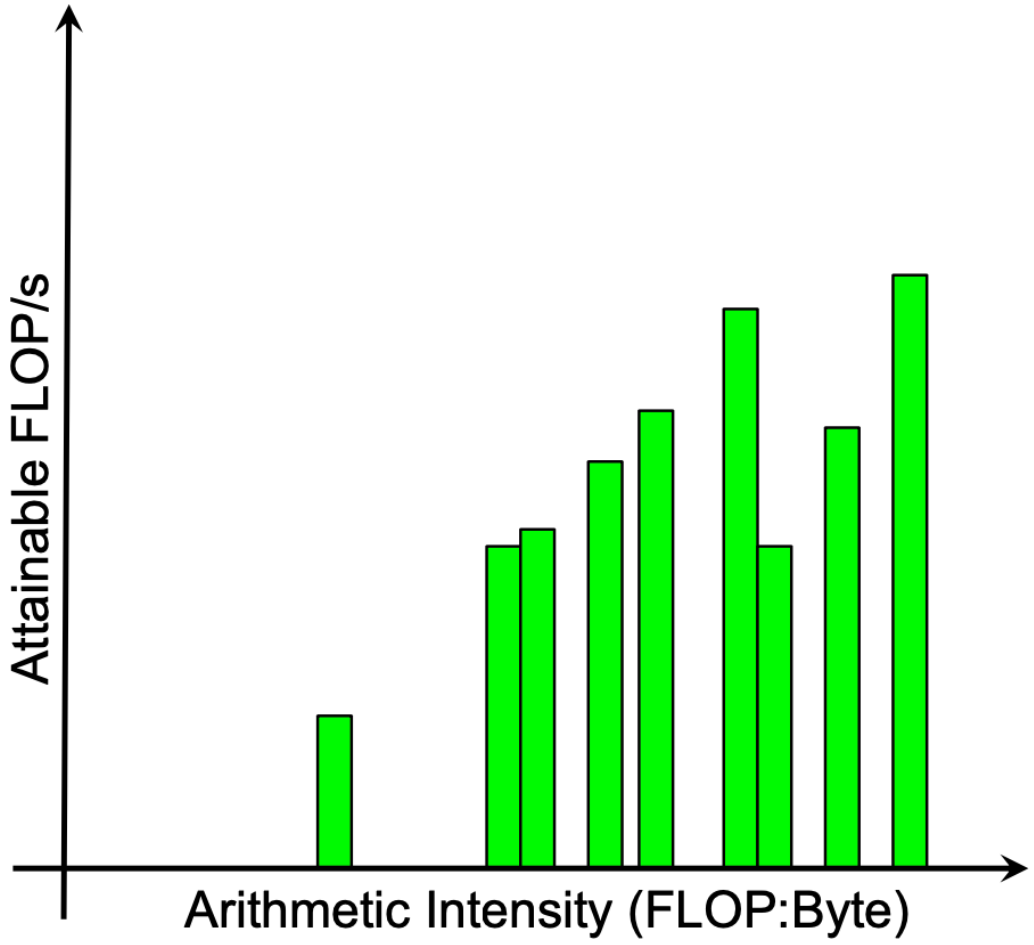
8 flops, 8 memory references (7 reads, 1 store) per point

Cache can filter all but 1 read and 1 write per point

q = 0.5 flops per byte

What is good performance

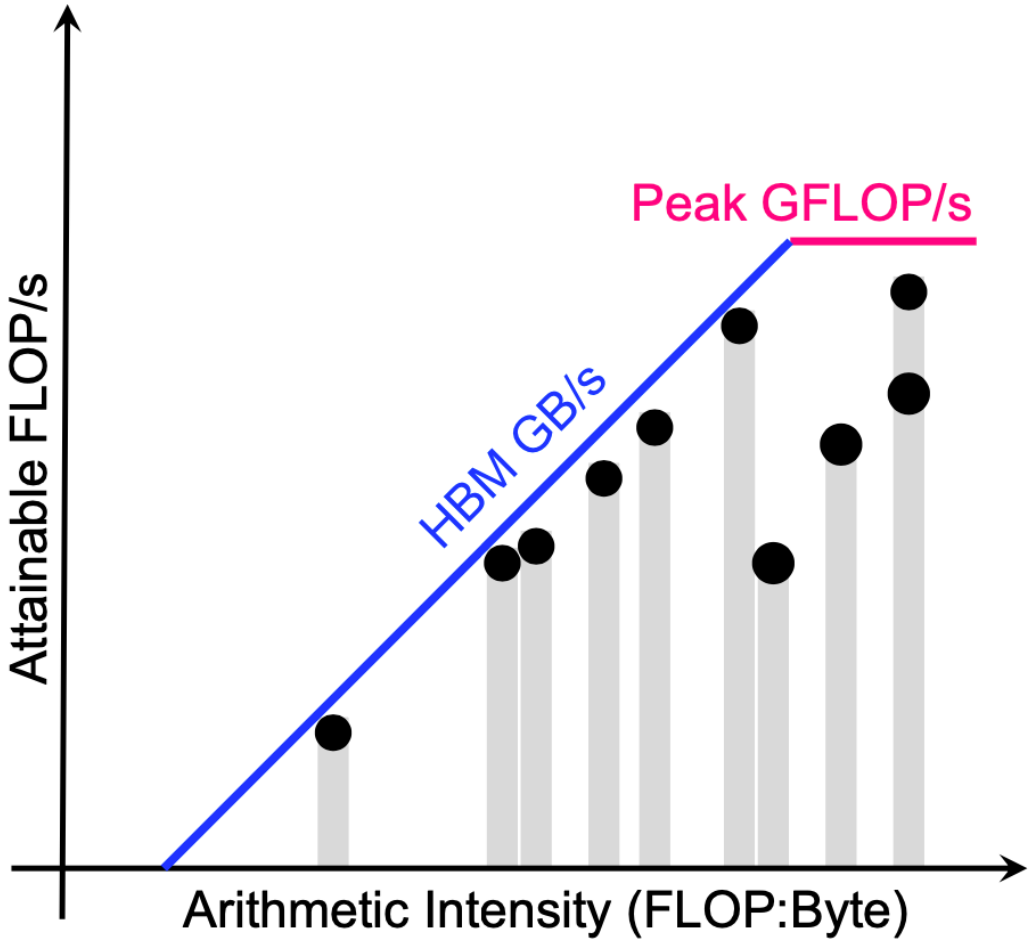
We may sort the implementation by Arithmetic Intensity



What is good performance

We may sort the implementation by Arithmetic Intensity

Compare performance relative to machine capabilities

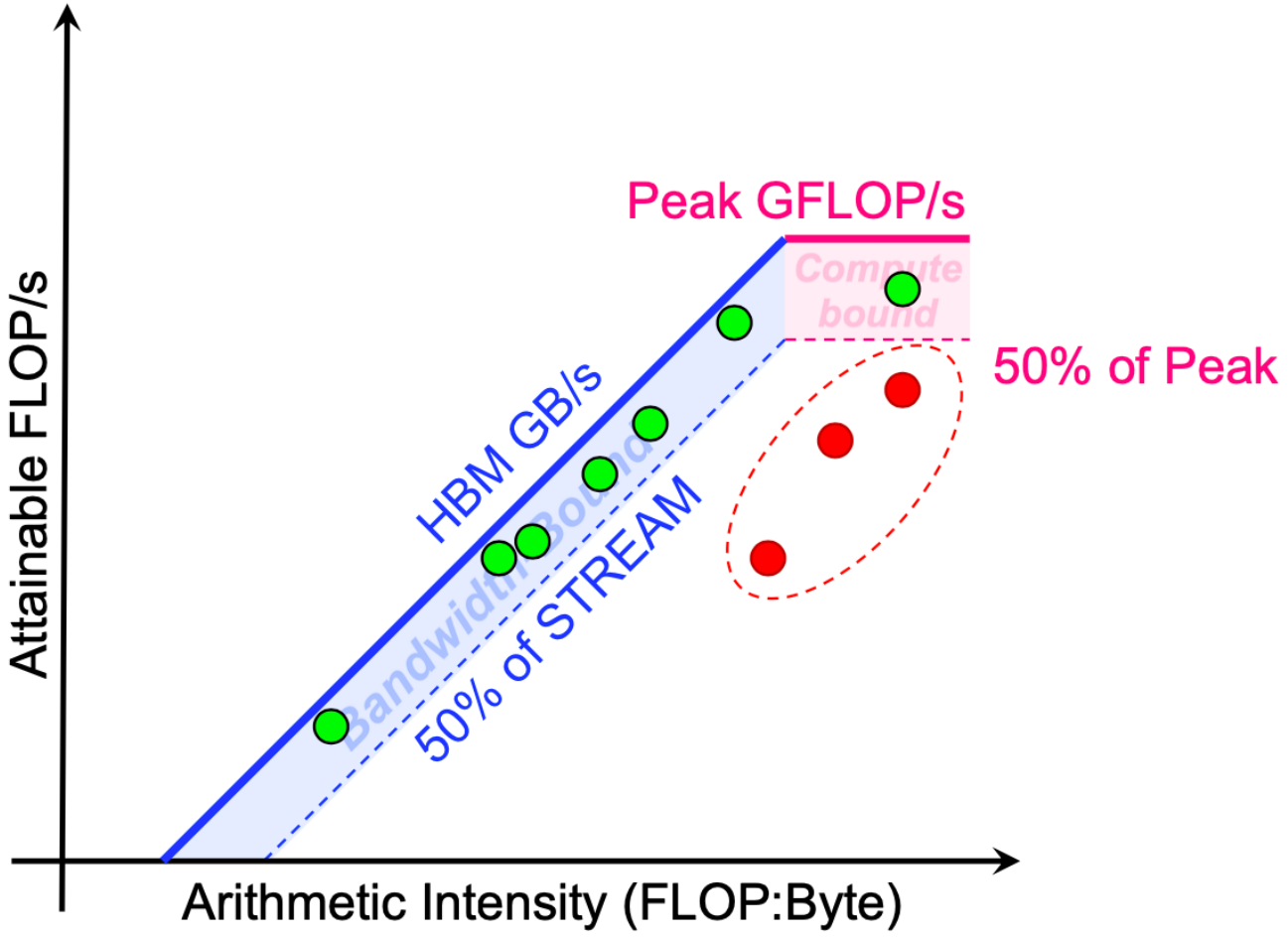


What is good performance

Kernels near the roofline are making good use of computational resources

- They may have low performance but make good use of a machine

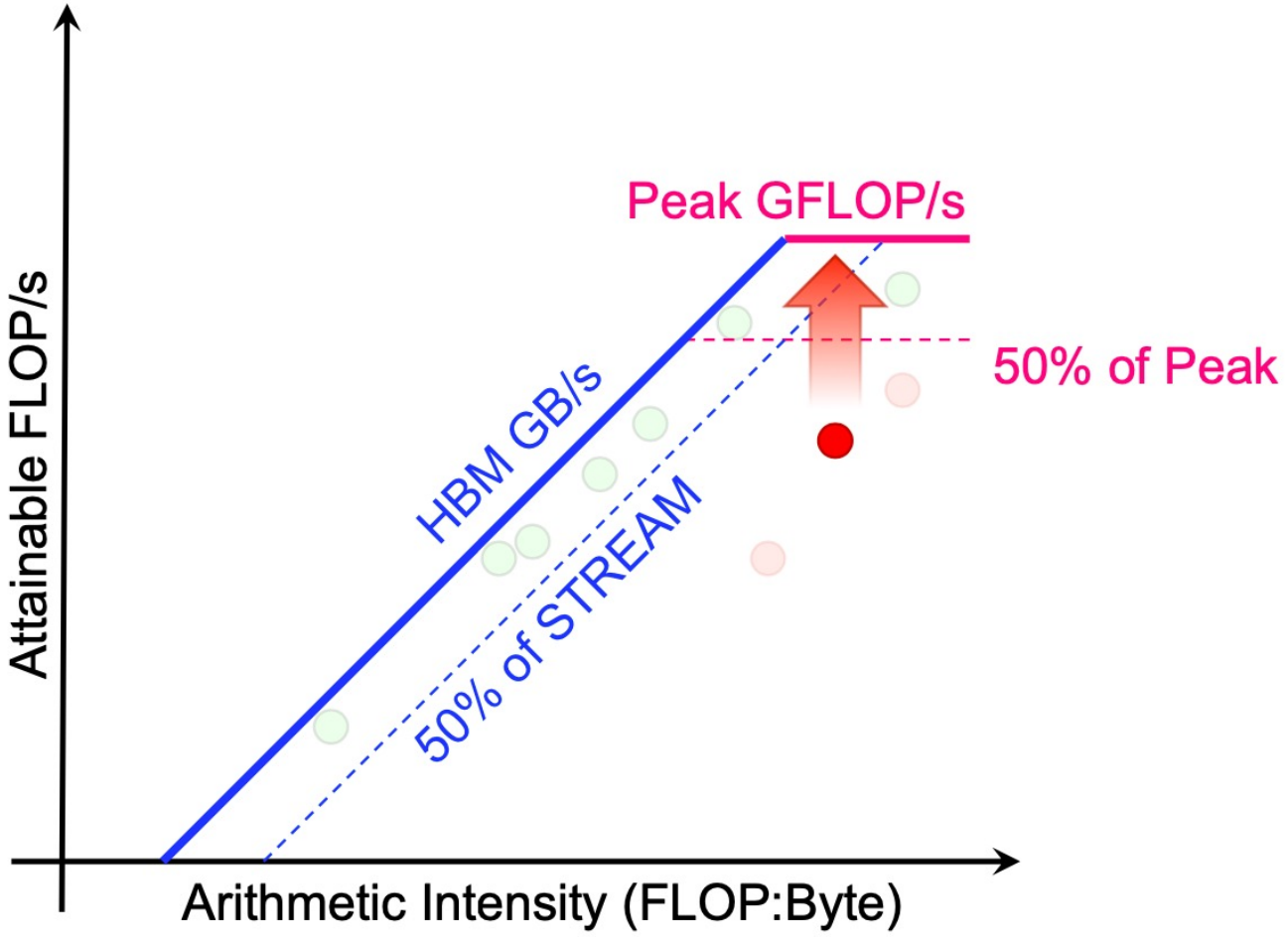
- They can have high performance but still make poor use of a machine



What is good performance

Get to the roofline

- Vectorization by SIMD
- Use FMA instructions
- Optimize memory access patterns
- Align memory access
- Use good math libraries (BLAS, etc.)
- Multi-thread



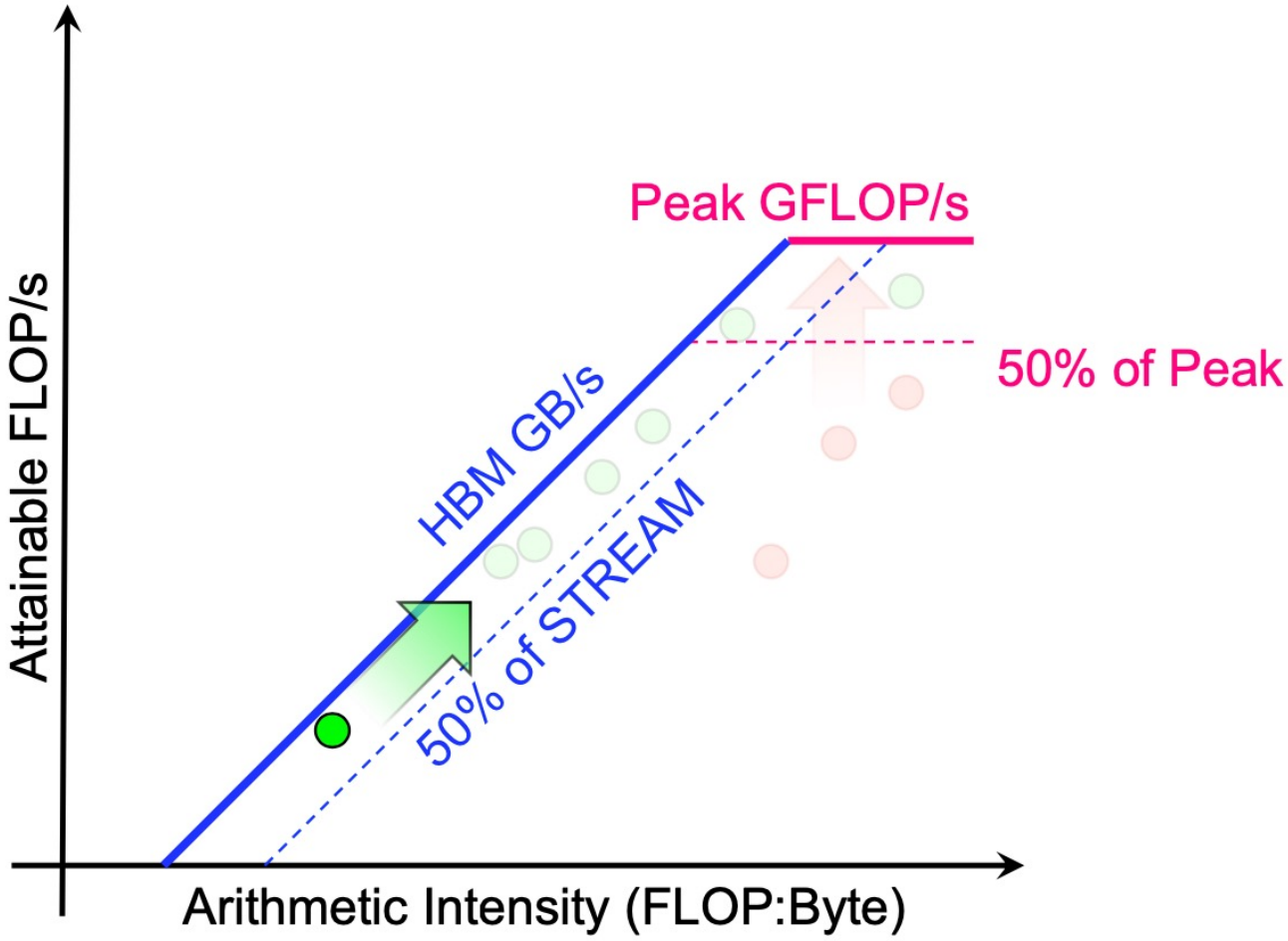
What is good performance

Get to the roofline

Increase the arithmetic intensity when bandwidth-limited

- Reducing data movement

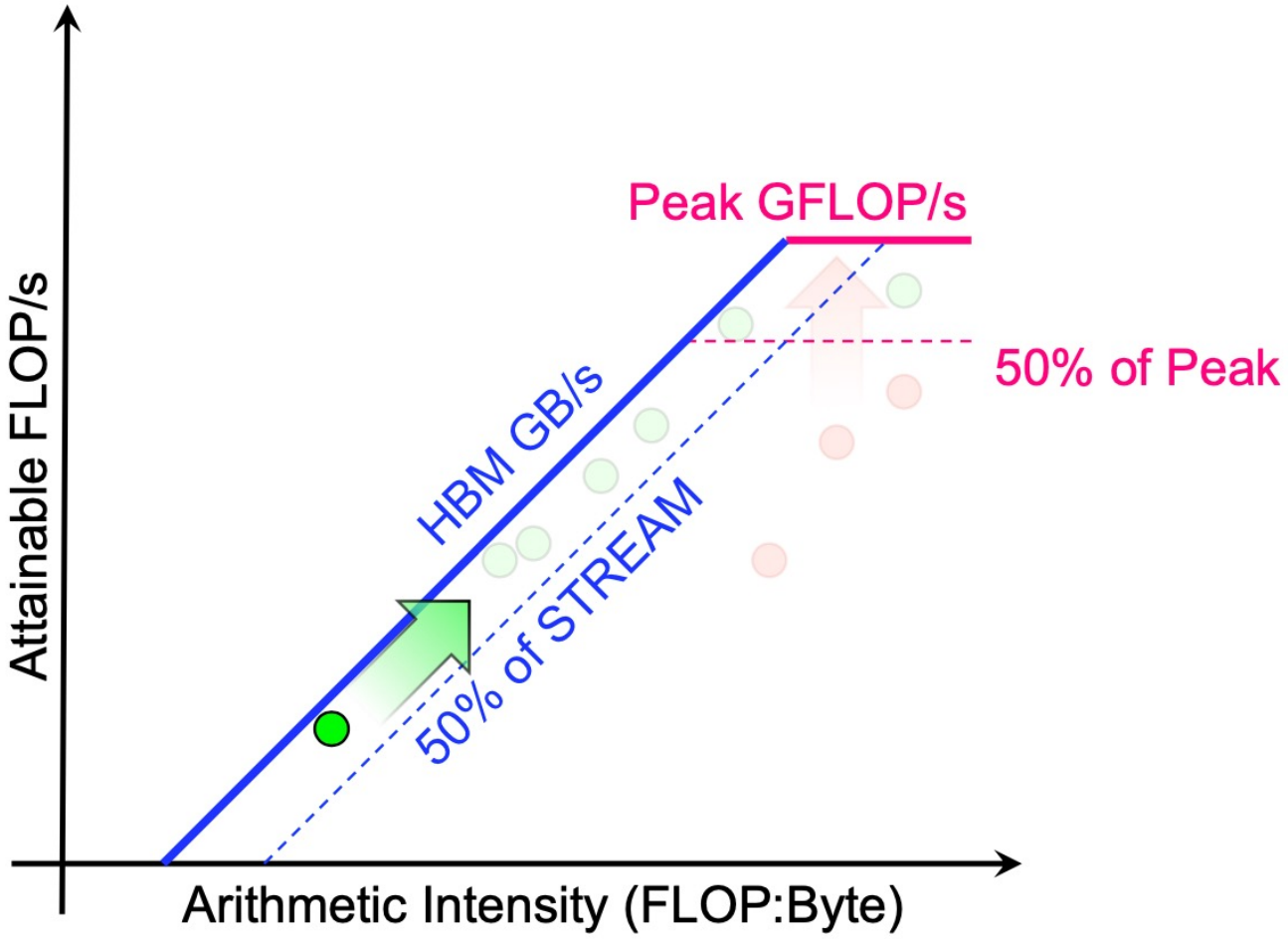
- Spatial locality, cache blocking, etc.



What is good performance

Machine model
- Lines defined by peak GB/s and GF/s
- Unique to each architecture
- Common to all apps on that machine

Application characteristics
- Dots defined by application Gflop/s and GB/s
- Unique to each application



Summary

- **Performance programming on uniprocessors requires**
  understanding of memory system
  understanding of fine-grained parallelism in processor
- **Simple performance models can aid in understanding**

  Two ratios are key to efficiency (relative to peak)

  1.computational intensity of the algorithm:

  q = f/m = # floating point operations / # slow memory references

  2.machine balance in the memory system:

  t_m/t_f = time for slow memory reference / time for floating point operation

- **Want q > t_m/t_f to get half machine peak**
- **Blocking (tiling) is a basic approach to increase q**

  Techniques apply generally, but the details (e.g., block size) are architecture dependent
  Similar techniques are possible on other data structures and algorithms

Conclusions

References:

- “Computer Systems: A Programmer’s Perspective” by R.E. Bryant and D.R. O’Hallaron

- “What every programmer should know about memory” by U. Drepper