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 finds may guide our programming styles.

Possible conclusions

“I want to optimize my code right now!”

DO NOT DO THAT!

Optimization can be done by compilers.

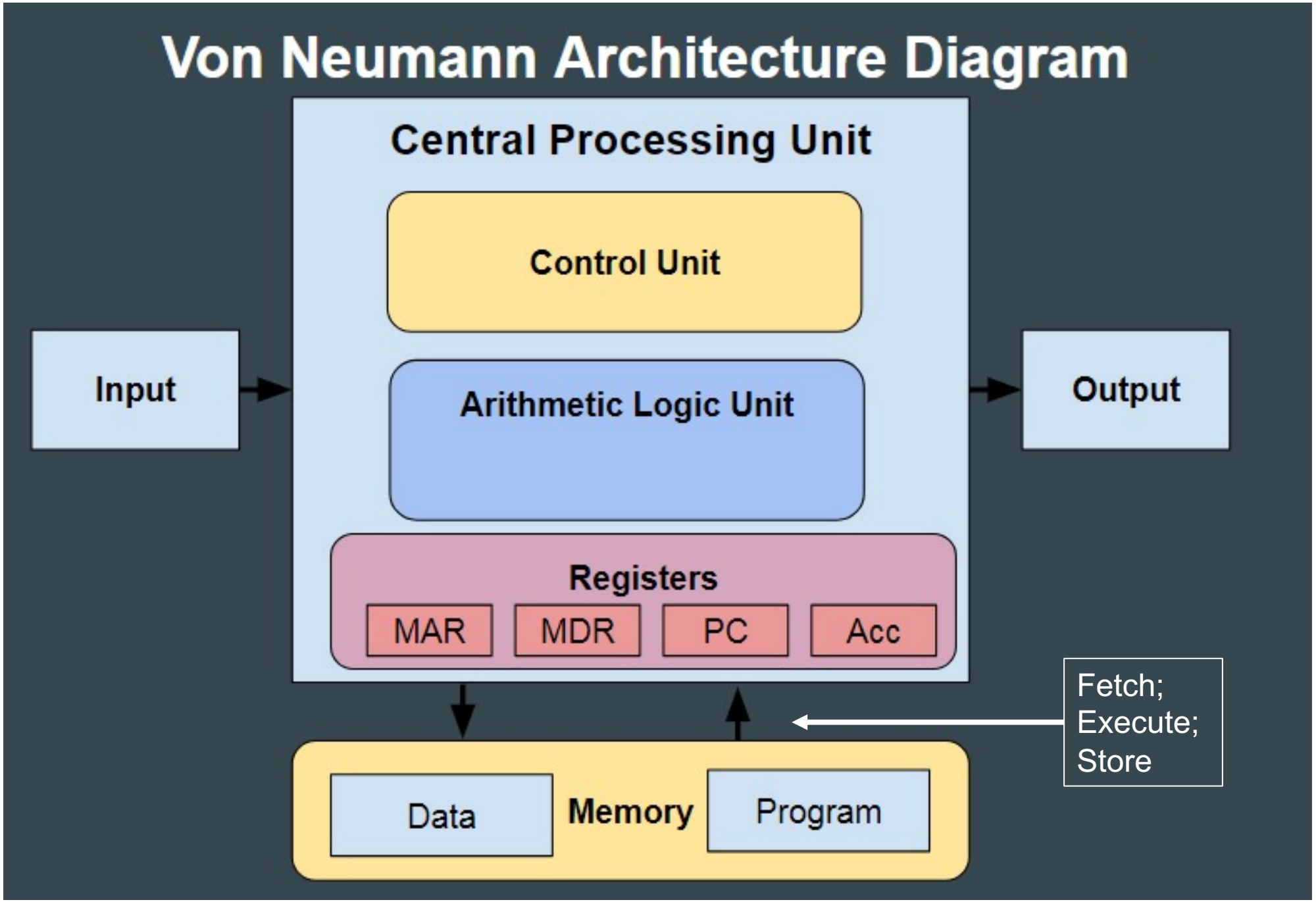
Early optimization is the root of all evil.

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. Illustrations

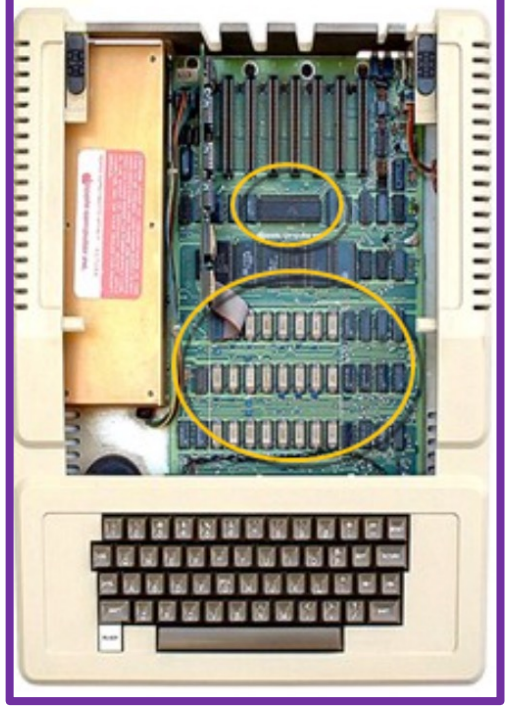

Von Neumann Architecture Diagram
Central Processing Unit
Control Unit
Arithmetic Logic Unit
Registers
MAR
MDR
PC
Acc
Input
Output
Fetch;
Execute;
Store
Data
Memory
Program

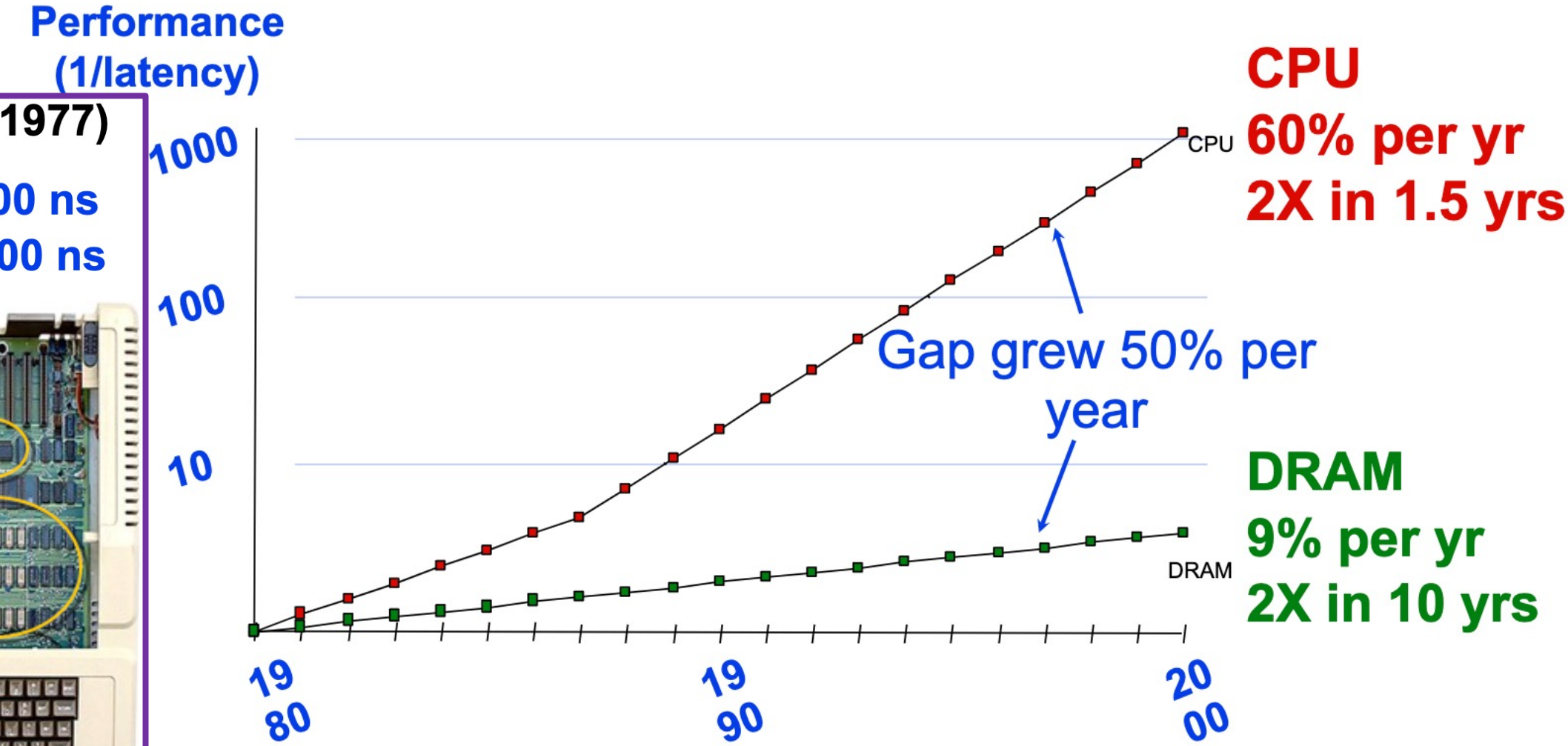
Memory latency gap

Apple][(1977)

CPU: 1000 ns

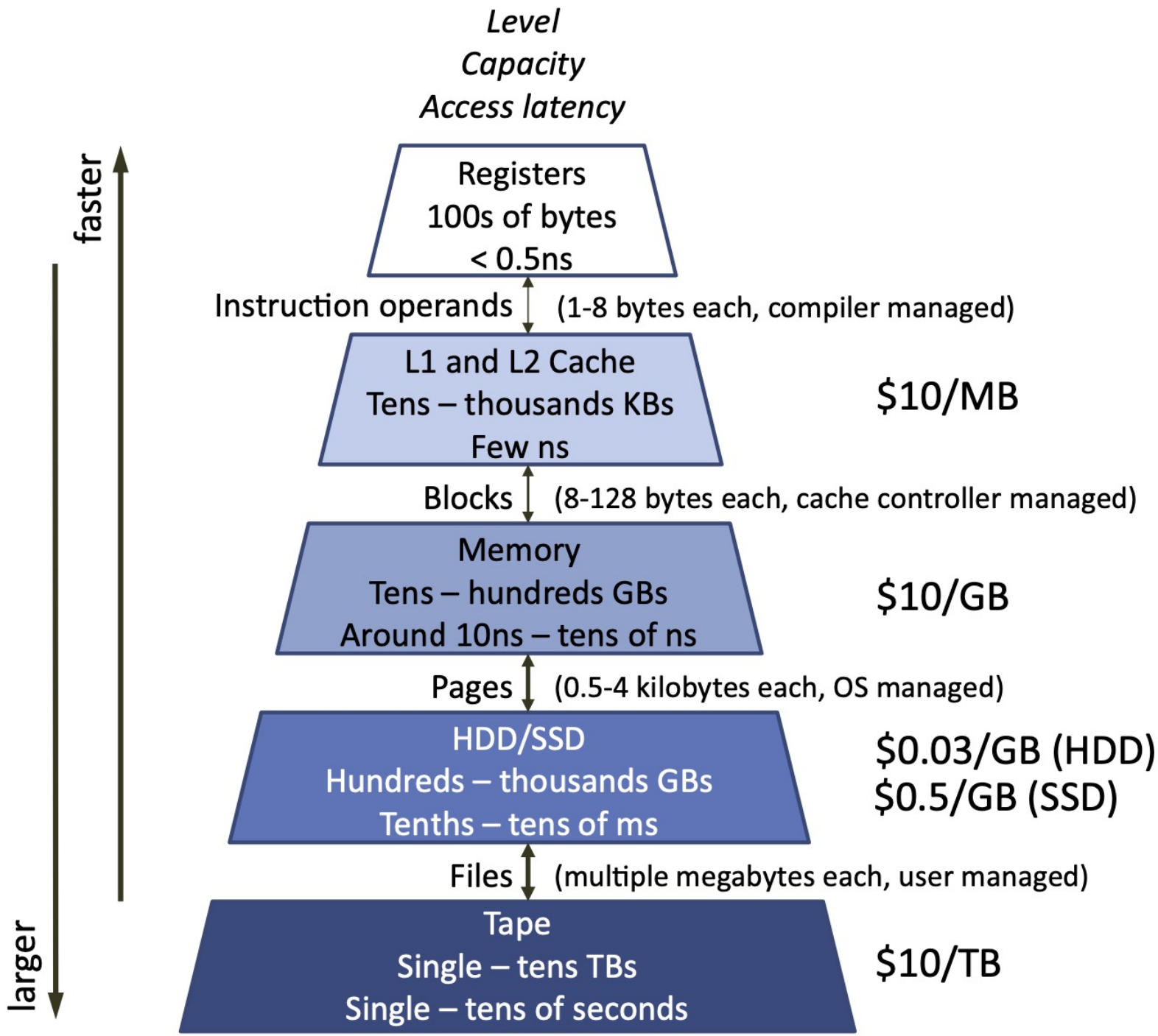
DRAM: 400 ns





CPU
60% per yr
2X in 1.5 yrs

DRAM
9% per yr
2X in 10 yrs


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 bottlenect)
- 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
  - …

Latency and bandwitdth

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 item actually arriving.
  - Various latencies: memory to cache, cache to register, or just memory to processor.
  - Units are usually nonoseconds 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 bandwitdth

Bandwidth β

≈ data throughput (bits/second)

Low Bandwidth

High Bandwidth

Latency α

≈ delay due data travel time (ms)

Low Latency

High Latency

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

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.

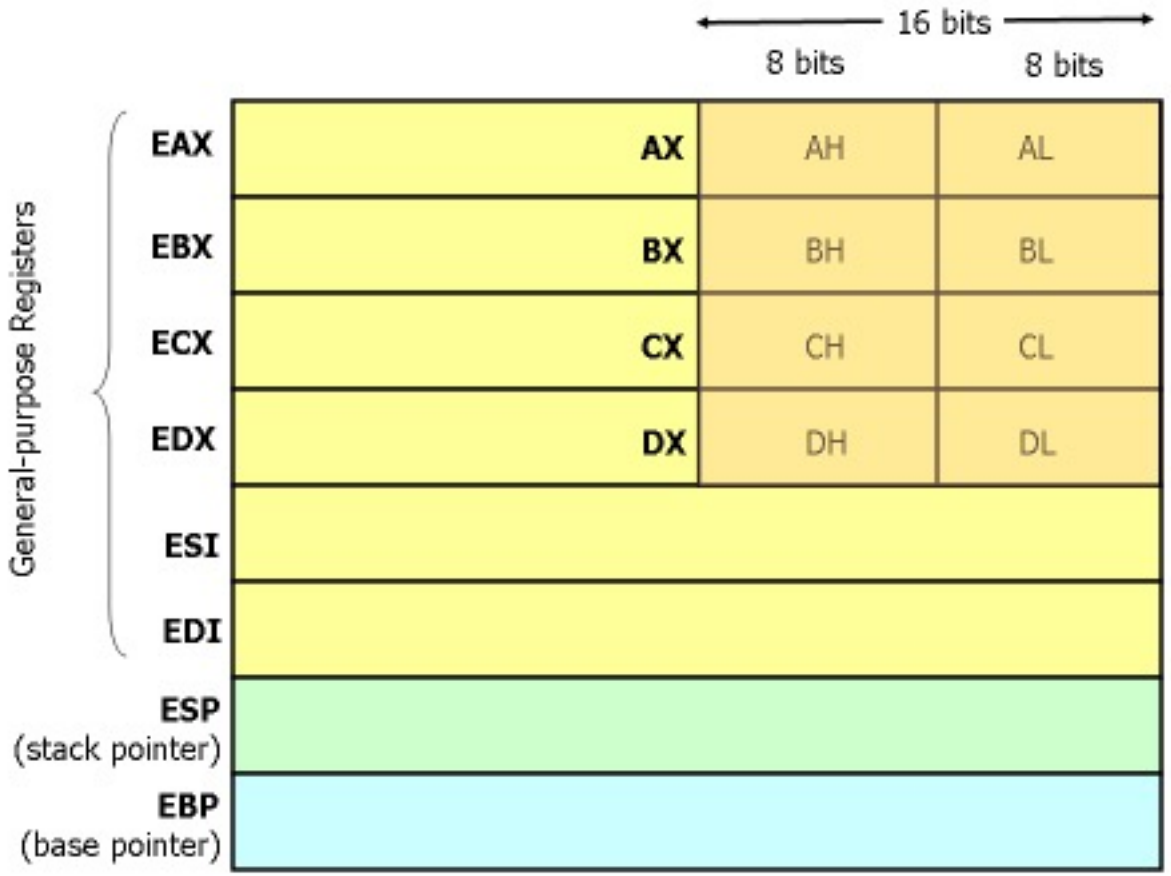
Approaches to handling memory latency

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

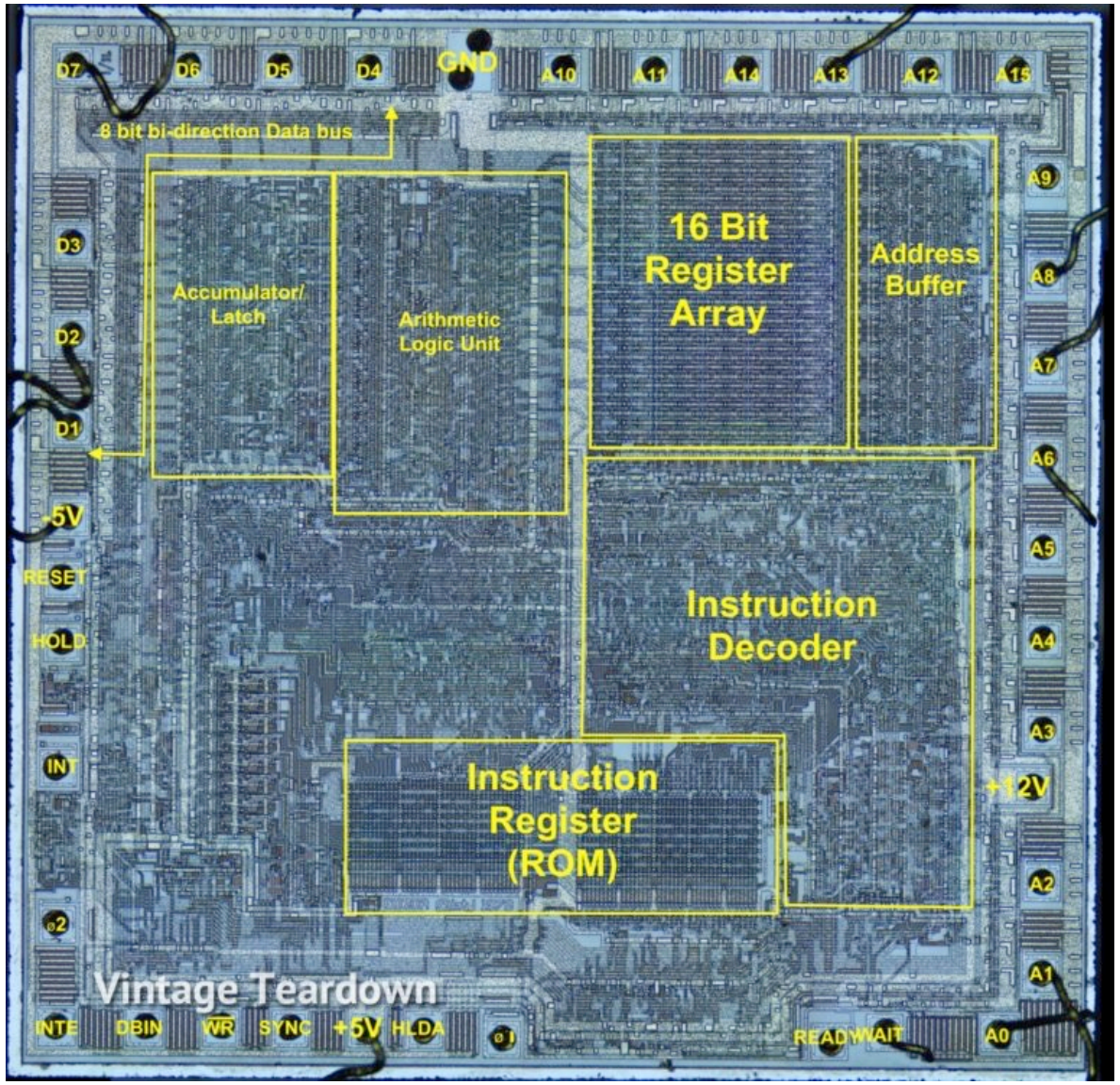
- **Processor instructions operate on registers directly.**
  - **Have assembly language names like: eax, ebx, ecx, etc.**
  - **Sample instruction: addl %eax, %edx**

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 varialbe in register is important if that variable appears in an inner loop.

Example:

```
for ii = …
    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,I
        a[ i, k ]=b[ i, k ] * c[ k ]
```

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

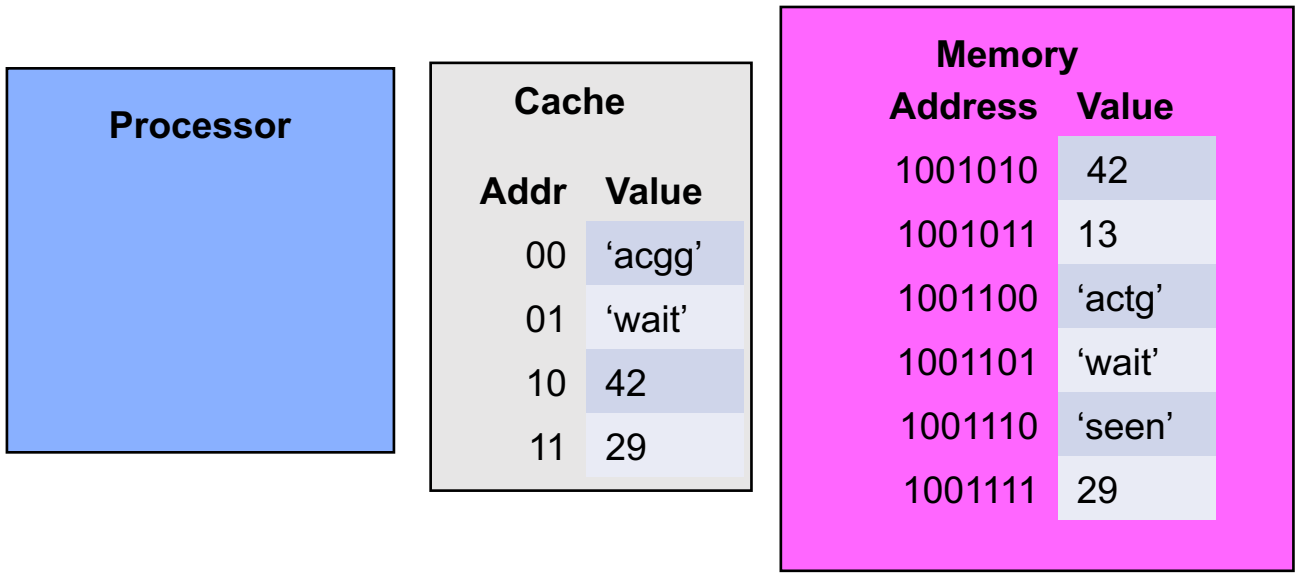
- In C, you can give a hint to the compiler to keep a variable in register by “register”.

```
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 16KB.

- **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.

Main memory

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

Main memory

Bandwidth:

- **L1: 32 bytes or 4 double precision numbers. Enough for half of peak performance.**
- **L2 and L3 are approximately the same as L1's. Partly wasted for coherence issues.**
- **Main memory: 5 bytes and shared by many cores. So effective bandwidth is even smaller.**

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*

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

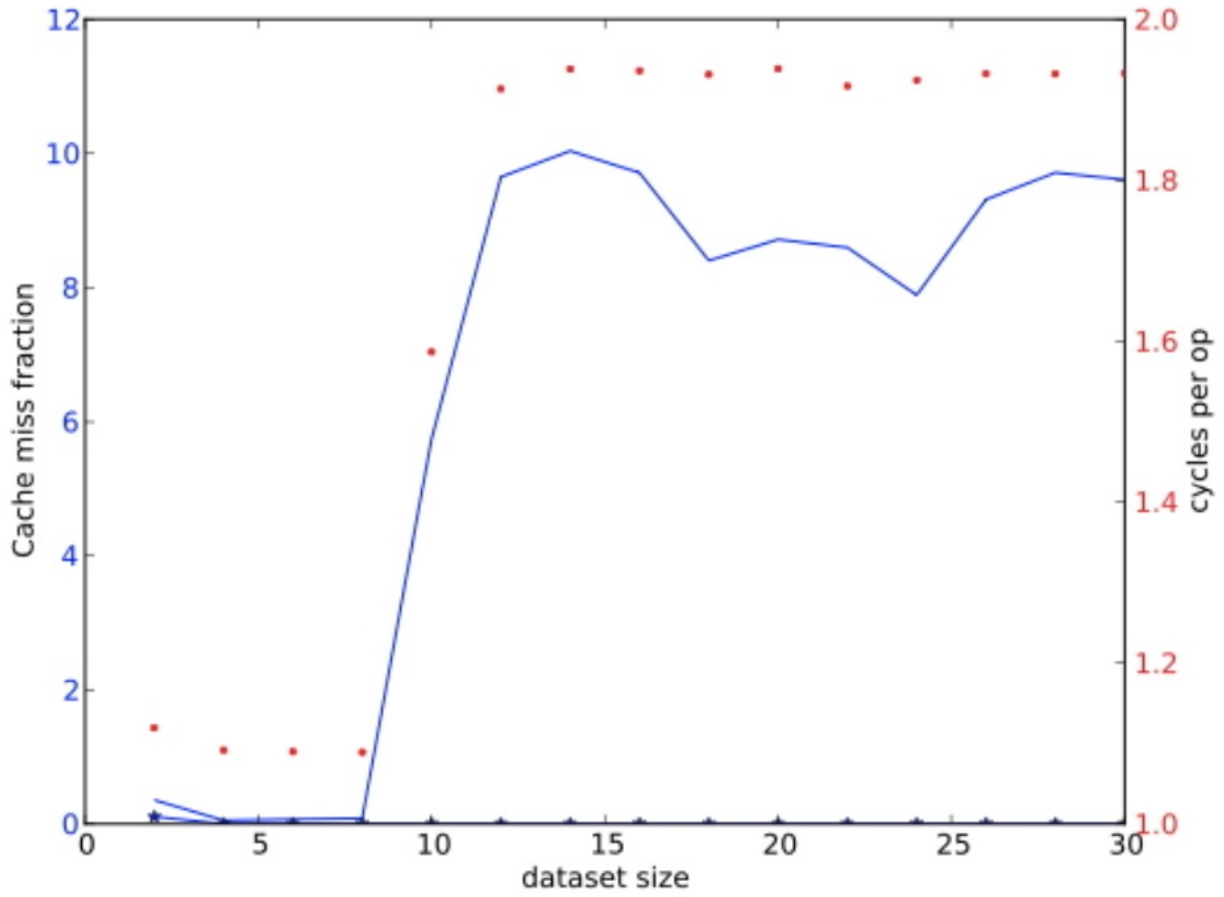
Cache access

- Access is *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 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;
 }
```

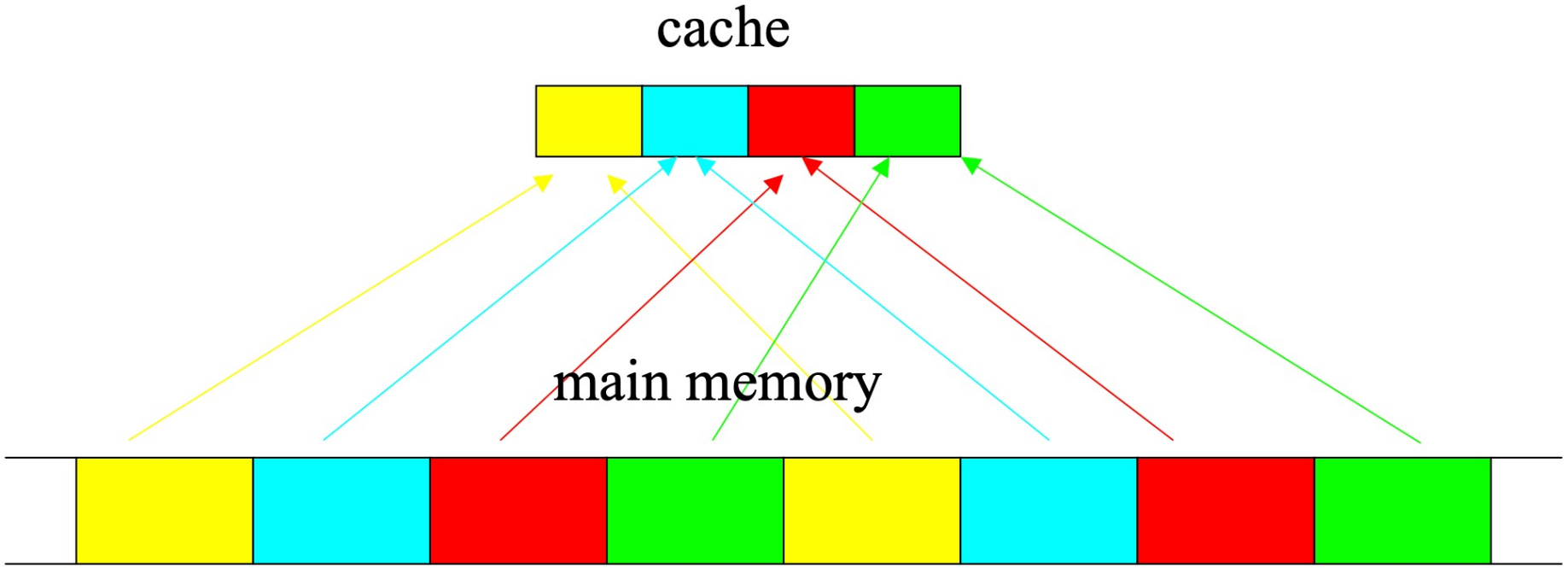
- Data can be arranged to fit in cache

Cache mapping

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

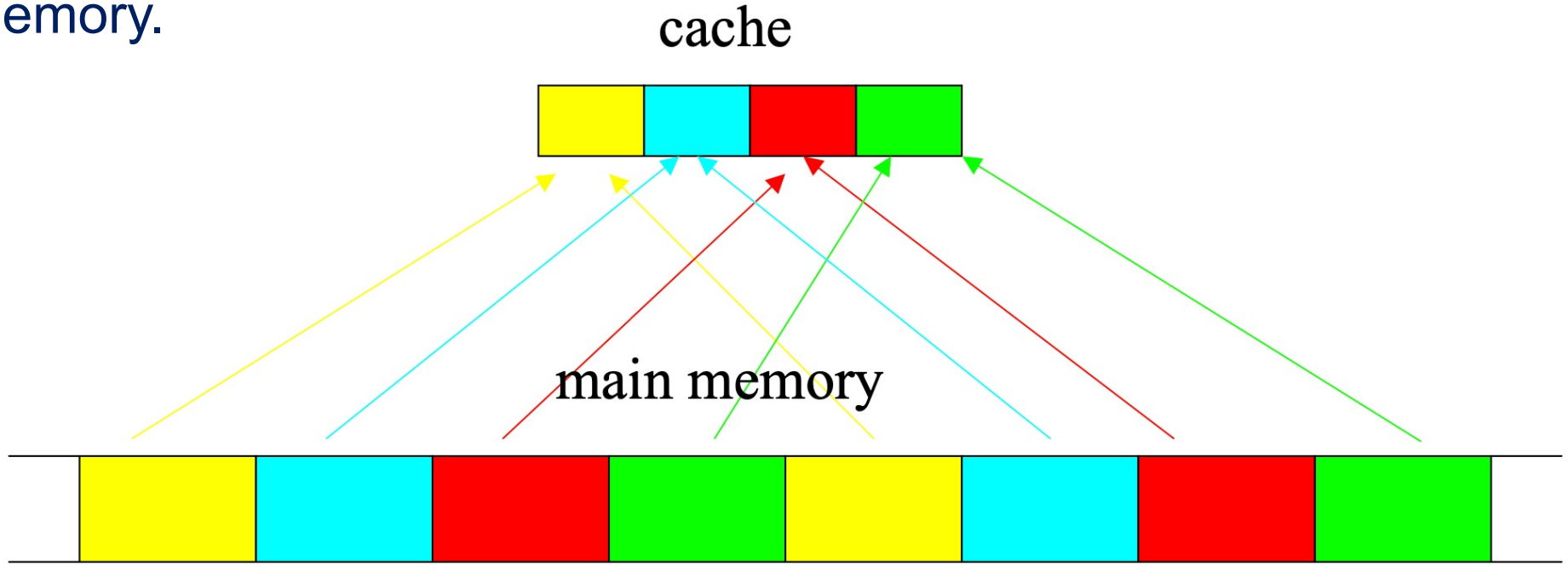
Direct Mapped Caches

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.



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
- 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

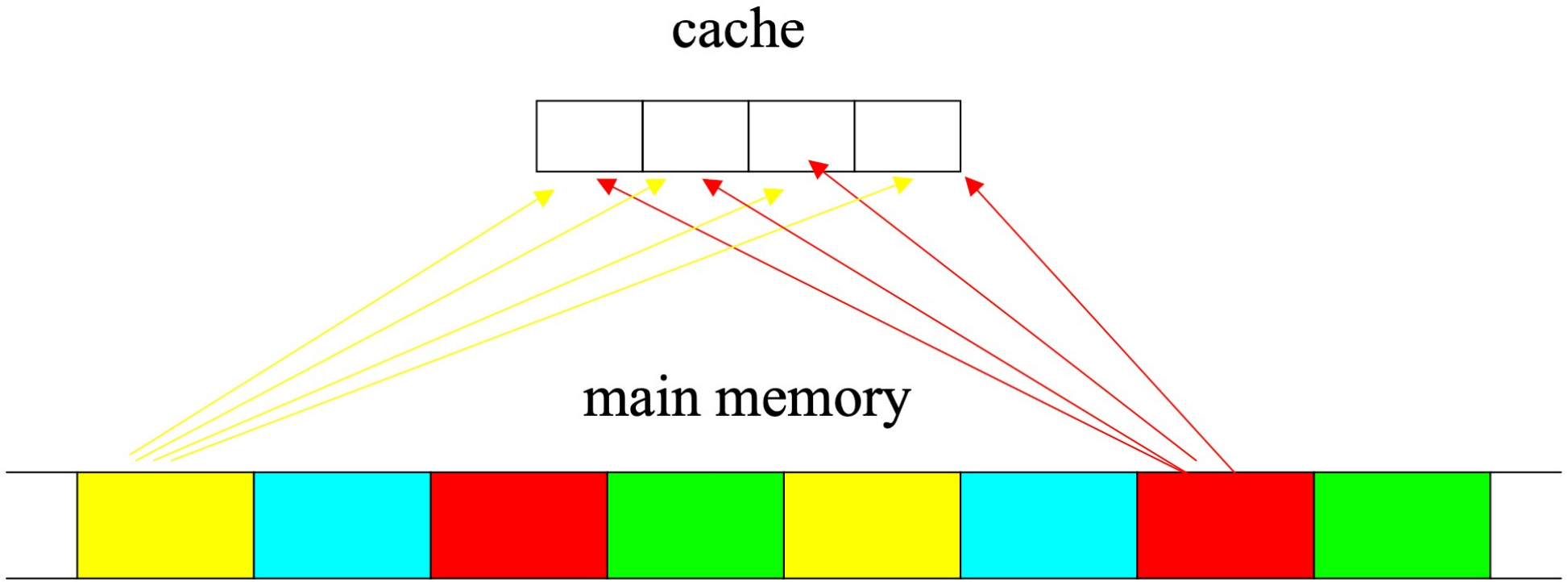
```
double a[8192],b[8192];
for (i=0; i<n; i++) {
  a[i] = b[i]
}
```

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

- 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

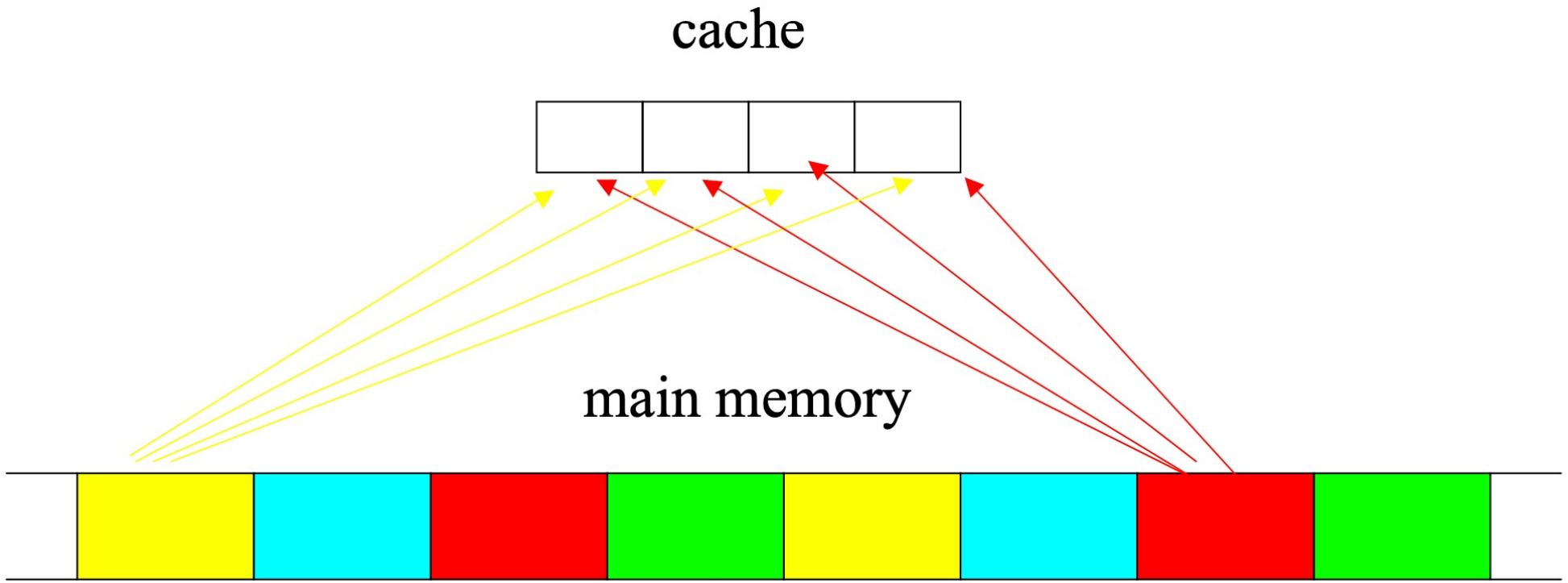
Fully Associative Caches

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.



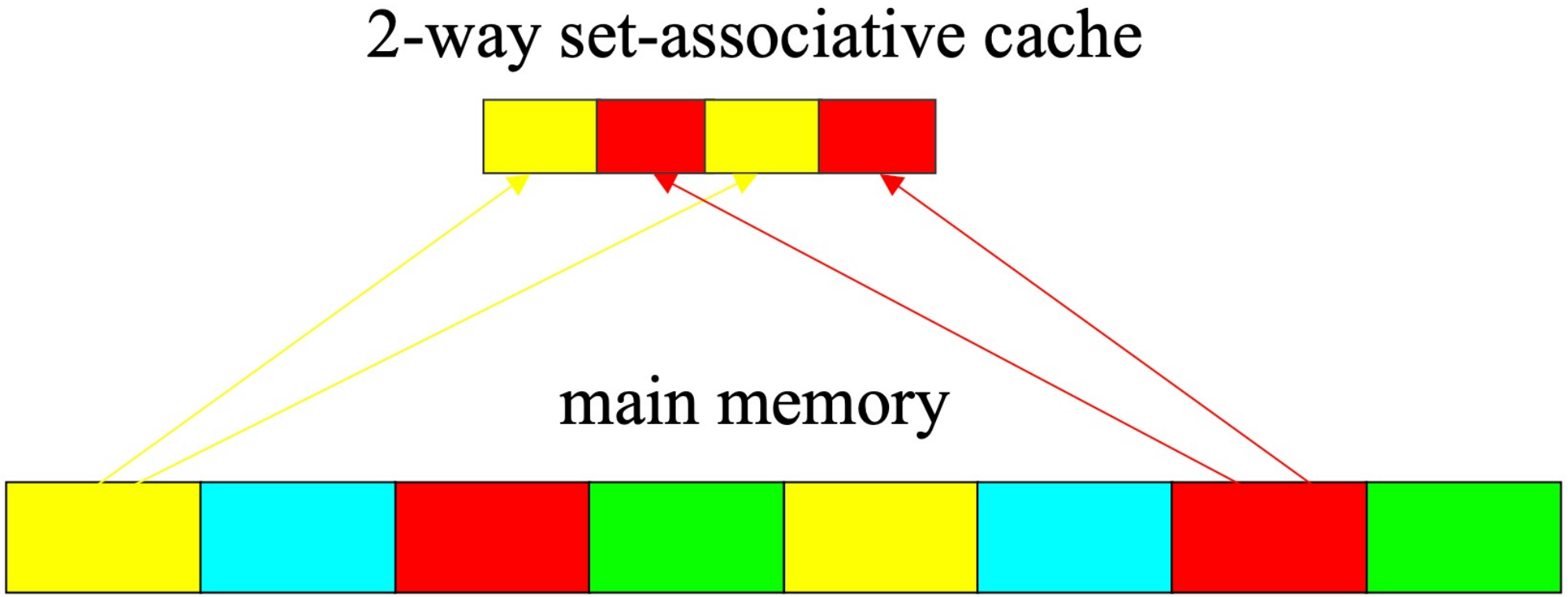
Fully Associative Caches

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



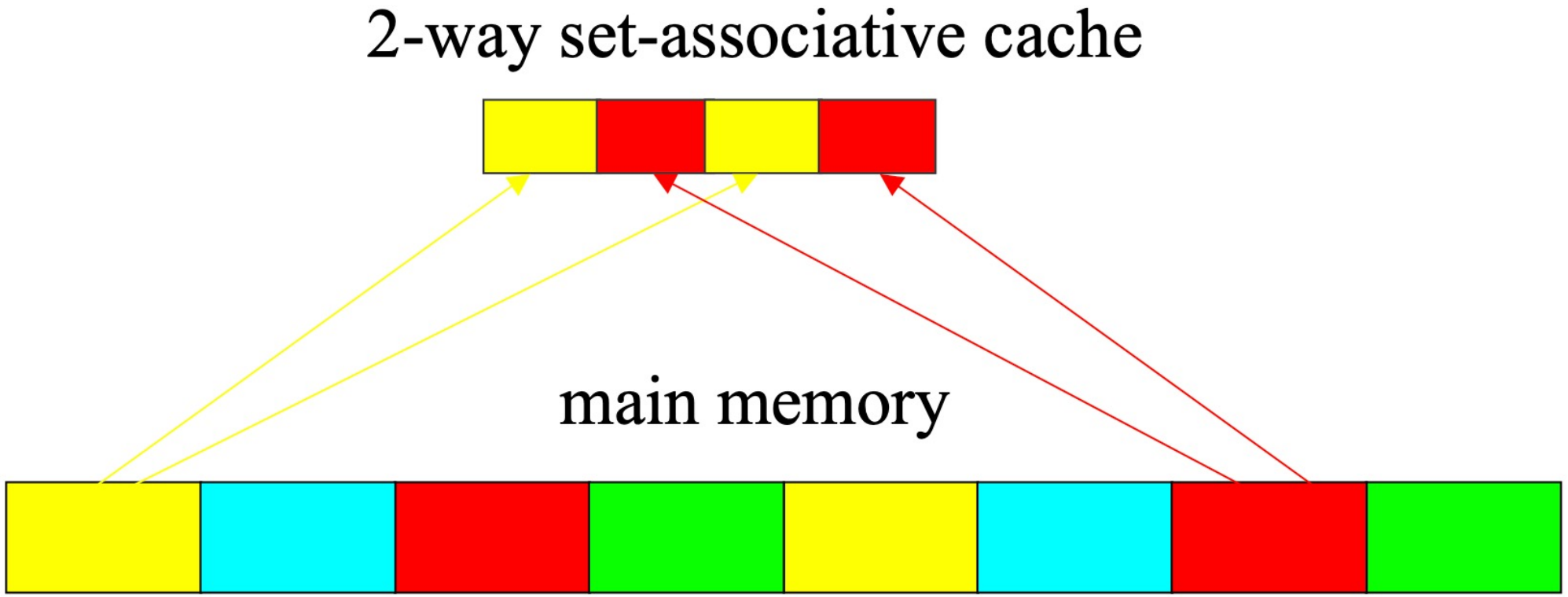
Set Associative Caches

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.



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

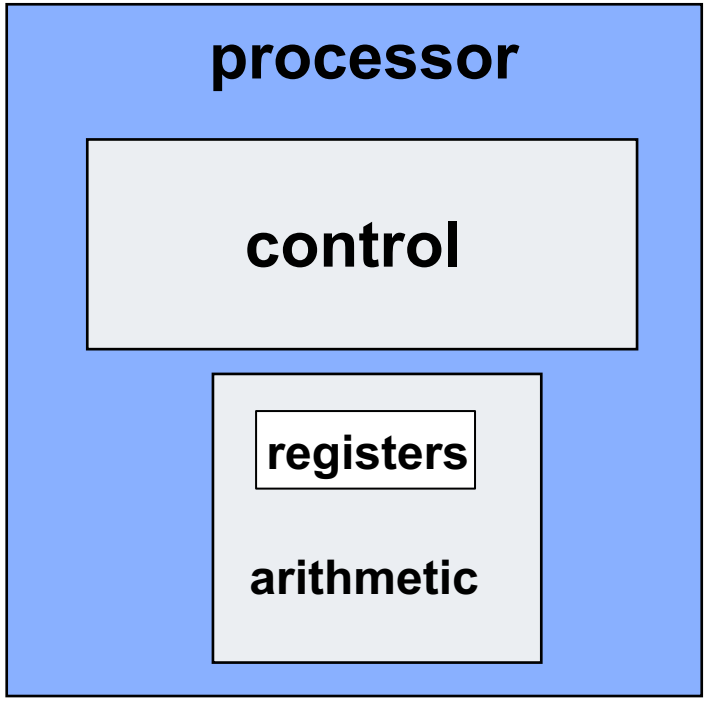
2. Parallelism within single processors
    - Pipelining
    - SIMD
    - Special instructions (FMA)

3. Case study: Matrix multiplication

4. Illustrations

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.



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-2

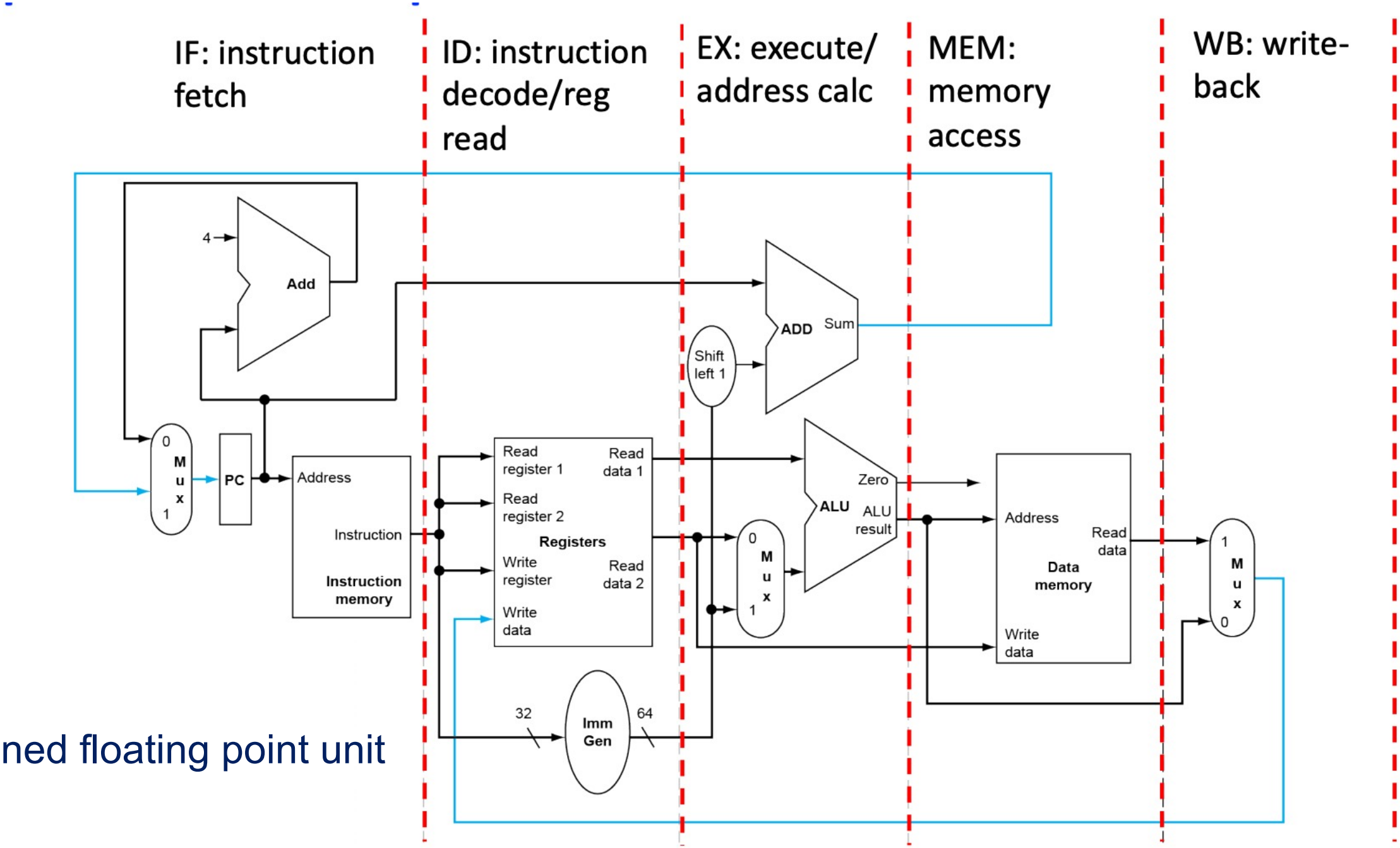
- Executing the addition, in this case 0.41 e-1
- Storing the result.

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

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

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 $t(n) = nl\tau$ to do *n* additions.

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

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

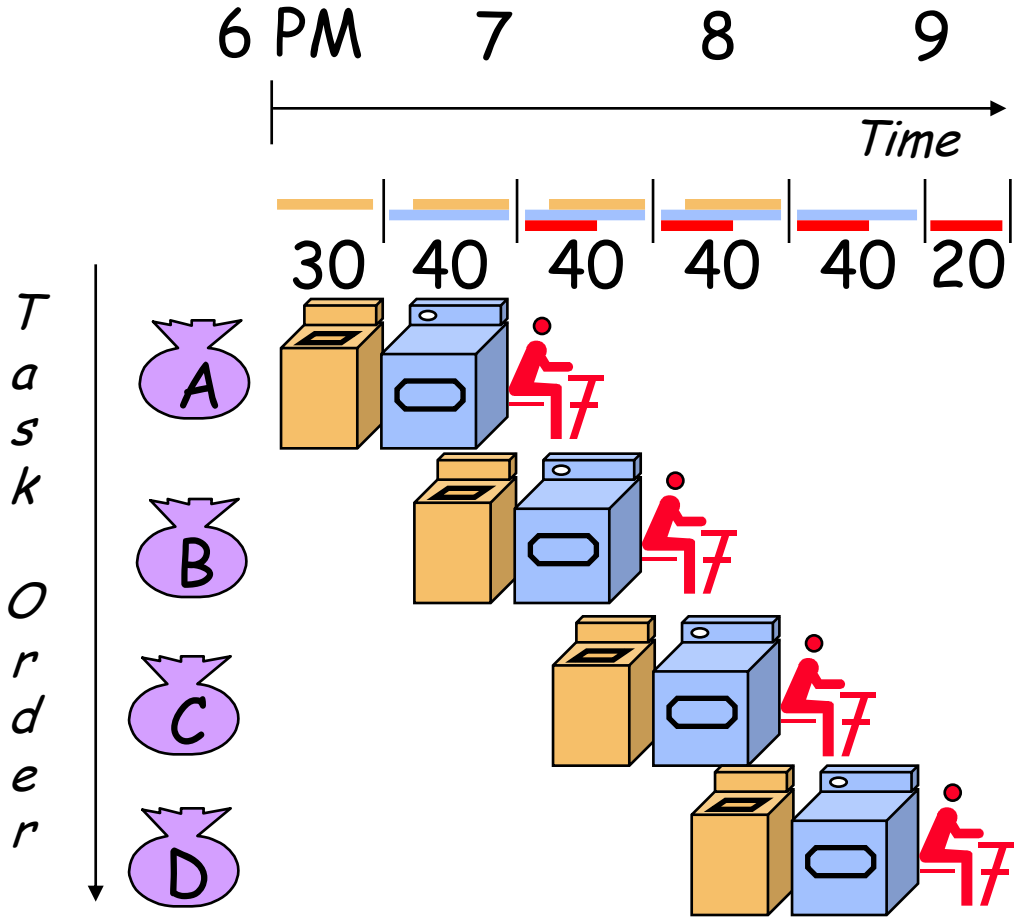
Traditional: $r_\infty = \frac{1}{l\tau}$ Pipelined: $r_\infty = \frac{1}{\tau}$

Speedup by the number of stages l

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

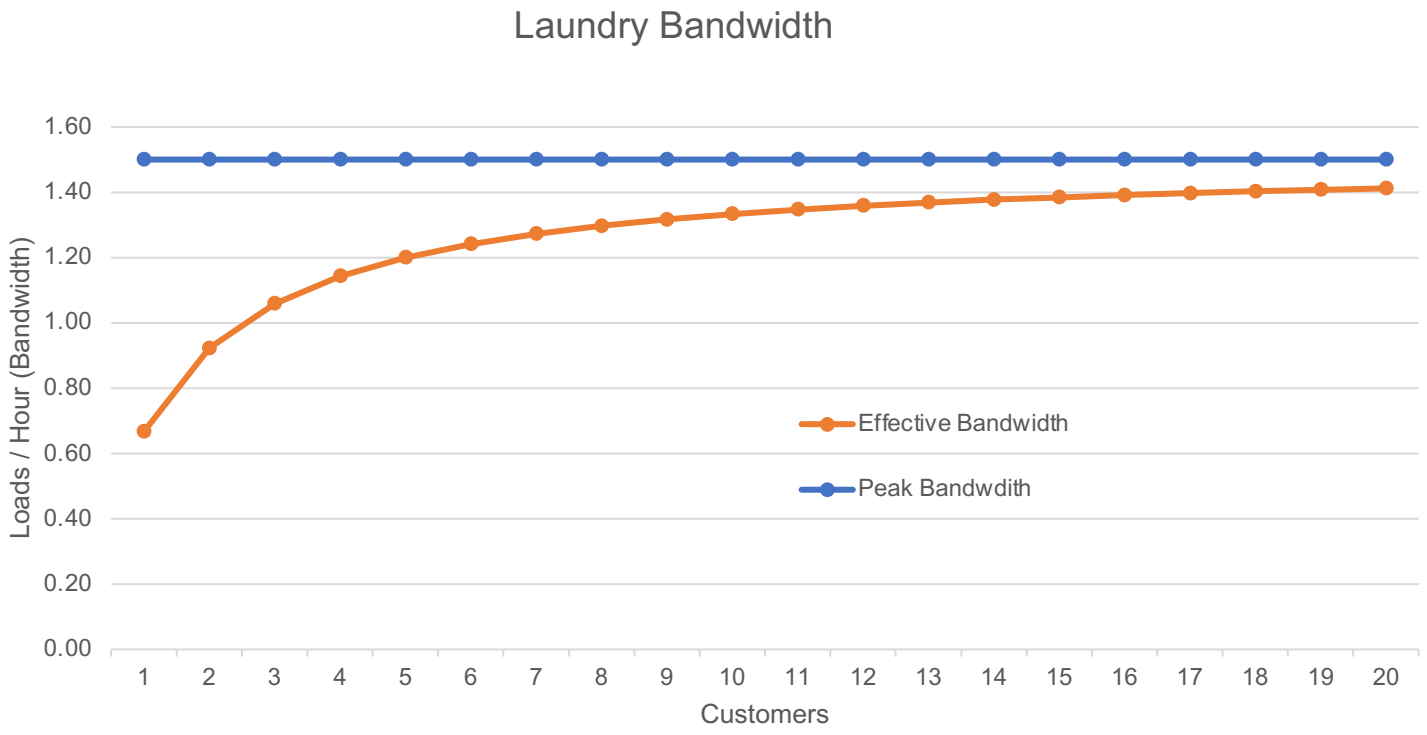
loads/hour

4/6	l/h	w/o pipelining
4/3.5	l/h	w pipelining
<= 4/2	l/h	w pipelining

Speedup <= # of stages

Laundry peak performance

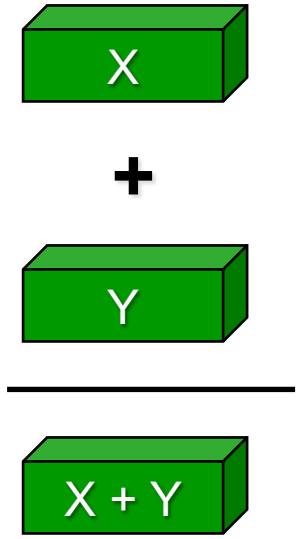
- The system bandwidth is 1 load per 40 minutes (1.5 Loads / hour)
- How much of that you see depends on available ”customers” ready to do laundry



SIMD: Single Instruction Multiple Data

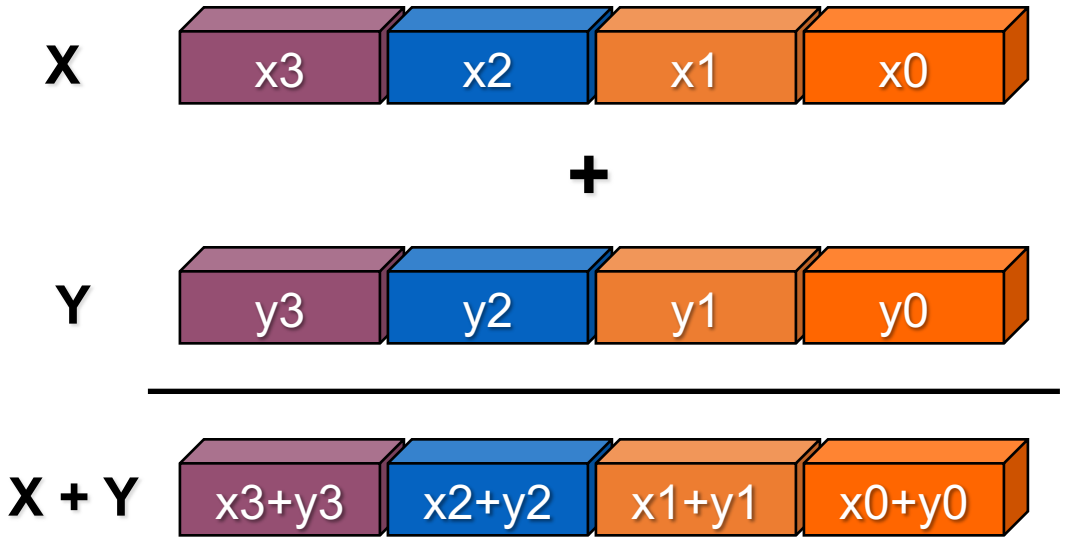
Scalar processing

- traditional mode
- one operation produces one result



SIMD processing: vectors

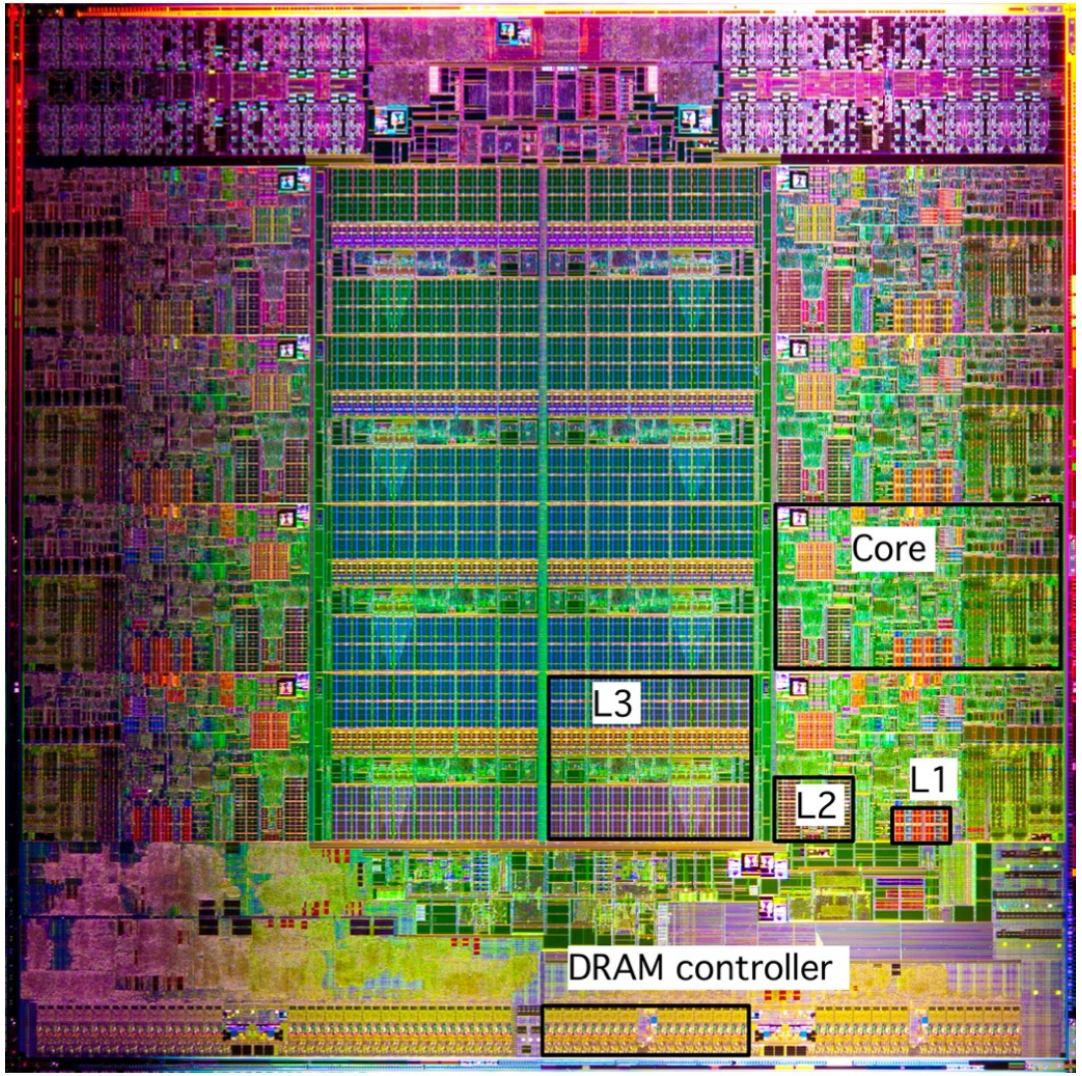
- Sandy Bridge: AVX (256 bit)
- Haswell: AVX2 (256 bit w/ FMA)
- KNL: AVX-512 (512 bit)



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

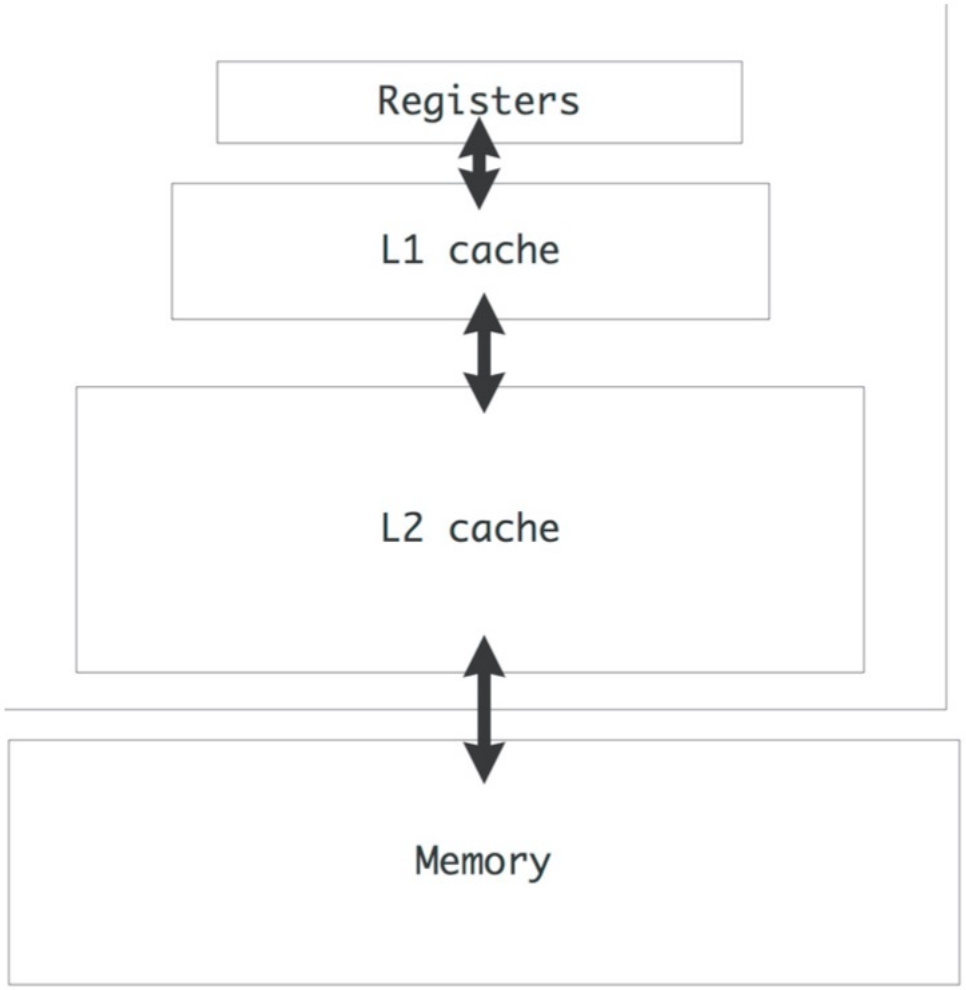
Multicore Architecture

- 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 chinp 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.

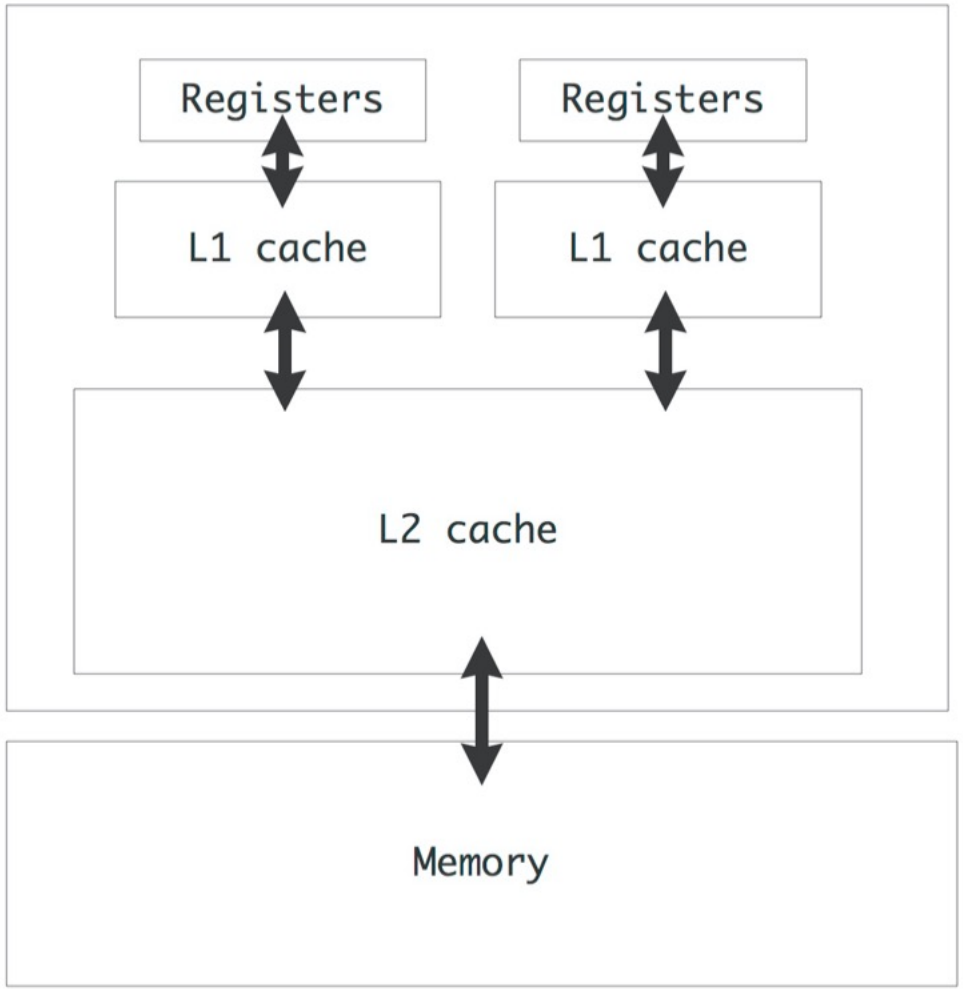


Multicore Architecture

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 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.

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

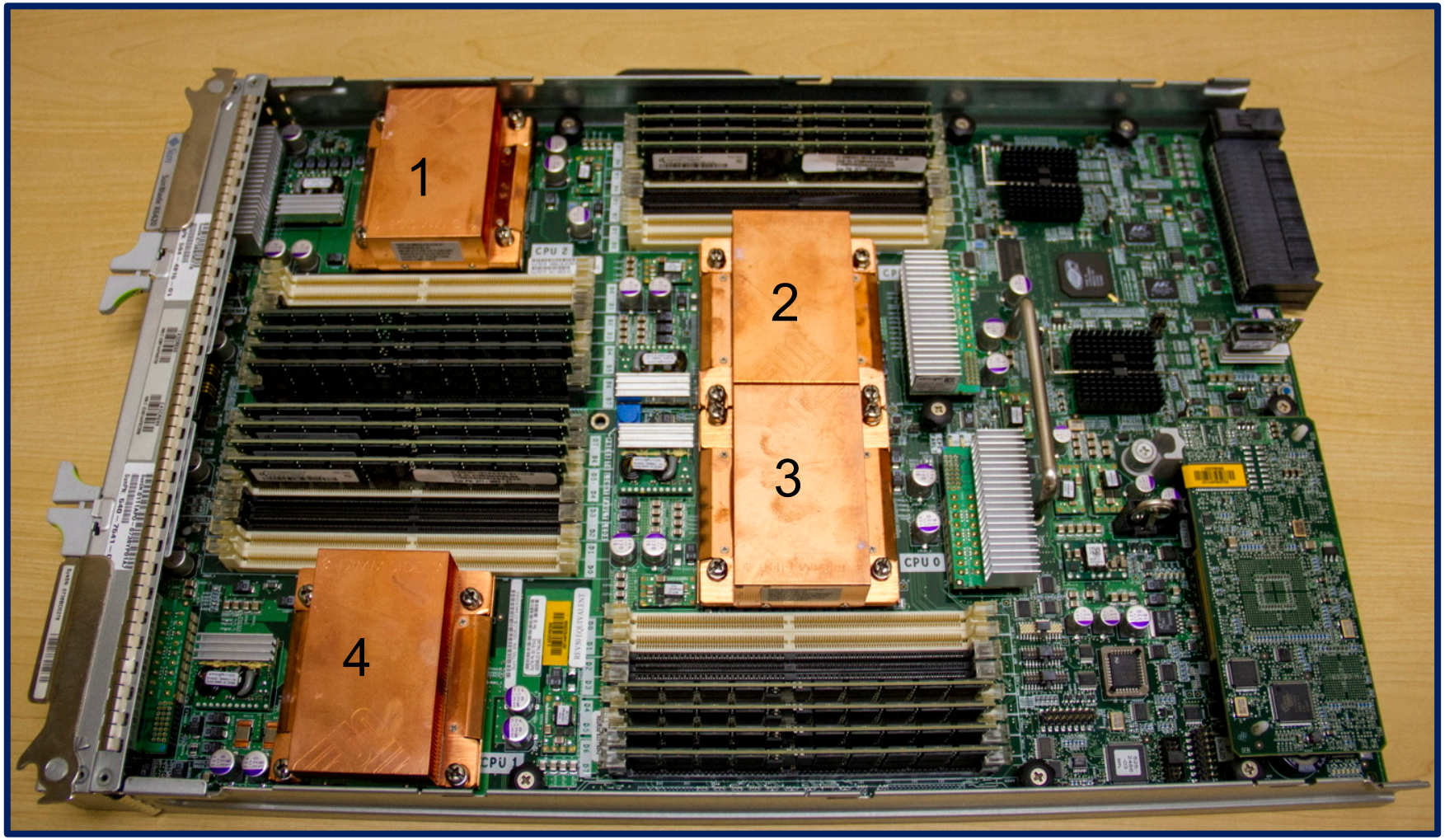
存储：528 TB

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



Instruction-level parallelism

- SIMD: instructions that are independent can be started at the same time
- 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 cycle

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]
```

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
  - 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

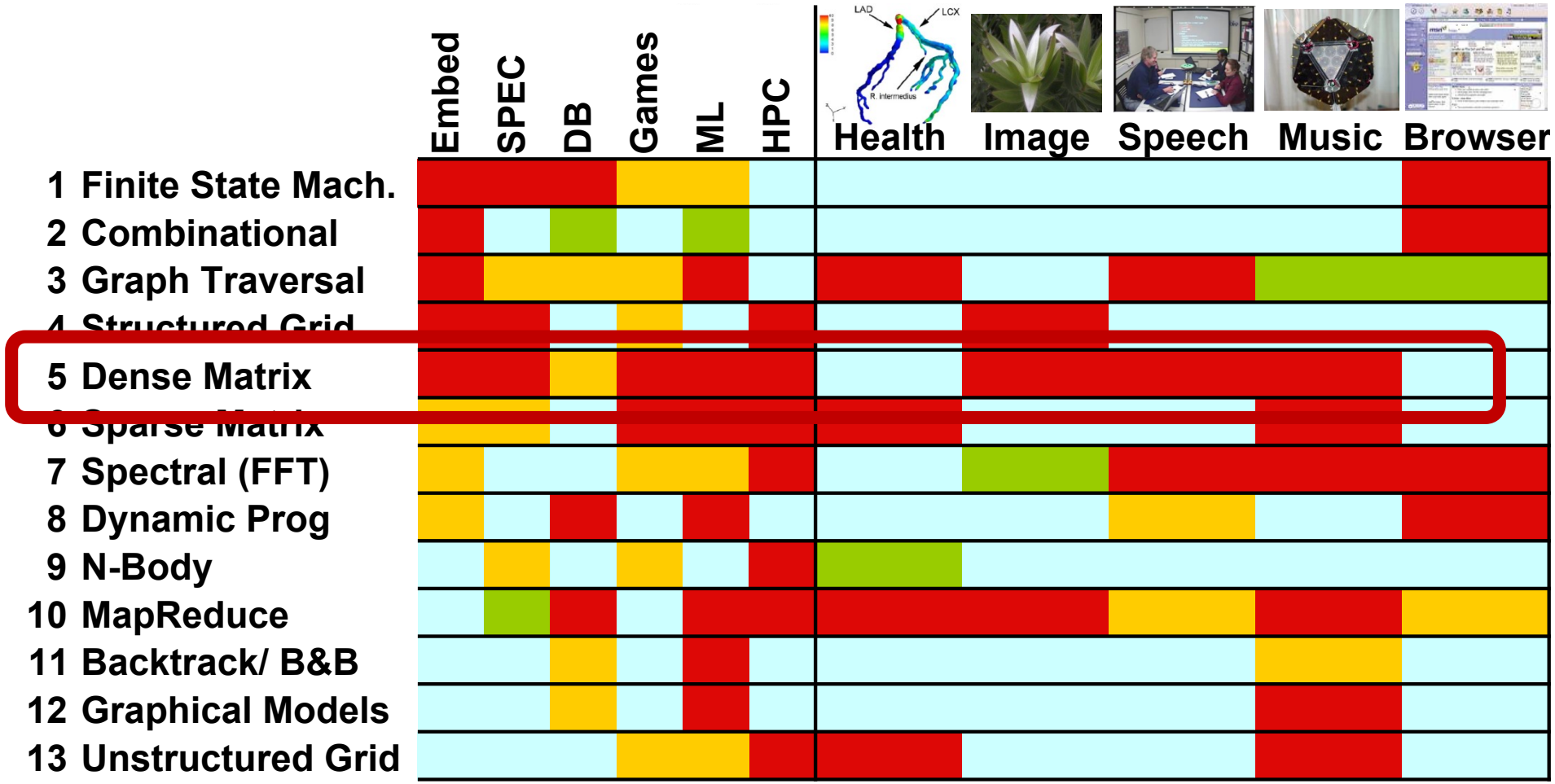
4. Illustrations

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

  recursive (later)

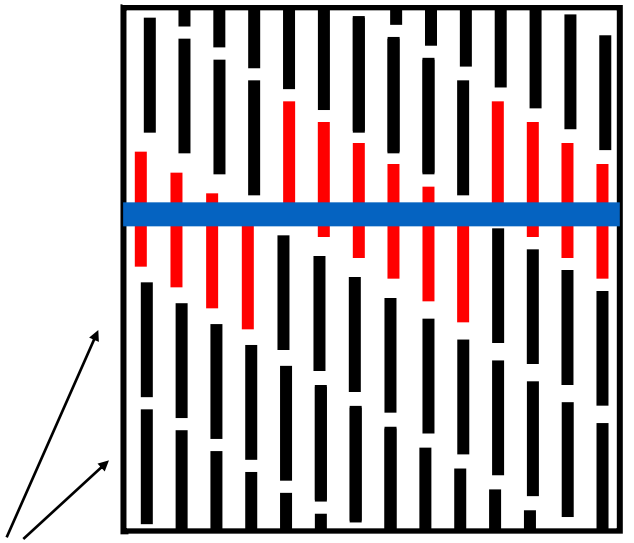
Column major

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



cachelines

Blue row of matrix is stored in red cache lines

- Column major (for now)

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 (words) 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)

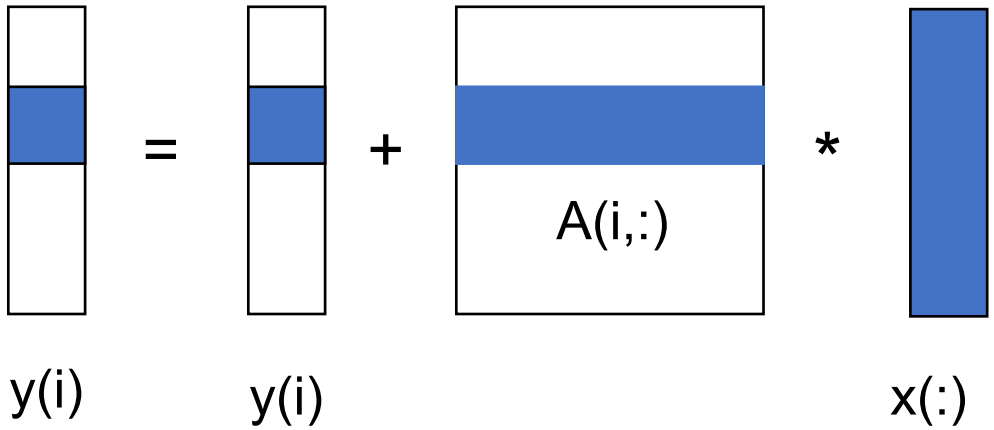
Key to machine efficiency

- 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 $\approx$ 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 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

	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 **(CI must be at least this for ½ peak speed)**

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

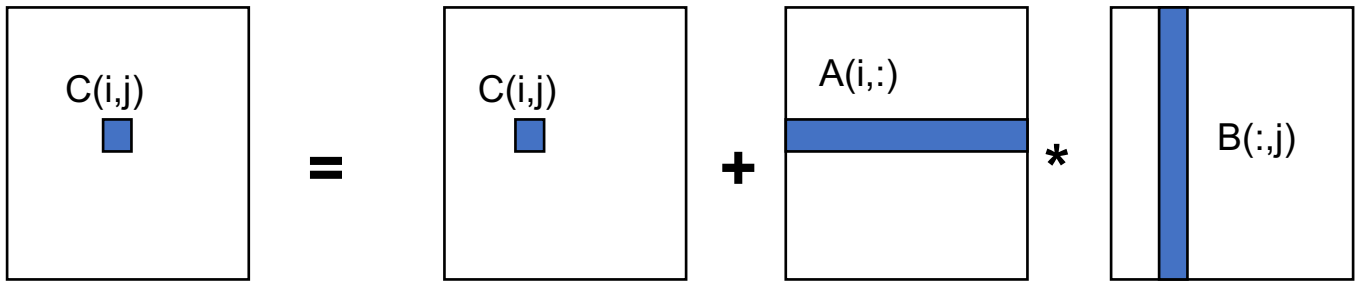
    Reasonable if we are talking about registers

    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

  Mflop rate/element = $2 / (2 * t_f + t_m)$

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) back to slow memory}
```

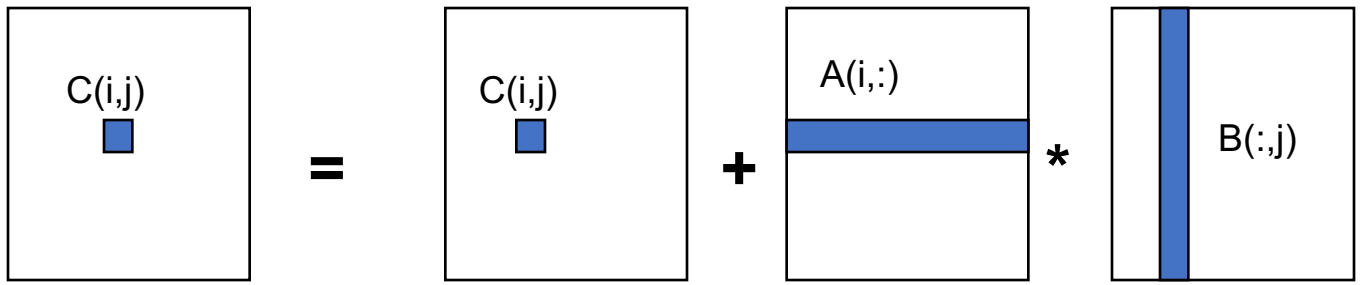


Matrix multiplication

```
{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

Number of slow memory references on unblocked matrix multiply

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

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

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

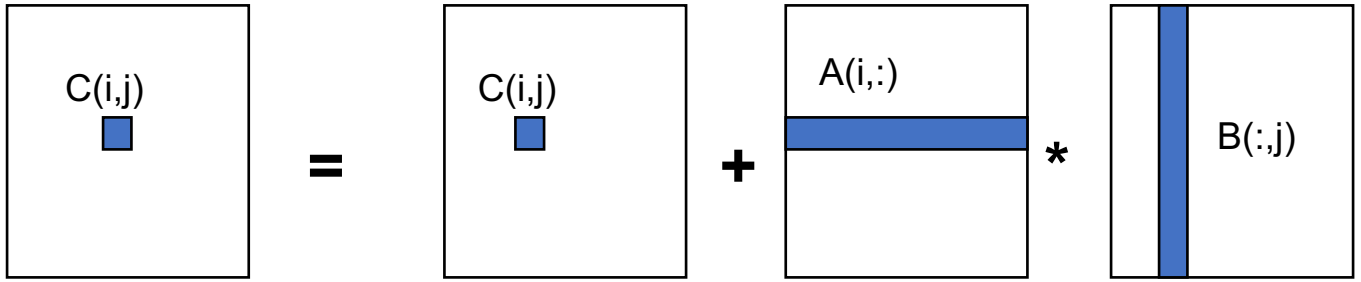
$= n^3 + 3n^2$

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

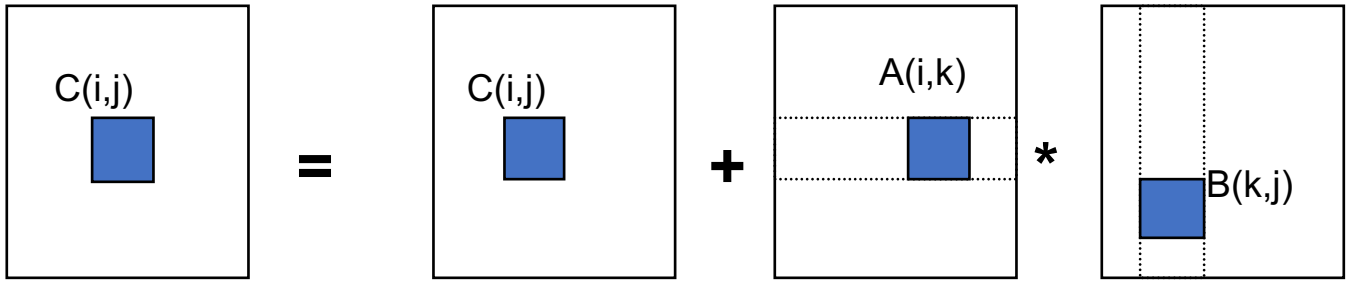
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
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 amount 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 $q = f / m = 2n^3 / ((2N + 2) * n^2)$

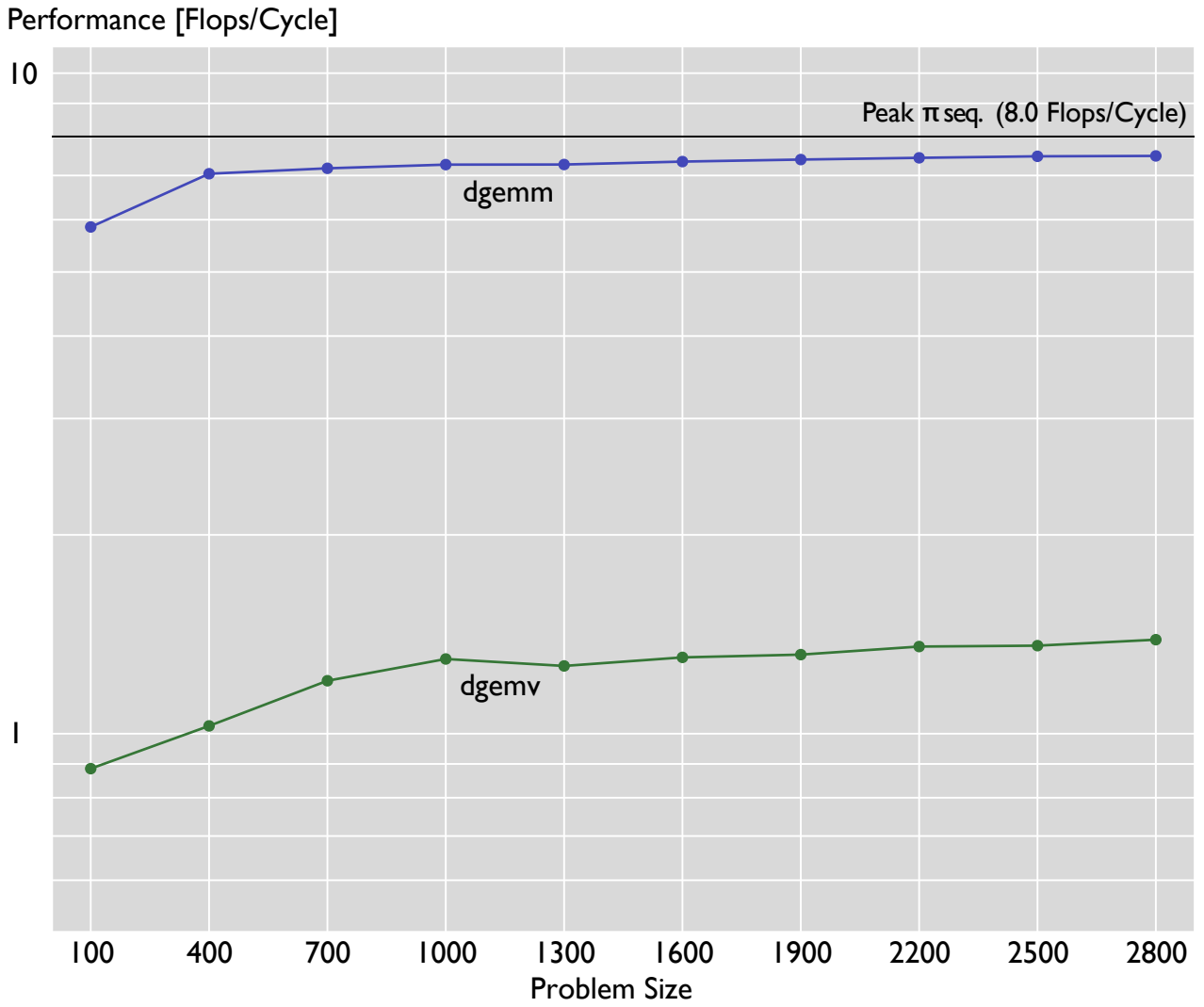
$\approx n / N = b$ for large n

So we can improve performance by increasing the blocksize b

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

Matrix multiplication

Measuring Performance — Flops/Cycle



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. Illustrations

Tuning code in practice

Tuning code can be tedious

- Many optimizations besides blocking
- Behavior of machine and compiler hard to predict

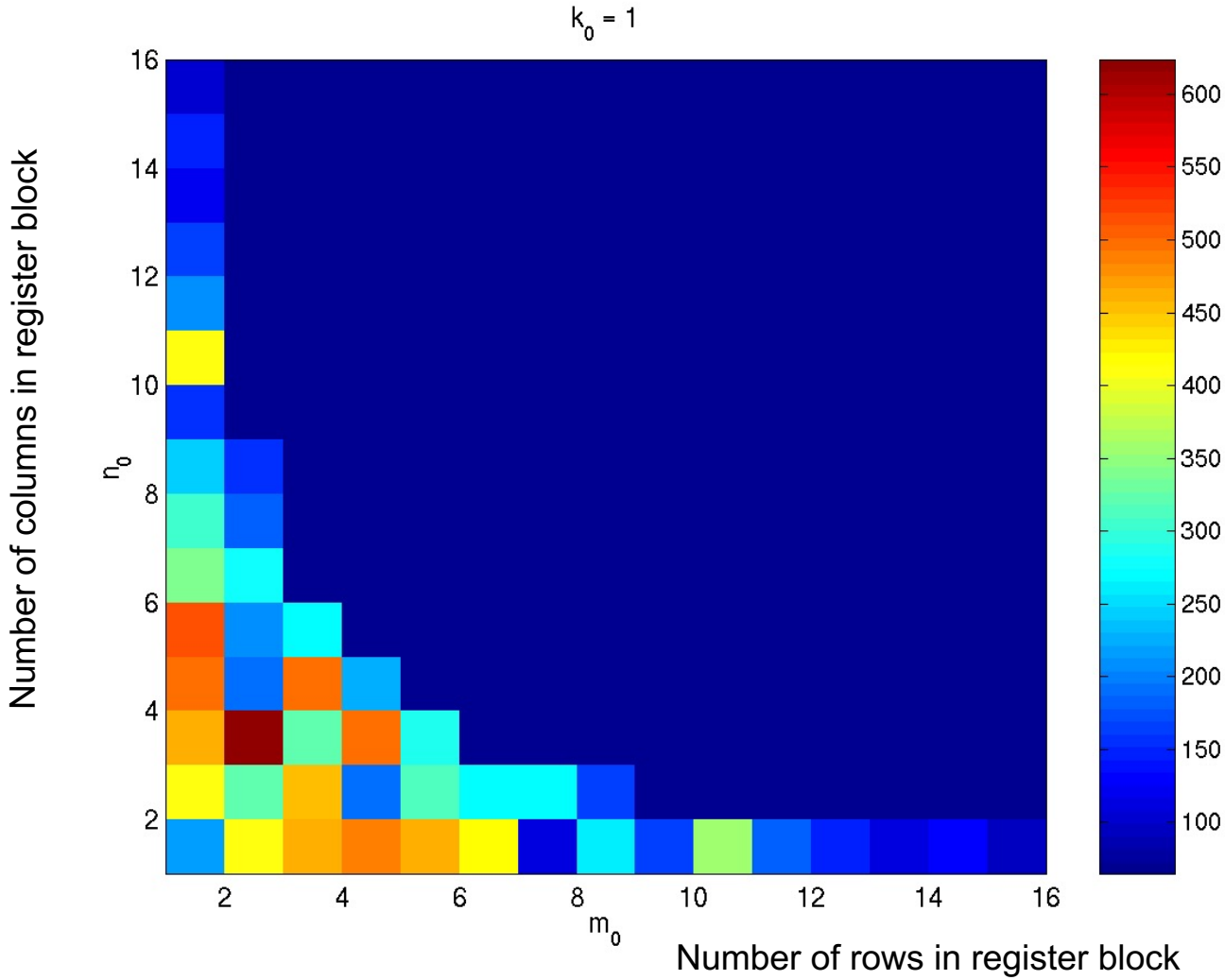
Approach #1: Analytical performance models

- Use model (of cache, memory costs, etc.) to select best code
- But model needs to be both simple and accurate ☹

Approach #2: “Autotuning”

- Let computer generate large set of possible code variations, and search for the fastest ones
- Sometimes all done “off-line”, sometimes at run-time

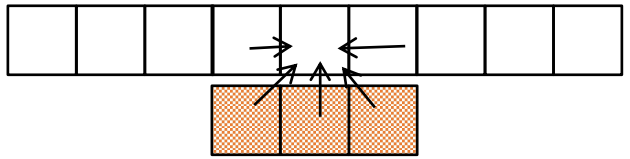
Tuning code in practice



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

Tuning code in practice

Reduce demands on memory bandwidth by pre-loading into local variables




```
while( … ) {
   *res++ = filter[0]*signal[0]
            + filter[1]*signal[1]
            + filter[2]*signal[2];
   signal++;
}
```

```
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( … ) {
   *res++ = f0*signal[0]
            + f1*signal[1]
            + f2*signal[2];
   signal++;
}
```

also: `register float f0 = …;`

Example is a convolution

Tuning code in practice

Expose instruction-level parallelism

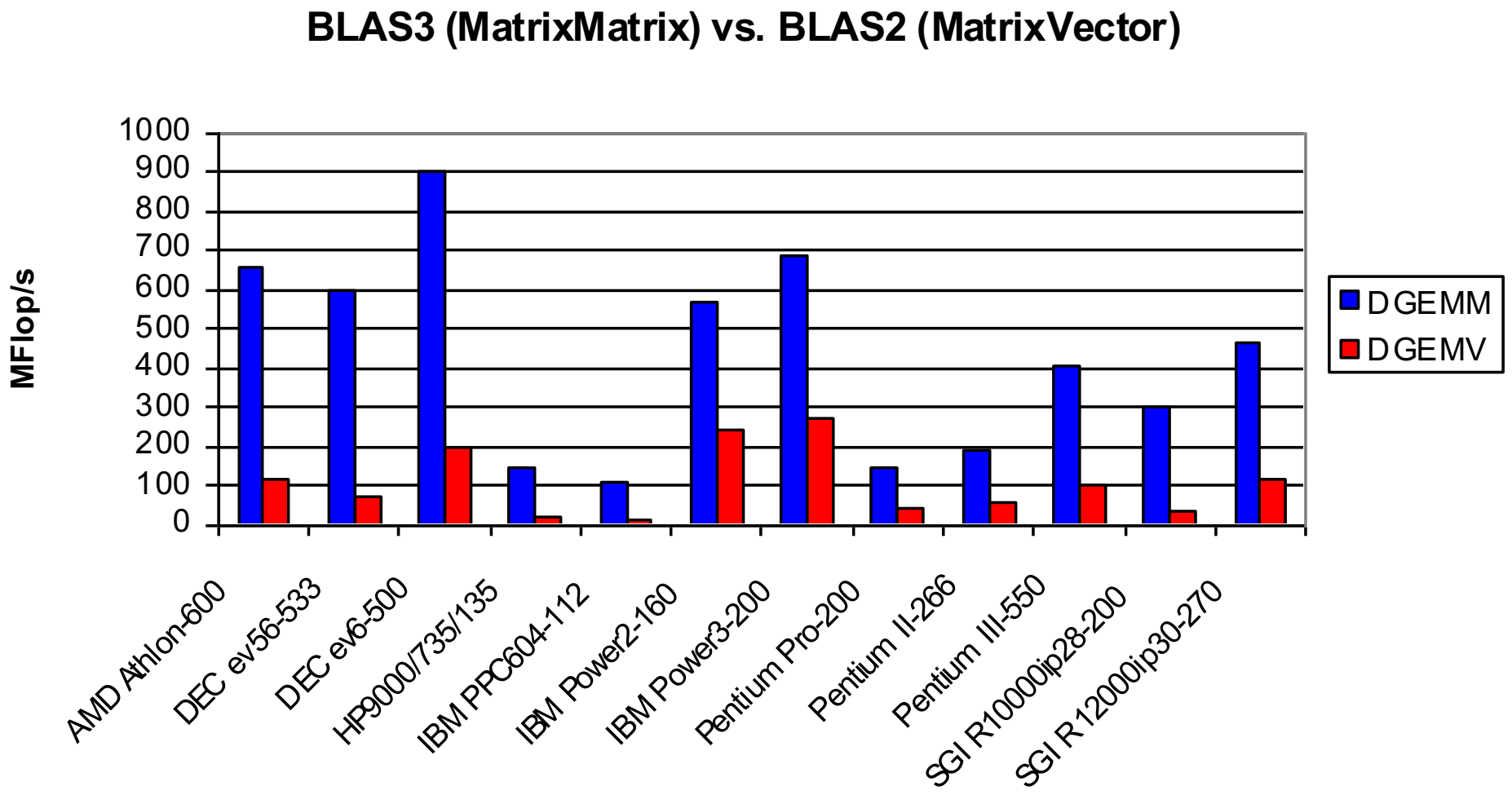
```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
   signal += 3;
   s0 = signal[0];
   res[0] = f0*s1 + f1*s2 + f2*s0;

   s1 = signal[1];
   res[1] = f0*s2 + f1*s0 + f2*s1;

   s2 = signal[2];
   res[2] = f0*s0 + f1*s1 + f2*s2;

   res += 3;
} while( … );
```

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

Dense Linear Algebra

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



Tuning code in practice

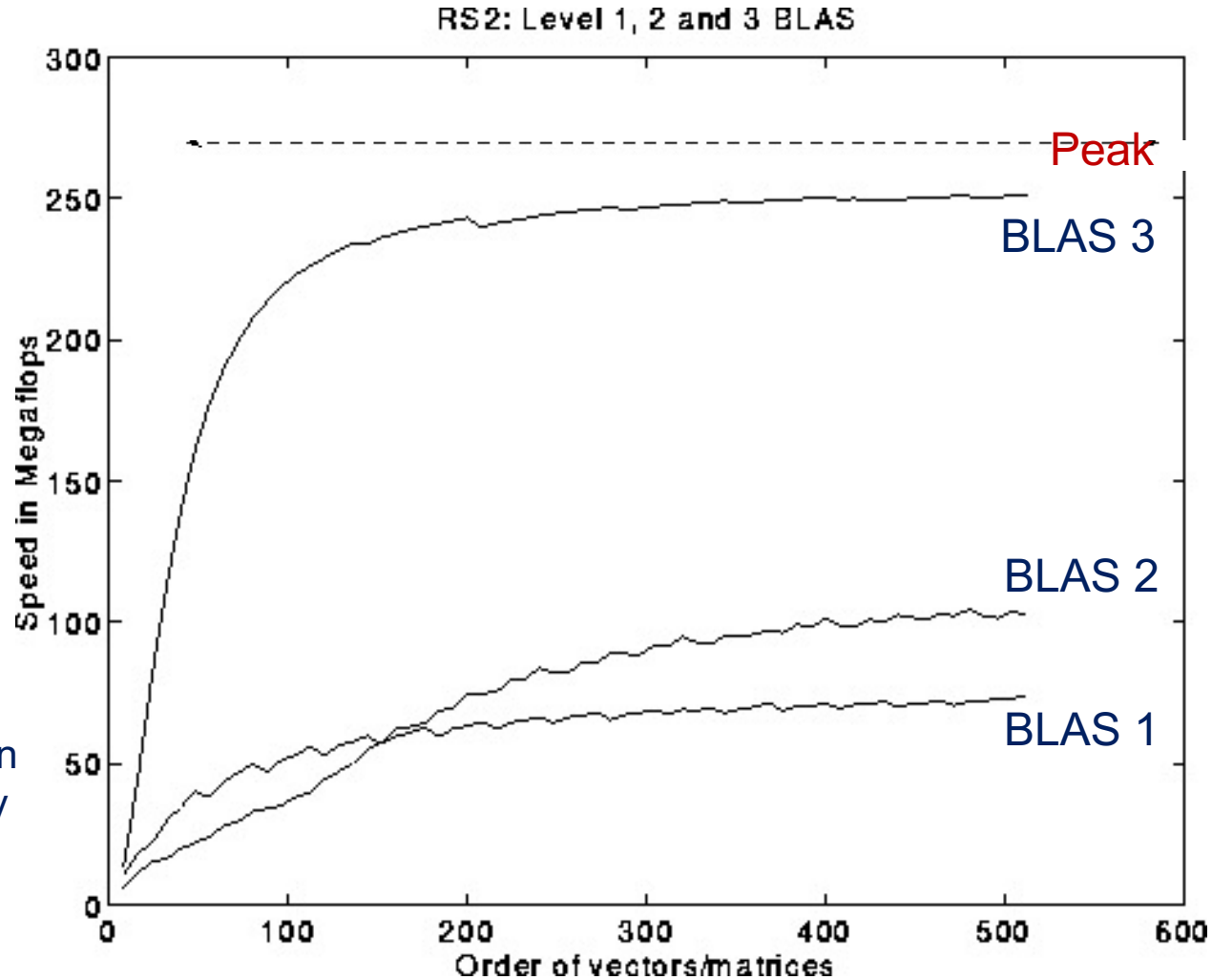
Peak speed = 266 Mflops

BLAS 3 (n-by-n matrix matrix multiply)

BLAS 2 (n-by-n matrix vector multiply)

BLAS 1 (saxpy of n vectors)

Matrices start in DRAM Memory



References

- "Performance Optimization of Numerically Intensive Codes", by Stefan Goedecker and Adolfy Hoisie, SIAM 2001.
- Web pages for reference:
  - BeBOP Homepage
  - ATLAS Homepage
  - 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.

Summary

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - Before you parallelize, make sure you're getting good serial performance
  - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
  - Pipelining, SIMD, etc
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby to recently used data
- Can rearrange code/data to improve locality
  - Goal: minimize communication = data movement

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.

Assignment: Read chapter 1 of “Introduction to High Performance Scientific Computing ” by V. Eijkhout