Supercomputing in Plain English

The Tyranny of the Storage Hierarchy

Henry Neeman, Director

Director, OU Supercomputing Center for Education & Research (OSCER)
Assistant Vice President, Information Technology – Research Strategy Advisor
Associate Professor, College of Engineering
Adjunct Associate Professor, School of Computer Science
University of Oklahoma

Tuesday January 27 2015

This is an experiment!

It’s the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

PLEASE MUTE YOURSELF

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with echo cancellation.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Download the Slides Beforehand

Before the start of the session, please download the slides from the Supercomputing in Plain English website:

http://www.oscer.ou.edu/education/

That way, if anything goes wrong, you can still follow along with just audio.

PLEASE MUTE YOURSELF.

H.323 (Polycom etc) #1

If you want to use H.323 videoconferencing – for example, Polycom – then:

- If you AREN'T registered with the OneNet gatekeeper (which is probably the case), then:
  - Dial **`164.58.250.47`**
  - Bring up the virtual keypad.
    On some H.323 devices, you can bring up the virtual keypad by typing:
    #
    (You may want to try without first, then with; some devices won't work with the #, but give cryptic error messages about it.)
  - When asked for the conference ID, or if there's no response, enter:
    `0409`
  - On most but not all H.323 devices, you indicate the end of the ID with:
    #

H.323 (Polycom etc) #2

If you want to use H.323 videoconferencing – for example, Polycom – then:

- If you ARE already registered with the OneNet gatekeeper (most institutions aren't), dial:

  `2500409`

Many thanks to James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman of OneNet for providing this.

PLEASE MUTE YOURSELF.

Wowza #1

You can watch from a Windows, MacOS or Linux laptop using Wowza from the following URL:

http://jwplayer.onenet.net/stream6/sipe.html

Wowza behaves a lot like YouTube, except live.

Many thanks to James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman of OneNet for providing this.

PLEASE MUTE YOURSELF.

Wowza #2

Wowza has been tested on multiple browsers on each of:

- Windows (7 and 8): IE, Firefox, Chrome, Opera, Safari
- MacOS X: Safari, Firefox
- Linux: Firefox, Opera

PLEASE MUTE YOURSELF.

Toll Free Phone Bridge

IF ALL ELSE FAILS, you can use our toll free phone bridge:

800-832-0736

* 623 2874 #

Please mute yourself and use the phone to listen.

Don’t worry, we’ll call out slide numbers as we go.

Please use the phone bridge **ONLY** if you cannot connect any other way: the phone bridge can handle only 100 simultaneous connections, and we have over 500 participants.

Many thanks to OU CIO Loretta Early for providing the toll free phone bridge.

PLEASE MUTE YOURSELF.

Please Mute Yourself

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

(For Wowza, you don't need to do that, because the information only goes from us to you, not from you to us.)

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with echo cancellation.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Questions via E-mail Only

Ask questions by sending e-mail to:

sipe2015@gmail.com

All questions will be read out loud and then answered out loud.

PLEASE MUTE YOURSELF.

Onsite: Talent Release Form

If you're attending onsite, you **MUST** do one of the following:

- complete and sign the Talent Release Form,

OR

- sit behind the cameras (where you can't be seen) and don't talk at all.

If you aren't onsite, then **PLEASE MUTE YOURSELF.**

TENTATIVE Schedule

Tue Jan 20: Storage Hierarchy: What the Heck is Supercomputing?
Tue Jan 27: The Tyranny of the Storage Hierarchy
Tue Feb 3: Instruction Level Parallelism
Tue Feb 10: Stupid Compiler Tricks
Tue Feb 17: Shared Memory Multithreading
Tue Feb 24: Distributed Multiprocessing
Tue March 3: Applications and Types of Parallelism
Tue March 10: Multicore Madness
Tue March 17: **NO SESSION** (OU's Spring Break)
Tue March 24: **NO SESSION** (Henry has a huge grant proposal due)
Tue March 31: High Throughput Computing
Tue Apr 7: GPGPU: Number Crunching in Your Graphics Card
Tue Apr 14: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Brandon George, Dave Akin, Brett Zimmerman, Josh Alexander, Patrick Calhoun)
  - Horst Severini, OSCER Associate Director for Remote & Heterogeneous Computing
  - Debi Gentis, OSCER Coordinator
  - Jim Summers
  - The OU IT network team
- James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman, OneNet
- Kay Avila, U Iowa
- Stephen Harrell, Purdue U

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

PLEASE MUTE YOURSELF.

Coming in 2015!

Linux Clusters Institute workshop May 18-22 2015 @ OU

http://www.linuxclustersinstitute.org/workshops/

Great Plains Network Annual Meeting, May 27-29, Kansas City

Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency May 31 - June 6 2015

XSEDE2015, July 26-30, St. Louis MO

https://conferences.xsede.org/xsede15

IEEE Cluster 2015, Sep 23-27, Chicago IL

http://www.mcs.anl.gov/ieeecluster2015/

OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2015, **Sep 22-23 2015** @ OU

SC13, Nov 15-20 2015, Austin TX

http://sc15.supercomputing.org/

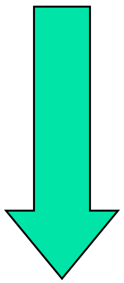
PLEASE MUTE YOURSELF.

Outline

- What is the storage hierarchy?
- Registers
- Cache
- Main Memory (RAM)
- The Relationship Between RAM and Cache
- The Importance of Being Local
- Hard Disk
- Virtual Memory

The Storage Hierarchy

Fast, expensive, few



Slow, cheap, a lot

[5]

- Registers
- Cache memory
- Main memory (RAM)
- Hard disk
- Removable media (CD, DVD etc)
- Internet

Henry's Laptop

Dell Latitude E5540[4]

http://content.hwigroup.net/images/products/xl/204419/dell_latitude_e5540_55405115.jpg

- Intel Core i3-4010U
  dual core, 1.7 GHz, 3 MB L3 Cache
- 12 GB 1600 MHz DDR3L SDRAM
- 340 GB SATA 5400 RPM Hard Drive
- DVD±RW/CD-RW Drive
- 1 Gbps Ethernet Adapter

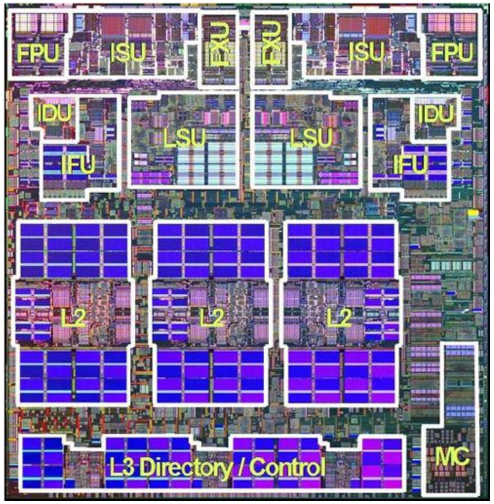
Storage Speed, Size, Cost

Henry's Laptop	Registers (Intel Core2 Duo 1.6 GHz)	Cache Memory (L3)	Main Memory (1600MHz DDR3L SDRAM)	Hard Drive	Ethernet (1000 Mbps)	DVD±R (16x)	Phone Modem (56 Kbps)
Speed (MB/sec) [peak]	668,672[6] (27.2 GFLOP/s*)	46,000	15,000 [7]	100[9]	125	32 [10]	0.007
Size (MB)	832 bytes** [11]	3	12,288 4096 times as much as cache	340,000	unlimited	unlimited	unlimited
Cost ($/MB)	–	$38 [12]	$0.0084 [12] ~1/4500 as much as cache	$0.00003 [12]	charged per month (typically)	$0.000045 [12]	charged per month (typically)

* GFLOP/s: billions of floating point operations per second

** 16 64-bit general purpose registers, 8 64-bit floating point registers, 8 128-bit floating point vector registers, 16 256-bit floating point registers

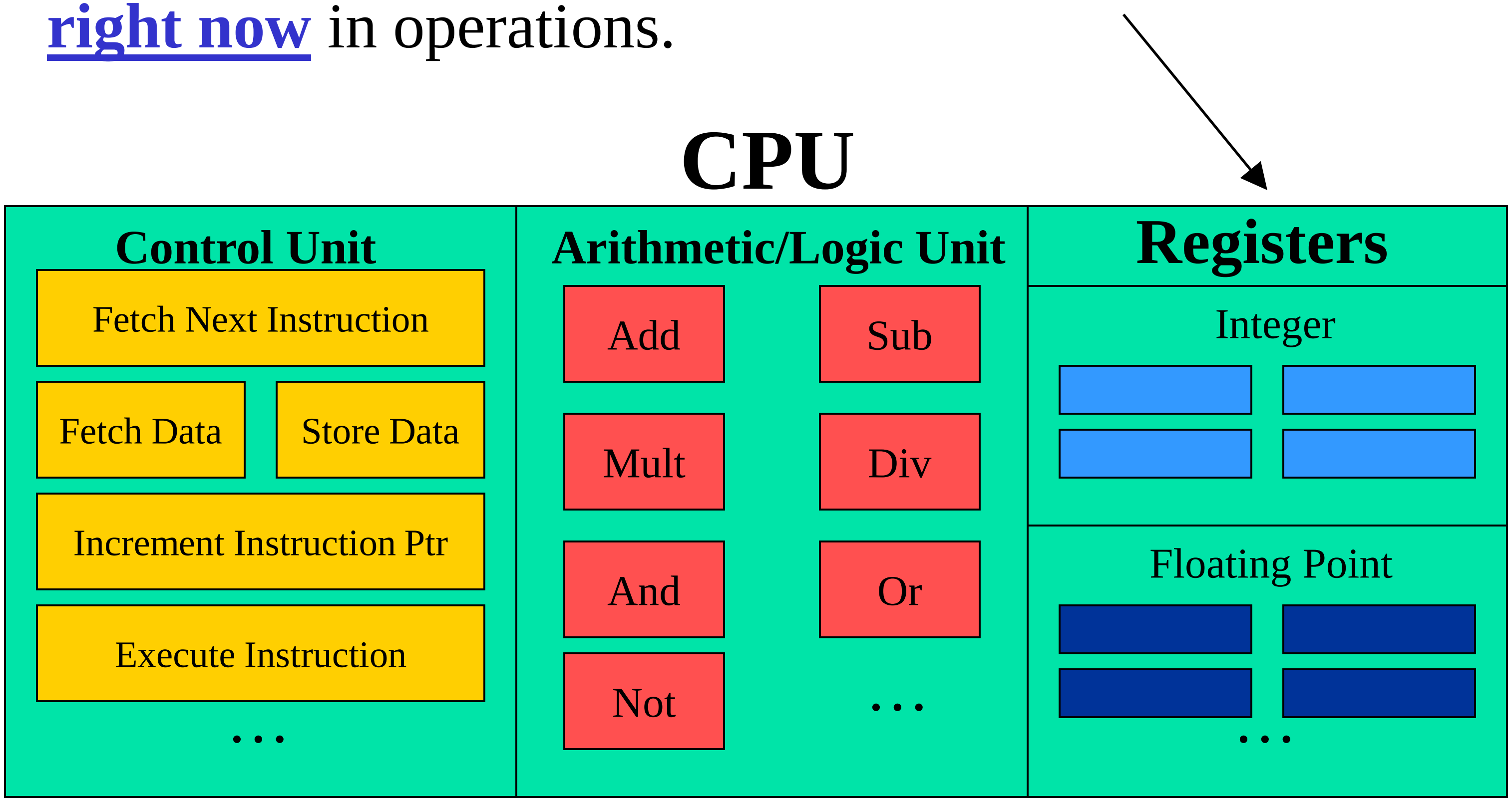
Registers



[25]

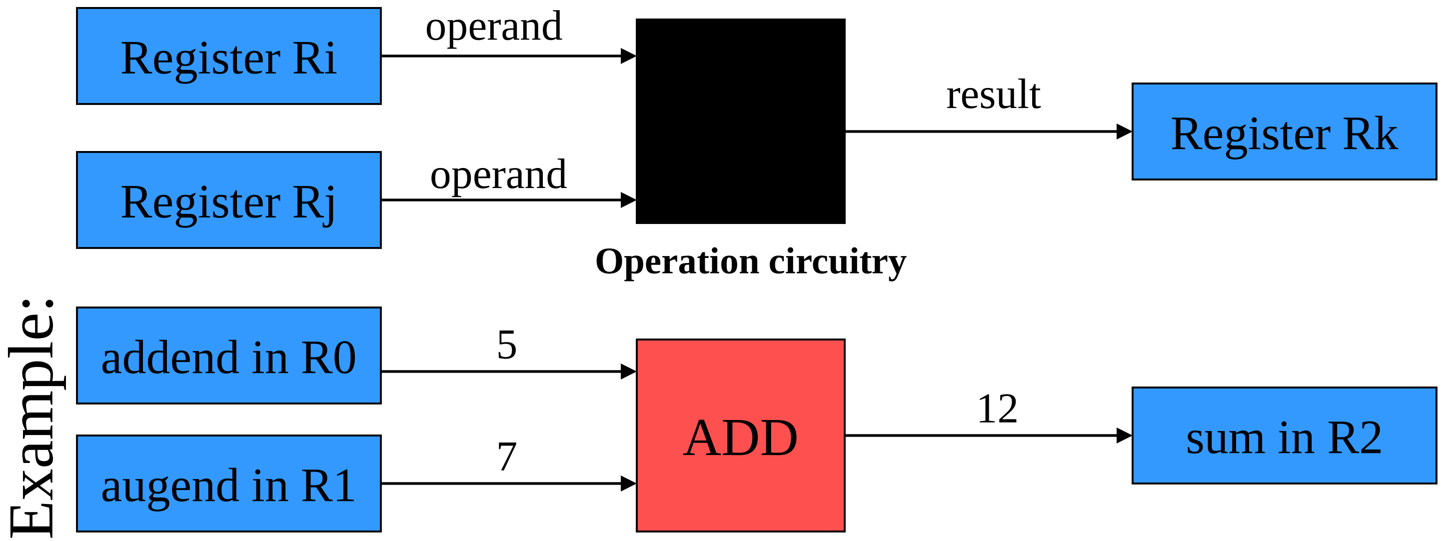
What Are Registers?

Registers are memory-like locations inside the Central Processing Unit that hold data that are **being used right now** in operations.



How Registers Are Used

- Every arithmetic or logical operation has one or more operands and one result.
- Operands are contained in source registers.
- A "black box" of circuits performs the operation.
- The result goes into a destination register.



How Many Registers?

Typically, a CPU has less than 8 KB (8192 bytes) of registers, usually split into registers for holding **integer** values and registers for holding **floating point** (real) values, plus a few special purpose registers.

Examples:

- **IBM POWER7** (found in IBM p-Series supercomputers): 226 64-bit integer registers and 348 128-bit merged vector/scalar registers (7376 bytes) [28]

- **Intel Sandy Bridge:** 16 64-bit general purpose registers, 8 64-bit floating point registers, 8 128-bit floating point vector registers, 16 256-bit floating point registers (832 bytes) [29]

- **Intel Itanium2**: 128 64-bit integer registers, 128 82-bit floating point registers (2304 bytes) [23]

Why So Few Registers?

Why so few registers?

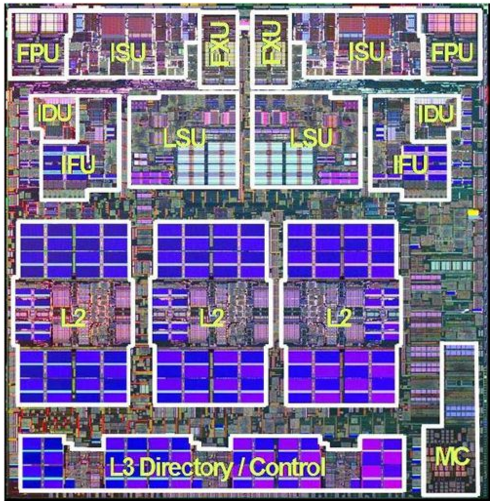
Because having more registers can be expensive, but doesn’t seem to help much:

“… [A]lthough for all applications, in average, the best size of [the] register file is 68 and above but in sizes near … half of this size performance penalty is lower that 5%.”

M. Alipour, M. E. Salehi, H. Shojaei Baghini, “Design Space Exploration to Find the Optimum Cache and Register File Size for Embedded Applications.” *Int'l Conf. Embedded Systems and Applications (ESA'11)*, 214-219.
http://arxiv.org/ftp/arxiv/papers/1205/1205.1871.pdf

In other words, you can add more registers, but your CPU will cost more, may draw more power, and your performance improvement will be modest.

Cache

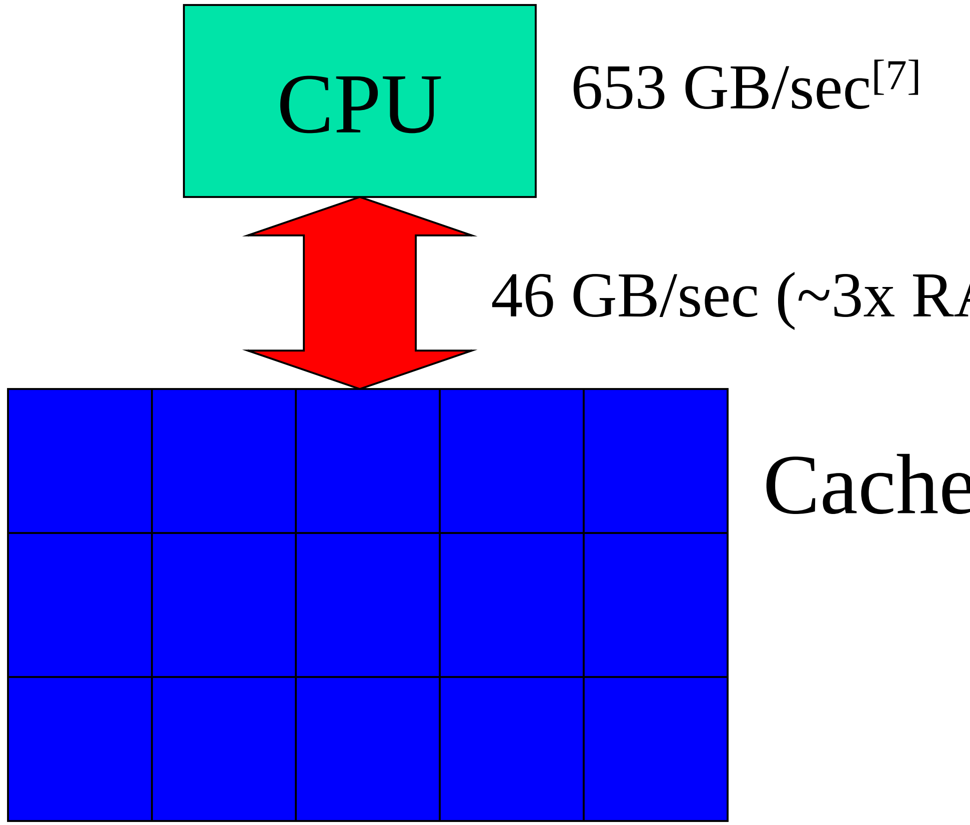

FPU
ISU
FXU
FXU
ISU
FPU
IDU
LSU
LSU
IDU
IFU
IFU
L2
L2
L2
L3 Directory / Control
MC

[4]

What is Cache?

- A special kind of memory where data reside that **are about to be used** or **have just been used**.
- Very fast => very expensive => very small (typically 100 to 10,000 times as expensive as RAM per byte)
- Data in cache can be loaded into or stored from registers at speeds comparable to the speed of performing computations.
- Data that are not in cache (but that are in Main Memory) take **much** longer to load or store.
- Cache is near the CPU: either inside the CPU or on the ***motherboard*** that the CPU sits on.

From Cache to the CPU



Typically, data move between cache and the CPU at speeds relatively near to that of the CPU performing calculations.

Multiple Levels of Cache

Most contemporary CPUs have more than one level of cache. For example:

- **Intel Sandy Bridge** [29]
  - Level 1 caches: 32 KB instruction, 32 KB data
  - Level 2 cache: 256 KB ***unified*** (instruction+data)
  - Level 3 cache: 20,480 KB, shared among all cores

- **IBM POWER7** [28]
  - Level 1 cache: 32 KB instruction, 32 KB data per core
  - Level 2 cache: 256 KB unified per core
  - Level 3 cache: 4096 KB unified per core

Why Multiple Levels of Cache?

The lower the level of cache:

- the faster the cache can transfer data to the CPU;
- the smaller that level of cache is **(faster => more expensive => smaller)**.

<u>Example</u>: IBM POWER7 latency to the CPU [28]

- L1 cache: 1 cycle = 0.29 ns for 3.5 GHz
- L2 cache: 8.5 cycles = 2.43 ns for 3.5 GHz (average)
- L3 cache: 23.5 cycles = 5.53 ns for 3.5 GHz (local to core)
- RAM: 346 cycles = 98.86 ns for 3.5 GHz (1066 MHz RAM)

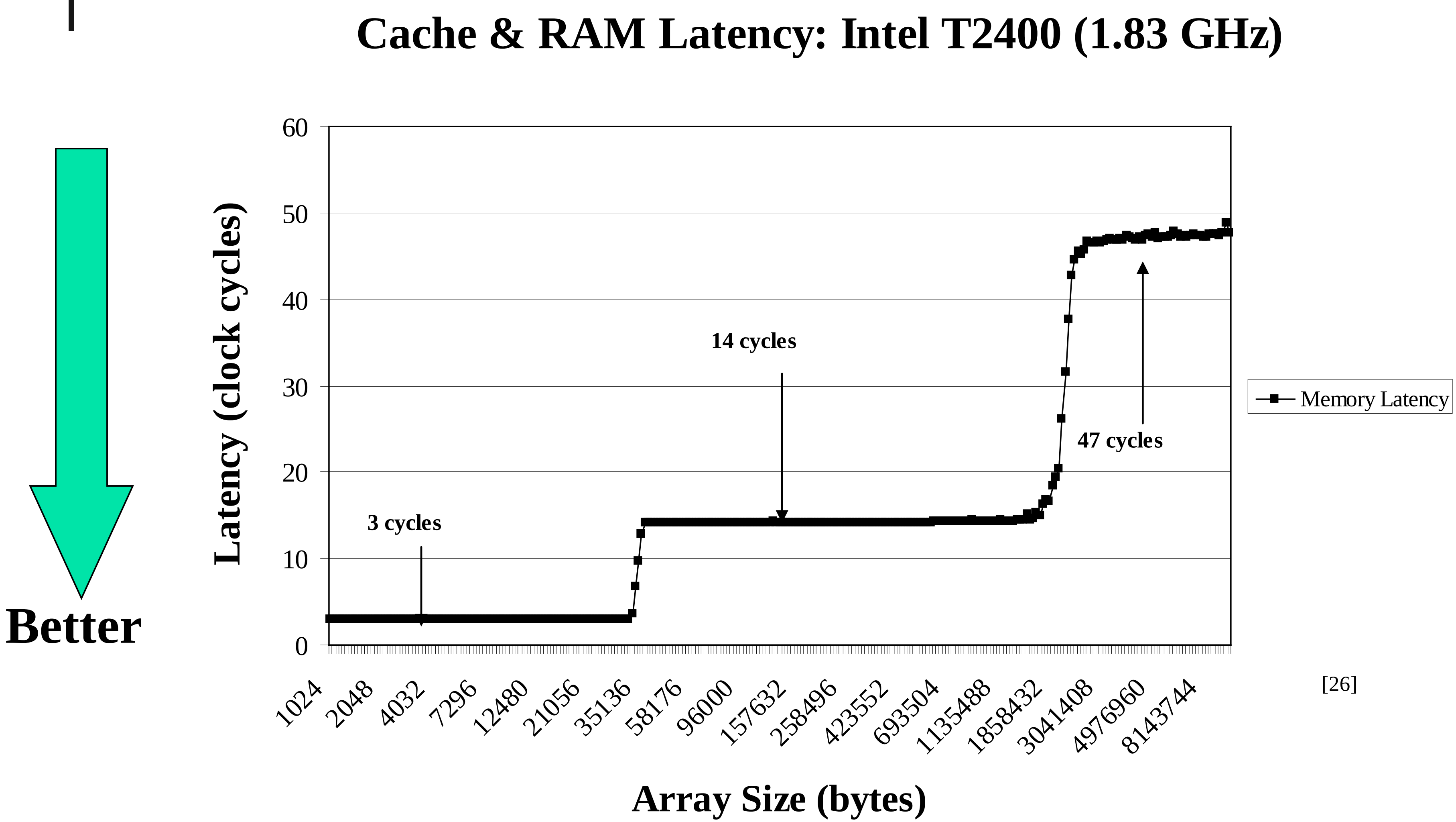
<u>Example</u>: Intel Itanium2 latency to the CPU [19]

- L1 cache: 1 cycle = 1.0 ns for 1.0 GHz
- L2 cache: 5 cycles = 5.0 ns for 1.0 GHz
- L3 cache: 12-15 cycles = 12 – 15 ns for 1.0 GHz

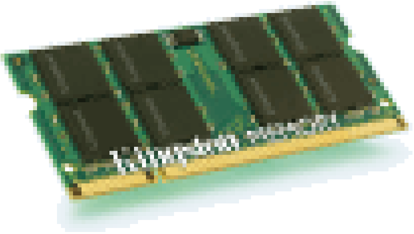
<u>Example</u>: Intel Sandy Bridge[29]

- L1 cache: 4 cycles = 2 ns @ 2.0 GHz = 32 calculations
- L2 cache: 12 cycles = 6 ns @ 2.0 GHz = 96 calculations
- RAM: 26-31 cycles = 13 – 15.5 ns @ 2.0 GHz = 200-248 calculations

Cache & RAM Latencies



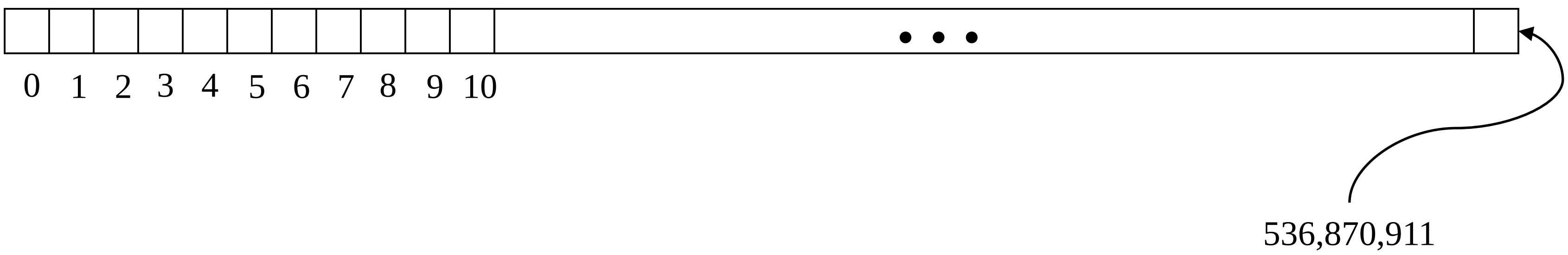
Main Memory


[13]

What is Main Memory?

- Where data reside for a program that is **currently running**
- Sometimes called ***RAM*** (Random Access Memory): you can load from or store into any main memory location at any time
- Sometimes called ***core*** (from magnetic "cores" that some memories used, many years ago)
- Much slower => much cheaper => much bigger

What Main Memory Looks Like

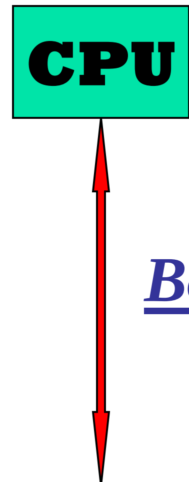


You can think of main memory as a big long 1D array of bytes.

The Relationship Between Main Memory & Cache

RAM is Slow

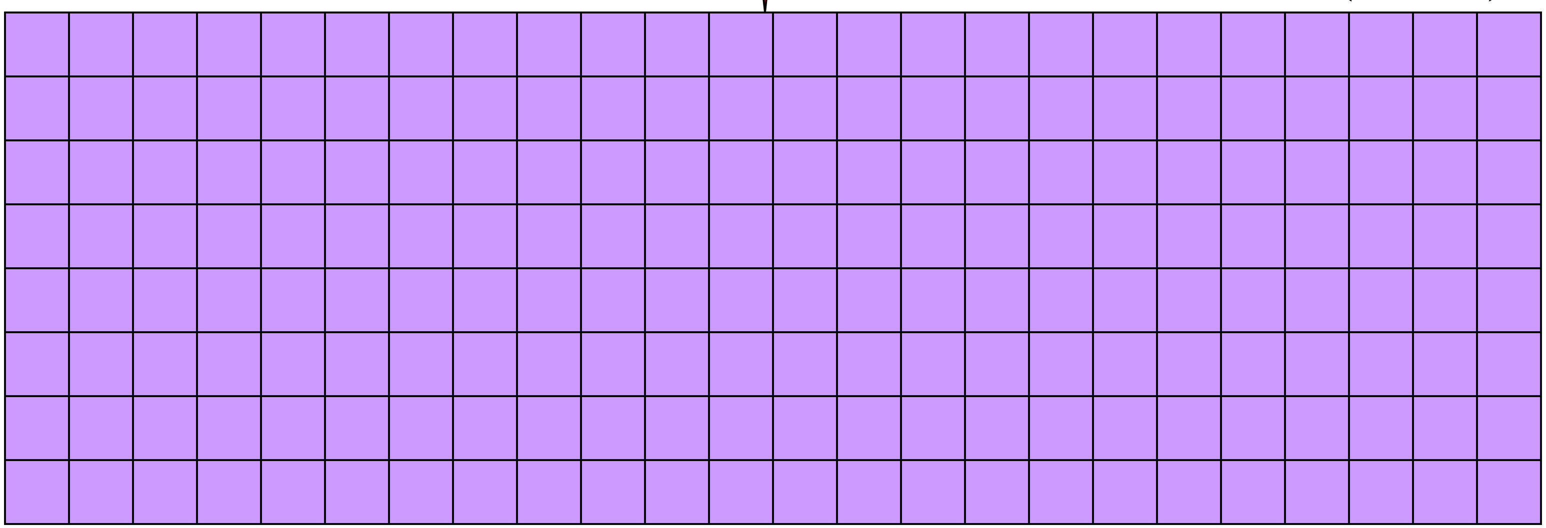
The speed of data transfer between Main Memory and the CPU is much slower than the speed of calculating, so the CPU spends most of its time waiting for data to come in or go out.



653 GB/sec

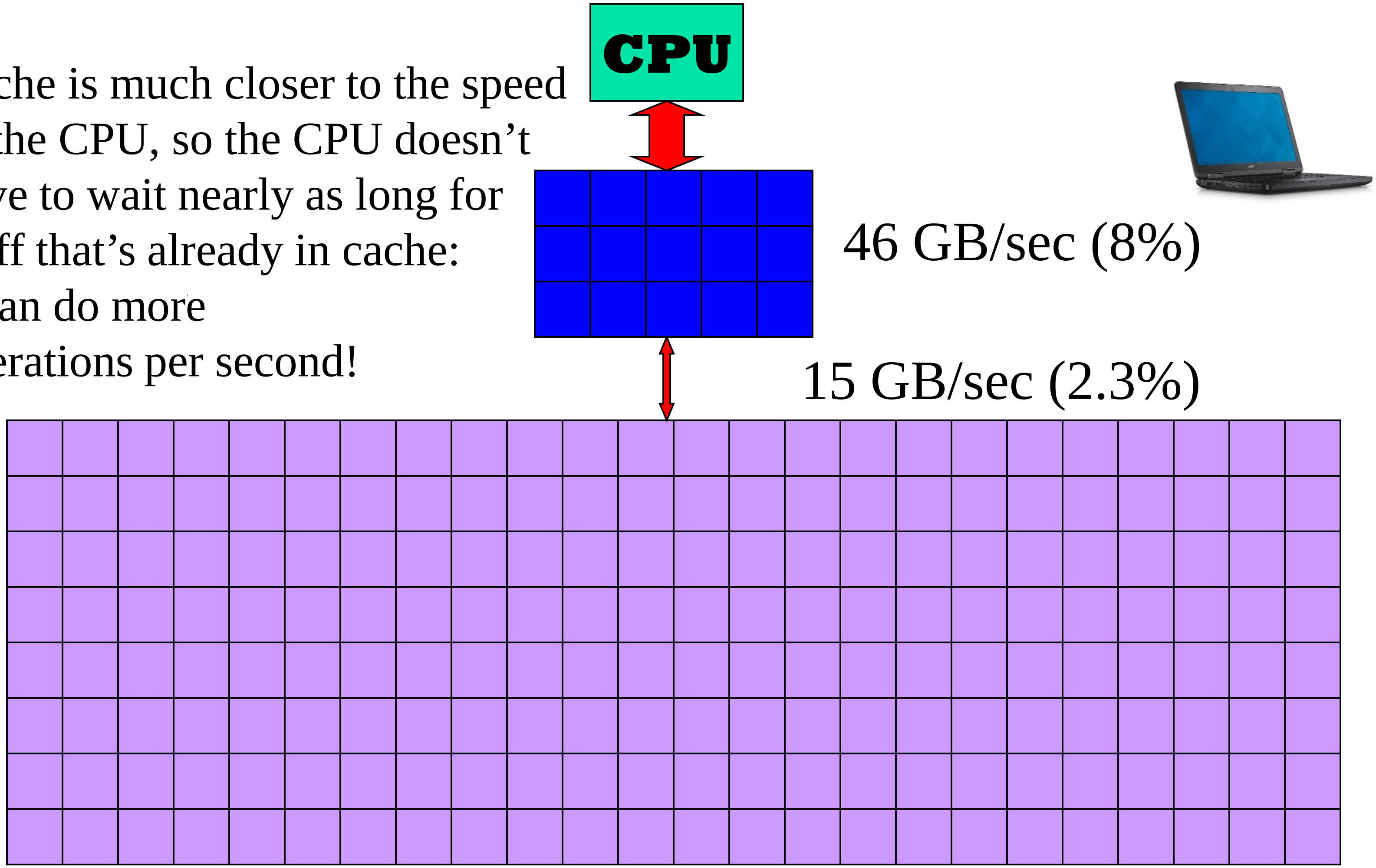
Bottleneck

15 GB/sec (2.3%)



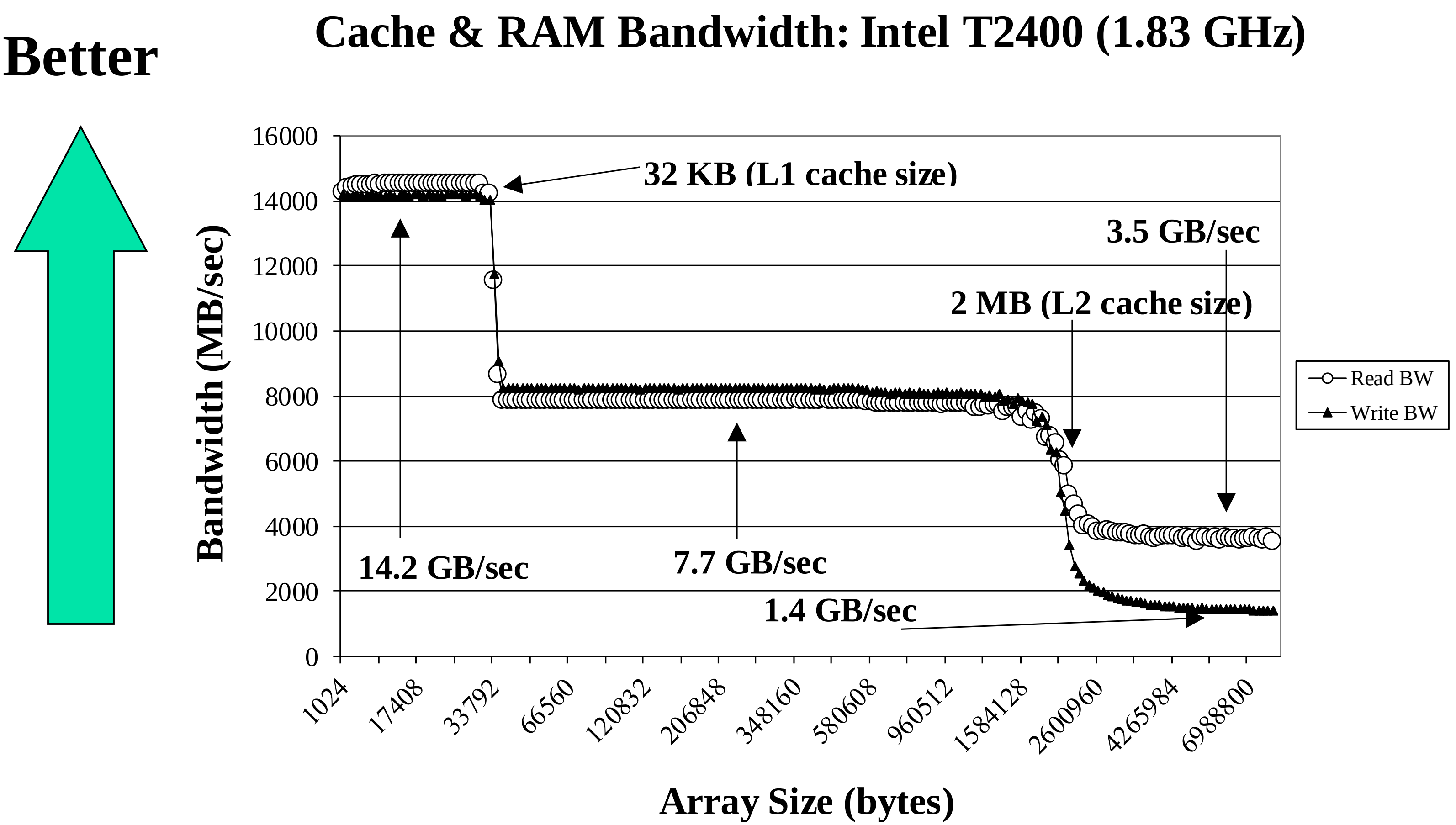
Why Have Cache?

Cache is much closer to the speed of the CPU, so the CPU doesn't have to wait nearly as long for stuff that's already in cache: it can do more operations per second!



Cache & RAM Bandwidths

Better



[26]

Cache Use Jargon

- ***Cache Hit***: the data that the CPU needs right now are **already in cache**.
- ***Cache Miss***: the data that the CPU needs right now are **not currently in cache**.

If all of your data are small enough to fit in cache, then when you run your program, you'll get almost all cache hits (except at the very beginning), which means that your performance could be excellent!

Sadly, this rarely happens in real life: most problems of scientific or engineering interest are bigger than just a few MB.

Cache Lines

- A ***cache line*** is a small, contiguous region in cache, corresponding to a contiguous region in RAM of the same size, that is loaded all at once.
- Typical size: 32 to 1024 bytes
- Examples
  - **Intel Sandy Bridge** [29]
    - L1 data cache: 64 bytes per line
    - L2 cache: 64 bytes per line

  - **POWER7** [28]
    - L1 instruction cache: 128 bytes per line
    - L1 data cache: 128 bytes per line
    - L2 cache: 128 bytes per line
    - L3 cache: 128bytes per line

How Cache Works

When you request data from a particular address in Main Memory, here's what happens:

1. The hardware checks whether the data for that address is already in cache. If so, it uses it.
2. Otherwise, it loads from Main Memory the entire cache line that contains the address.

For example, on a 2.0 GHz Sandy Bridge, a cache miss makes the program ***stall*** (wait) at least 26 cycles (13 nanoseconds) for the next cache line to load – time that could have been spent performing up to 208 calculations! [29]

If It’s in Cache, It’s Also in RAM

If a particular memory address is currently in cache, then it’s **<u>also</u>** in Main Memory (RAM).

That is, **<u>all</u>** of a program’s data are in Main Memory, but **<u>some</u>** are **<u>also</u>** in cache.

We’ll revisit this point shortly.

Mapping Cache Lines to RAM

Main memory typically maps into cache in one of three ways:

- Direct mapped (occasionally)
- Fully associative (very rare these days)
- Set associative (common)

DON'T PANIC!

Direct Mapped Cache

Direct Mapped Cache is a scheme in which each location in main memory corresponds to exactly one location in cache (but not the reverse, since cache is much smaller than main memory).

Typically, if a cache address is represented by **`c`** bits, and a main memory address is represented by **`m`** bits, then the cache location associated with main memory address **`A`** is **`MOD(A,2`**^{c}**`);`** that is, the lowest **`c`** bits of **`A`**.

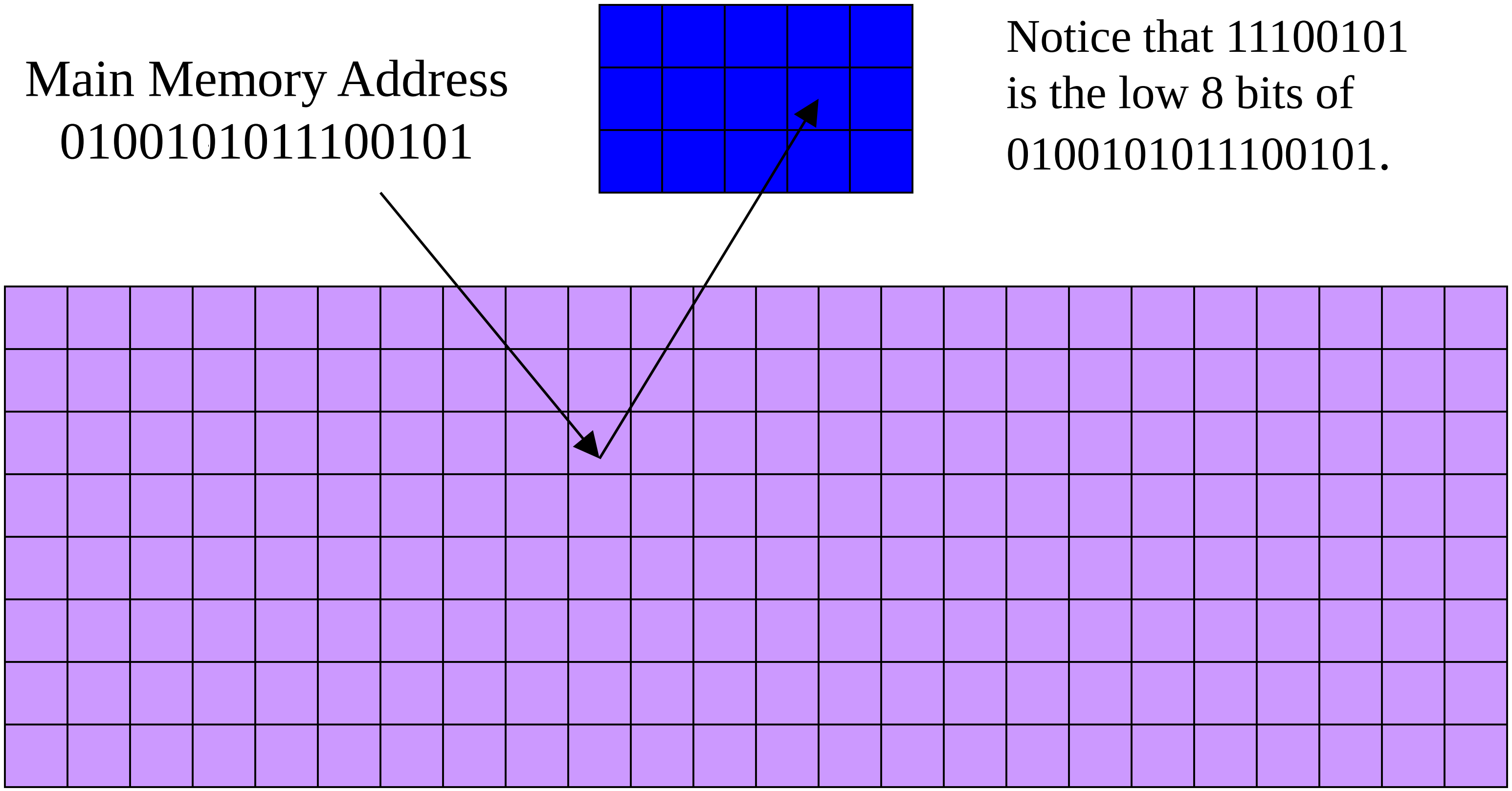
Example: POWER4 L1 instruction cache

Direct Mapped Cache Illustration

Must go into
cache address
11100101

Main Memory Address
0100101011100101

Notice that 11100101 is the low 8 bits of 0100101011100101.



Jargon: Cache Conflict

Suppose that the cache address 11100101 currently contains RAM address 0100101011100101.

But, we now need to load RAM address 1100101011100101, which maps to the same cache address as 0100101011100101.

This is called a ***cache conflict*** : the CPU needs a RAM location that maps to a cache line already in use.

In the case of direct mapped cache, every cache conflict leads to the new cache line clobbering the old cache line.

This can lead to serious performance problems.

Problem with Direct Mapped: F90

If you have two arrays that start in the same place relative to cache, then they might clobber each other all the time: no cache hits!

```
REAL,DIMENSION(multiple_of_cache_size) :: a, b, c
INTEGER :: index

DO index = 1, multiple_of_cache_size
    a(index) = b(index) + c(index)
END DO
```

In this example, **`a(index)`**, **`b(index)`** and **`c(index)`** all map to the same cache line, so loading **`c(index)`** clobbers **`b(index)`** – **no cache reuse!**

Problem with Direct Mapped: C

If you have two arrays that start in the same place relative to cache, then they might clobber each other all the time: no cache hits!

```
float a[multiple_of_cache_size],
      b[multiple_of_cache_size,
      c[multiple_of_cache_size];
int index;

for (index = 0; index < multiple_of_cache_size;
     index++)
    { a[index] = b[index] + c[index]; }
```

In this example, **`a[index]`**, **`b[index]`** and **`c[index]`** all map to the same cache line, so loading **`c[index]`** clobbers **`b[index]`** – **no cache reuse!**

Fully Associative Cache

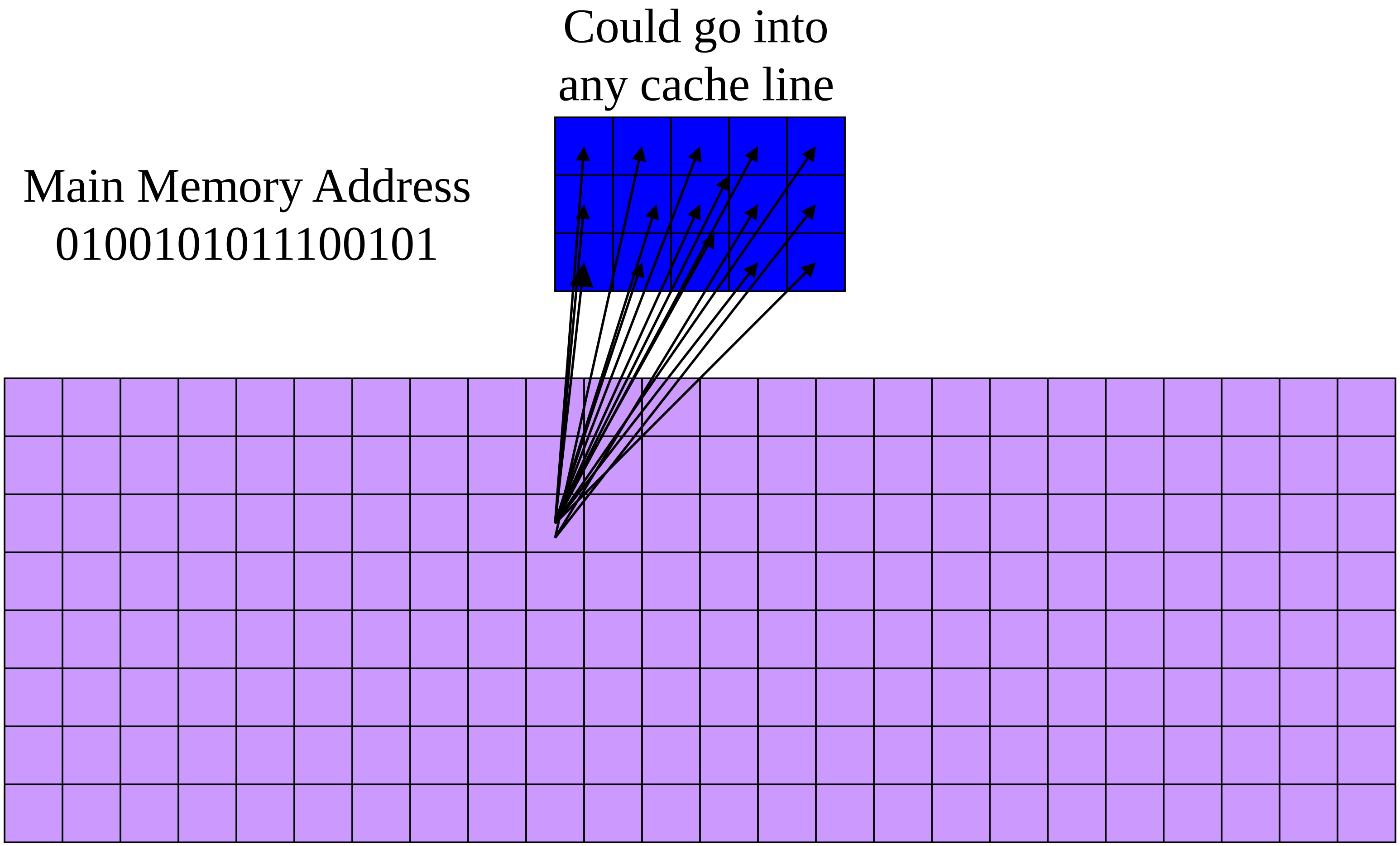
Fully Associative Cache can put **any** line of main memory into **any** cache line.

Typically, the cache management system will put the newly loaded data into the ***Least Recently Used*** cache line, though other strategies are possible (e.g., ***Random***, ***First In First Out***, ***Round Robin***, ***Least Recently Modified***).

So, this can solve, or at least reduce, the cache conflict problem.

But, fully associative cache tends to be **expensive**, so it's pretty rare: you need $N_{\text{cache}} \cdot N_{\text{RAM}}$ connections!

Fully Associative Illustration



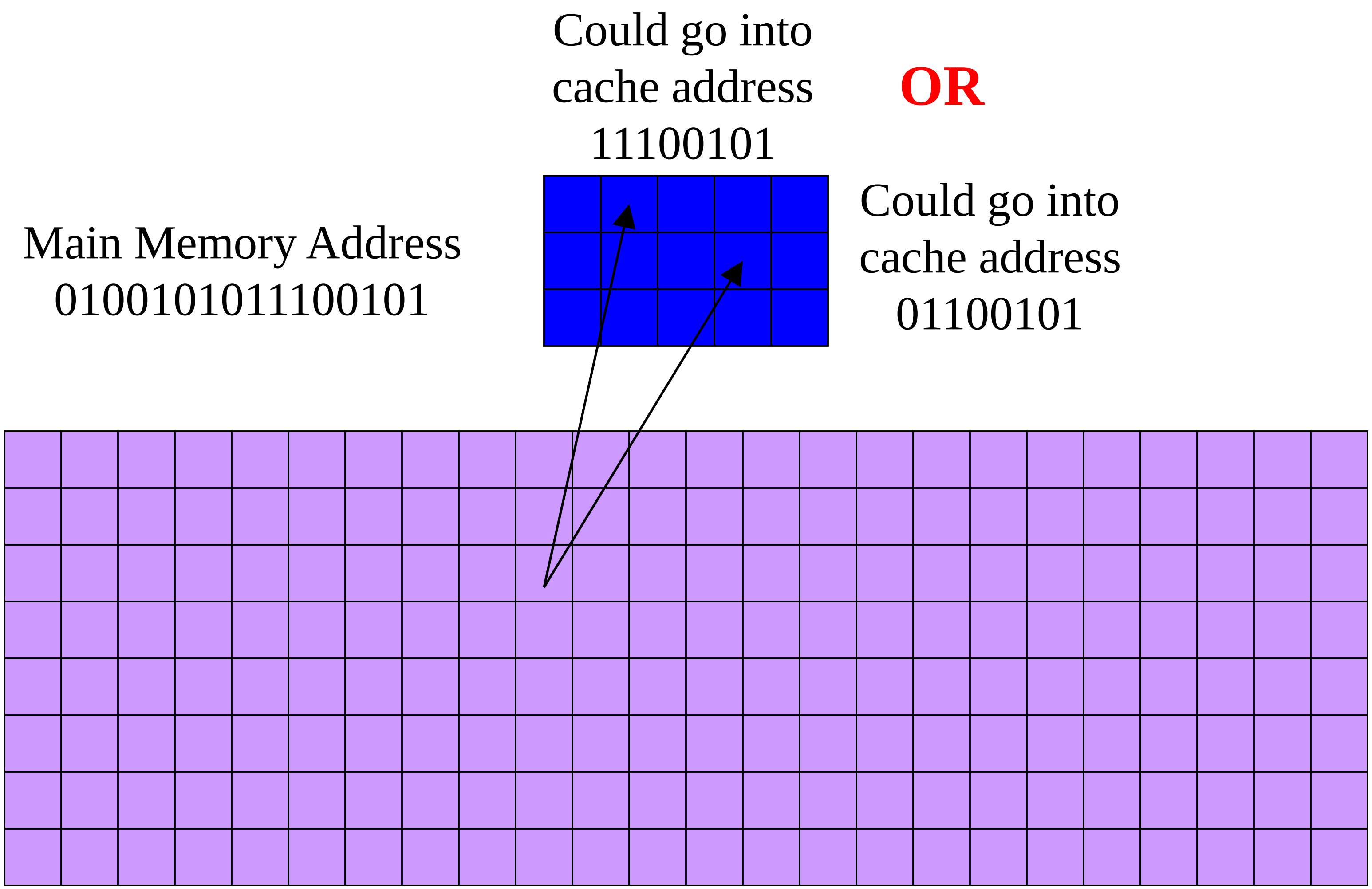
Set Associative Cache

Set Associative Cache is a compromise between direct mapped and fully associative. A line in main memory can map to any of a **fixed number** of cache lines.

For example, ***2-way Set Associative Cache*** can map each main memory line to either of 2 cache lines (e.g., to the Least Recently Used), 3-way maps to any of 3 cache lines, 4-way to 4 lines, and so on.

Set Associative cache is **cheaper** than fully associative – you need $K \cdot N_{\text{RAM}}$ connections – but **more robust** than direct mapped.

2-Way Set Associative Illustration



Cache Associativity Examples

- **Sandy Bridge** [29]
  - L1 data cache: 8-way set associative
  - L2 cache: 8-way set associative
  - L3 cache: varies with cache size
- **POWER4** [12]
  - L1 instruction cache: direct mapped
  - L1 data cache: 2-way set associative
  - L2 cache: 8-way set associative
  - L3 cache: 8-way set associative
- **POWER7** [28]
  - L1 instruction cache: 4-way set associative
  - L1 data cache: 8-way set associative
  - L2 cache: 8-way set associative
  - L3 cache: 8-way set associative

If It's in Cache, It's Also in RAM

As we saw earlier:

If a particular memory address is currently in cache, then it's **also** in Main Memory (RAM).

That is, **all** of a program's data are in Main Memory, but **some** are **also** in cache.

Changing a Value That's in Cache

Suppose that you have in cache a particular line of main memory (RAM).

If you don't change the contents of any of that line's bytes while it's in cache, then when it gets clobbered by another main memory line coming into cache, there's no loss of information.

But, if you change the contents of any byte while it's in cache, then you need to store it back out to main memory before clobbering it.

Cache Store Strategies

Typically, there are two possible cache store strategies:

- ***Write-through***: every single time that a value in cache is changed, that value is also stored back into main memory (RAM).
- ***Write-back***: every single time that a value in cache is changed, the cache line containing that cache location gets marked as ***dirty***. When a cache line gets clobbered, then if it has been marked as dirty, then it is stored back into main memory (RAM). [14]

Cache Store Examples

- **Intel Sandy Bridge** [29]
  - L1 cache: write-back

- **Pentium D** [26]
  - L1 cache: write-through

The Importance of Being Local

[15]

More Data Than Cache

Let's say that you have 1000 times more data than cache. Then won't most of your data be outside the cache?

YES!

Okay, so how does cache help?

Improving Your Cache Hit Rate

Many scientific codes use a lot more data than can fit in cache all at once.

Therefore, you need to ensure a high cache hit rate even though you've got much more data than cache.

So, how can you improve your cache hit rate?

Use the same solution as in Real Estate:

Location, Location, Location!

Data Locality

Data locality is the principle that, if you use data in a particular memory address, then **very soon** you'll use either **the same address** or **a nearby address**.

- ***Temporal locality***: if you're using address **A** now, then you'll probably soon use address **A** again.
- ***Spatial locality***: if you're using address **A** now, then you'll probably soon use addresses between **A-k** and **A+k**, where **k** is small.

Note that this principle works well for sufficiently small values of "soon."

Cache is designed to exploit locality, which is why a cache miss causes a whole line to be loaded.

Data Locality Is Empirical: C

Data locality has been observed empirically in many, many programs.

```
void ordered_fill (float* array, int array_length)
{ /* ordered_fill */
  int index;

  for (index = 0; index < array_length; index++) {
    array[index] = index;
  } /* for index */
} /* ordered_fill */
```

Data Locality Is Empirical: F90

Data locality has been observed empirically in many, many programs.

```
SUBROUTINE ordered_fill (array, array_length)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: array_length
  REAL,DIMENSION(array_length),INTENT(OUT) :: array
  INTEGER :: index

  DO index = 1, array_length
    array(index) = index
  END DO
END SUBROUTINE ordered_fill
```

No Locality Example: C

In principle, you could write a program that exhibited **<u>absolutely no data locality at all</u>**:

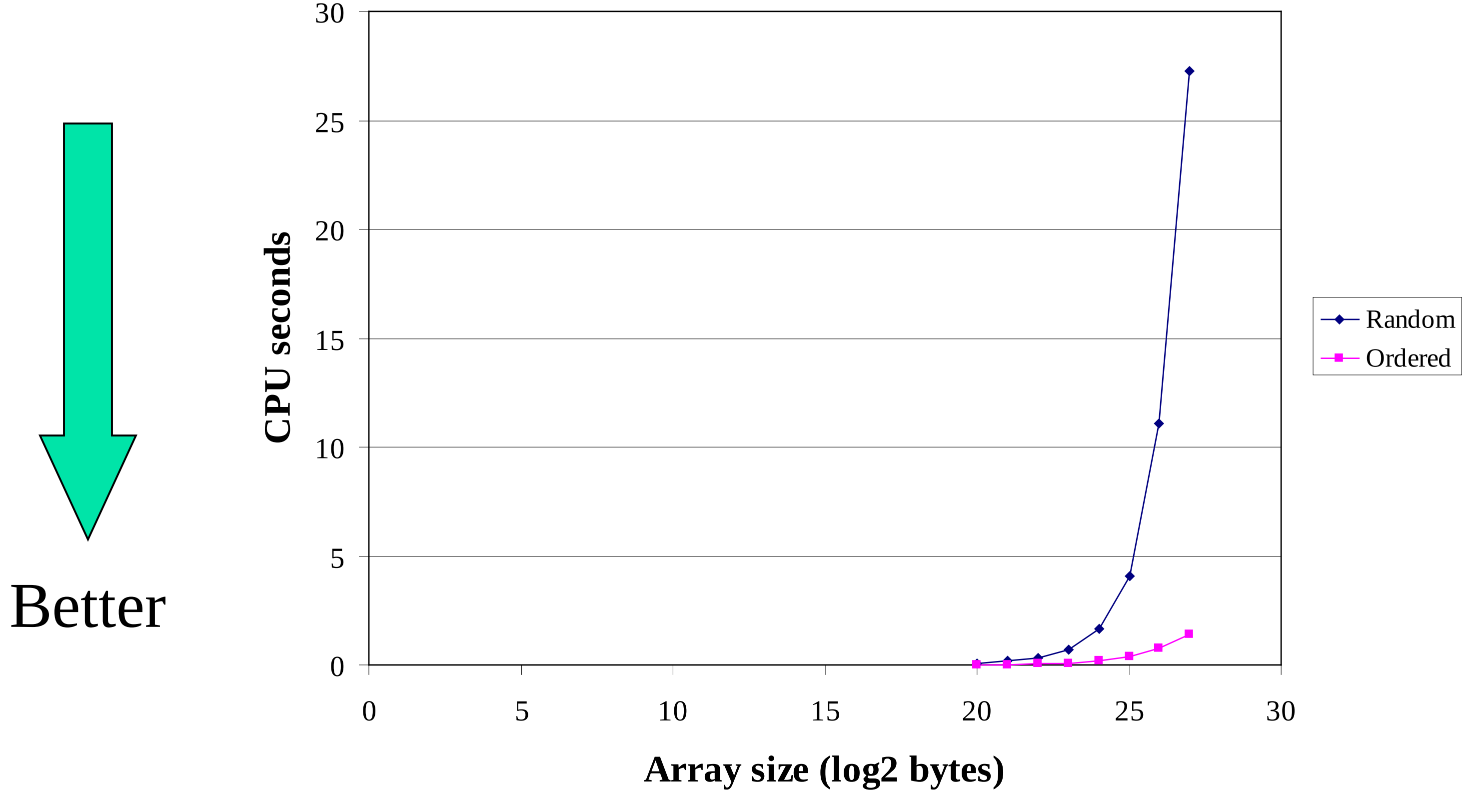
```
void random_fill (float* array,
                  int* random_permutation_index,
                  int array_length)
{ /* random_fill */
  int index;

  for (index = 0; index < array_length; index++) {
    array[random_permutation_index[index]] = index;
  } /* for index */
} /* random_fill */
```

No Locality Example: F90

In principle, you could write a program that exhibited **<u>absolutely no data locality at all</u>**:

```
SUBROUTINE random_fill (array,
              random_permutation_index, array_length)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: array_length
  INTEGER,DIMENSION(array_length),INTENT(IN) :: &
&   random_permutation_index
  REAL,DIMENSION(array_length),INTENT(OUT) :: array
  INTEGER :: index

  DO index = 1, array_length
    array(random_permutation_index(index)) = index
  END DO
END SUBROUTINE random_fill
```

Permuted vs. Ordered



In a simple array fill, locality provides a factor of 8 to 20 speedup over a randomly ordered fill on a Pentium4.

Exploiting Data Locality

If you know that your code is capable of operating with a decent amount of data locality, then you can get speedup by focusing your energy on improving the locality of the code's behavior.

This will substantially increase your **cache reuse**.

A Sample Application

Matrix-Matrix Multiply

Let A, B and C be matrices of sizes
$nr \times nc$, $nr \times nk$ and $nk \times nc$, respectively:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,nc} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,nc} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,nc} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{nr,1} & a_{nr,2} & a_{nr,3} & \cdots & a_{nr,nc} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,nk} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,nk} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,nk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{nr,1} & b_{nr,2} & b_{nr,3} & \cdots & b_{nr,nk} \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & \cdots & c_{1,nc} \\ c_{2,1} & c_{2,2} & c_{2,3} & \cdots & c_{2,nc} \\ c_{3,1} & c_{3,2} & c_{3,3} & \cdots & c_{3,nc} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{nk,1} & c_{nk,2} & c_{nk,3} & \cdots & c_{nk,nc} \end{bmatrix}$$

The definition of A = B • C is

$$a_{r,c} = \sum_{k=1}^{nk} b_{r,k} \cdot c_{k,c} = b_{r,1} \cdot c_{1,c} + b_{r,2} \cdot c_{2,c} + b_{r,3} \cdot c_{3,c} + \ldots + b_{r,nk} \cdot c_{nk,c}$$

for $r \in \{1, nr\}$, $c \in \{1, nc\}$.

Matrix Multiply w/Initialization

```
SUBROUTINE matrix_matrix_mult_by_init (dst, src1, src2, &
 &                                       nr, nc, nq)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2

  INTEGER :: r, c, q

  DO c = 1, nc
    DO r = 1, nr
      dst(r,c) = 0.0
      DO q = 1, nq
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO !! q
    END DO !! r
  END DO !! c
END SUBROUTINE matrix_matrix_mult_by_init
```

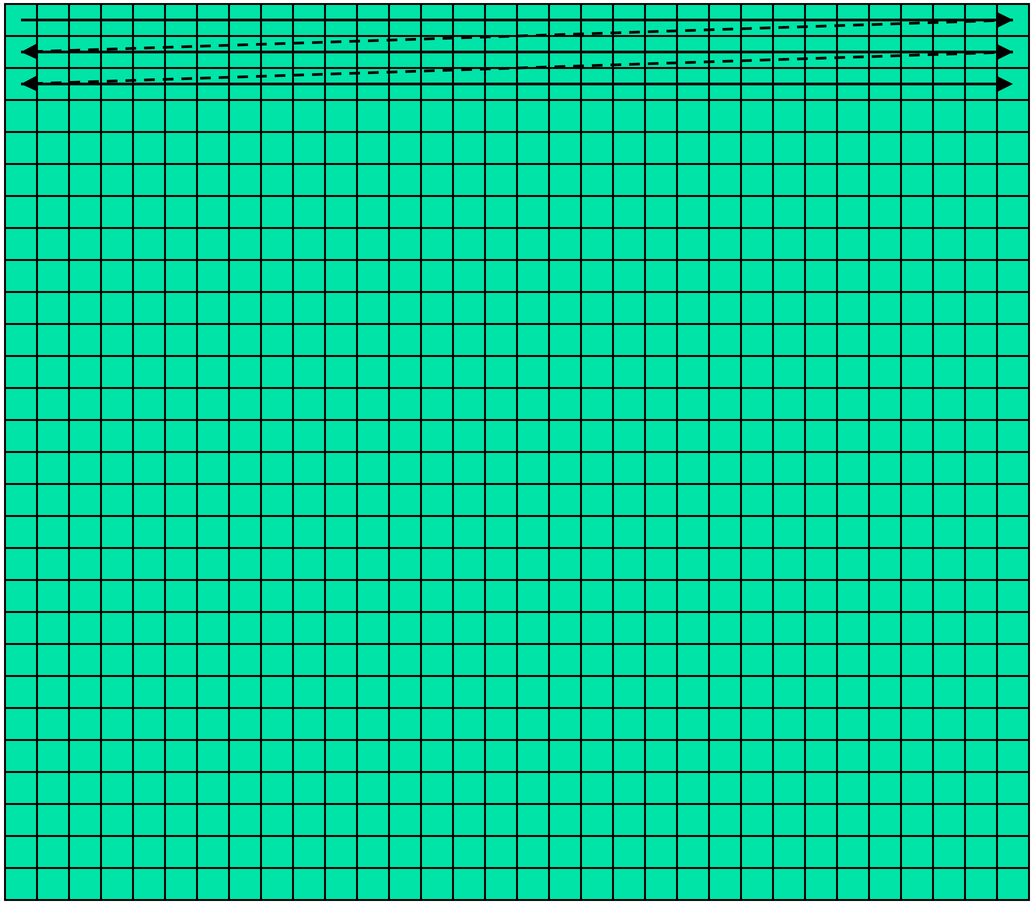
Matrix Multiply w/Initialization

```
void matrix_matrix_mult_by_init (
         float** dst, float** src1, float** src2,
         int nr, int nc, int nq)
{ /* matrix_matrix_mult_by_init */
  int r, c, q;

  for (r = 0; r < nr; r++) {
    for (c = 0; c < nc; c++) {
      dst[r][c] = 0.0;
      for (q = 0; q < nq; q++) {
        dst[r][c] = dst[r][c] + src1[r][q] * src2[q][c];
      } /* for q */
    } /* for c */
  } /* for r */
} /* matrix_matrix_mult_by_init */
```

Matrix Multiply Via Intrinsic

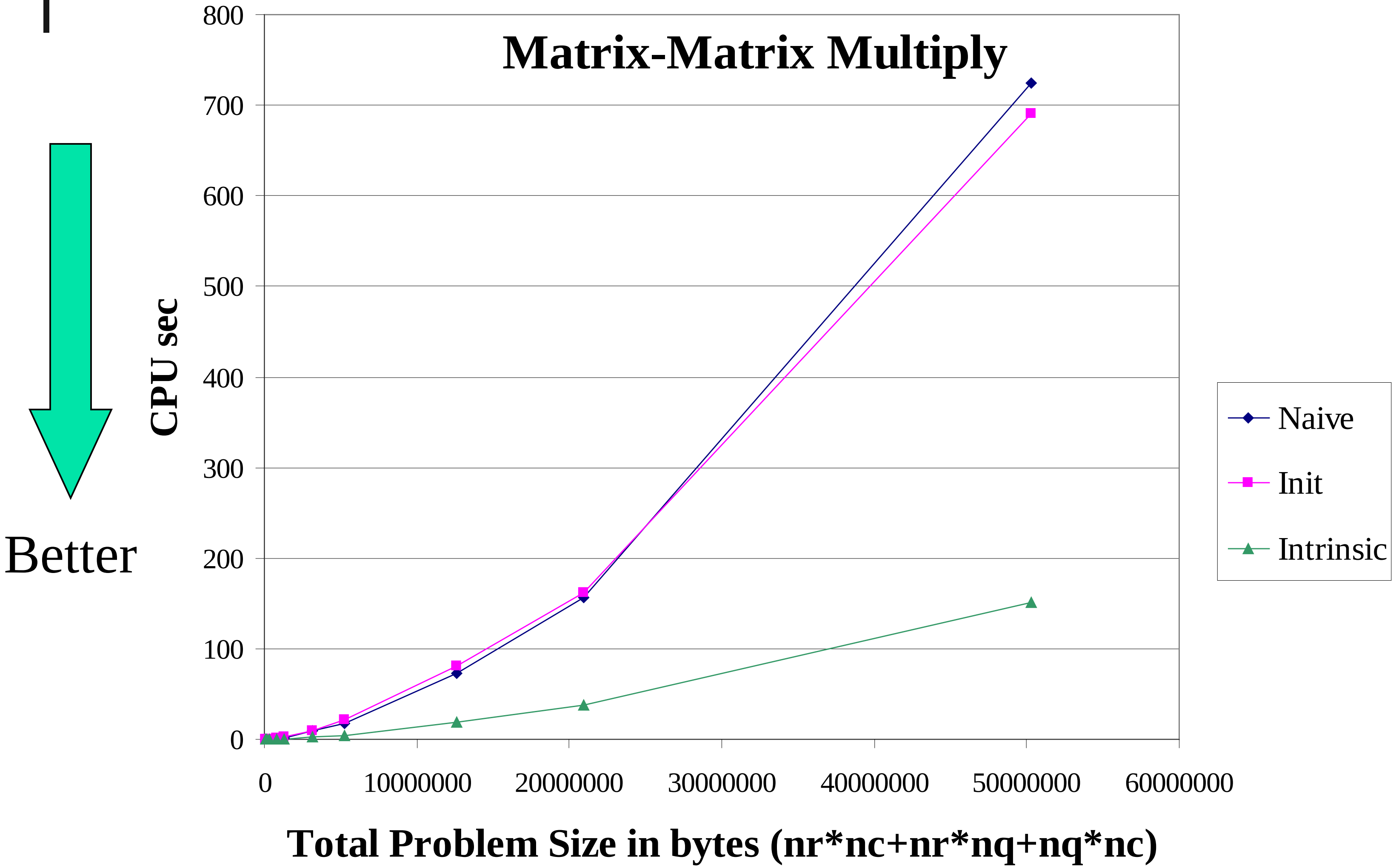
```
SUBROUTINE matrix_matrix_mult_by_intrinsic ( &
 &           dst, src1, src2, nr, nc, nq)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2

  dst = MATMUL(src1, src2)
END SUBROUTINE matrix_matrix_mult_by_intrinsic
```

Matrix Multiply Behavior

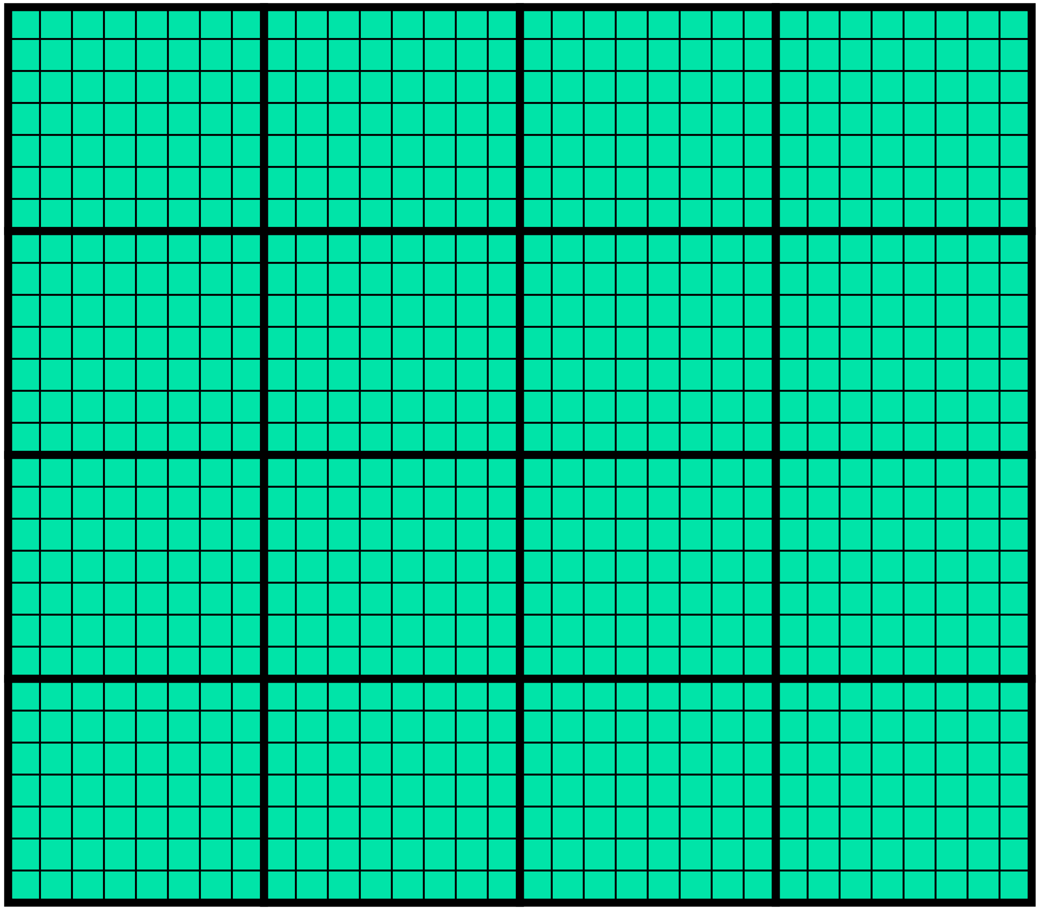


If the matrix is big, then each sweep of a row will clobber nearby values in cache.

Performance of Matrix Multiply



Tiling



Tiling

- ***Tile***: a small rectangular subdomain of a problem domain. Sometimes called a ***block*** or a ***chunk***.
- ***Tiling***: breaking the domain into tiles.
- Tiling strategy: operate on each tile to completion, then move to the next tile.
- Tile size can be set at runtime, according to what's best for the machine that you're running on.

Tiling Code: F90

```
SUBROUTINE matrix_matrix_mult_by_tiling (dst, src1, src2, nr, nc, nq, &
 &           rtilesize, ctilesize, qtilesize)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2
  INTEGER,INTENT(IN) :: rtilesize, ctilesize, qtilesize

  INTEGER :: rstart, rend, cstart, cend, qstart, qend

  DO cstart = 1, nc, ctilesize
    cend = cstart + ctilesize - 1
    IF (cend > nc) cend = nc
    DO rstart = 1, nr, rtilesize
      rend = rstart + rtilesize - 1
      IF (rend > nr) rend = nr
      DO qstart = 1, nq, qtilesize
        qend = qstart + qtilesize - 1
        IF (qend > nq) qend = nq
        CALL matrix_matrix_mult_tile(dst, src1, src2, nr, nc, nq, &
 &                                  rstart, rend, cstart, cend, qstart, qend)
      END DO !! qstart
    END DO !! rstart
  END DO !! cstart
END SUBROUTINE matrix_matrix_mult_by_tiling
```

Tiling Code: C

```
void matrix_matrix_mult_by_tiling (
         float** dst, float** src1, float** src2,
         int nr, int nc, int nq,
         int rtilesize, int ctilesize, int qtilesize)
{ /* matrix_matrix_mult_by_tiling */
  int rstart, rend, cstart, cend, qstart, qend;

  for (rstart = 0; rstart < nr; rstart += rtilesize) {
    rend = rstart + rtilesize – 1;
    if (rend >= nr) rend = nr - 1;
    for (cstart = 0; cstart < nc; cstart += ctilesize) {
      cend = cstart + ctilesize – 1;
      if (cend >= nc) cend = nc - 1;
      for (qstart = 0; qstart < nq; qstart += qtilesize) {
        qend = qstart + qtilesize – 1;
        if (qend >= nq) qend = nq - 1;
        matrix_matrix_mult_tile(dst, src1, src2, nr, nc, nq,
                                rstart, rend, cstart, cend, qstart, qend);
      } /* for qstart */
    } /* for cstart */
  } /* for rstart */
} /* matrix_matrix_mult_by_tiling */
```

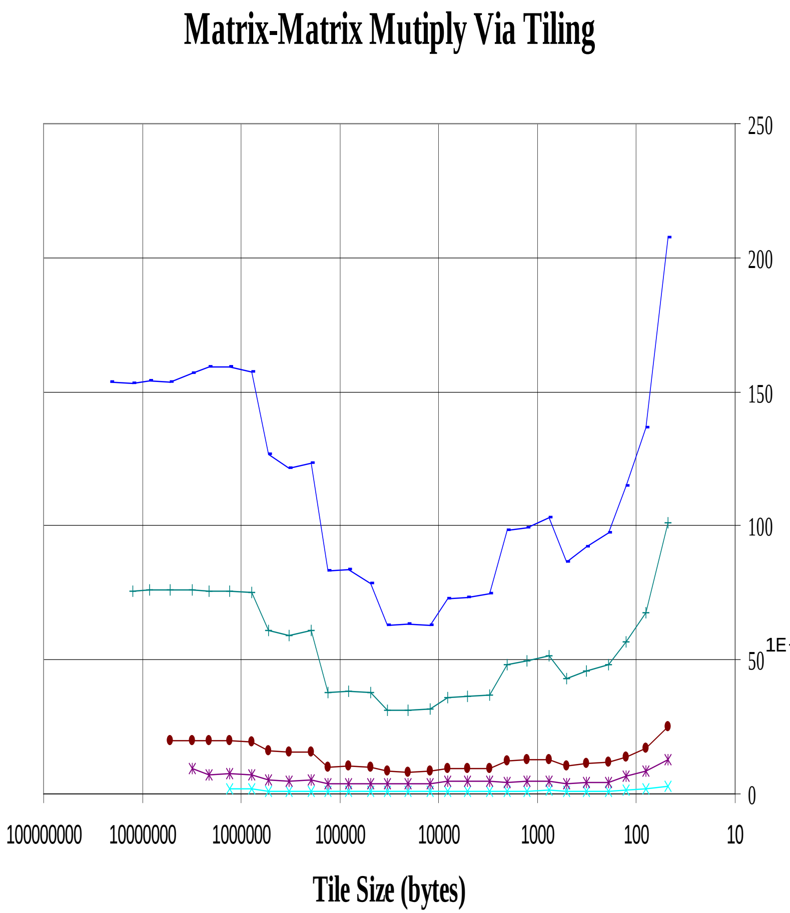
Multiplying Within a Tile: F90

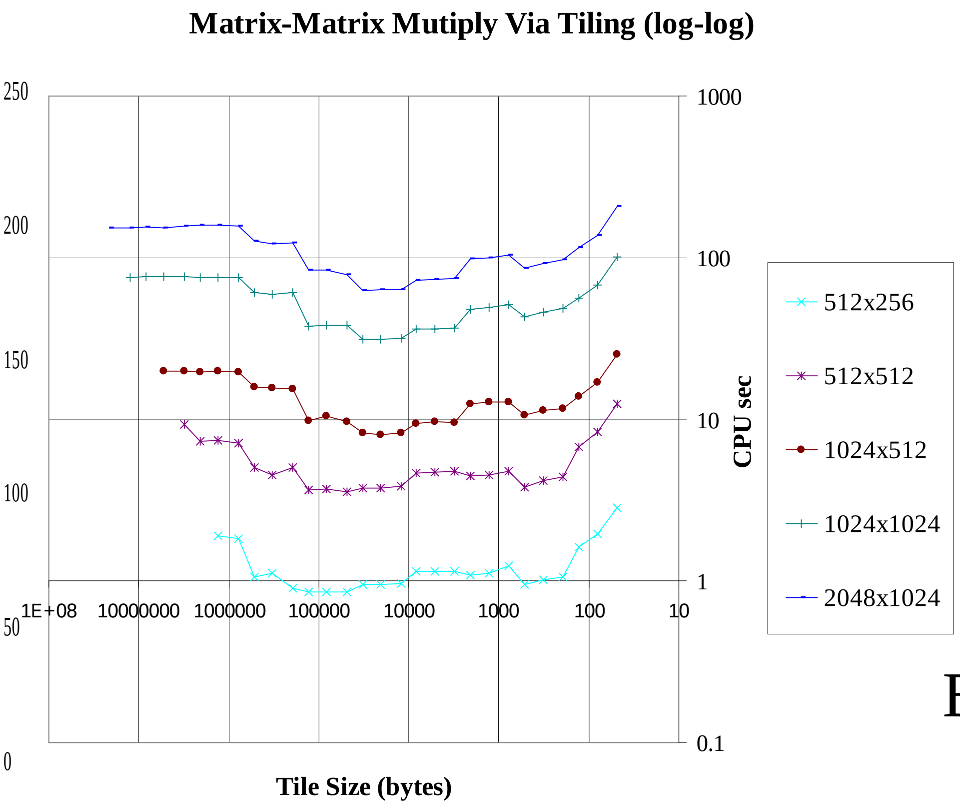
```
SUBROUTINE matrix_matrix_mult_tile (dst, src1, src2, nr, nc, nq, &
 &              rstart, rend, cstart, cend, qstart, qend)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2
  INTEGER,INTENT(IN) :: rstart, rend, cstart, cend, qstart, qend

  INTEGER :: r, c, q

  DO c = cstart, cend
    DO r = rstart, rend
      IF (qstart == 1) dst(r,c) = 0.0
      DO q = qstart, qend
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO !! q
    END DO !! r
  END DO !! c
END SUBROUTINE matrix_matrix_mult_tile
```

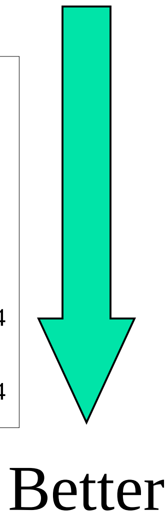
Multiplying Within a Tile: C

```
void matrix_matrix_mult_tile (
          float** dst, float** src1, float** src2,
          int nr, int nc, int nq,
          int rstart, int rend, int cstart, int cend,
          int qstart, int qend)
{ /* matrix_matrix_mult_tile */
  int r, c, q;

  for (r = rstart; r <= rend; r++) {
    for (c = cstart; c <= cend; c++) {
      if (qstart == 0) dst[r][c] = 0.0;
      for (q = qstart; q <= qend; q++) {
        dst[r][c] = dst[r][c] + src1[r][q] * src2[q][c];
      } /* for q */
    } /* for c */
  } /* for r */
} /* matrix_matrix_mult_tile */
```

Performance with Tiling







The Advantages of Tiling

- It allows your code to **exploit data locality** better, to get much more cache reuse: your code runs faster!
- It's a relatively **modest amount of extra coding** (typically a few wrapper functions and some changes to loop bounds).
- **If you don't need** tiling – because of the hardware, the compiler or the problem size – then you can **turn it off by simply** setting the tile size equal to the problem size.

Will Tiling Always Work?

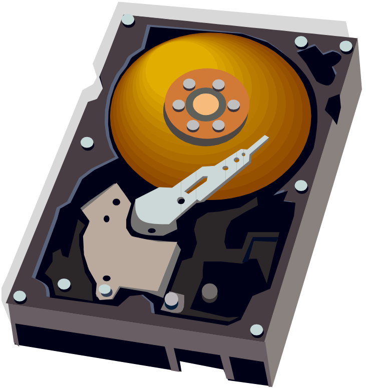
Tiling **<u>WON'T</u>** always work. Why?

Well, tiling works well when:

- the order in which calculations occur doesn't matter much, AND
- there are lots and lots of calculations to do for each memory movement.

If either condition is absent, then tiling won't help.

Hard Disk



Why Is Hard Disk Slow?

Your hard disk is **much much** slower than main memory (factor of 1000+). **Why?**

Well, accessing data on the hard disk involves physically moving:

- the disk platter
- the read/write head

In other words, hard disk is slow because **objects** move much slower than **electrons**: Newtonian speeds are much slower than Einsteinian speeds.

I/O Strategies

Read and write the absolute minimum amount.

- Don’t reread the same data if you can keep it in memory.
- Write binary instead of characters.
- Use optimized I/O libraries like NetCDF [17] and HDF [18].

Avoid Redundant I/O: C

An actual piece of code seen at OU:

```
for (thing = 0; thing < number_of_things; thing++) {
  for (timestep = 0; timestep < number_of_timesteps; timestep++) {
    read_file(filename[timestep]);
    do_stuff(thing, timestep);
  } /* for timestep */
} /* for thing */
```

Improved version:

```
for (timestep = 0; timestep < number_of_timesteps; timestep++) {
  read_file(filename[timestep]);
  for (thing = 0; thing < number_of_things; thing++) {
    do_stuff(thing, timestep);
  } /* for thing */
} /* for timestep */
```

Savings (in real life): **factor of 500!**

Avoid Redundant I/O: F90

An actual piece of code seen at OU:

```
DO thing = 1, number_of_things
  DO timestep = 1, number_of_timesteps
    CALL read_file(filename(timestep))
    CALL do_stuff(thing, timestep)
  END DO !! timestep
END DO !! thing
```

Improved version:

```
DO timestep = 1, number_of_timesteps
  CALL read_file(filename(timestep))
  DO thing = 1, number_of_things
    CALL do_stuff(thing, timestep)
  END DO !! thing
END DO !! timestep
```

Savings (in real life): **factor of 500!**

Write Binary, Not ASCII

When you write binary data to a file, you're writing (typically) 4 bytes per value.

When you write ASCII (character) data, you're writing (typically) 8-16 bytes per value.

So binary saves a factor of 2 to 4 (typically).

Problem with Binary I/O

There are many ways to represent data inside a computer, especially floating point (real) data.

Often, the way that one kind of computer (e.g., an Intel i7) saves binary data is different from another kind of computer (e.g., an IBM POWER7).

So, a file written on an Intel i7 machine may not be readable on an IBM POWER7.

Portable I/O Libraries

NetCDF and HDF are the two most commonly used I/O libraries for scientific computing.

Each has its own internal way of representing numerical data. When you write a file using, say, HDF, it can be read by a HDF on **<u>any</u>** kind of computer.

Plus, these libraries are optimized to make the I/O **<u>very fast</u>**.

Virtual Memory

Virtual Memory

- Typically, the amount of main memory (RAM) that a CPU can ***address*** is larger than the amount of data physically present in the computer.
- For example, consider a laptop that can address 1 TB of main memory (roughly 1 trillion bytes), but only contains 4 GB (roughly 4 billion bytes).

Virtual Memory (cont'd)

- **Locality**: Most programs don't jump all over the memory that they use; instead, they work in a particular area of memory for a while, then move to another area.
- So, you can offload onto hard disk much of the ***memory image*** of a program that's running.

Virtual Memory (cont'd)

- Memory is chopped up into many ***pages*** of modest size (e.g., 1 KB – 32 KB; typically 4 KB).
- Only pages that have been recently used actually reside in memory; the rest are stored on hard disk.
- Hard disk is 1,000+ times slower than main memory, so you get better performance if you rarely get a ***page fault***, which forces a read from (and maybe a write to) hard disk: **exploit data locality!**

Cache vs. Virtual Memory

- Lines (cache) vs. pages (VM)
- Cache faster than RAM (cache) vs.
  RAM faster than disk (VM)

Storage Use Strategies

- **Register reuse**: do a lot of work on the same data before working on new data.
- **Cache reuse**: the program is much more efficient if all of the data and instructions fit in cache; if not, try to use what's in cache a lot before using anything that isn't in cache (e.g., tiling).
- **Data locality**: try to access data that are near each other in memory before data that are far.
- **I/O efficiency**: do a bunch of I/O all at once rather than a little bit at a time; don't mix calculations and I/O.

TENTATIVE Schedule

Tue Jan 20: Storage Hierarchy: What the Heck is Supercomputing?
Tue Jan 27: The Tyranny of the Storage Hierarchy
Tue Feb 3: Instruction Level Parallelism
Tue Feb 10: Stupid Compiler Tricks
Tue Feb 17: Shared Memory Multithreading
Tue Feb 24: Distributed Multiprocessing
Tue March 3: Applications and Types of Parallelism
Tue March 10: Multicore Madness
Tue March 17: **NO SESSION** (OU's Spring Break)
Tue March 24: **NO SESSION** (Henry has a huge grant proposal due)
Tue March 31: High Throughput Computing
Tue Apr 7: GPGPU: Number Crunching in Your Graphics Card
Tue Apr 14: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Brandon George, Dave Akin, Brett Zimmerman, Josh Alexander, Patrick Calhoun)
  - Horst Severini, OSCER Associate Director for Remote & Heterogeneous Computing
  - Debi Gentis, OSCER Coordinator
  - Jim Summers
  - The OU IT network team
- James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman, OneNet
- Kay Avila, U Iowa
- Stephen Harrell, Purdue U

Coming in 2015!

Red Hat Tech Day, Thu Jan 22 2015 @ OU
    http://goo.gl/forms/jORZCz9xh7
Linux Clusters Institute workshop May 18-22 2015 @ OU
    http://www.linuxclustersinstitute.org/workshops/
Great Plains Network Annual Meeting, May 27-29, Kansas City
Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency May 31 - June 6 2015
XSEDE2015, July 26-30, St. Louis MO
    https://conferences.xsede.org/xsede15
IEEE Cluster 2015, Sep 23-27, Chicago IL
    http://www.mcs.anl.gov/ieeecluster2015/
OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2015, **Sep 22-23 2015** @ OU
SC13, Nov 15-20 2015, Austin TX
    http://sc15.supercomputing.org/

OK Supercomputing Symposium 2015

2003 Keynote:
Peter Freeman
NSF
Computer & Information Science & Engineering
Assistant Director

2004 Keynote:
Sangtae Kim
NSF Shared Cyberinfrastructure
Division Director

2005 Keynote:
Walt Brooks
NASA Advanced Supercomputing
Division Director

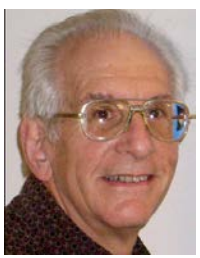
2006 Keynote:
Dan Atkins
Head of NSF's Office of Cyberinfrastructure

2007 Keynote:
Jay Boisseau
Director
Texas Advanced Computing Center
U. Texas Austin

2008 Keynote:
José Munoz
Deputy Office Director/Senior Scientific Advisor
NSF Office of Cyberinfrastructure

2009 Keynote:
Douglass Post
Chief Scientist
US Dept of Defense
HPC Modernization Program

2010 Keynote:
Horst Simon
Deputy Director
Lawrence Berkeley National Laboratory



2011 Keynote:
Barry Schneider
Program Manager
National Science Foundation

2012 Keynote:
Thom Dunning
Director
National Center for Supercomputing Applications

2013 Keynote:
John Shalf
Dept Head CS
Lawrence Berkeley Lab
CTO, NERSC

2014 Keynote:
Irene Qualters
Division Director
Advanced Cyberinfarstructure Division, NSF

FREE!
Wed Sep 23 2015
@ OU

Reception/Poster Session
Tue Sep 22 2015 @ OU
Symposium
Wed Sep 23 2015 @ OU

Thanks for your attention!

Questions?

www.oscer.ou.edu

References

[1] http://graphics8.nytimes.com/images/2007/07/13/sports/auto600.gif
[2] http://www.vw.com/newbeetle/
[3] http://img.dell.com/images/global/products/resultgrid/sm/latit_d630.jpg
[4] http://en.wikipedia.org/wiki/X64
[5] Richard Gerber, The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture. Intel Press, 2002, pp. 161-168.
[6] http://www.anandtech.com/showdoc.html?i=1460&p=2
[8] http://www.toshiba.com/taecdpd/products/features/MK2018gas-Over.shtml
[9] http://www.toshiba.com/taecdpd/techdocs/sdr2002/2002spec.shtml
[10] ftp://download.intel.com/design/Pentium4/manuals/24896606.pdf
[11] http://www.pricewatch.com/
[12] http://en.wikipedia.org/wiki/POWER7
[13] http://www.kingston.com/branded/image_files/nav_image_desktop.gif
14] M. Wolfe, High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, Redwood City CA, 1996.
[15] http://www.visit.ou.edu/vc_campus_map.htm
[16] http://www.storagereview.com/
[17] http://www.unidata.ucar.edu/packages/netcdf/
[18] http://hdf.ncsa.uiuc.edu/
[23] http://en.wikipedia.org/wiki/Itanium
[19] ftp://download.intel.com/design/itanium2/manuals/25111003.pdf
[20] http://images.tomshardware.com/2007/08/08/extreme_fsb_2/qx6850.jpg (em64t)
[21] http://www.pcdo.com/images/pcdo/20031021231900.jpg (power5)
[22] http://vnuuk.typepad.com/photos/uncategorized/itanium2.jpg (i2)
[??] http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2353&p=2 (Prescott cache latency)
[??] http://www.xbitlabs.com/articles/mobile/print/core2duo.html (T2400 Merom cache)
[??] http://www.lenovo.hu/kszf/adatlap/Prosi_Proc_Core2_Mobile.pdf (Merom cache line size)
[25] http://www.lithium.it/nove3.jpg
[26] http://cpu.rightmark.org/
[27] Tribuvan Kumar Prakash, "Performance Analysis of Intel Core 2 Duo Processor." MS Thesis, Dept of Electrical and Computer Engineering, Louisiana State University, 2007.
[28] R. Kalla, IBM, personal communication, 10/26/2010.
[29] Intel® 64 and IA-32 Architectures Optimization Reference Manual
http://www.intel.com/Assets/en_US/PDF/manual/248966.pdf