

Parallel & Cluster Computing

The Storage Hierarchy

National Computational Science Institute
August 8-14 2004

Paul Gray, University of Northern Iowa
David Joiner, Shodor Education Foundation
Tom Murphy, Contra Costa College
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College





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

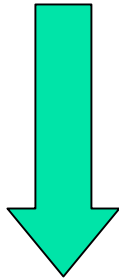


What is the Storage Hierarchy?



[1]

Fast, expensive, few



Slow, cheap, a lot

- Registers
- Cache memory
- Main memory (RAM)
- Hard disk
- Removable media (e.g., CDROM)
- Internet



[2]


Henry's Laptop

Dell Latitude C840^[3]



- Pentium 4 1.6 GHz w/512 KB L2 Cache
- 512 MB 400 MHz DDR SDRAM
- 30 GB Hard Drive
- Floppy Drive
- DVD/CD-RW Drive
- 10/100 Mbps Ethernet
- 56 Kbps Phone Modem

Storage Speed, Size, Cost

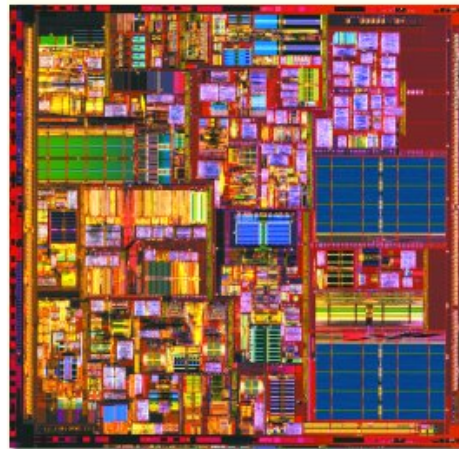
	Registers (Pentium 4 1.6 GHz)	Cache Memory (L2)	Main Memory (400 MHz DDR SDRAM)	Hard Drive	Ethernet (100 Mbps)	CD-RW	Phone Modem (56 Kbps)
Speed (MB/sec) [peak]	73,232 ^[7] (3200 MFLOP/s*)	52,428 ^[8]	3,277 ^[9]	100 ^[10]	12	4 ^[11]	0.007
Size (MB)	304 bytes** ^[12]	0.5	512	30,000	unlimited	unlimited	unlimited
Cost (\$/MB)	—	\$254 ^[13]	\$0.24 ^[13]	\$0.0005 ^[13]	charged per month (typically)	\$0.0015 ^[13]	charged per month (typically)

* MFLOP/s: millions of floating point operations per second

** 8 32-bit integer registers, 8 80-bit floating point registers, 8 64-bit MMX integer registers, 8 128-bit floating point XMM registers



Registers

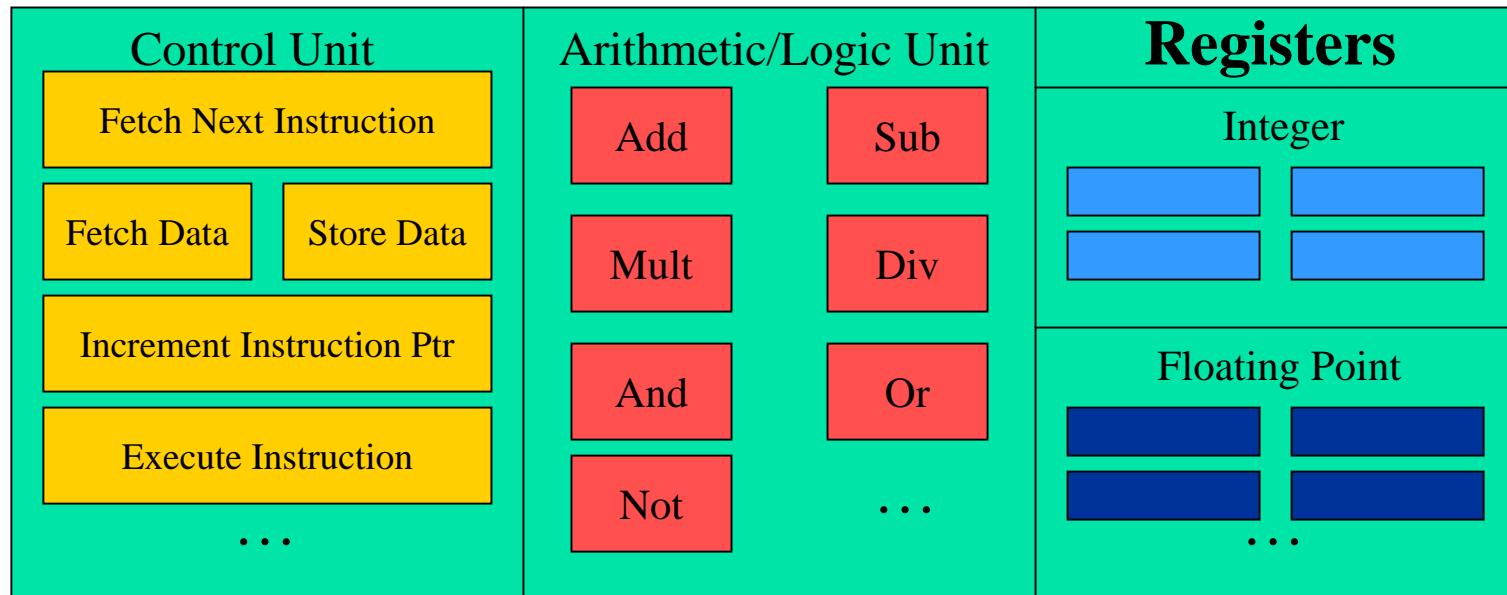


[4]

What Are Registers?

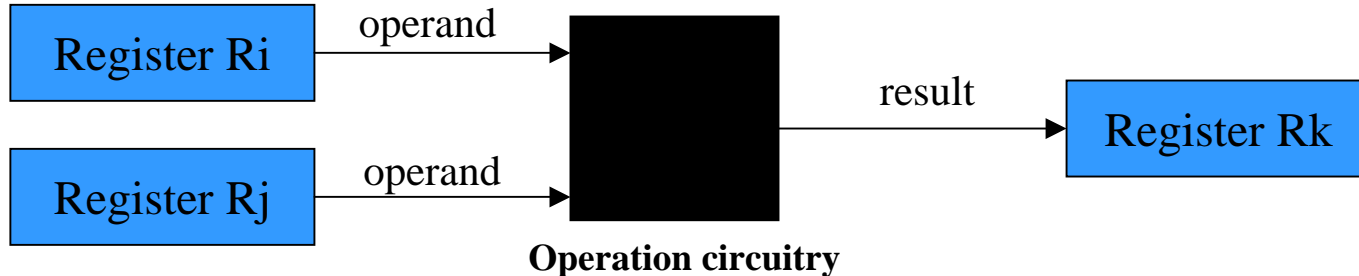
Registers are memory-like locations inside the Central Processing Unit that hold data that are **being used right now** in operations.

CPU

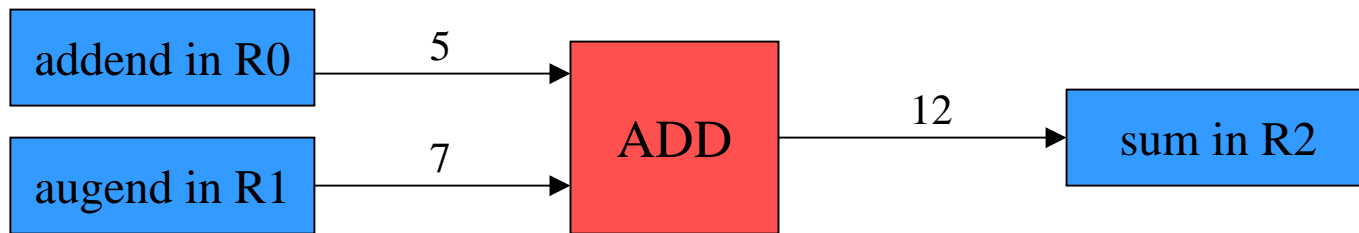


How Registers Are Used

- 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 the destination register.



Example:



How Many Registers?

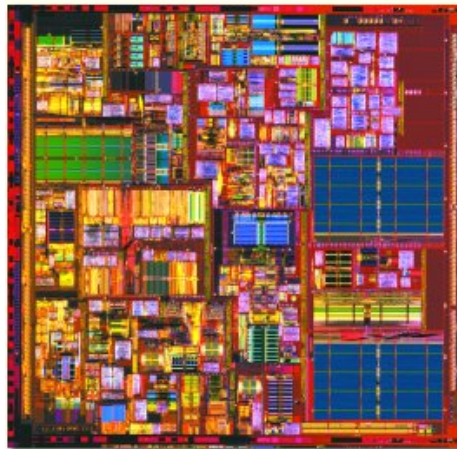
Typically, a CPU has less than 2 KB (2048 bytes) of registers, usually split into registers for holding integer values and registers for holding floating point (real) values, plus a few special purpose registers.

Examples:

- IBM POWER4 (found in IBM Regatta supercomputers): 80 64-bit integer registers and 72 64-bit floating point registers (1,216 bytes) [12]
- Intel Pentium4: 8 32-bit integer registers, 8 80-bit floating point registers, 8 64-bit integer vector registers, 8 128-bit floating point vector registers (304 bytes) [10]



Cache

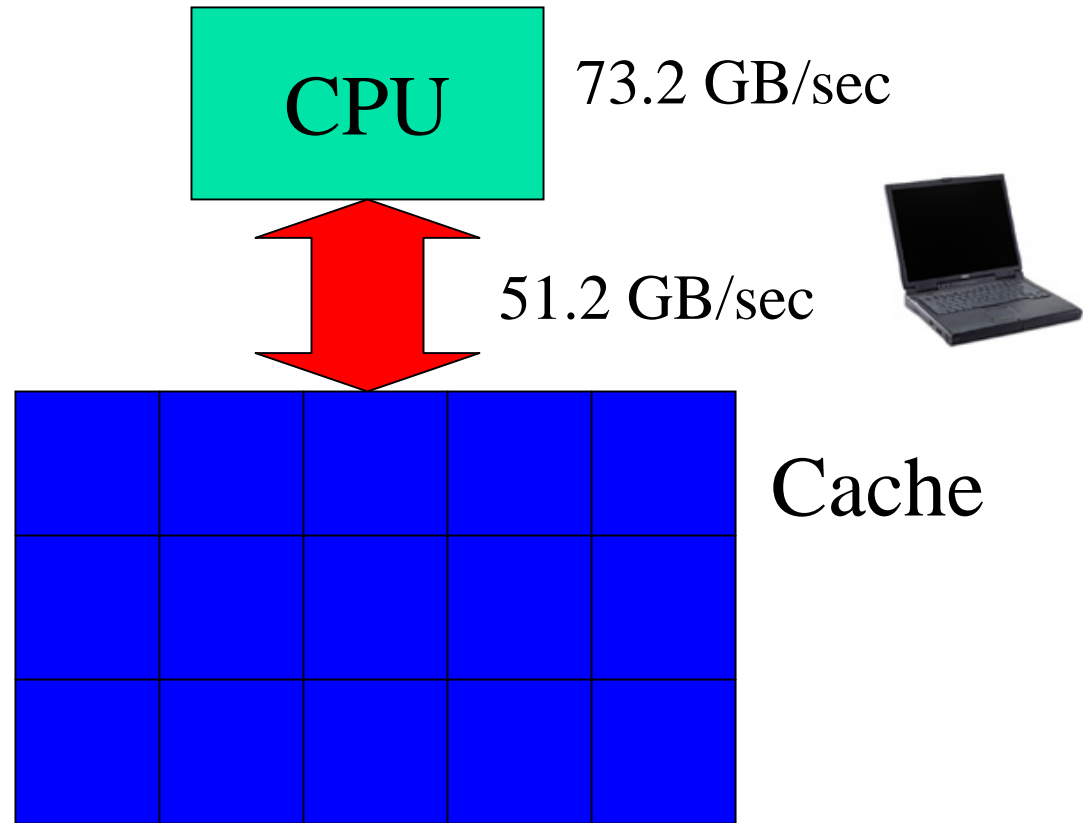


[4]

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 comparable to that of the CPU performing calculations.

Multiple Levels of Cache

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

- Intel Pentium4 [5,10] 
 - Level 1 caches: 12 KB instruction*, 8 KB data
 - Level 2 cache: 512 KB unified (instruction + data)
- IBM POWER4 [12]
 - Level 1 cache: 64 KB instruction, 32 KB data
 - Level 2 cache: 1440 KB unified for each 2 CPUs
 - Level 3 cache: 32 MB unified for each 2 CPUS



*Pentium 4 L1 instruction cache is called “trace cache.”

Why Multiple Levels of Cache?

The lower the level of cache:

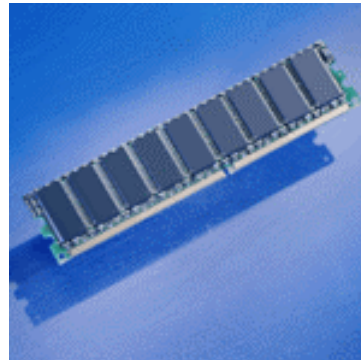
- the faster the cache can transfer data to the CPU;
- the smaller that level of cache is, because faster => more expensive => smaller.

Example: IBM POWER4 latency to the CPU [12]

- L1 cache: 4 cycles = 3.6 ns for 1.1 GHz CPU
- L2 cache: 14 cycles = 12.7 ns for 1.1 GHz CPU



Main Memory

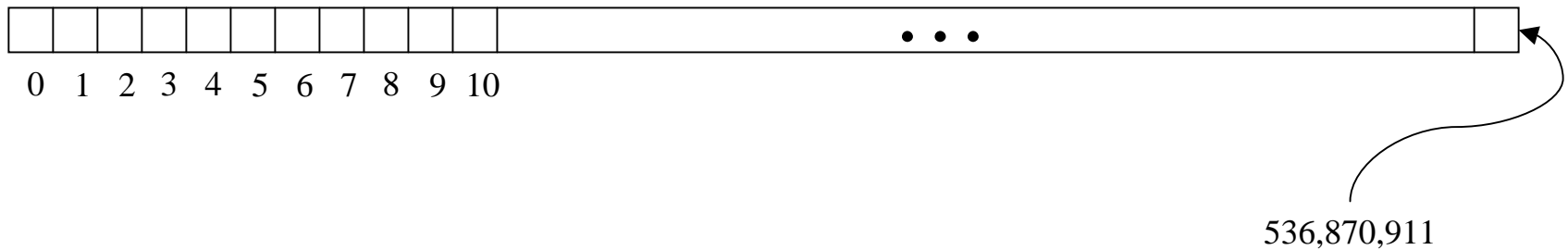


[13]

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.

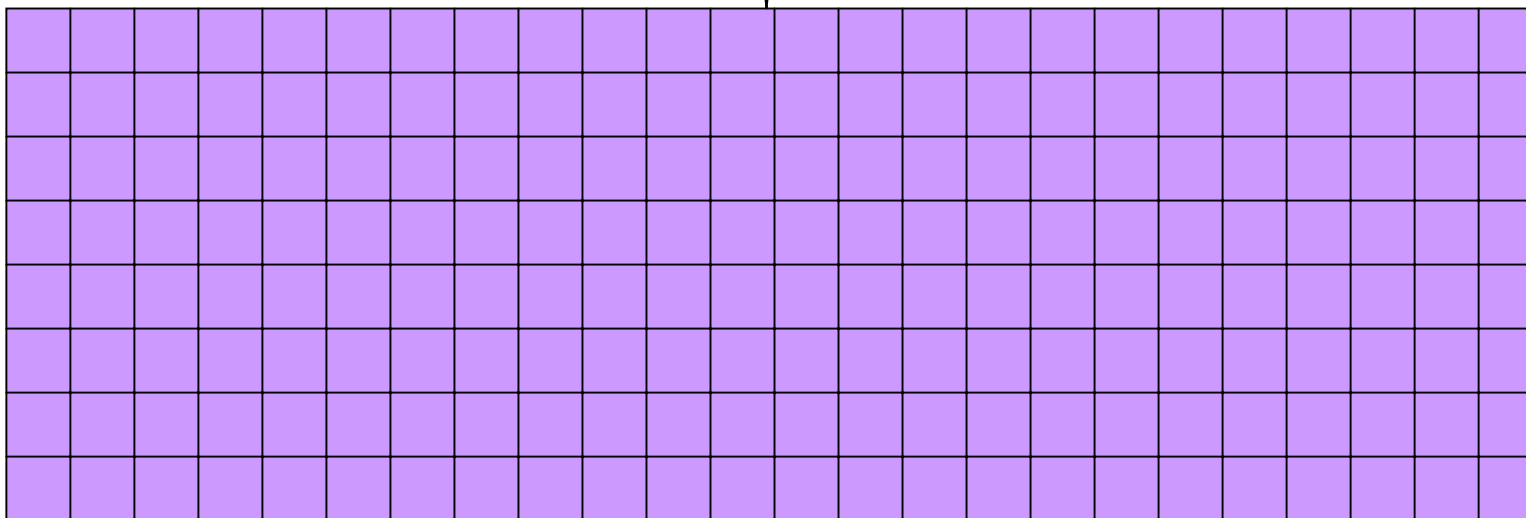
CPU

73.2 GB/sec



Bottleneck

3.2 GB/sec

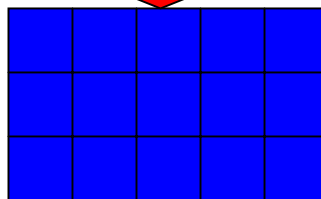


Why Have Cache?

Cache is nearly the same speed as 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!

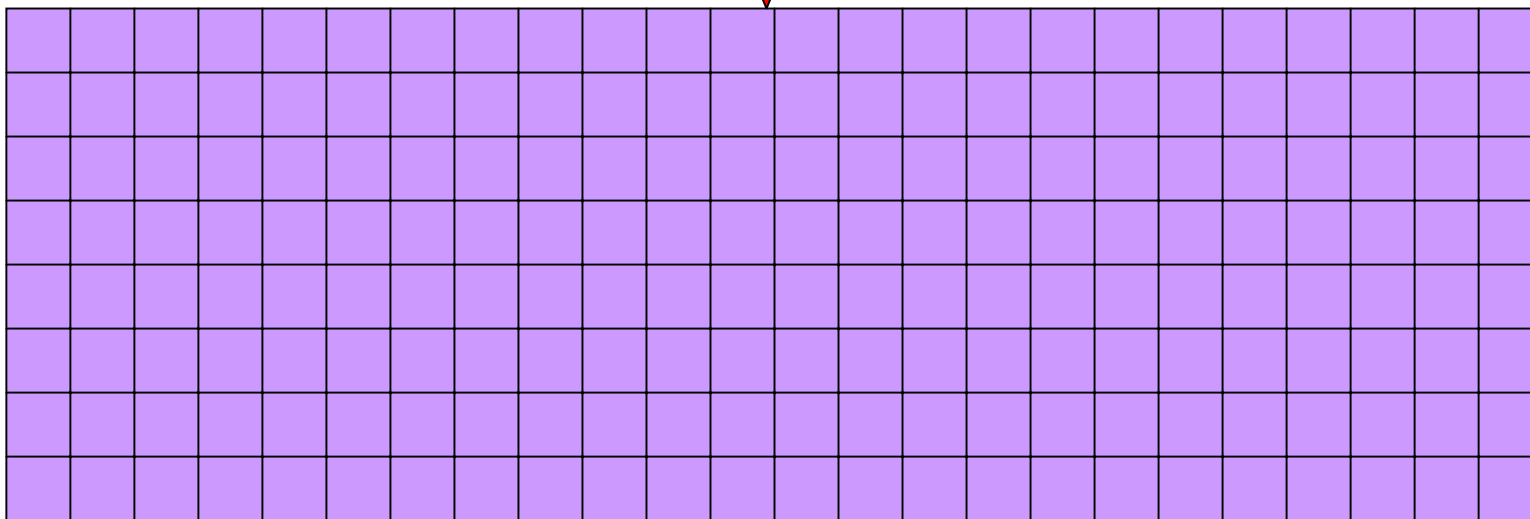
CPU

73.2 GB/sec



51.2 GB/sec

3.2 GB/sec



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

- Pentium 4 [5,10]



- L1 data cache: 64 bytes per line
- L2 cache: 128 bytes per line

- POWER4 [12]

- L1 instruction cache: 128 bytes per line
- L1 data cache: 128 bytes per line
- L2 cache: 128 bytes per line
- L3 cache: 512 bytes 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 1.6 GHz Pentium 4, a cache miss makes the program stall (wait) at least 37.25 nanoseconds for the next cache line to load – a time that could have been spent performing up to 149 calculations! [5,10]





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

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.

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

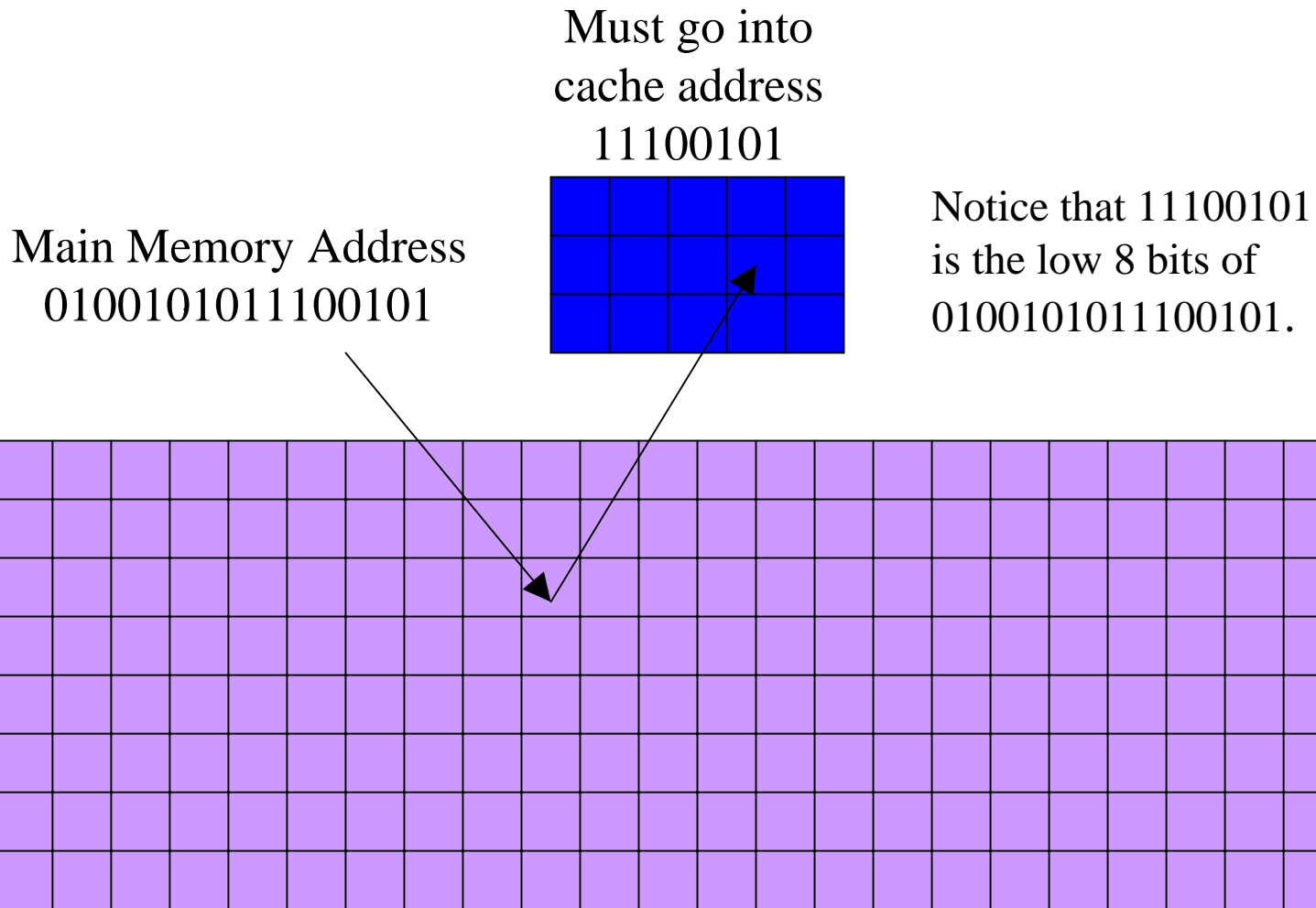
Direct Mapped Cache is a scheme in which each location in main memory corresponds to exactly one location in cache (but not the reverse, since cache is much smaller than main memory).

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 $\text{MOD}(A, 2^c)$; that is, the lowest c bits of A .

Example: POWER4 L1 instruction cache



Direct Mapped Cache Illustration





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

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 !! index = 1, multiple_of_cache_size
```

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., First In First Out, Round Robin, Least Recently Modified).

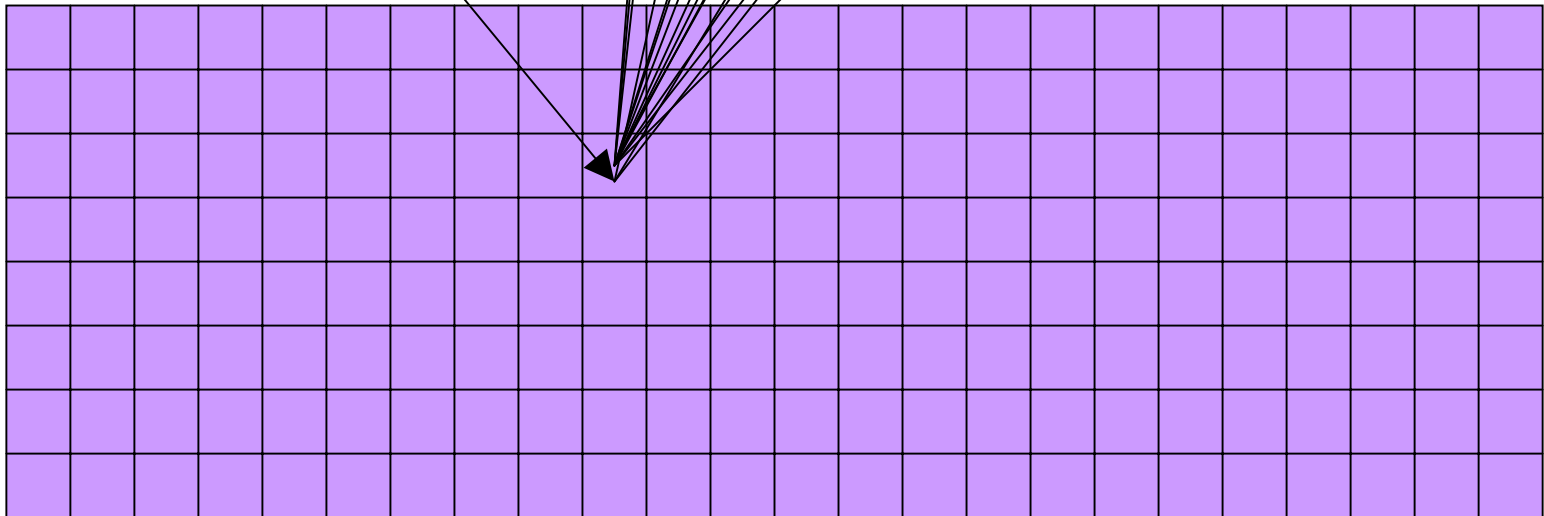
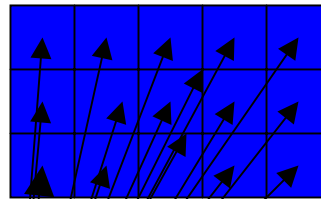
So, this can solve, or at least reduce, the cache conflict problem.

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

Main Memory Address
0100101011100101

Could go into
any cache line



Set Associative Cache

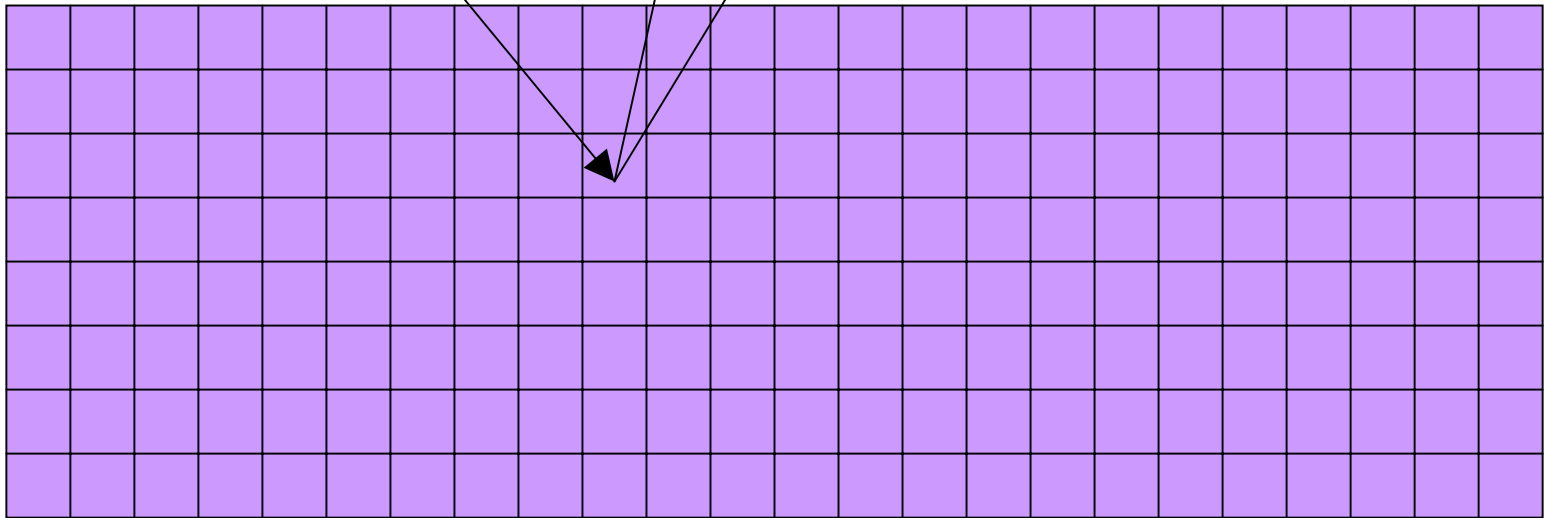
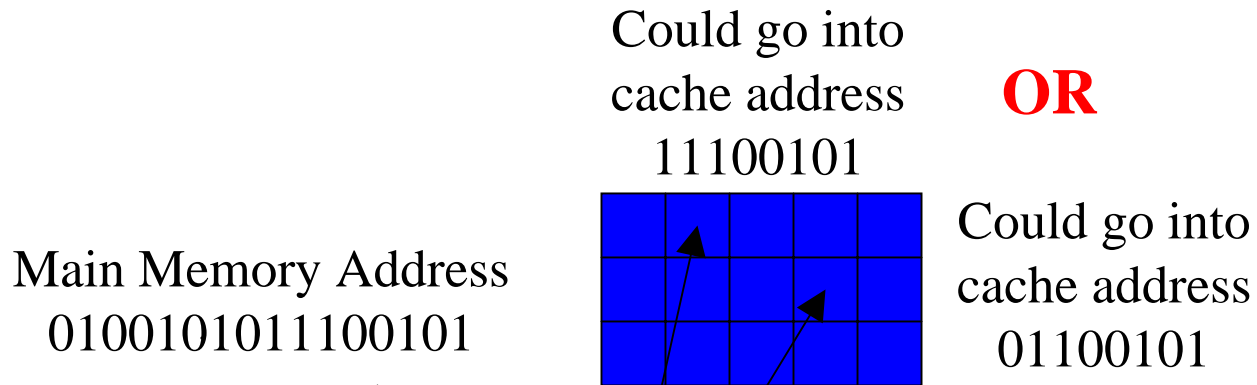
Set Associative Cache is a compromise between direct mapped and fully associative. A line in main memory can map to any of a **fixed number** of cache lines.

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

- Pentium 4 [5,10]
 - L1 data cache: 4-way set associative
 - L2 cache: 8-way set associative
- 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





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 RAM.

That is, all of your data and instructions are in RAM, 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 and only if it's been marked as dirty, then it is stored back into main memory (RAM). [14]

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.

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



Data Locality Is Empirical

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

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

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





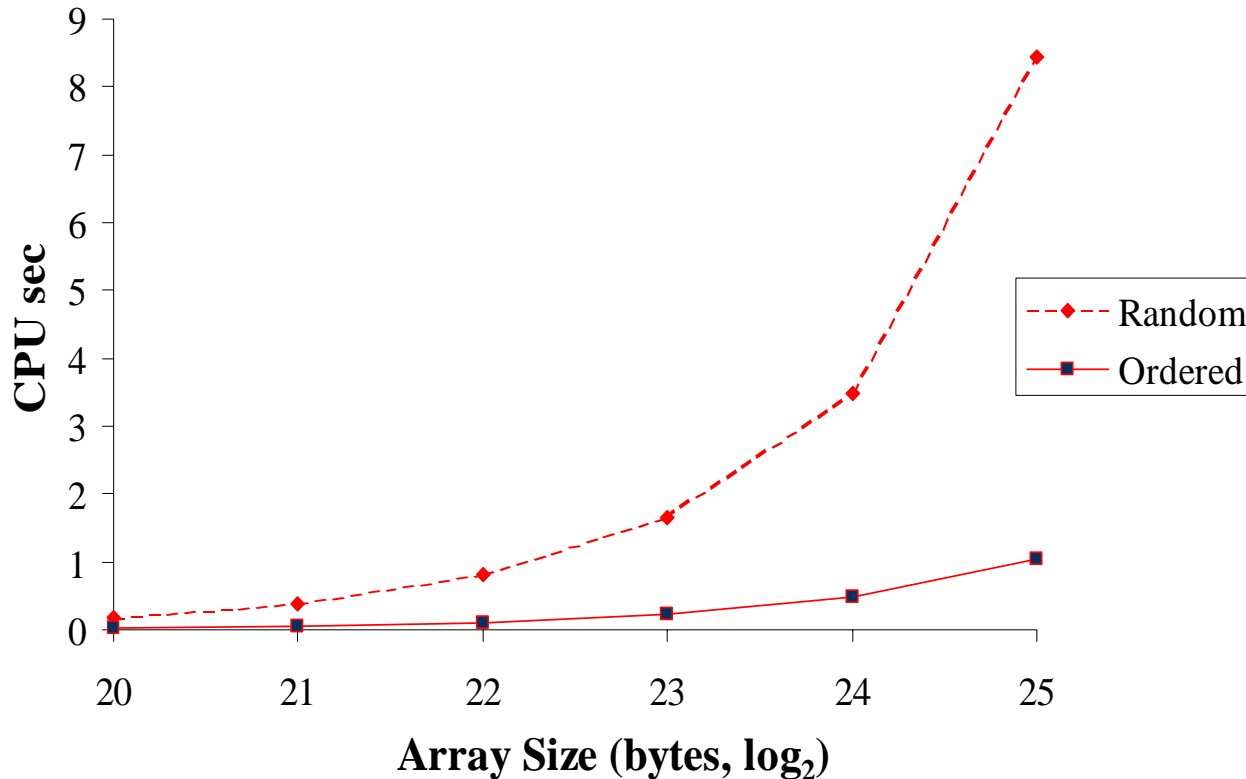
No Locality Example

In principle, you could write a program that exhibited absolutely no data locality at all:

```
void random_fill (int* 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 */
```

Permuted vs. Ordered



In a simple array fill, locality provides a factor of 6 to 8 speedup over a randomly ordered fill on a Pentium III.



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 $\mathbf{A} = \mathbf{B} \cdot \mathbf{C}$ is

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

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

Matrix Multiply: Naïve Version

```
SUBROUTINE matrix_matrix_mult_by_naive (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

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



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 = 1, nq
    END DO !! r = 1, nr
  END DO !! c = 1, nc
END SUBROUTINE matrix_matrix_mult_by_init
```



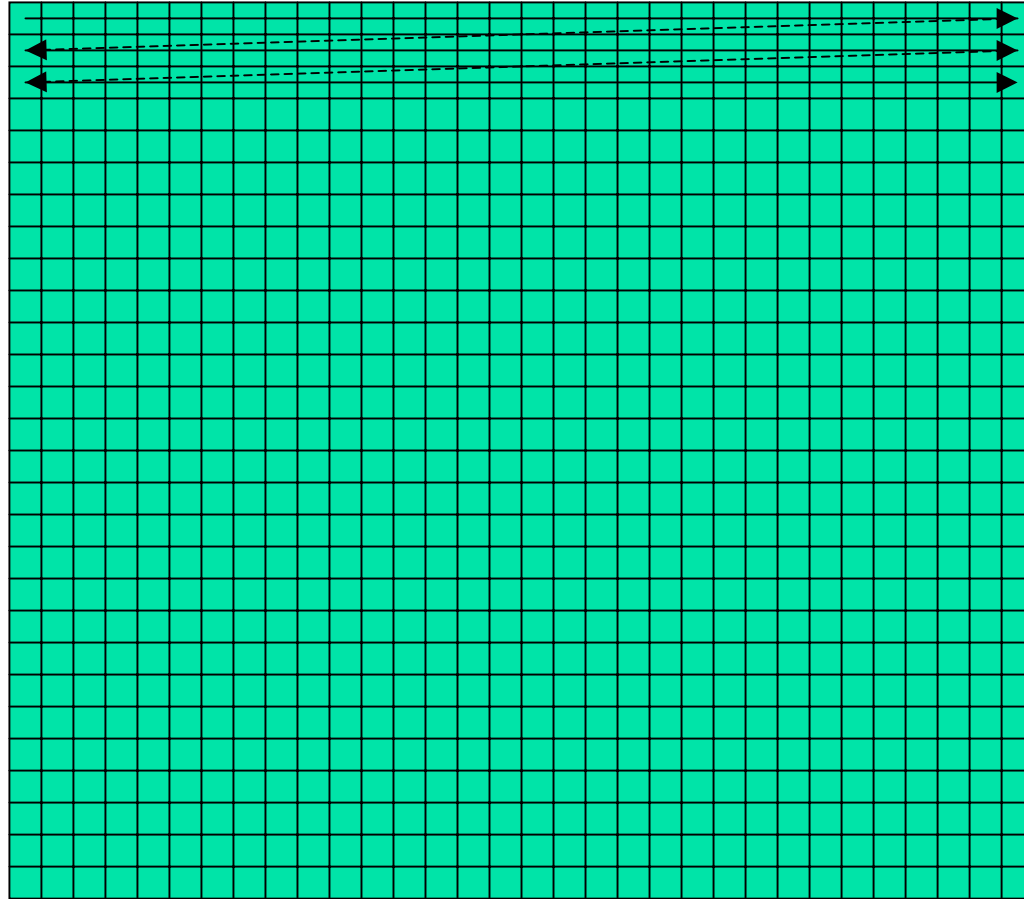
Matrix Multiply Via Intrinsic

```
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
```



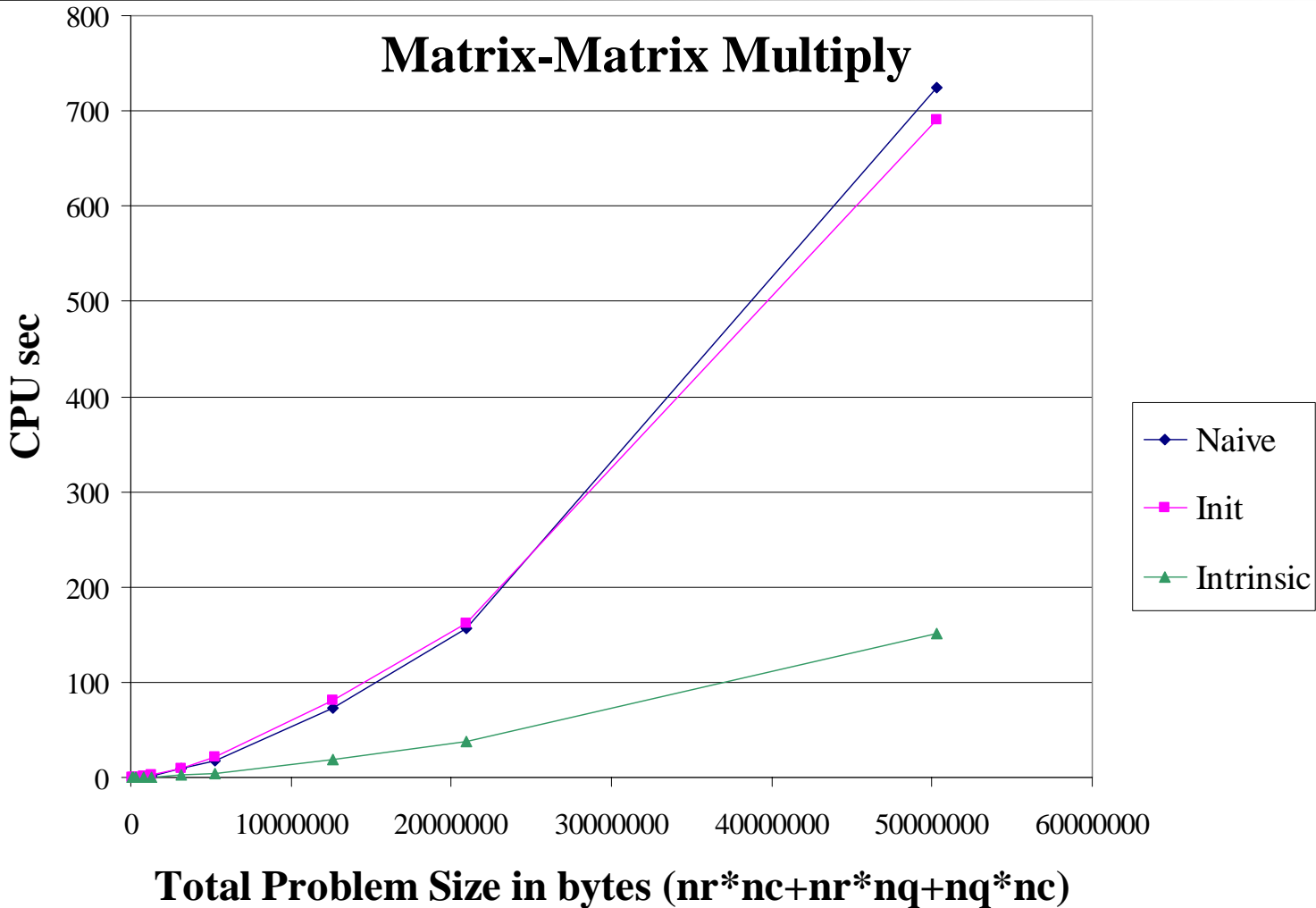
Matrix Multiply Behavior



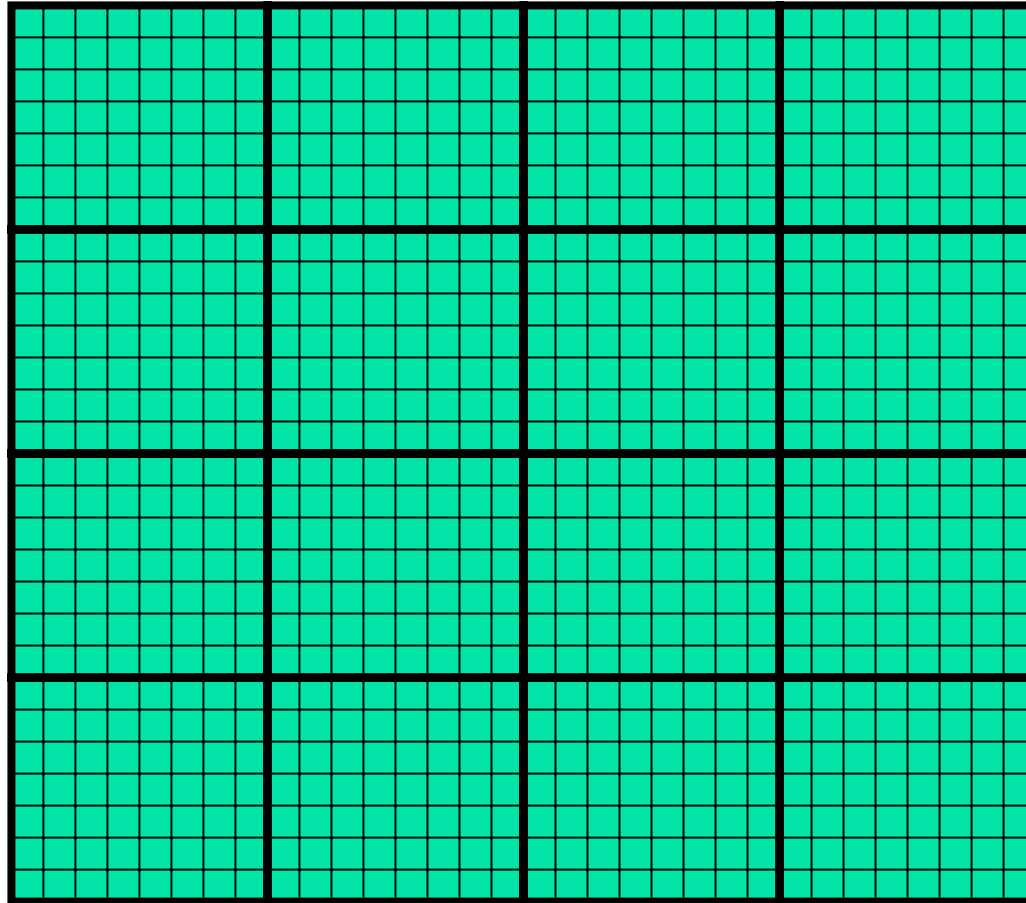
If the matrix is big, then each sweep of a row will clobber nearby values in cache.



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

```
SUBROUTINE matrix_matrix_mult_by_tiling (dst, src1, src2, nr, nc, nq, &
&      rtilsize, 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) :: rtilsize, 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, rtilsize
      rend = rstart + rtilsize - 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 = 1, nq, qtilesize
    END DO !! rstart = 1, nr, rtilsize
  END DO !! cstart = 1, nc, ctilesize
END SUBROUTINE matrix_matrix_mult_by_tiling
```



Multiplying Within a Tile

```
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

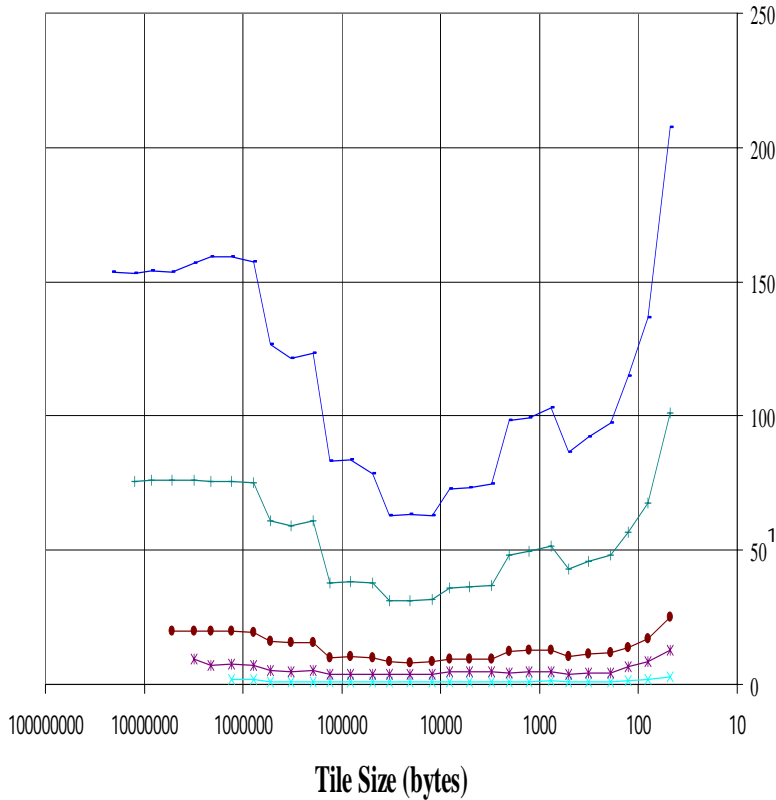
  INTEGER :: r, c, q

  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 = qstart, qend
    END DO !! r = rstart, rend
  END DO !! c = cstart, cend
END SUBROUTINE matrix_matrix_mult_tile
```

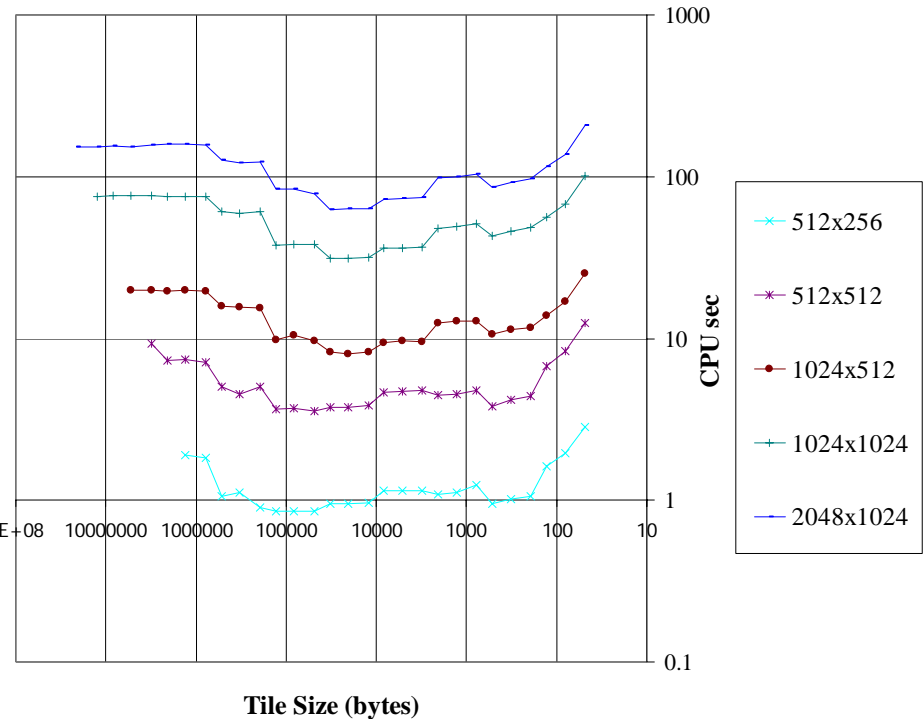


Performance with Tiling

Matrix-Matrix Multiply Via Tiling



Matrix-Matrix Multiply Via Tiling (log-log)



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.



Hard Disk



[16]



Why Is Hard Disk Slow?

Your hard disk is **much much** slower than main memory (factor of 10-1000). **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**.



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

An actual piece of code seen at OU:

```
for (thing = 0; thing < number_of_things; thing++) {  
    for (time = 0; time < number_of_timesteps; time++) {  
        read(file[time]);  
        do_stuff(thing, time);  
    } /* for time */  
} /* for thing */
```

Improved version:

```
for (time = 0; time < number_of_timesteps; time++) {  
    read(file[time]);  
    for (thing = 0; thing < number_of_things; thing++) {  
        do_stuff(thing, time);  
    } /* for thing */  
} /* for time */
```

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., a Pentium) saves binary data is different from another kind of computer (e.g., a Cray).

So, a file written on a Pentium machine may not be readable on a Cray.





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 **any** kind of computer.

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





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, Henry's laptop can address 2 GB of main memory (roughly a billion bytes), but only contains 512 MB (roughly 512 million 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).
- Only pages that have been recently used actually reside in memory; the rest are stored on hard disk.
- Hard disk is 10 to 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!**



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.



References

- [1] <http://www.flphoto.com/>
- [2] <http://www.vw.com/newbeetle/>
- [3] http://www.dell.com/us/en/bsd/products/model_latit_latit_c840.htm
- [4] <http://www6.tomshardware.com/cpu/02q1/020107/p42200-04.html>
- [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>
- [7] <ftp://download.intel.com/design/Pentium4/papers/24943801.pdf>
- [8] <http://www.toshiba.com/taecdpc/products/features/MK2018gas-Over.shtml>
- [9] <http://www.toshiba.com/taecdpc/techdocs/sdr2002/2002spec.shtml>
- [10] <ftp://download.intel.com/design/Pentium4/manuals/24896606.pdf>
- [11] <http://www.pricewatch.com/>
- [12] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell and W. Weir, "The POWER4 Processor Introduction and Tuning Guide." IBM Redbooks, 2001.
- [13] <http://www.smartmodulartech.com/memory/ddr.html>
- [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/>

