Parallel Programming & Cluster Computing

The Tyranny of the Storage Hierarchy

Joshua Alexander, U Oklahoma
Ivan Babic, Earlham College
Michial Green, Contra Costa College
Mobeen Ludin, Earlham College

Tom Murphy, Contra Costa College
Kristin Muterspaw, Earlham College
Henry Neeman, U 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
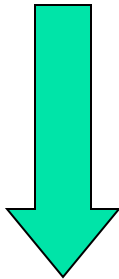- Hard Disk
- Virtual Memory

The Storage Hierarchy

Fast, expensive, few

Slow, cheap, a lot

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

[5]

A Laptop

Dell Latitude Z600[4]

- Intel Core2 Duo SU9600 1.6 GHz w/3 MB L2 Cache
- 4 GB 1066 MHz DDR3 SDRAM
- 256 GB SSD Hard Drive
- DVD+RW/CD-RW Drive (8x)
- 1 Gbps Ethernet Adapter

Storage Speed, Size, Cost

Laptop	Registers (Intel Core2 Duo 1.6 GHz)	Cache Memory (L2)	Main Memory (1066MHz DDR3 SDRAM)	Hard Drive (SSD)	Ethernet (1000 Mbps)	DVD+R (16x)	Phone Modem (56 Kbps)
Speed (MB/sec) [peak]	314,573[6] (12,800 MFLOP/s*)	27,276 [7]	4500 [7]	250 [9]	125	22 [10]	0.007
Size (MB)	464 bytes** [11]	3	4096	256,000	unlimited	unlimited	unlimited
Cost ($/MB)	–	$285 [13]	$0.03 [12]	$0.002 [12]	charged per month (typically)	$0.00005 [12]	charged per month (typically)

*   MFLOP/s: millions of floating point operations per second
** 16 64-bit general purpose registers, 8 80-bit floating point registers,
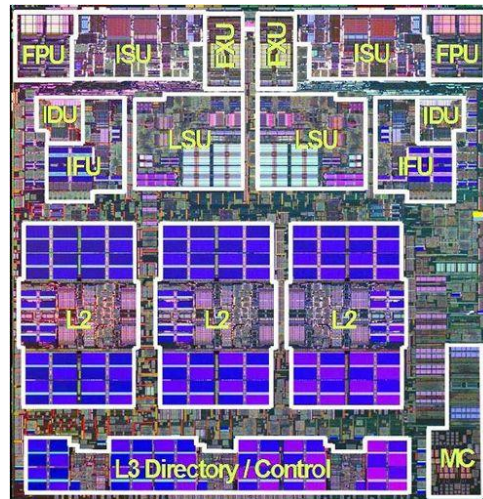16 128-bit floating point vector registers

NCSI Parallel & Cluster: Storage Hierarchy
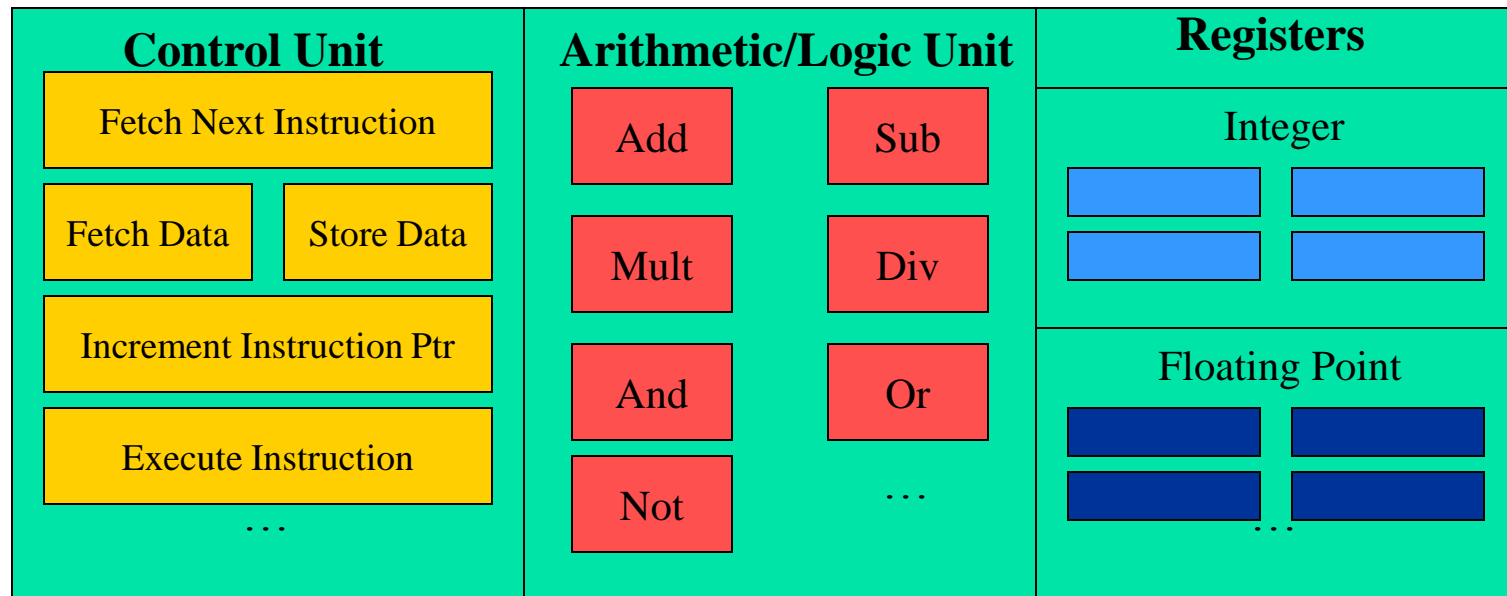U Oklahoma, July 29 - Aug 4 2012

5

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.

CPU

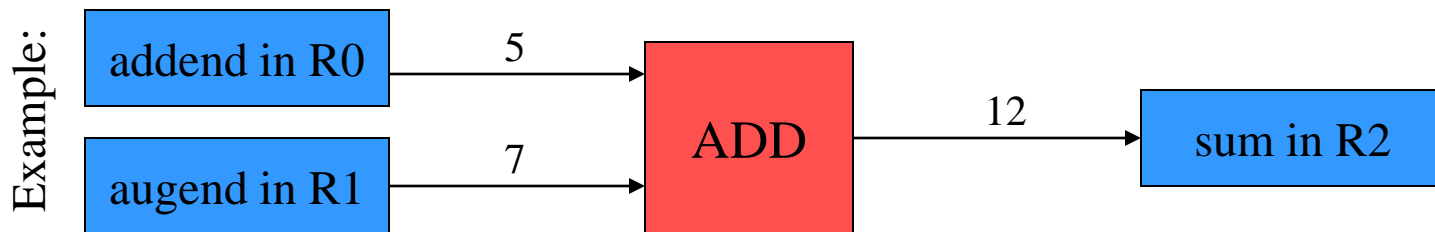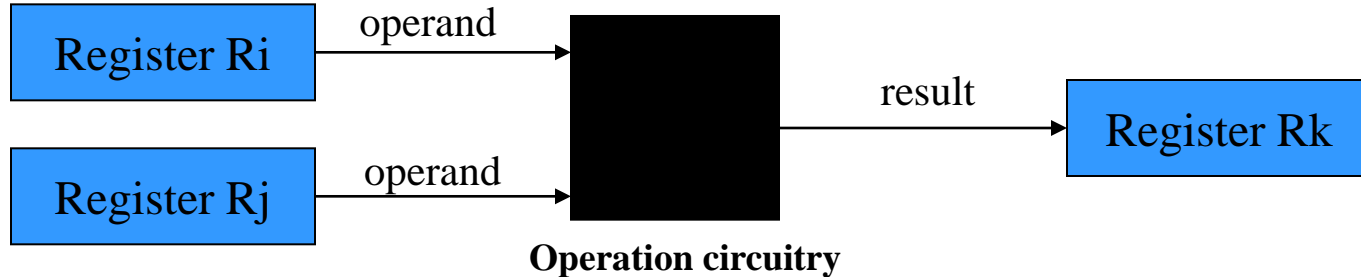| Control Unit	Arithmetic/Logic Unit	Registers
Fetch Next Instruction	Add Sub	**Integer**
Fetch Data Store Data	Mult Div	
Increment Instruction Ptr	And Or	**Floating Point**
Execute Instruction	Not …	
…		…

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.

Register Ri		

operand

result

Operation circuitry

Register Rj

Register Rk

Example:

addend in R0 — 5 — ADD — 12 — sum in R2

augend in R1 — 7 —

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 Core2 Duo**: 16 64-bit general purpose registers, 8 80-bit floating point registers, 16 128-bit floating point vector registers (464 bytes) [11]

- **Intel Itanium2**: 128 64-bit integer registers, 128 82-bit floating point registers (2304 bytes) [23]
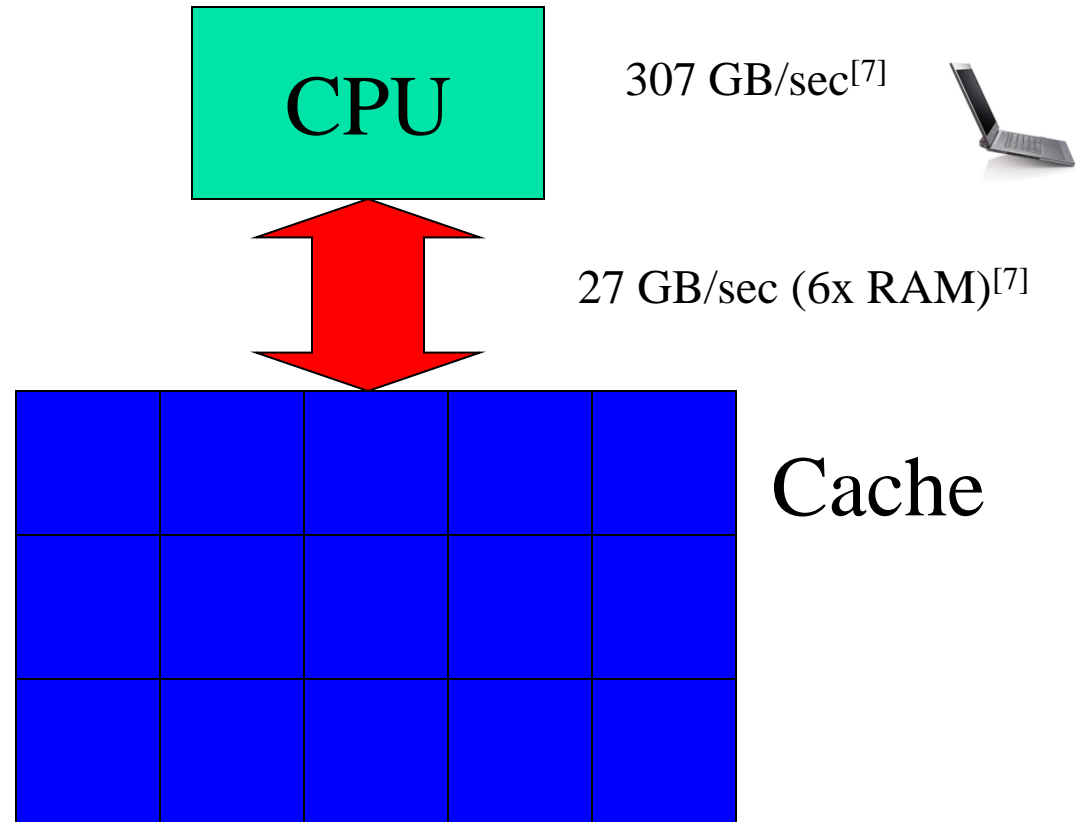
Cache
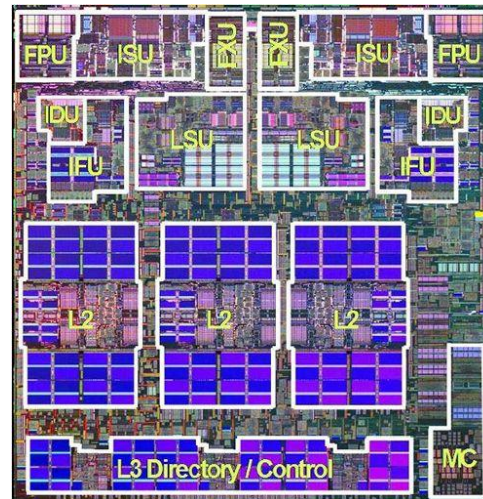
[4]

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

CPU

307 GB/sec[7]

27 GB/sec (6x RAM)[7]

Cache

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:

- **<u>Intel Pentium4 EM64T (Yonah)</u>** [??]
  - Level 1 caches: 32 KB instruction, 32 KB data
  - Level 2 cache: 2048 KB *<u>unified</u>* (instruction+data)
- **<u>IBM POWER7</u>** [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)**.

Example: 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)

Example: 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

Example: Intel Pentium4 (Yonah)
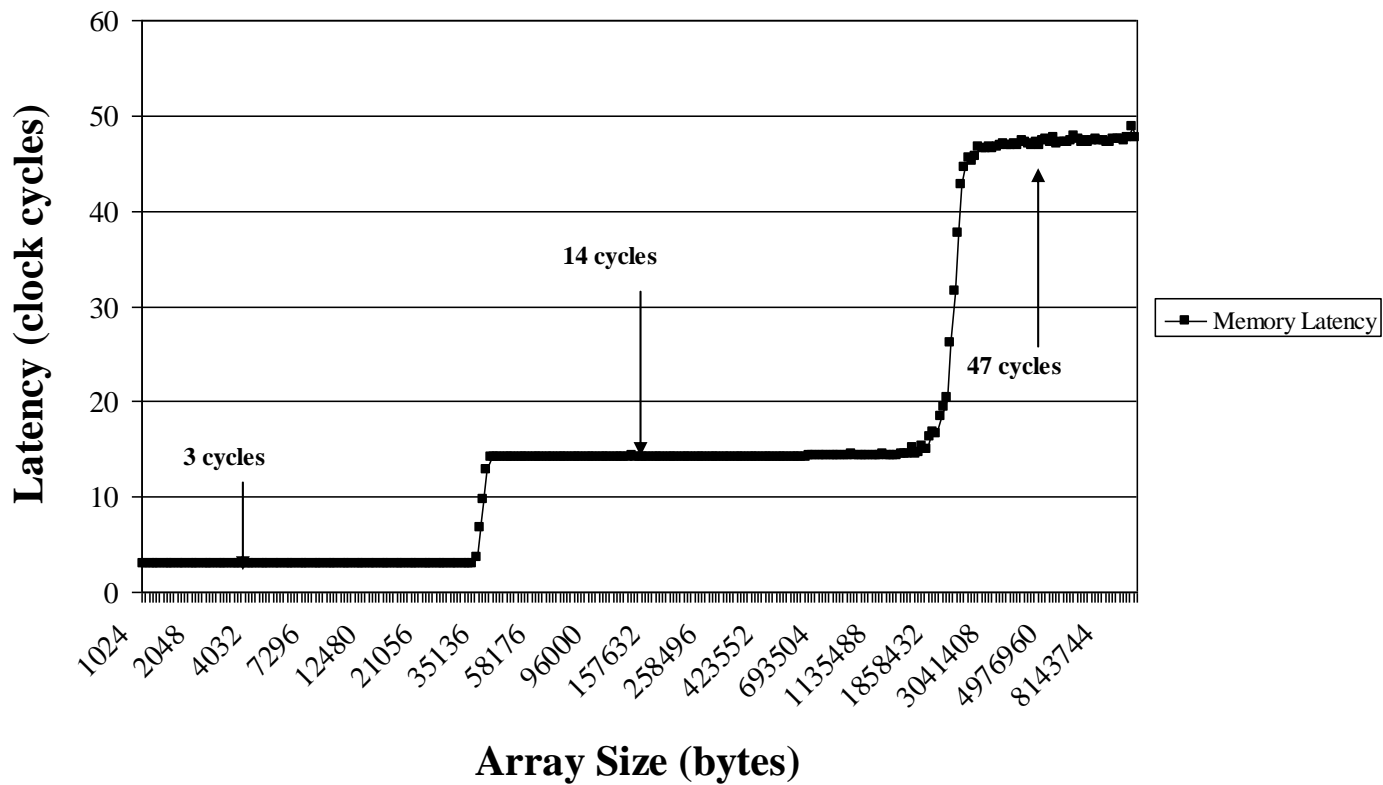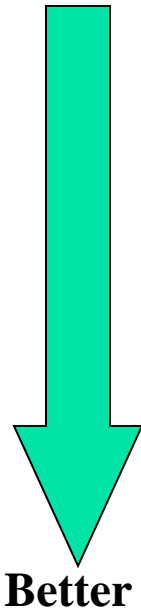- L1 cache: 3 cycles = 1.64 ns for a 1.83 GHz CPU =   12 calculations
- L2 cache: 14 cycles = 7.65 ns for a 1.83 GHz CPU =   56 calculations
- RAM: 48 cycles = 26.2 ns for a 1.83 GHz CPU = 192 calculations

Cache & RAM Latencies

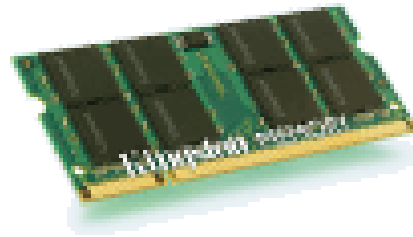**Cache & RAM Latency: Intel T2400 (1.83 GHz)**

Better

[26]

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)
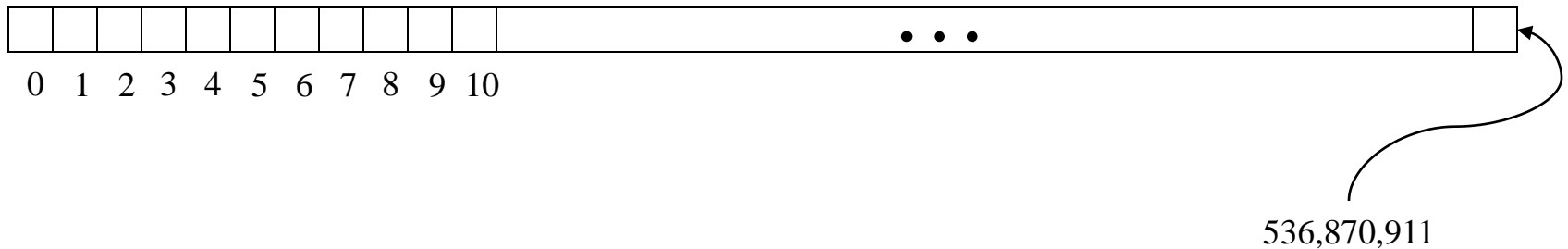
- Much slower => much cheaper => much bigger

What Main Memory Looks Like

0 1 2 3 4 5 6 7 8 9 10

• • •

536,870,911

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

The Relationship Between
Main Memory & Cache

RAM is Slow

CPU

307 GB/sec[6]
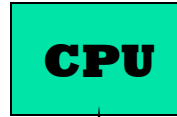
The speed of data transfer between Main Memory and the CPU is much slower than the speed of calculating, so the CPU spends most of its time waiting for data to come in or go out.
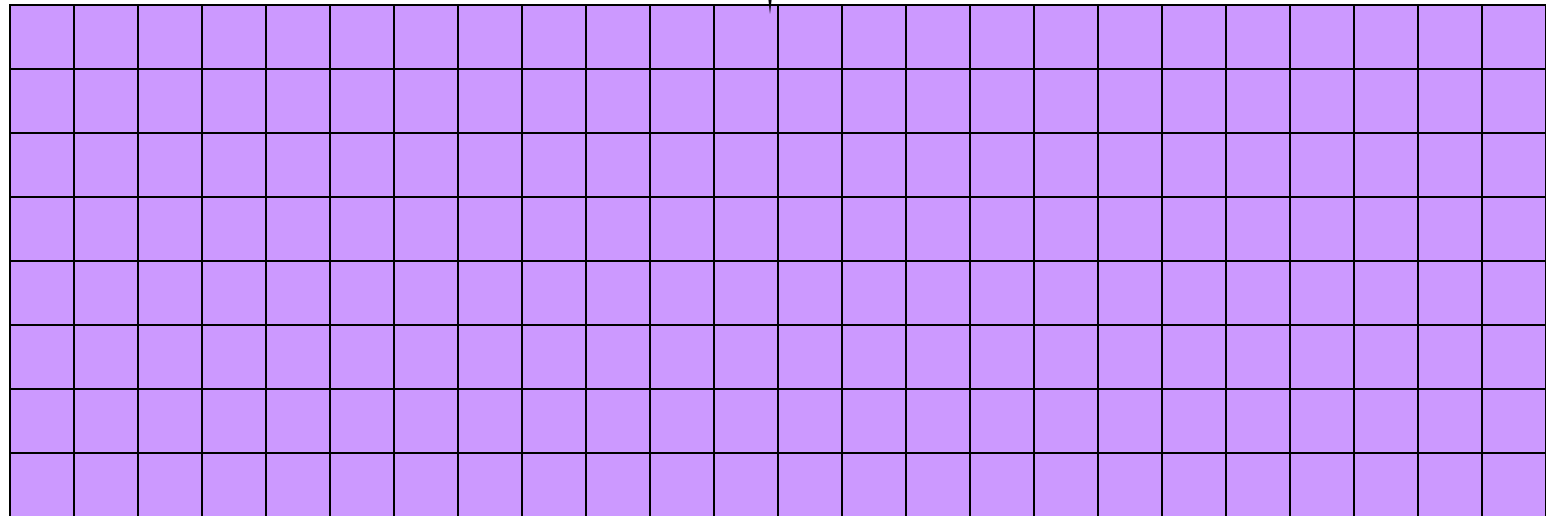
Bottleneck

4.4 GB/sec[7] (1.4%)

Why Have Cache?

CPU

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!
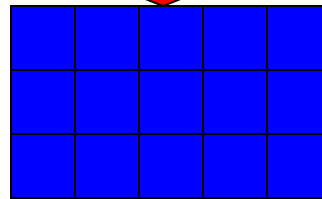
27 GB/sec (9%)[7]

4.4 GB/sec[7]

Cache & RAM Bandwidths

Cache & RAM Bandwidth: Intel T2400 (1.83 GHz)

Better

[26]
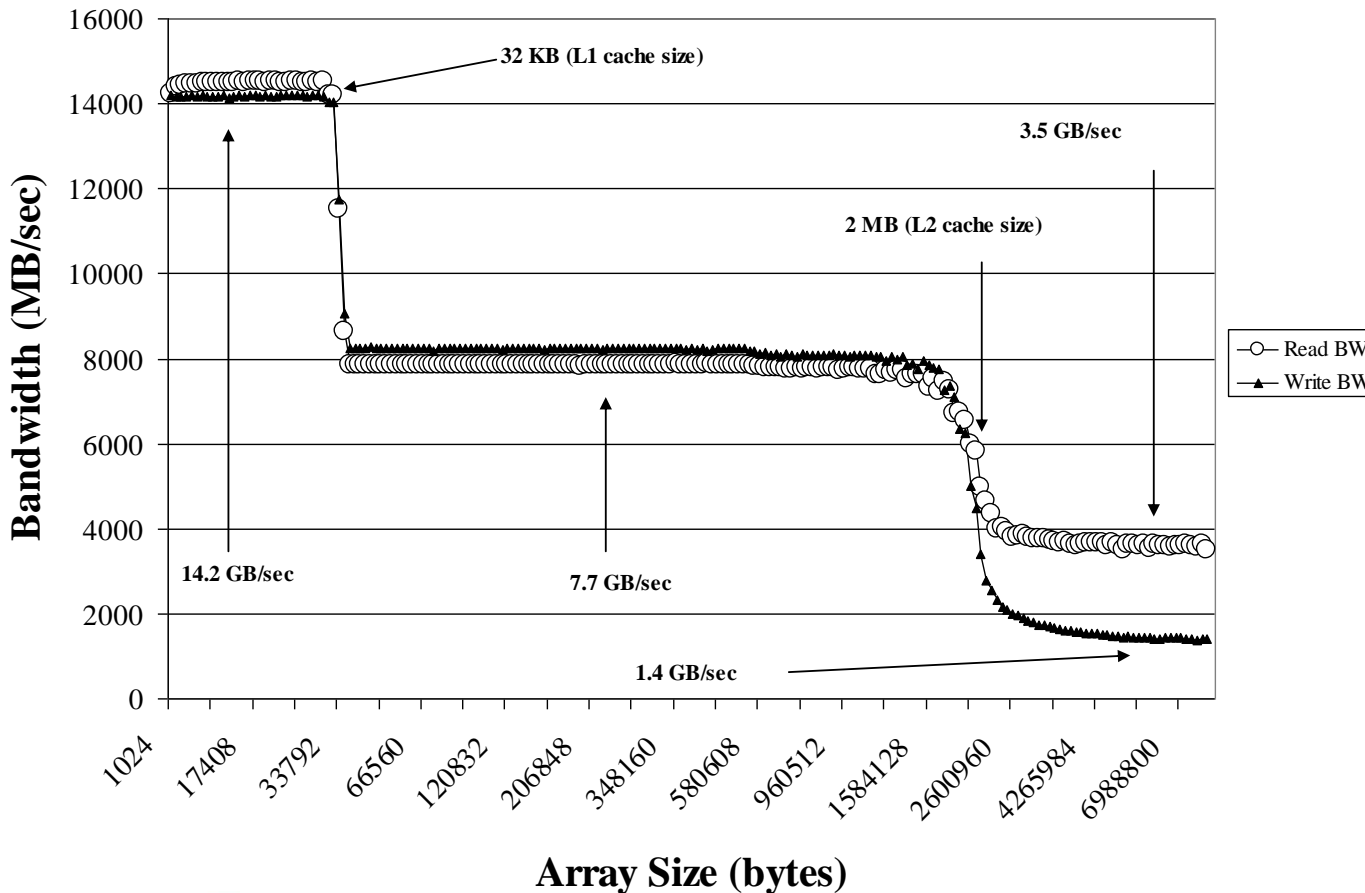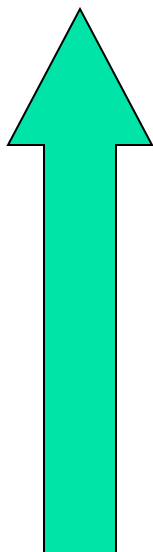
Cache Use Jargon

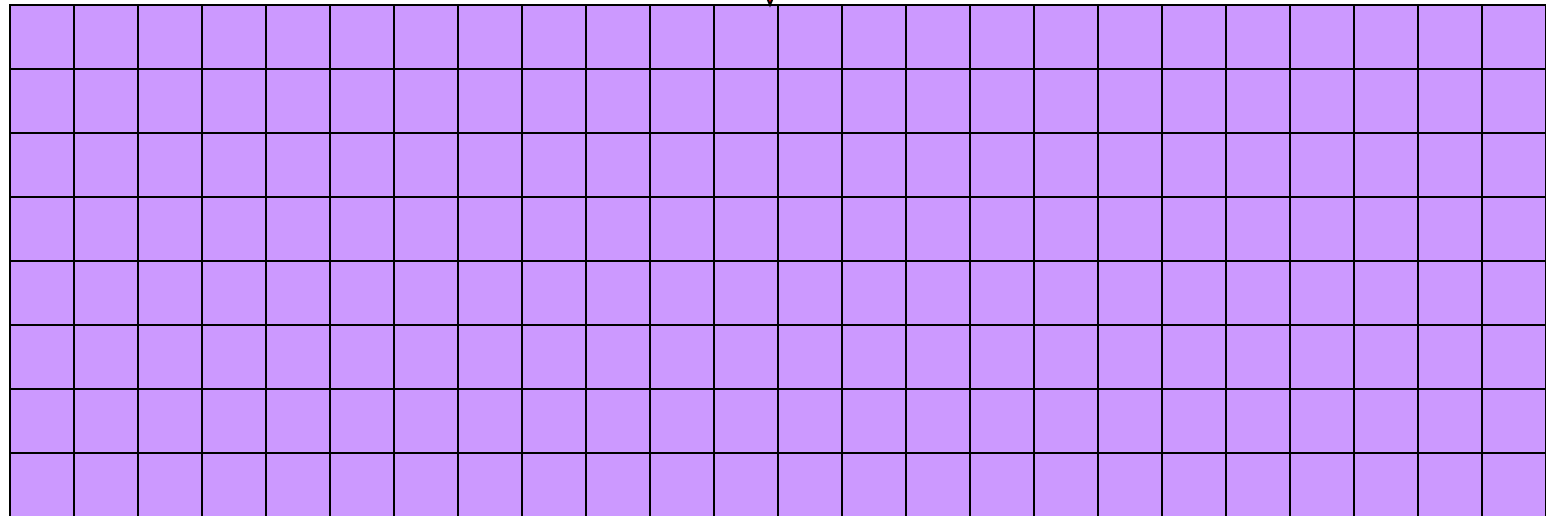- ***Cache Hit***: the data that the CPU needs right now are **already in cache**.

- ***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
  - **Core 2 Duo** [26]
    - 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 1.83 GHz Pentium4 Core Duo (Yonah), a cache miss makes the program ***stall*** (wait) at least 48 cycles (26.2 nanoseconds) for the next cache line to load – time that could have been spent performing up to 192 calculations! [26]

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.

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`.
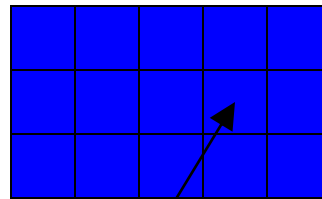
Example: POWER4 L1 instruction cache

Direct Mapped Cache Illustration

Must go into
cache address
11100101

Main Memory Address
0100101011100101

Notice that 11100101
is the low 8 bits of
0100101011100101.
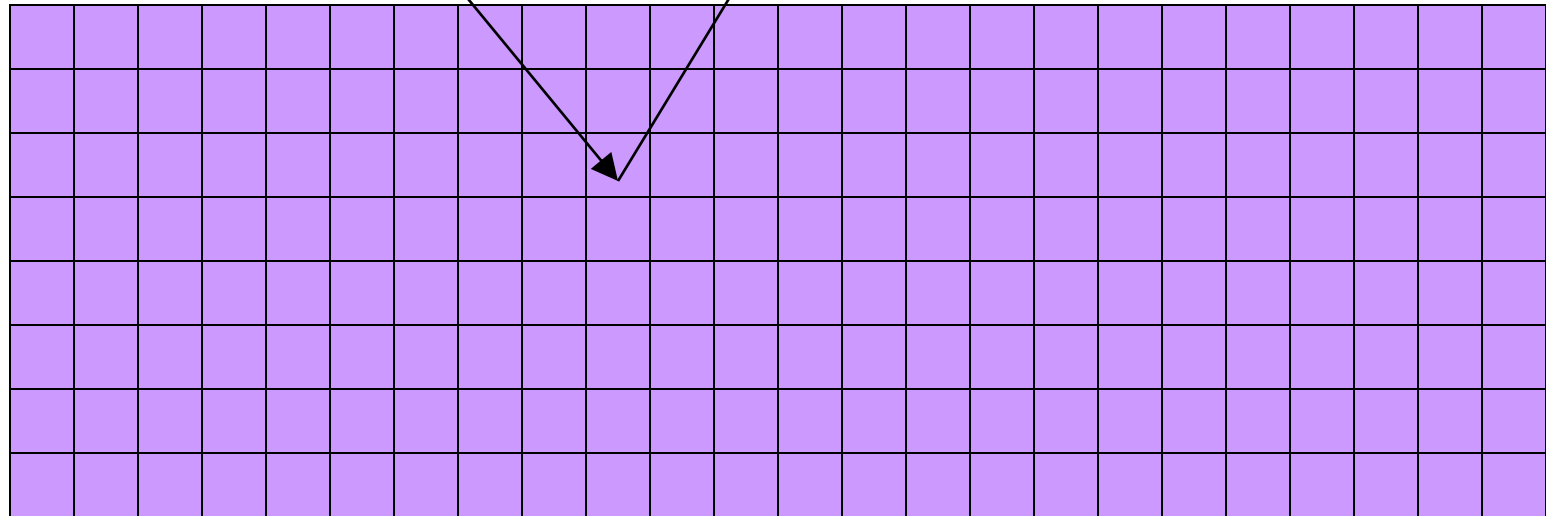
Jargon: Cache Conflict

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]** – <ins>**no cache reuse!**</ins>

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 $N_{cache} \cdot N_{RAM}$ connections!
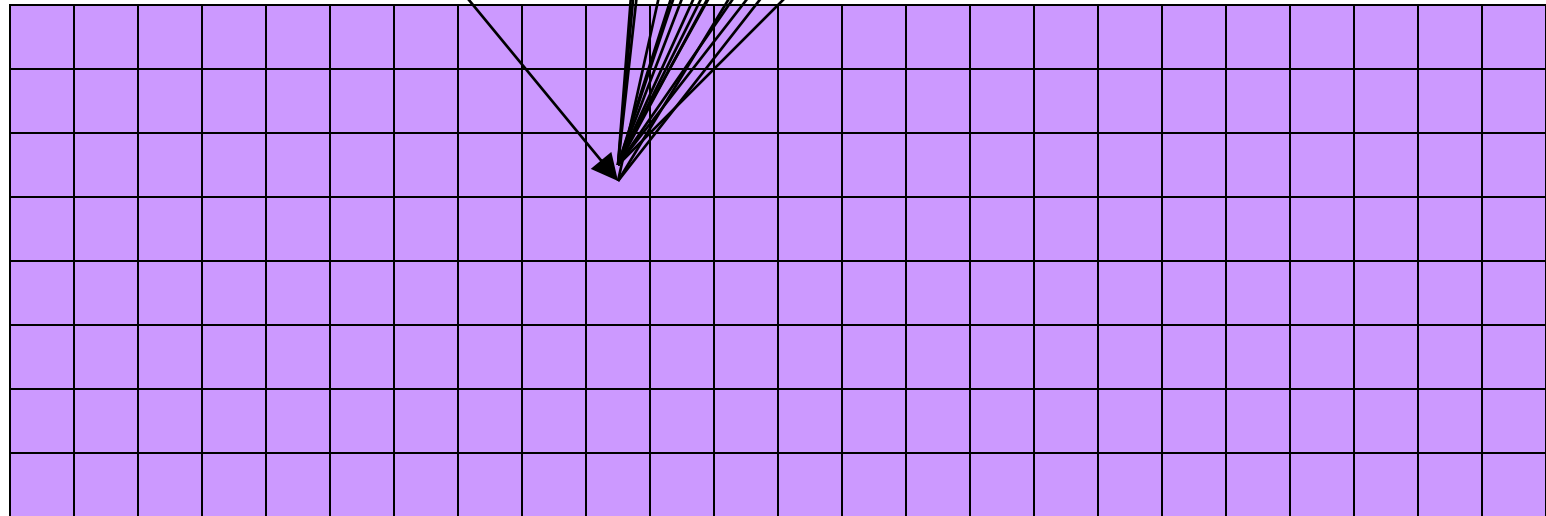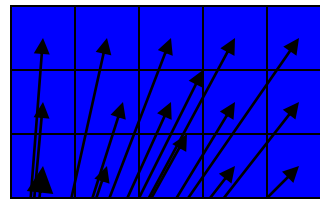
Fully Associative Illustration

Could go into
any cache line

Main Memory Address
0100101011100101

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 $K \cdot N_{\mathrm{RAM}}$ connections – but **more robust** than direct mapped.

2-Way Set Associative Illustration

Could go into
cache address
11100101

OR

Main Memory Address
0100101011100101

Could go into
cache address
01100101
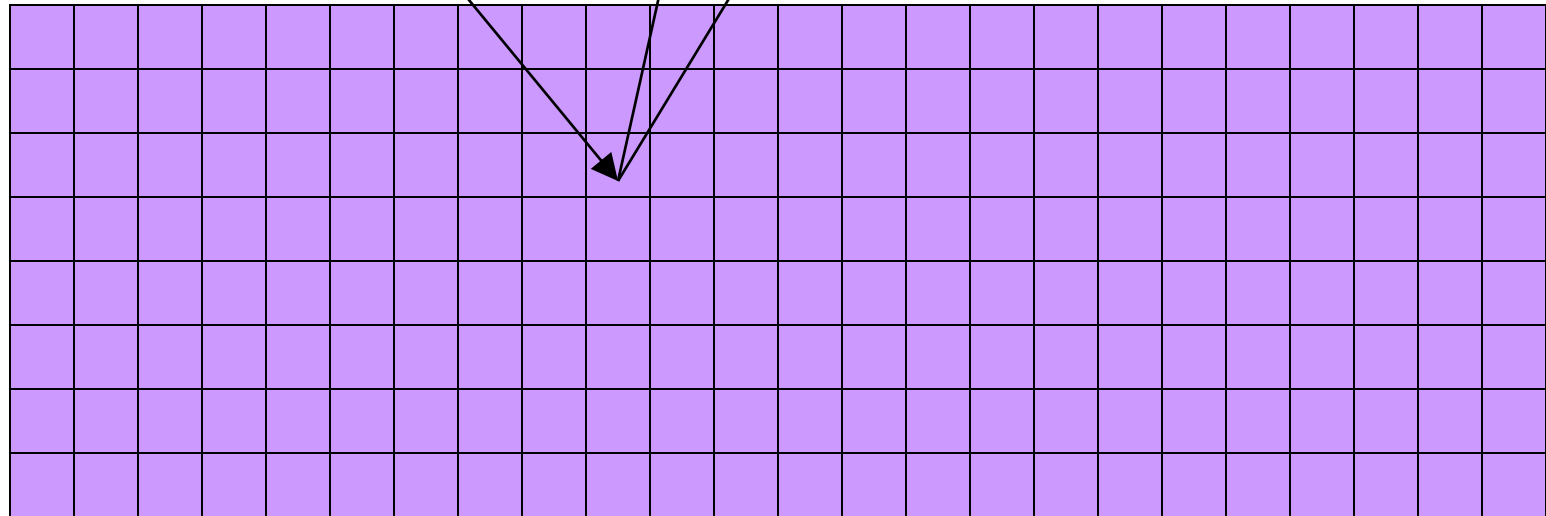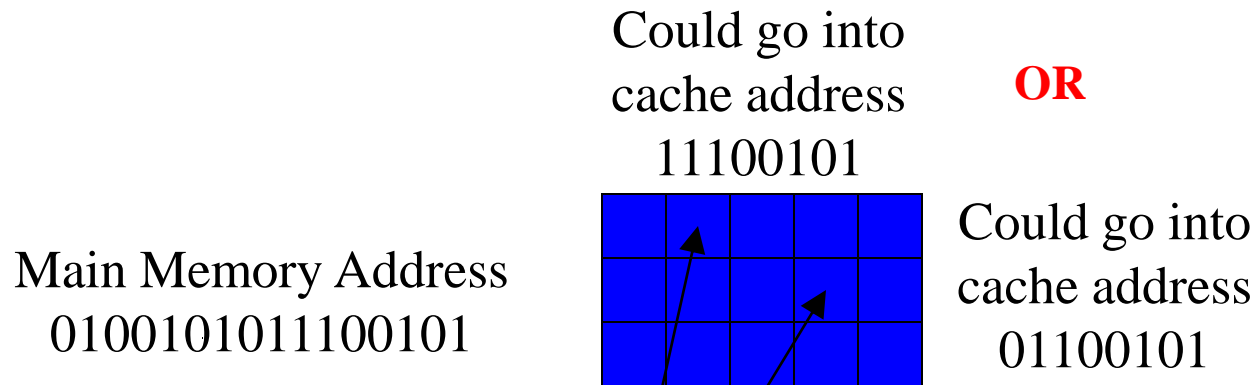
Cache Associativity Examples

- **Core 2 Duo** [26]
  - L1 data cache: 8-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
- **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

- ## **<u>Core 2 Duo</u>** [26]

  - ### L1 cache: write-back

- ## **<u>Pentium D</u>** [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:

<u>Location, Location, Location!</u>

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 **absolutely no data locality at all**:

```c
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 **absolutely no data locality at all**:

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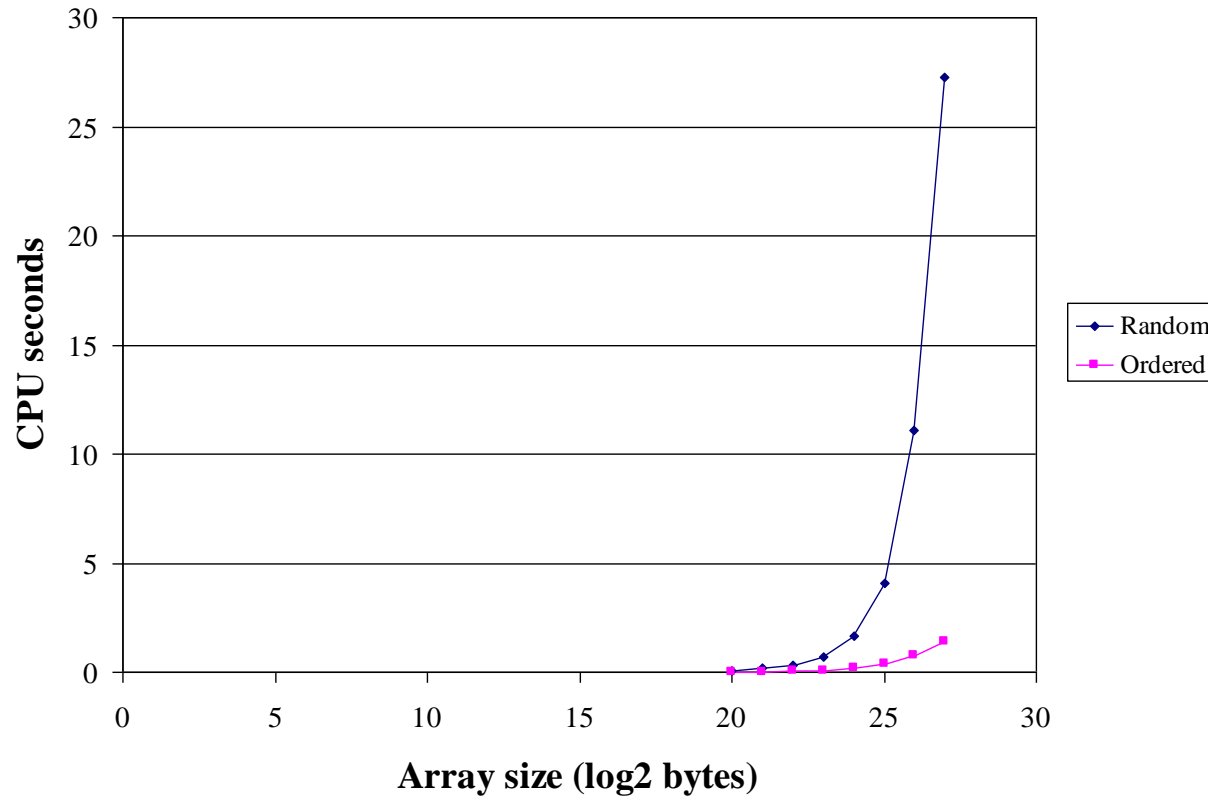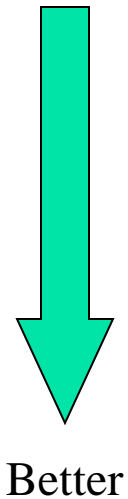
In a simple array fill, locality provides a factor of 8 to 20 speedup over a randomly ordered fill on a Pentium4.

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 $A = B \bullet C$ 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: F90

```fortran
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: C

```c
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: F90

```fortran
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
```
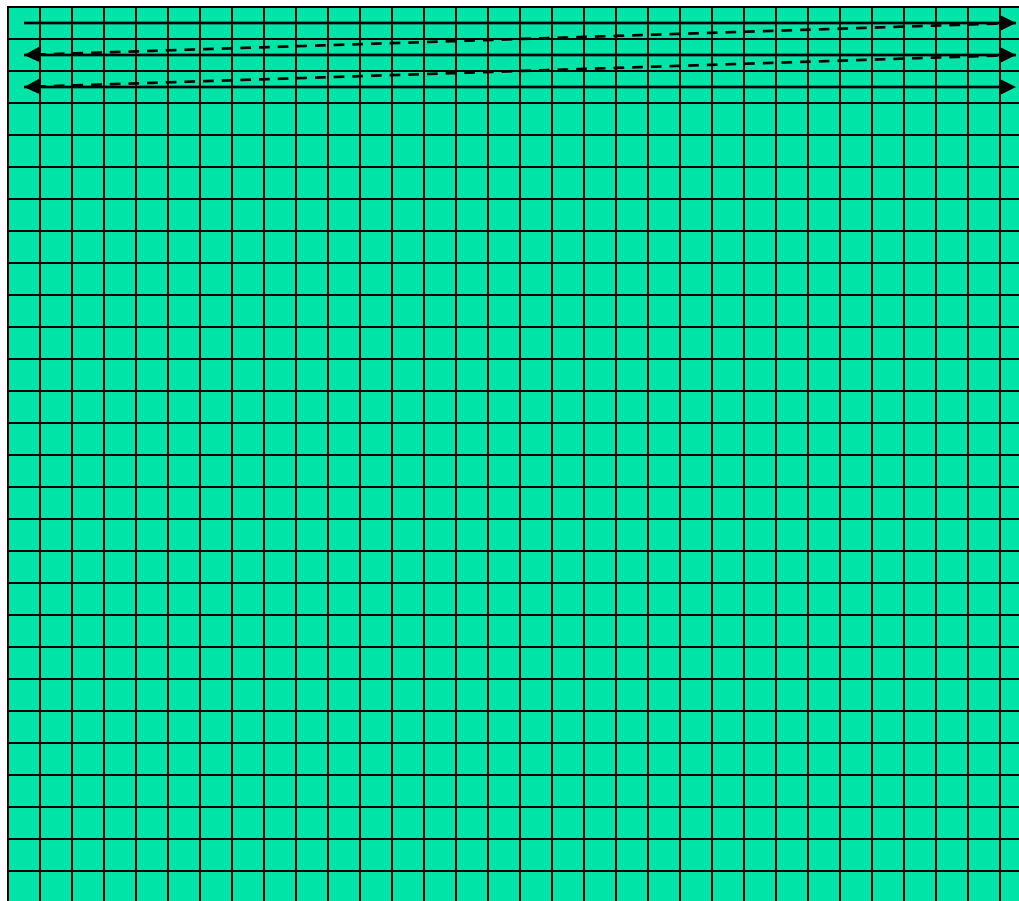
Matrix Multiply Behavior

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

Performance of Matrix Multiply
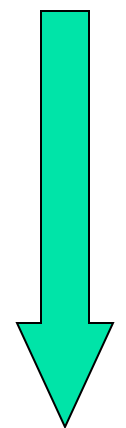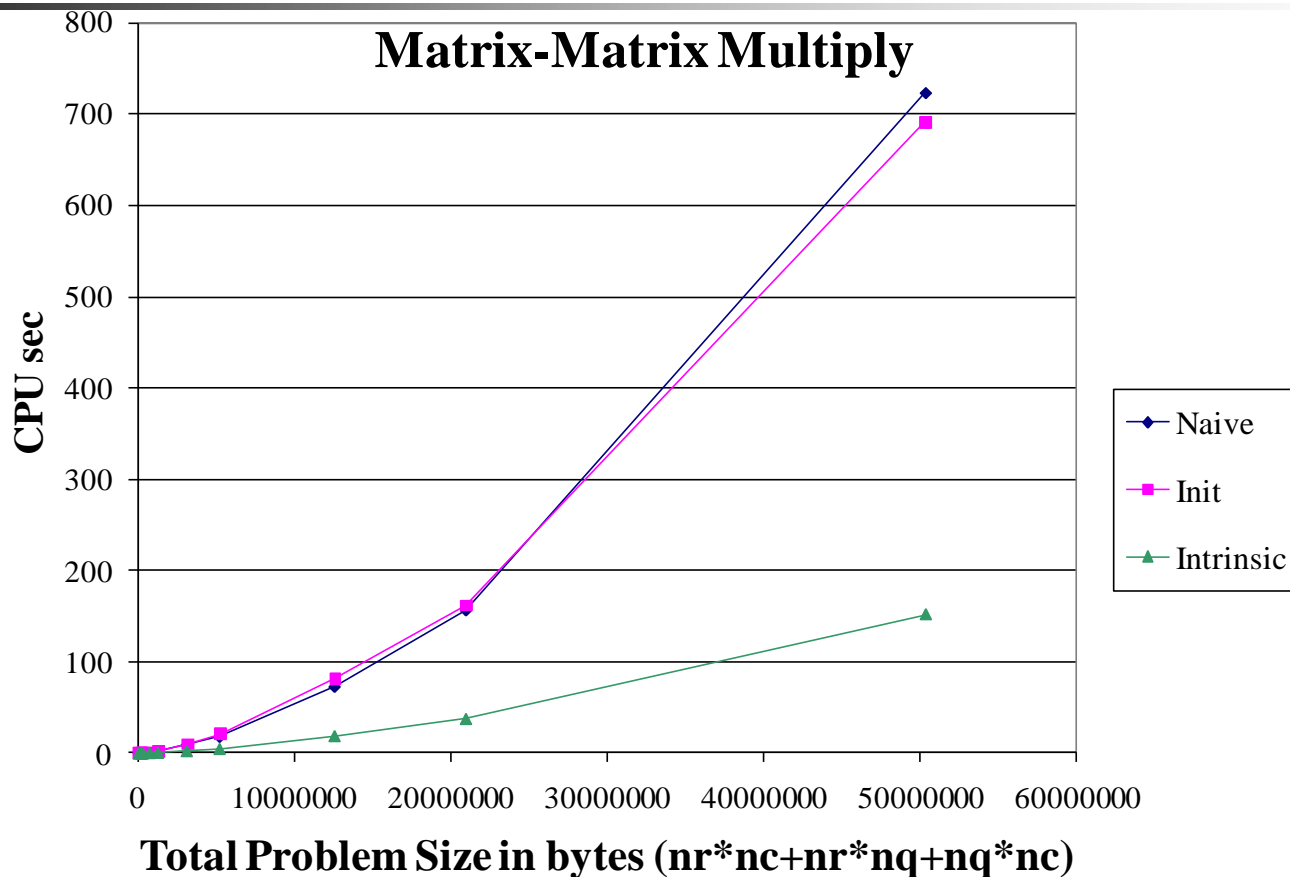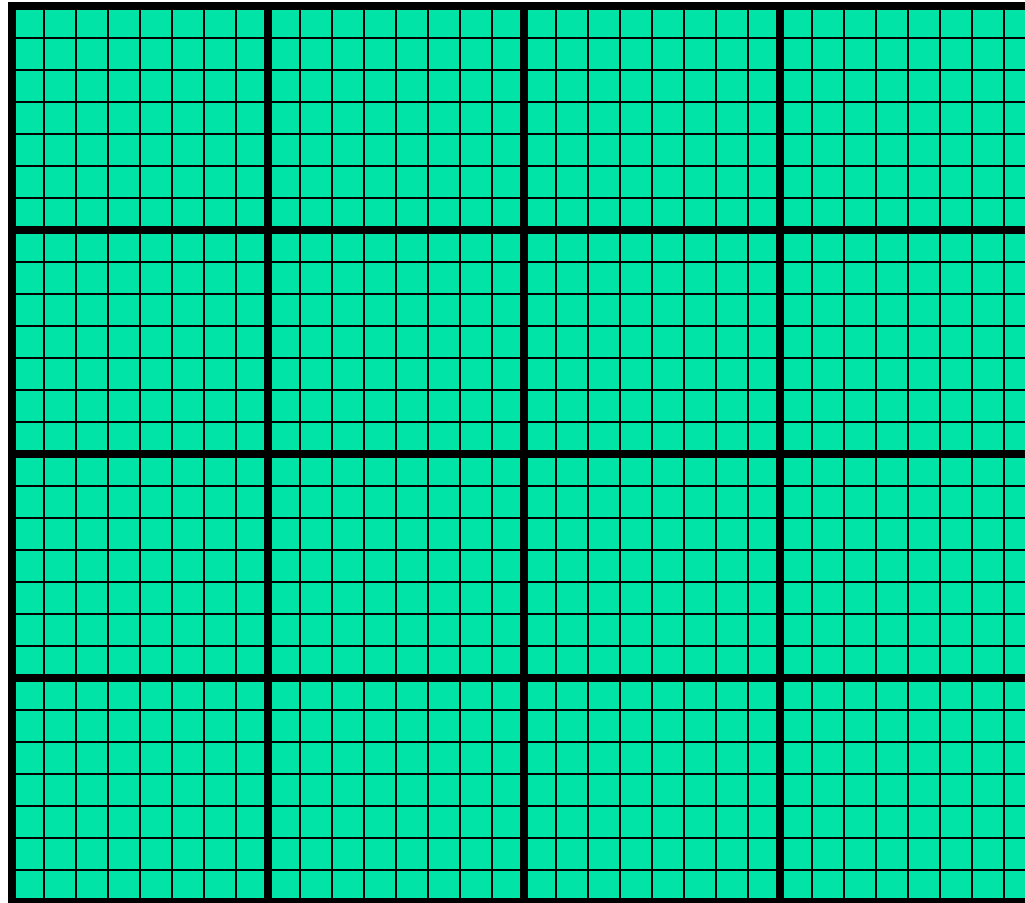
Tiling

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.

```fortran
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

```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

```fortran
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

```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 */
```
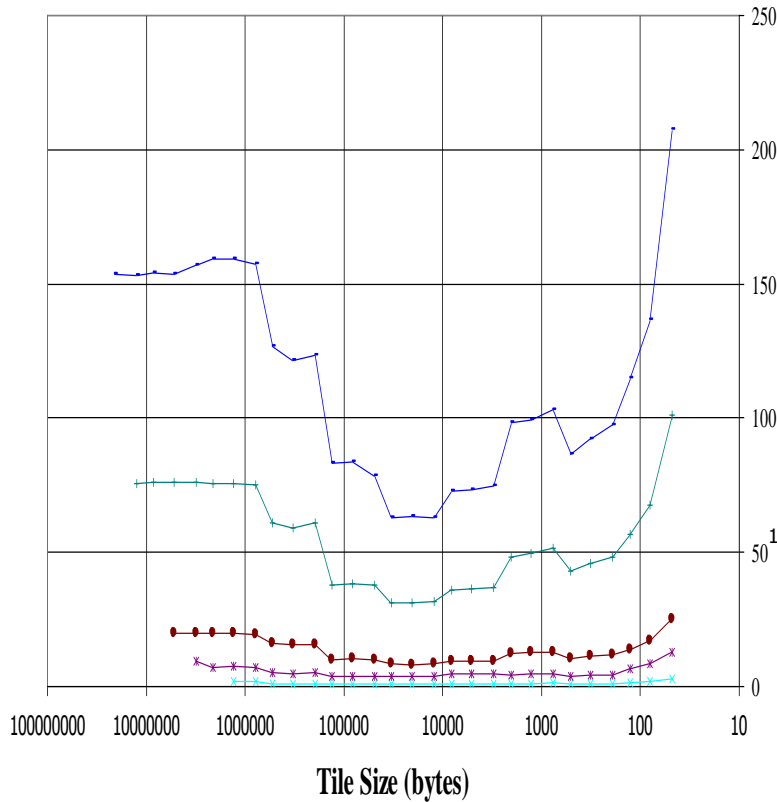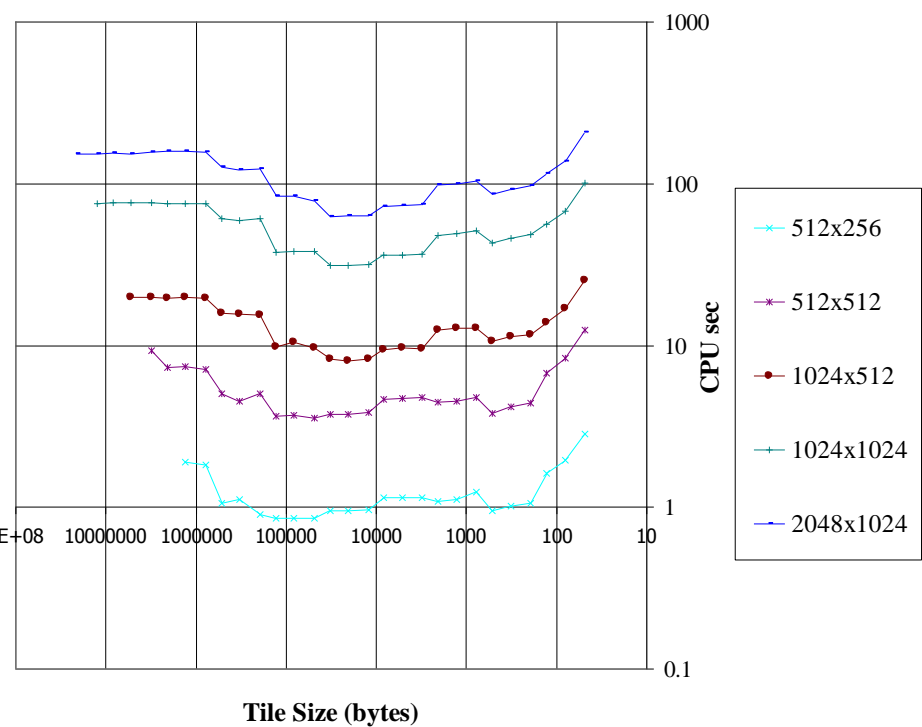
Performance with Tiling

Matrix-Matrix Mutiply Via Tiling

Matrix-Matrix Mutiply Via Tiling (log-log)

Better

512x256
512x512
1024x512
1024x1024
2048x1024

Tile Size (bytes)

Tile Size (bytes)

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 **WON'T** 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.
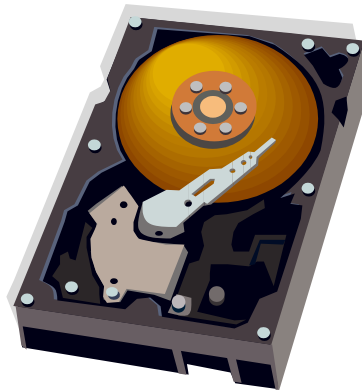
Hard Disk

Why Is Hard Disk Slow?

Your hard disk is **much much** slower than main memory (factor of 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**: 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:

```c
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:

```c
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:

```fortran
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:

```fortran
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 **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, consider a laptop that can address 16 GB of main memory (roughly 16 billion 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 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!**

Cache vs. Virtual Memory

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

Storage Use Strategies

- **<u>Register reuse</u>**: do a lot of work on the same data before working on new data.
- **<u>Cache reuse</u>**: 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).
- **<u>Data locality</u>**: try to access data that are near each other in memory before data that are far.
- **<u>I/O efficiency</u>**: do a bunch of I/O all at once rather than a little bit at a time; don't mix calculations and I/O.

OK Supercomputing Symposium 2012

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
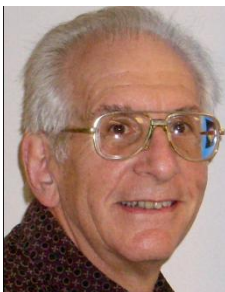Computing Center
U. Texas Austin

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

Thom Dunning, Director
National Center for Supercomputing
Applications

FREE! Wed Oct 3 2012 @ OU
http://symposium2012.oscer.ou.edu/

Reception/Poster Session
FREE! Tue Oct 2 2012 @ OU
FREE! Symposium Wed Oct 3 2012 @ OU

NCSI Parallel & Cluster: Storage Hierarchy
U Oklahoma, July 29 - Aug 4 2012

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.