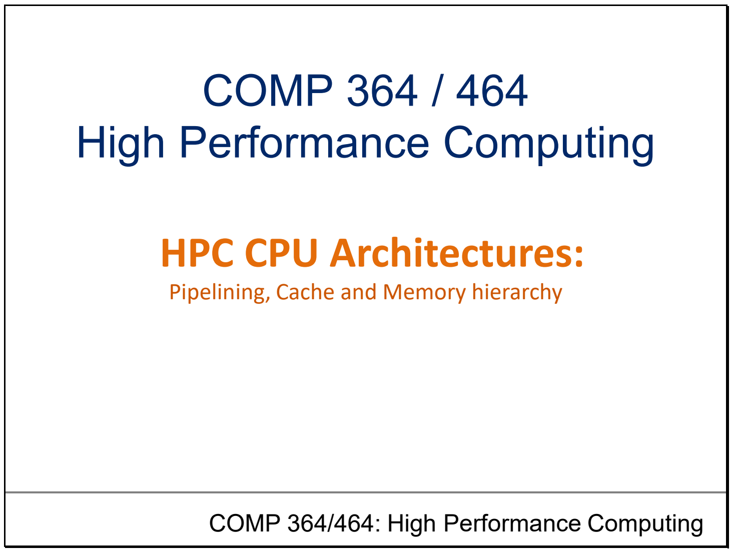
Slide 1



COMP 364 / 464 High Performance Computing

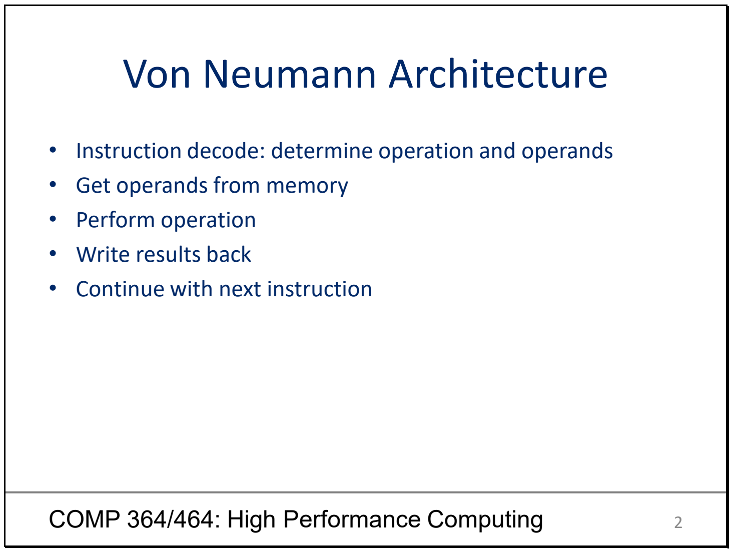
HPC CPU Architectures:

Pipelining, Cache and Memory hierarchy

COMP 364/464: High Performance Computing

This topic lays the groundwork for serious work in performance optimization. It covers performance-related hardware topics, especially caches and memory subsystems. The primary focus is on a single Central Processing Unit (CPU); the corresponding material for parallel systems (admittedly at a higher level) appears in L1 and L2.

Slide 2



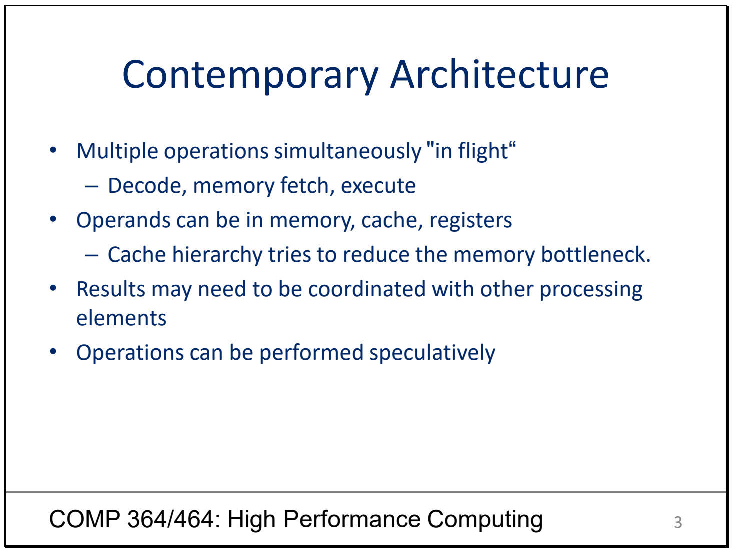
Von Neumann Architecture

- Instruction decode: determine operation and operands
- Get operands from memory
- Perform operation
- Write results back
- Continue with next instruction

COMP 364/464: High Performance Computing 2

Von Neumann: single data stream and single instruction stream. Everything in lock-step. Only one thing can happen at a time. Big bottleneck (at modern speeds) is waiting for memory operations.

Slide 3



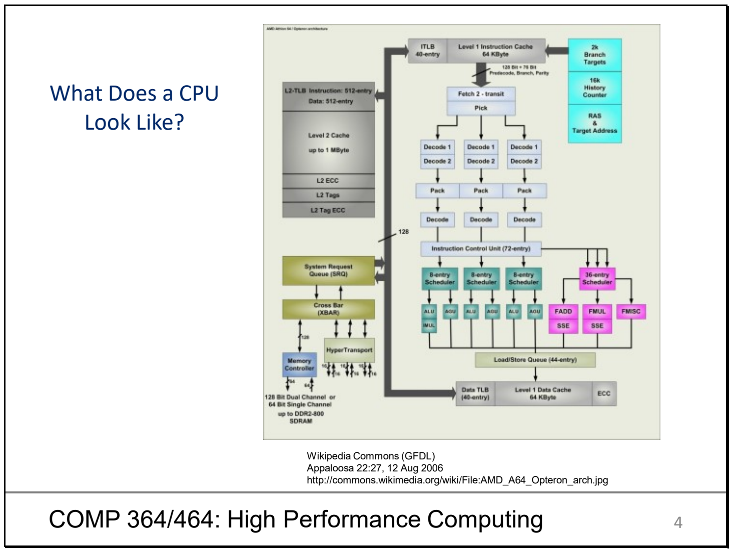
Contemporary Architecture

- Multiple operations simultaneously "in flight“
  - Decode, memory fetch, execute
- Operands can be in memory, cache, registers
  - Cache hierarchy tries to reduce the memory bottleneck.
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively

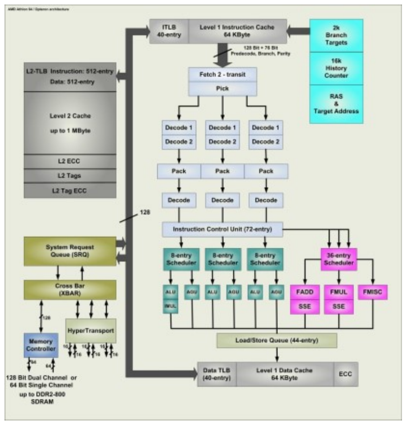
COMP 364/464: High Performance Computing 3

"Speculative" in this context means that operations may be completed before the results are needed; they are available in case they are needed, and are discarded otherwise. Some processors (e.g. the Xeon Phi KNC coprocessor) do not support this capability. Examples are evaluating both sides of a conditional if shallow. Prevents stall. Better to throw away a FLOP than stall the whole pipeline.

Slide 4



What Does a CPU Look Like?


AMD Athlon 64 / Opteron architecture
ITLB 40-entry
Level 1 Instruction Cache 64 KByte
2k Branch Targets
16k History Counter
RAS & Target Address
128 Bit + 76 Bit Predecode, Branch, Parity
Fetch 2 - transit
Pick
L2-TLB Instruction: 512-entry Data: 512-entry
Level 2 Cache up to 1 MByte
L2 ECC
L2 Tags
L2 Tag ECC
Decode 1
Decode 2
Pack
Decode
128
Instruction Control Unit (72-entry)
System Request Queue (SRQ)
Cross Bar (XBAR)
8-entry Scheduler
36-entry Scheduler
ALU
AGU
IMUL
FADD
FMUL
FMISC
SSE
HyperTransport
Memory Controller
Load/Store Queue (44-entry)
Data TLB (40-entry)
Level 1 Data Cache 64 KByte
ECC
128 Bit Dual Channel or 64 Bit Single Channel up to DDR2-800 SDRAM

Wikipedia Commons (GFDL)
Appaloosa 22:27, 12 Aug 2006
http://commons.wikimedia.org/wiki/File:AMD_A64_Opteron_arch.jpg

COMP 364/464: High Performance Computing 4

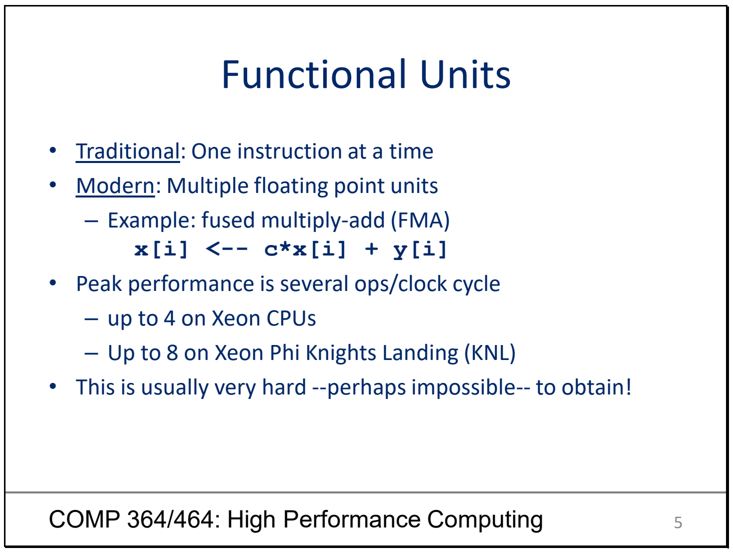
The slide shows a schematic of a single Opteron core. For a top-level view of six-core chip (labeled with overlays) see:

http://www.google.com/imgres?imgurl=http%3A%2F%2Fhothardware.com%2Fnewsimages%2FItem14534%2FMagny_Cours.jpg&imgrefurl=http%3A%2F%2Fhothardware.com%2FNews%2FAMD-Customers-Demand-More-Cores%2F&docid=SyBUCF97FQBUUM&tbnid=RcT0GpPCIS4CMM&w=500&h=344&ei=H067Ud2oFbO30QHwp4DwCg&ved=0CAkQxiAwBw&iact=rics

For a view of the chip without overlays see:

http://www.google.com/imgres?imgurl=http://www.dvhardware.net/news/amd_opteron_istanbul_die.jpg&imgrefurl=http://forums.overclockers.com.au/showthread.php?t=845484&docid=crWLbmeYL_uwuM&tbnid=-p4-JfRZUCRhxM&w=500&h=665&ei=H067Ud2oFbO30QHwp4DwCg&ved=0CAMQxiAwAQ&iact=rics

Slide 5



Functional Units

- Traditional: One instruction at a time
- Modern: Multiple floating point units
  - Example: fused multiply-add (FMA)

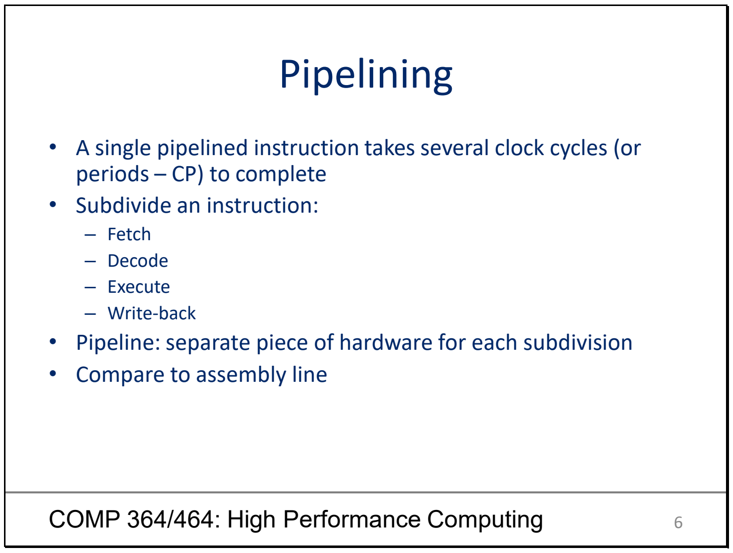
```
x[i] <-- c*x[i] + y[i]
```

- Peak performance is several ops/clock cycle
  - up to 4 on Xeon CPUs
  - Up to 8 on Xeon Phi Knights Landing (KNL)
- This is usually very hard --perhaps impossible-- to obtain!

COMP 364/464: High Performance Computing 5

See http://en.wikipedia.org/wiki/SAXPY for related material. In a classical SAXPY or DAXPY, the value of c is scalar. In the FMA implemented on the Intel Xeon Phi Vector Processor Unit (VPU), all three of c, x, and y are vectors. 4 and 8 on the Intel products is because of SIMD vector units.

Slide 6

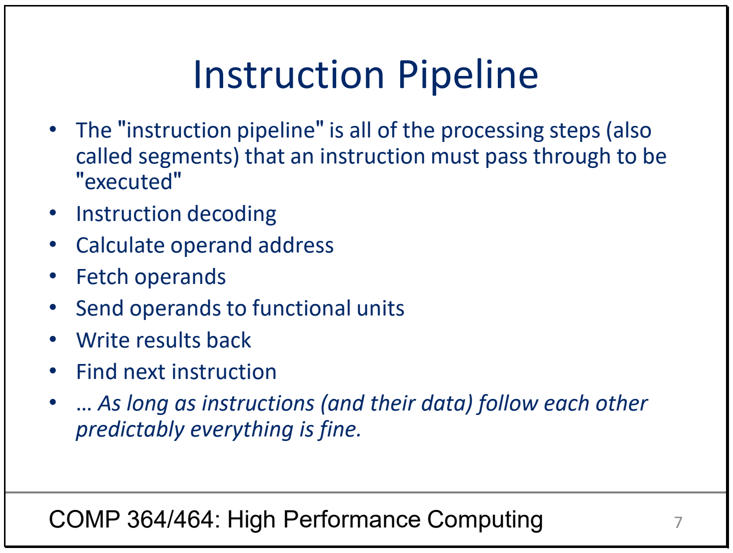


Pipelining

- A single pipelined instruction takes several clock cycles (or periods – CP) to complete
- Subdivide an instruction:
  - Fetch
  - Decode
  - Execute
  - Write-back
- Pipeline: separate piece of hardware for each subdivision
- Compare to assembly line

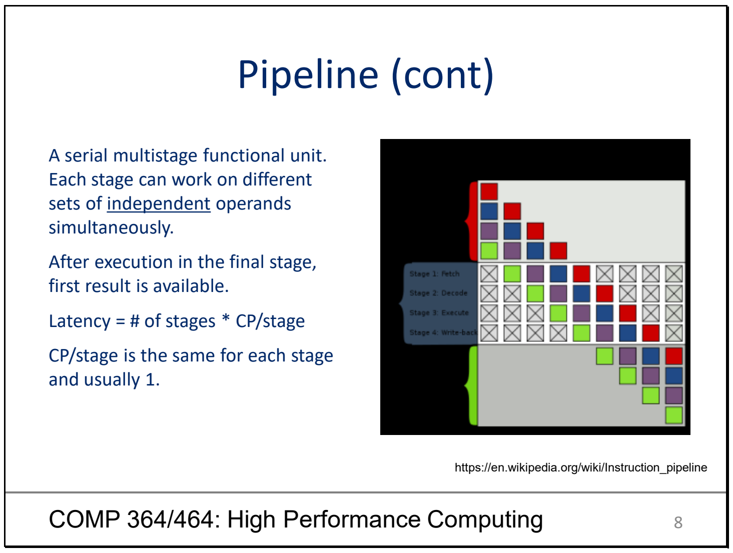
Each operation is handled by a separate (dedicated) piece of hardware.

Slide 7



Instruction Pipeline

- The "instruction pipeline" is all of the processing steps (also called segments) that an instruction must pass through to be "executed"
- Instruction decoding
- Calculate operand address
- Fetch operands
- Send operands to functional units
- Write results back
- Find next instruction
- *… As long as instructions (and their data) follow each other predictably everything is fine.*



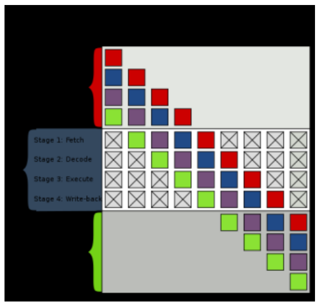
Pipeline (cont)

A serial multistage functional unit. Each stage can work on different sets of independent operands simultaneously.

After execution in the final stage, first result is available.

Latency = # of stages * CP/stage

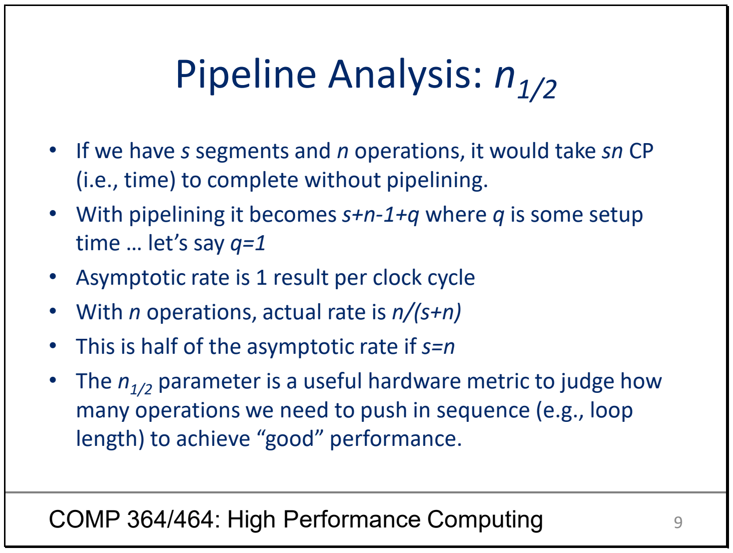
CP/stage is the same for each stage and usually 1.



https://en.wikipedia.org/wiki/Instruction_pipeline

CP = clock period or cycle period … 1/(clock frequency)

Slide 9



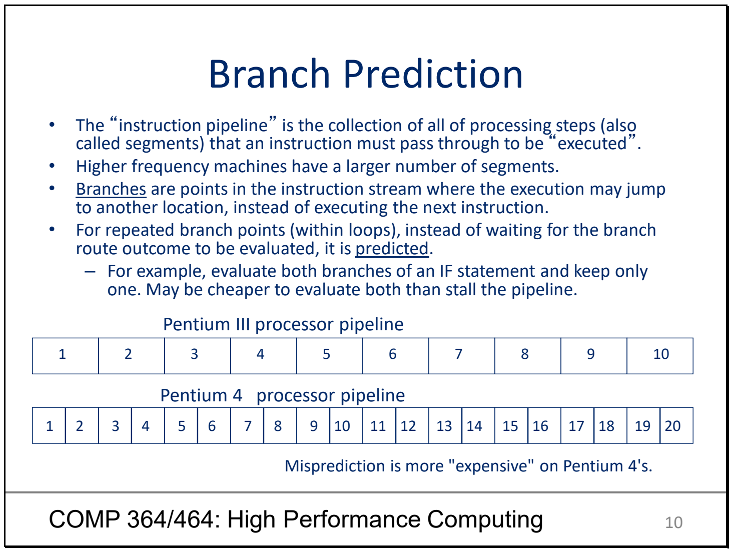
Pipeline Analysis: $n_{1/2}$

- If we have *s* segments and *n* operations, it would take *sn* CP (i.e., time) to complete without pipelining.
- With pipelining it becomes $s+n-1+q$ where *q* is some setup time … let's say $q=1$
- Asymptotic rate is 1 result per clock cycle
- With *n* operations, actual rate is $n/(s+n)$
- This is half of the asymptotic rate if $s=n$
- The $n_{1/2}$ parameter is a useful hardware metric to judge how many operations we need to push in sequence (e.g., loop length) to achieve "good" performance.

COMP 364/464: High Performance Computing 9

n_1/2 is used as a description of the available parallelism and efficiency of the architecture and the algorithm running.

Slide 10



Branch Prediction

- The "instruction pipeline" is the collection of all of processing steps (also called segments) that an instruction must pass through to be "executed".
- Higher frequency machines have a larger number of segments.
- Branches are points in the instruction stream where the execution may jump to another location, instead of executing the next instruction.
- For repeated branch points (within loops), instead of waiting for the branch route outcome to be evaluated, it is predicted.
  - For example, evaluate both branches of an IF statement and keep only one. May be cheaper to evaluate both than stall the pipeline.

Pentium III processor pipeline

1	2	3	4	5	6	7	8	9	10

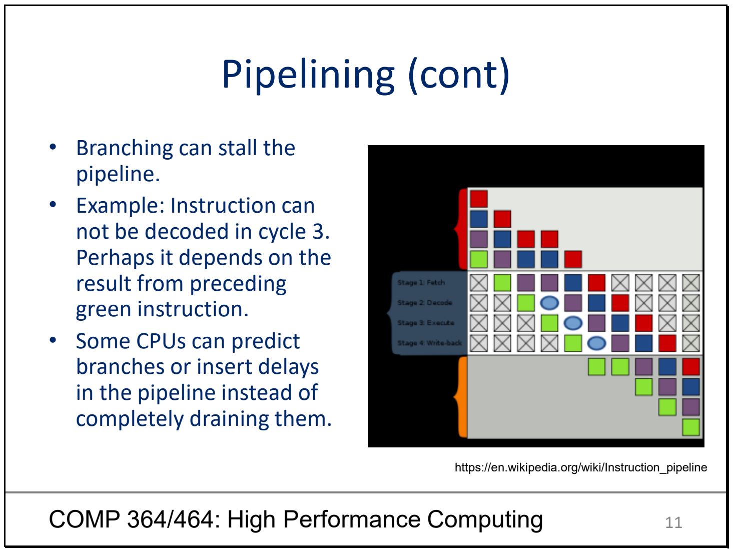
Pentium 4 processor pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Misprediction is more "expensive" on Pentium 4's.

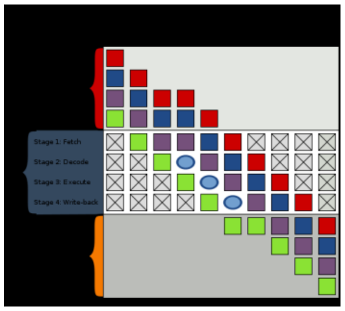
The deeper the pipeline, the more costly is a stall or mis-prediction. Modern Xeon cores have 14-19 stage pipelines. Diminishing return on depth.

Slide 11



Pipelining (cont)

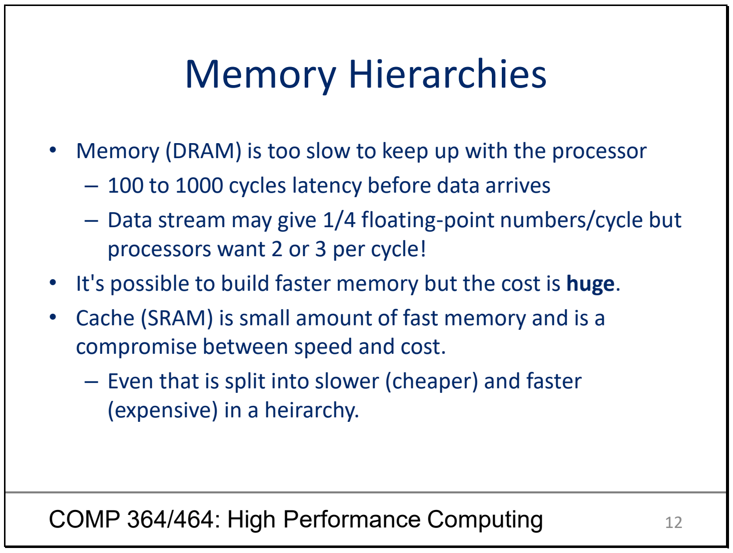
- Branching can stall the pipeline.
- Example: Instruction can not be decoded in cycle 3. Perhaps it depends on the result from preceding green instruction.
- Some CPUs can predict branches or insert delays in the pipeline instead of completely draining them.



https://en.wikipedia.org/wiki/Instruction_pipeline

Interesting graphic depiction of instruction pipelining with a some problem decoding the instruction in cycle 3. Perhaps it depends on the result from the green instruction so it can’t proceed.

Slide 12



Memory Hierarchies

- Memory (DRAM) is too slow to keep up with the processor
  - 100 to 1000 cycles latency before data arrives
  - Data stream may give 1/4 floating-point numbers/cycle but processors want 2 or 3 per cycle!
- It's possible to build faster memory but the cost is **huge**.
- Cache (SRAM) is small amount of fast memory and is a compromise between speed and cost.
  - Even that is split into slower (cheaper) and faster (expensive) in a heirarchy.

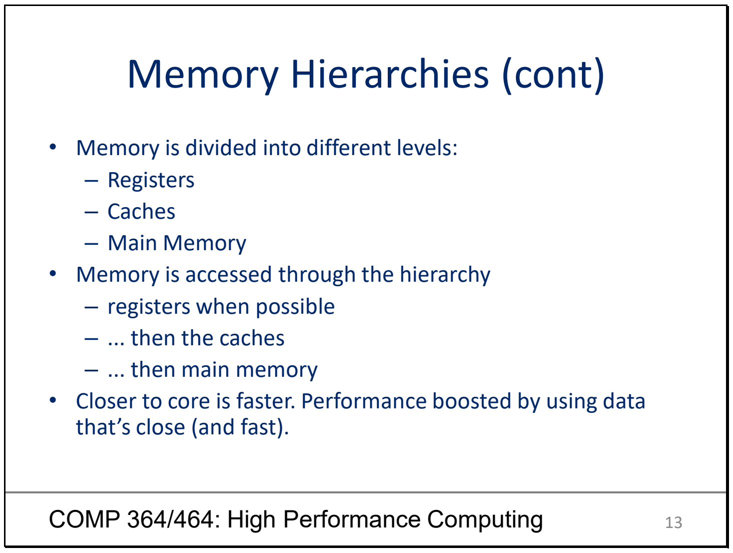
COMP 364/464: High Performance Computing 12

An oversimplified characterization of this slide's message: "flops are free". The spirit of this admittedly exaggerated expression: computations take place so fast these days that it's very hard to feed them data fast enough to keep them satisfied. The long pole in the tent is no longer flops -- it's more about data locality and movement.

FPUs need 2-3 FP/cycle for basic operations such as y <- c*x + y

FLOP/word is a useful metric to determine the FP intensity of an algorithm. Higher intensity means less reliant on memory bandwidth and generally gives better efficiency.

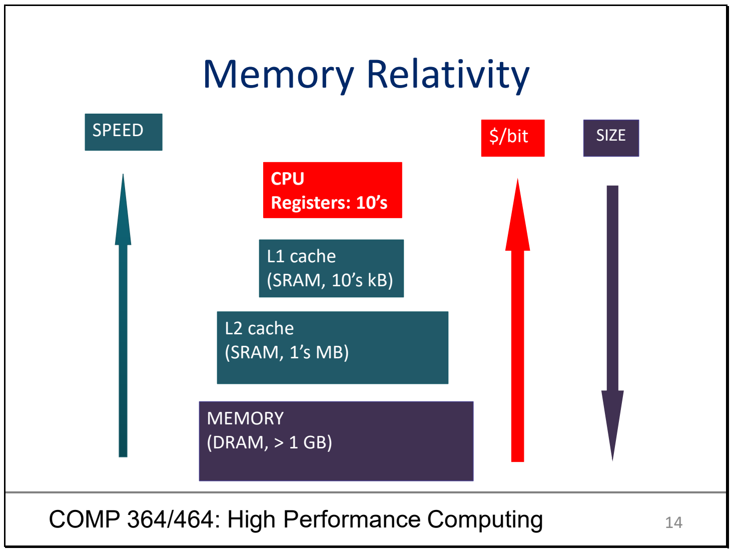
Slide 13



Memory Hierarchies (cont)

- Memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory
- Memory is accessed through the hierarchy
  - registers when possible
  - ... then the caches
  - ... then main memory
- Closer to core is faster. Performance boosted by using data that's close (and fast).

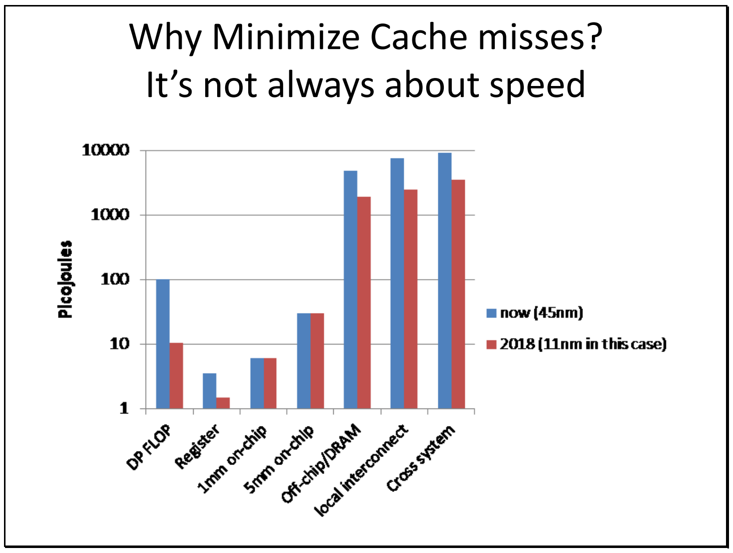
Slide 14



Memory Relativity

L3 is common on multi-core platforms. This is shared across all cores.

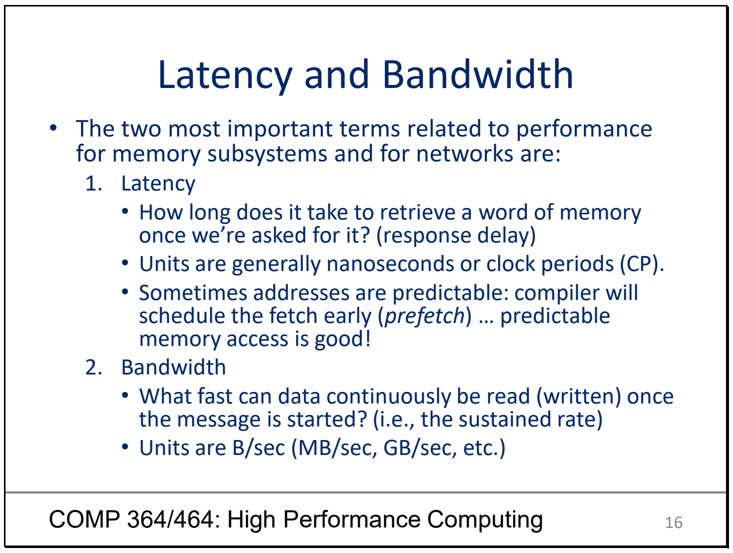
Slide 15



Why Minimize Cache misses? It's not always about speed

Broadwell Xeon (last commercial core) uses 14nm.

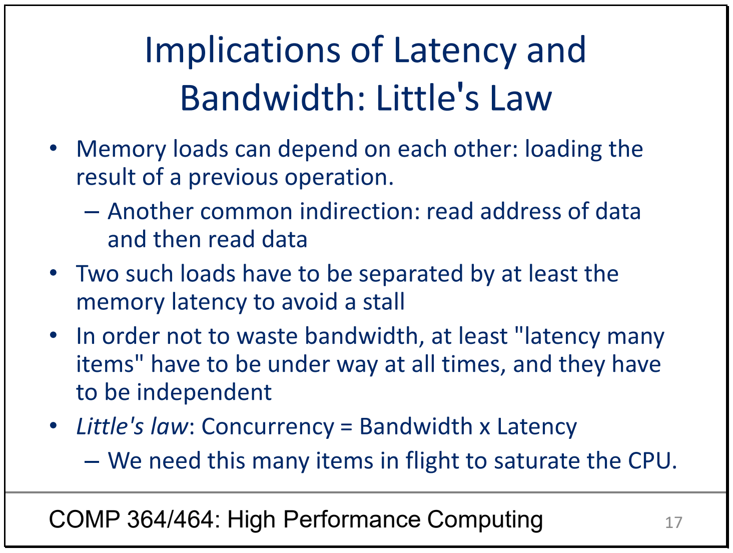
Slide 16



Latency and Bandwidth

- The two most important terms related to performance for memory subsystems and for networks are:
  1. Latency
     - How long does it take to retrieve a word of memory once we're asked for it? (response delay)
     - Units are generally nanoseconds or clock periods (CP).
     - Sometimes addresses are predictable: compiler will schedule the fetch early (*prefetch*) … predictable memory access is good!
  2. Bandwidth
     - What fast can data continuously be read (written) once the message is started? (i.e., the sustained rate)
     - Units are B/sec (MB/sec, GB/sec, etc.)

Slide 17



Implications of Latency and Bandwidth: Little's Law

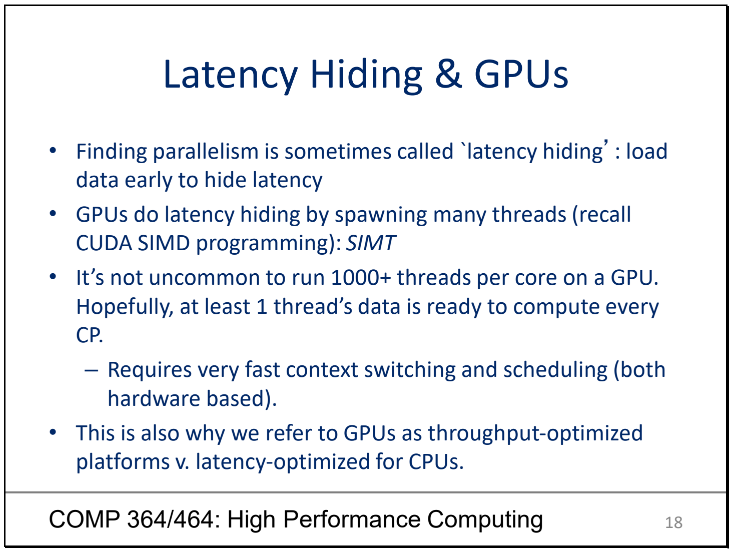
- Memory loads can depend on each other: loading the result of a previous operation.
  - Another common indirection: read address of data and then read data
- Two such loads have to be separated by at least the memory latency to avoid a stall
- In order not to waste bandwidth, at least "latency many items" have to be under way at all times, and they have to be independent
- *Little's law*: Concurrency = Bandwidth x Latency
  - We need this many items in flight to saturate the CPU.

COMP 364/464: High Performance Computing 17

Little's Law is actually a well known result in queuing theory. See http://www.davidhbailey.com/dhbpapers/little.pdf for a nice discussion. A potentially helpful and enjoyable non-computational metaphor for Little's Law appears in http://web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf.

That states L = (lambda) W:
L = average # of items in the queue.
Lambda = average # of items arriving per unit time
W = average wait time for an item

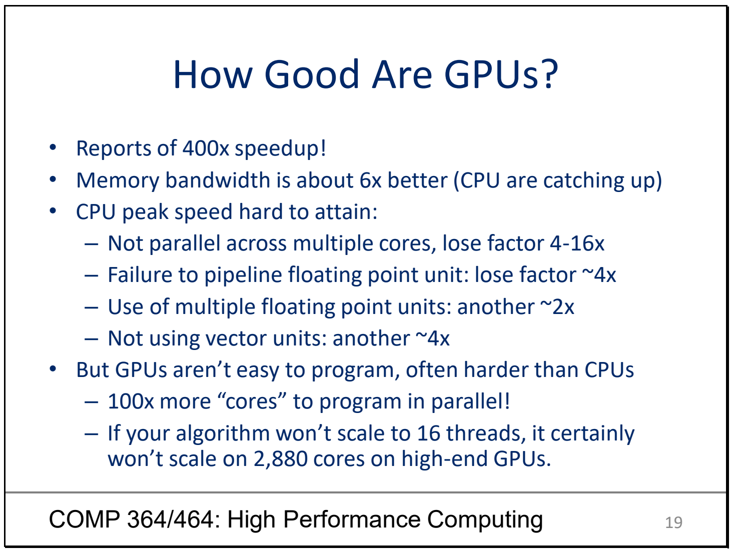
Quick way to estimate how many memory operations must be ‘in flight’ in order to saturate the CPU. Depends on the algorithm and the hardware. Good way to see if your algorithm is efficient.



Latency Hiding & GPUs

- Finding parallelism is sometimes called `latency hiding' : load data early to hide latency
- GPUs do latency hiding by spawning many threads (recall CUDA SIMD programming): *SIMT*
- It's not uncommon to run 1000+ threads per core on a GPU. Hopefully, at least 1 thread's data is ready to compute every CP.
  - Requires very fast context switching and scheduling (both hardware based).
- This is also why we refer to GPUs as throughput-optimized platforms v. latency-optimized for CPUs.

Slide 19



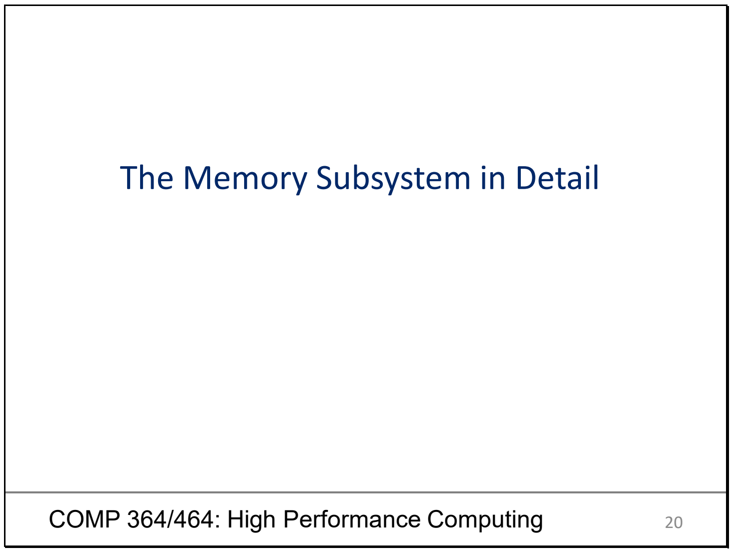
How Good Are GPUs?

- Reports of 400x speedup!
- Memory bandwidth is about 6x better (CPU are catching up)
- CPU peak speed hard to attain:
  - Not parallel across multiple cores, lose factor 4-16x
  - Failure to pipeline floating point unit: lose factor ~4x
  - Use of multiple floating point units: another ~2x
  - Not using vector units: another ~4x
- But GPUs aren't easy to program, often harder than CPUs
  - 100x more "cores" to program in parallel!
  - If your algorithm won't scale to 16 threads, it certainly won't scale on 2,880 cores on high-end GPUs.

COMP 364/464: High Performance Computing 19

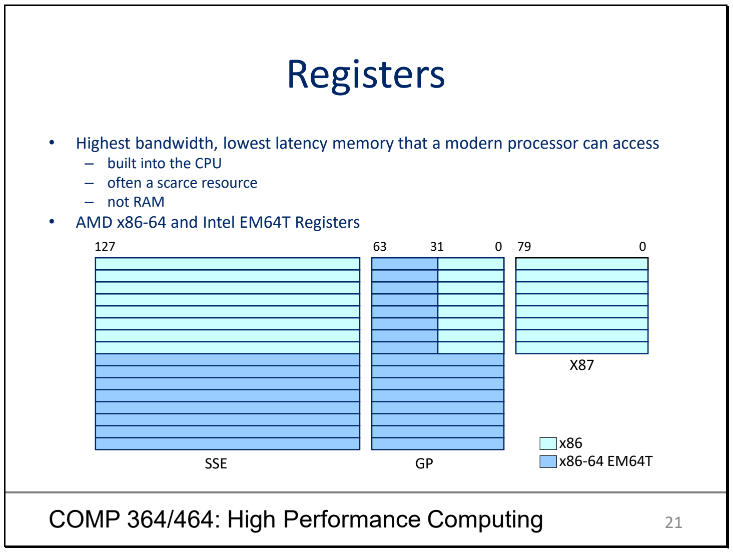
Regarding "[r]eports of 400x speedup" -- critics sometimes argue that these claims are misleading. The argument is that extreme multiples are comparing optimized GPU code vs. non-optimized host code (because the GPU optimizations do not benefit the non-GPU host code). In contrast, the process of optimizing code for the Xeon Phi MIC coprocessor usually benefits the performance of code on the host.

Slide 20



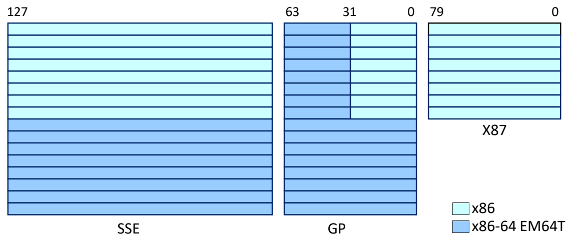
The Memory Subsystem in Detail

Slide 21



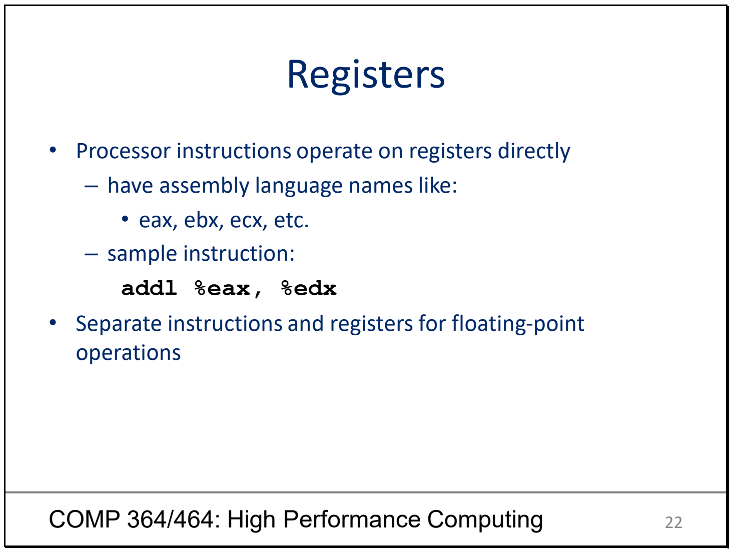
Registers

- Highest bandwidth, lowest latency memory that a modern processor can access
  - built into the CPU
  - often a scarce resource
  - not RAM
- AMD x86-64 and Intel EM64T Registers



x87 is float-point unit (FPU).
GP = general purpose.
SSE = vector registers.

Slide 22



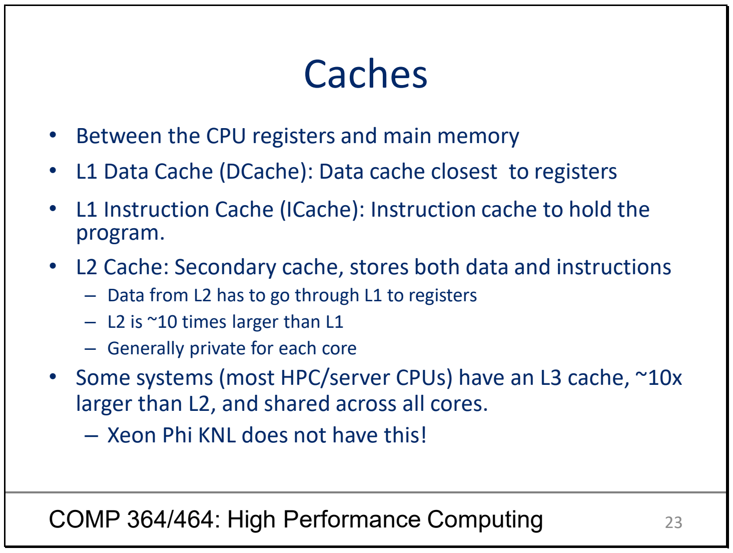
Registers

- Processor instructions operate on registers directly
  - have assembly language names like:
    - eax, ebx, ecx, etc.
  - sample instruction:

```
addl %eax, %edx
```

- Separate instructions and registers for floating-point operations

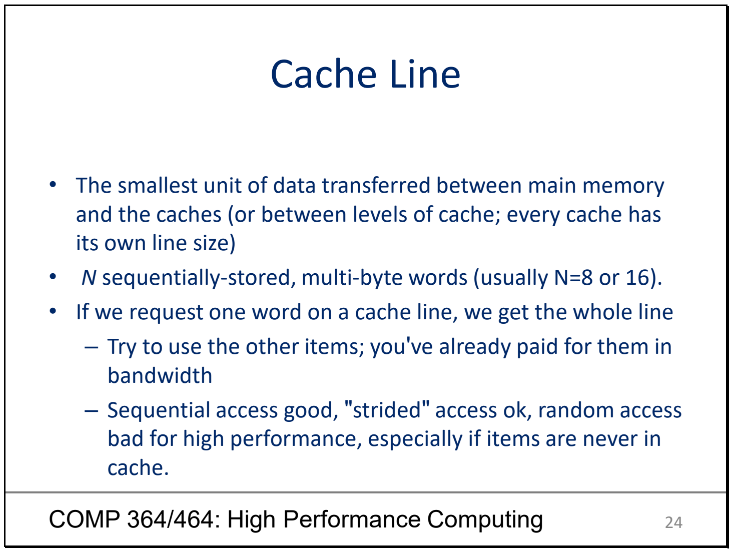
Slide 23



Caches

- Between the CPU registers and main memory
- L1 Data Cache (DCache): Data cache closest to registers
- L1 Instruction Cache (ICache): Instruction cache to hold the program.
- L2 Cache: Secondary cache, stores both data and instructions
  - Data from L2 has to go through L1 to registers
  - L2 is ~10 times larger than L1
  - Generally private for each core
- Some systems (most HPC/server CPUs) have an L3 cache, ~10x larger than L2, and shared across all cores.
  - Xeon Phi KNL does not have this!

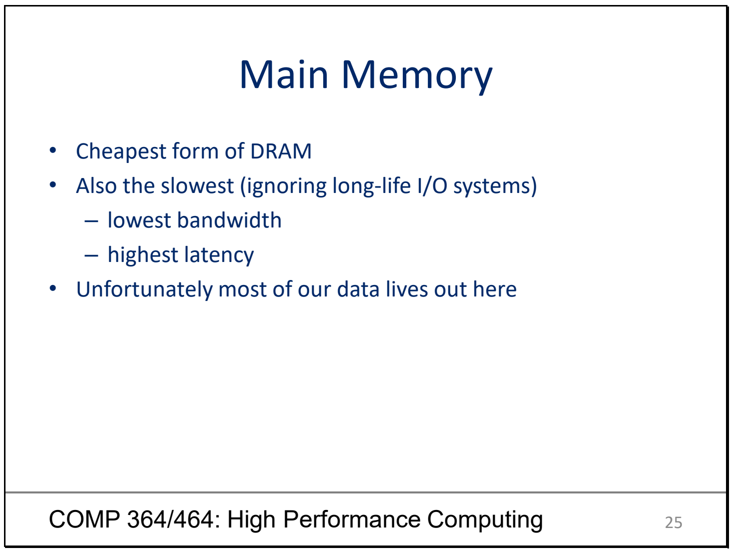
Slide 24



Cache Line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache; every cache has its own line size)
- *N* sequentially-stored, multi-byte words (usually N=8 or 16).
- If we request one word on a cache line, we get the whole line
  - Try to use the other items; you've already paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad for high performance, especially if items are never in cache.

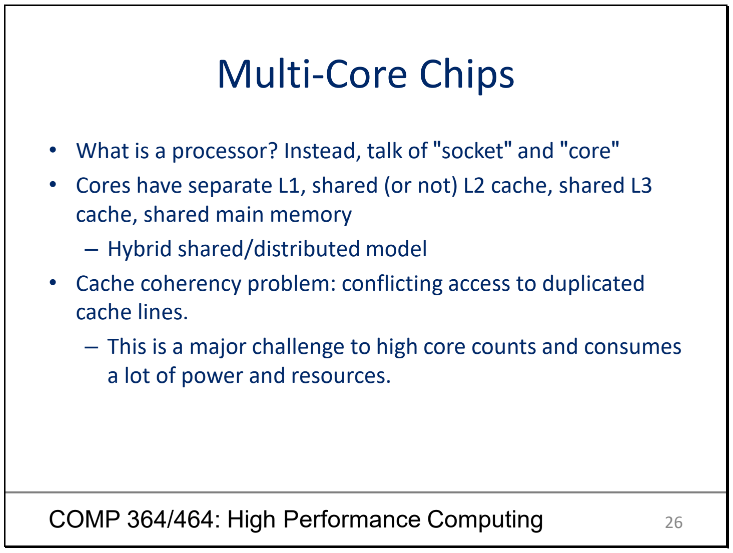
Slide 25



Main Memory

- Cheapest form of DRAM
- Also the slowest (ignoring long-life I/O systems)
  - lowest bandwidth
  - highest latency
- Unfortunately most of our data lives out here

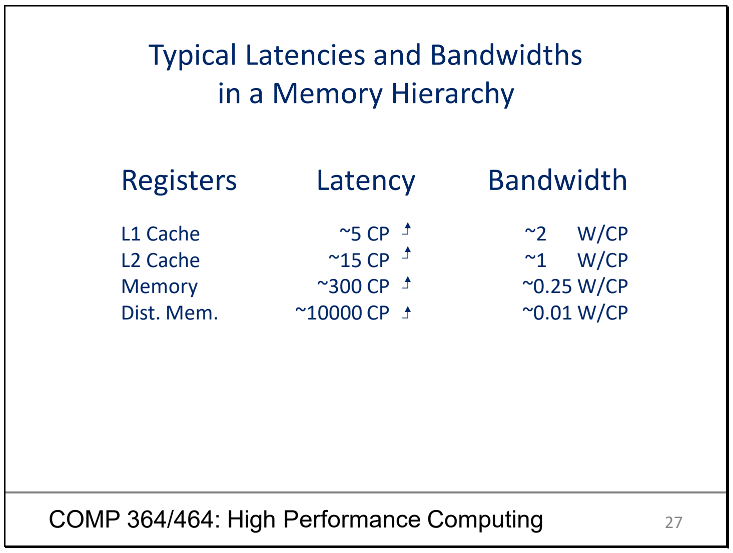
Slide 26



Multi-Core Chips

- What is a processor? Instead, talk of "socket" and "core"
- Cores have separate L1, shared (or not) L2 cache, shared L3 cache, shared main memory
  - Hybrid shared/distributed model
- Cache coherency problem: conflicting access to duplicated cache lines.
  - This is a major challenge to high core counts and consumes a lot of power and resources.

Slide 27

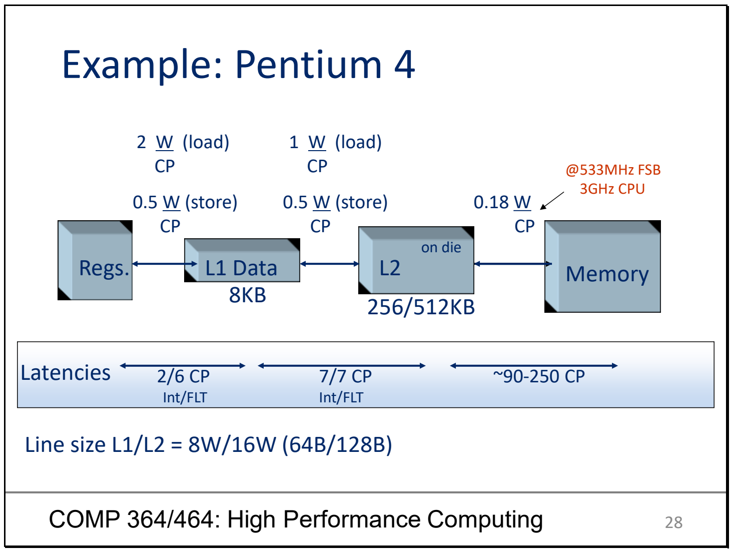


Typical Latencies and Bandwidths in a Memory Hierarchy

Registers	Latency	Bandwidth
L1 Cache	~5 CP	~2 W/CP
L2 Cache	~15 CP	~1 W/CP
Memory	~300 CP	~0.25 W/CP
Dist. Mem.	~10000 CP	~0.01 W/CP

"CP" means "clock periods" or cycles. .. The CPU frequency.

Slide 28



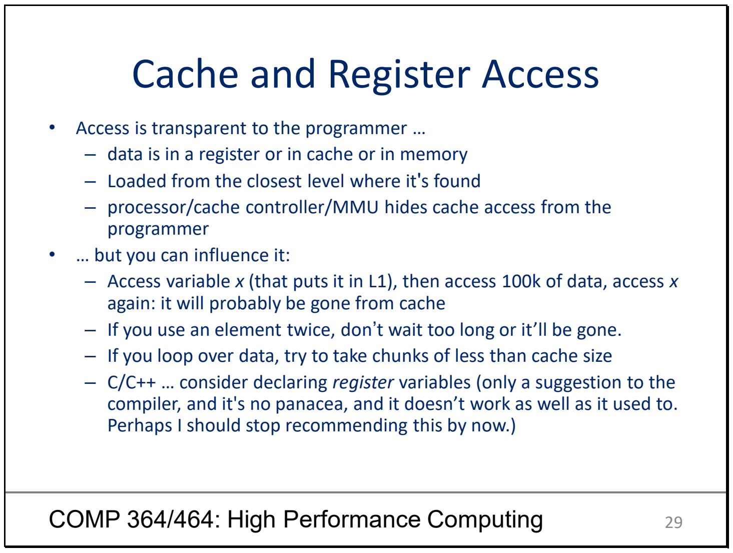
Example: Pentium 4

Line size L1/L2 = 8W/16W (64B/128B)

What is the width of these connections?

"FSB" means "Front-Side Bus" -- it connects main memory to L2 cache. There are other equivalent terms (and FSB tends to be Intel-specific). See http://en.wikipedia.org/wiki/Front-side_bus for more info. It's rather obsolete now but the 'new tech' is not much different.

Slide 29

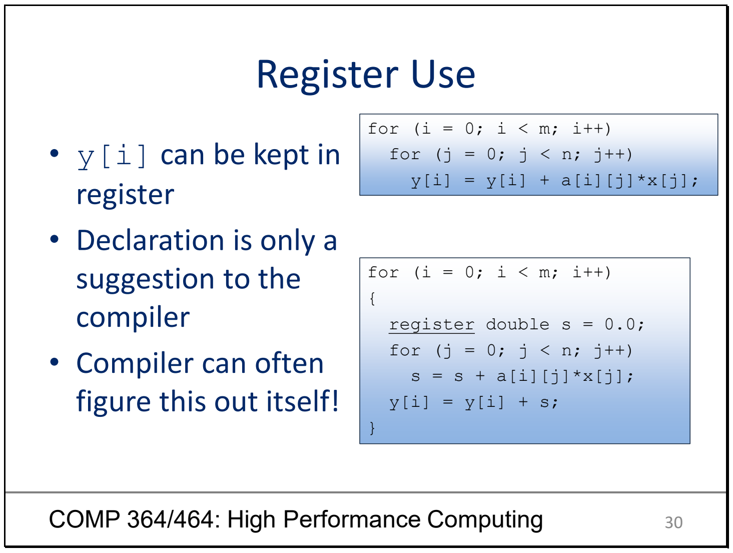


Cache and Register Access

- Access is transparent to the programmer …
  - data is in a register or in cache or in memory
  - Loaded from the closest level where it's found
  - processor/cache controller/MMU hides cache access from the programmer
- … but you can influence it:
  - Access variable *x* (that puts it in L1), then access 100k of data, access *x* again: it will probably be gone from cache
  - If you use an element twice, don't wait too long or it'll be gone.
  - If you loop over data, try to take chunks of less than cache size
  - C/C++ … consider declaring *register* variables (only a suggestion to the compiler, and it's no panacea, and it doesn't work as well as it used to. Perhaps I should stop recommending this by now.)

memory management unit (MMU)

Slide 30

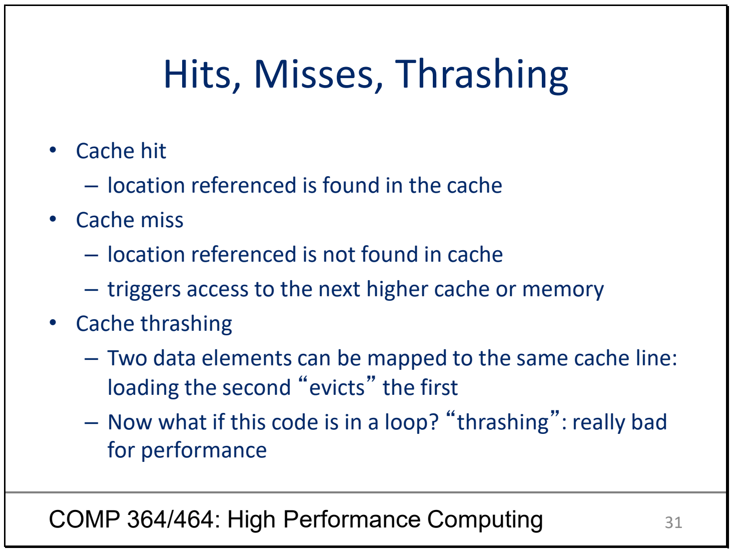


Register Use

- `y[i]` can be kept in register
- Declaration is only a suggestion to the compiler
- Compiler can often figure this out itself!

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    y[i] = y[i] + a[i][j]*x[j];
```

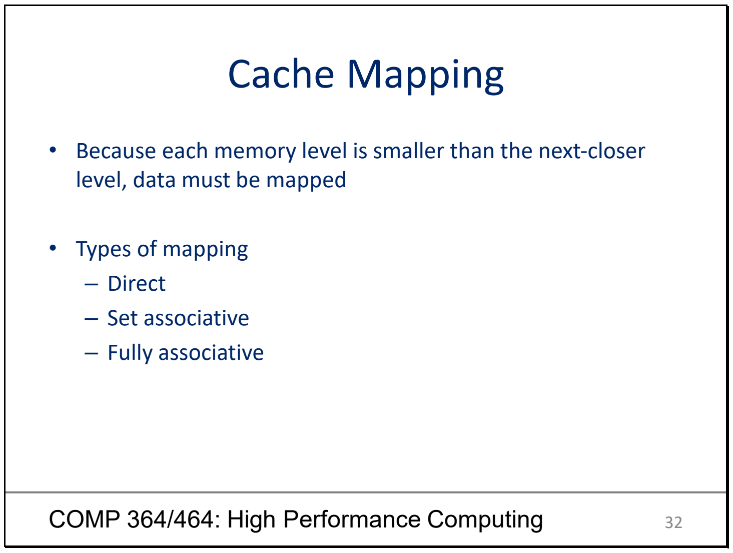
```
for (i = 0; i < m; i++)
{
  register double s = 0.0;
  for (j = 0; j < n; j++)
    s = s + a[i][j]*x[j];
  y[i] = y[i] + s;
}
```



Hits, Misses, Thrashing

- Cache hit
  - location referenced is found in the cache
- Cache miss
  - location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - Two data elements can be mapped to the same cache line: loading the second “evicts” the first
  - Now what if this code is in a loop? “thrashing”: really bad for performance

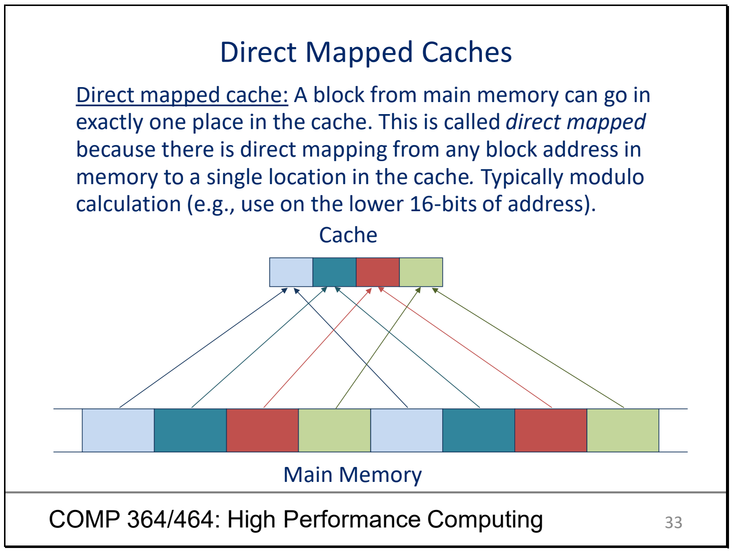
Slide 32



Cache Mapping

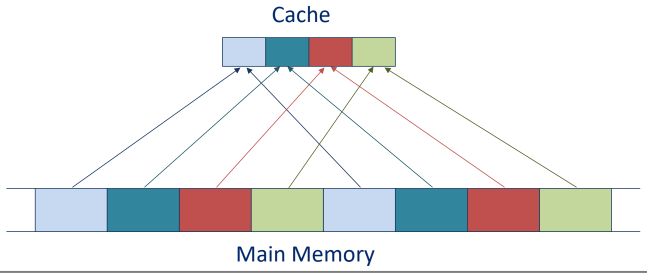
- Because each memory level is smaller than the next-closer level, data must be mapped
- Types of mapping
  - Direct
  - Set associative
  - Fully associative

Slide 33

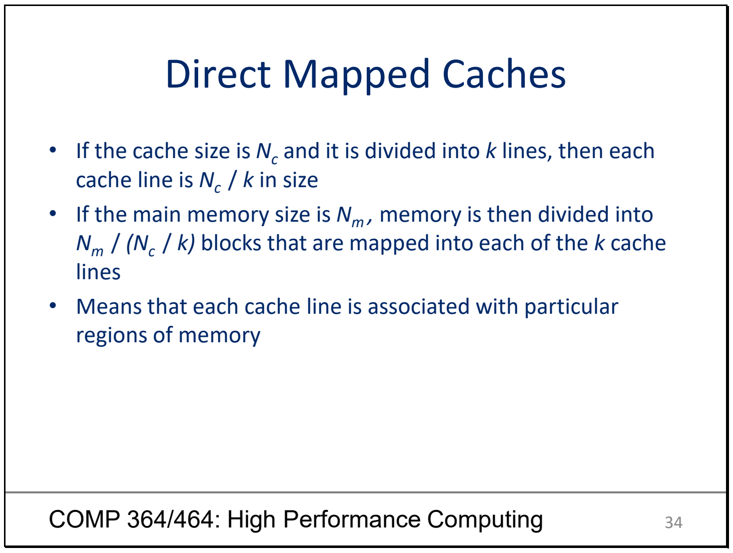


Direct Mapped Caches

Direct mapped cache: A block from main memory can go in exactly one place in the cache. This is called *direct mapped* because there is direct mapping from any block address in memory to a single location in the cache. Typically modulo calculation (e.g., use on the lower 16-bits of address).



Slide 34

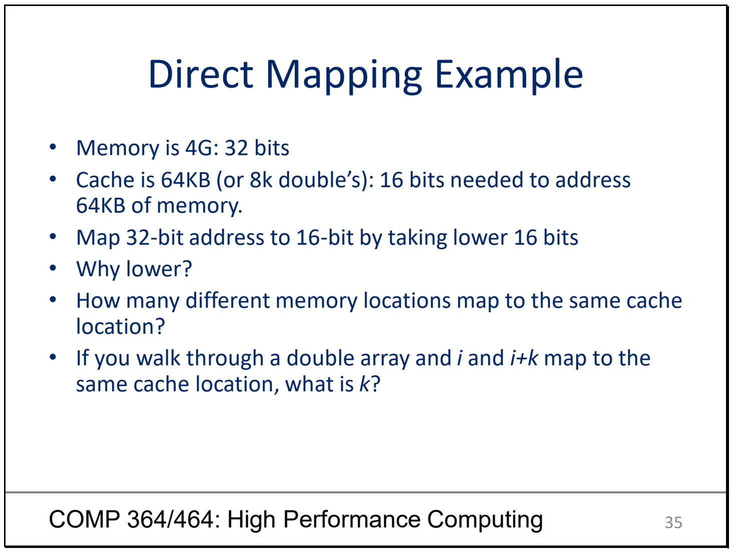

Direct Mapped Caches

- If the cache size is N_c and it is divided into k lines, then each cache line is N_c / k in size
- If the main memory size is N_m, memory is then divided into N_m / (N_c / k) blocks that are mapped into each of the k cache lines
- Means that each cache line is associated with particular regions of memory

COMP 364/464: High Performance Computing 34

Are these blocks contiguous? In other words, is assignment done by dropping leading or trailing bits?

Slide 35



Direct Mapping Example

- Memory is 4G: 32 bits
- Cache is 64KB (or 8k double's): 16 bits needed to address 64KB of memory.
- Map 32-bit address to 16-bit by taking lower 16 bits
- Why lower?
- How many different memory locations map to the same cache location?
- If you walk through a double array and *i* and *i+k* map to the same cache location, what is *k*?

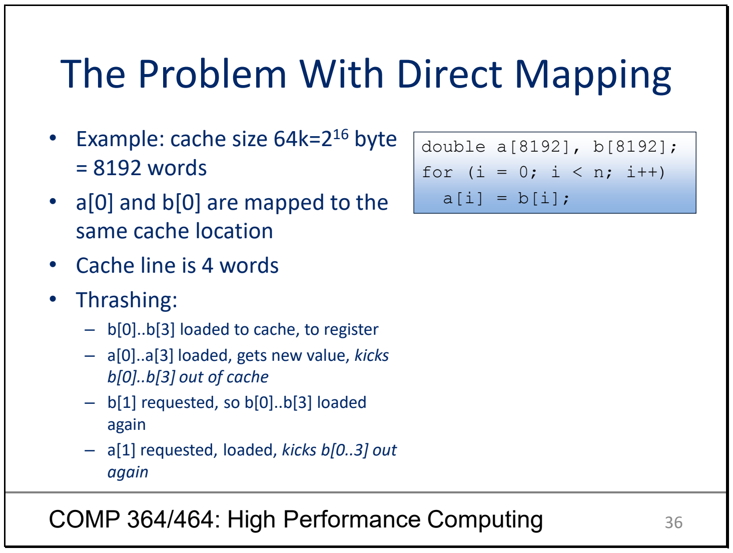
"(why last)?" -- Why does a direct mapped cache use the LOWER sixteen bits instead of the first? Answer: if it used the first 16 bits, the a block of contiguous main memory would get allocated to the same cache location, which would more or less defeat any benefit to having multiple cache locations.

k = (2^16 bytes addressable) / (8 bytes / word) = 8192 words

How many different memory locations map to the same cache location? For this example, (which is typical of a 32 bit architecture), all 32 bit addresses that end in the same 16 bits will map to the same cache line. There are 2^16=65,536 different values available for the first 16 bits. But remember: but a given 8-byte word uses 8 address locations.

"What is k?" -- for what value(s) of k do stride k accesses map to the same cache line? Answer: Any multiple of 2^16 bytes, which means (2^16)/8 = 8192 double precision words. This is the reason that multi-dimensional arrays (or several 1d arrays) with dimensions that are large multiples of 2 can be a problem. Imagine three arrays a, b, and c, all of length 8192. Then the operation c[i] = a[i] + b[i] would involve three numbers that map to the same cache location! See slide 34.

Slide 36

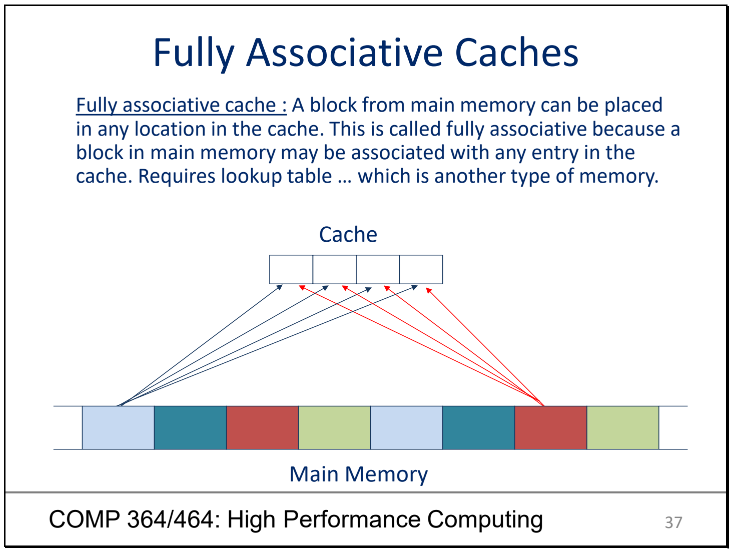


The Problem With Direct Mapping

- Example: cache size 64k=2^{16} byte = 8192 words
- a[0] and b[0] are mapped to the same cache location
- Cache line is 4 words
- Thrashing:
  - b[0]..b[3] loaded to cache, to register
  - a[0]..a[3] loaded, gets new value, *kicks b[0]..b[3] out of cache*
  - b[1] requested, so b[0]..b[3] loaded again
  - a[1] requested, loaded, *kicks b[0..3] out again*

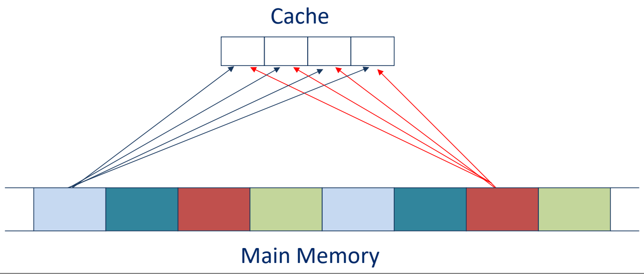
```
double a[8192], b[8192];
for (i = 0; i < n; i++)
  a[i] = b[i];
```

Slide 37

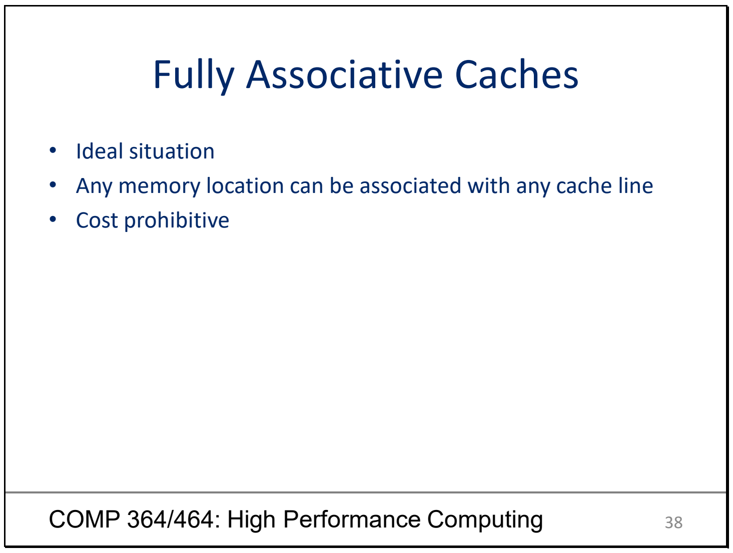


Fully Associative Caches

Fully associative cache : A block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache. Requires lookup table … which is another type of memory.



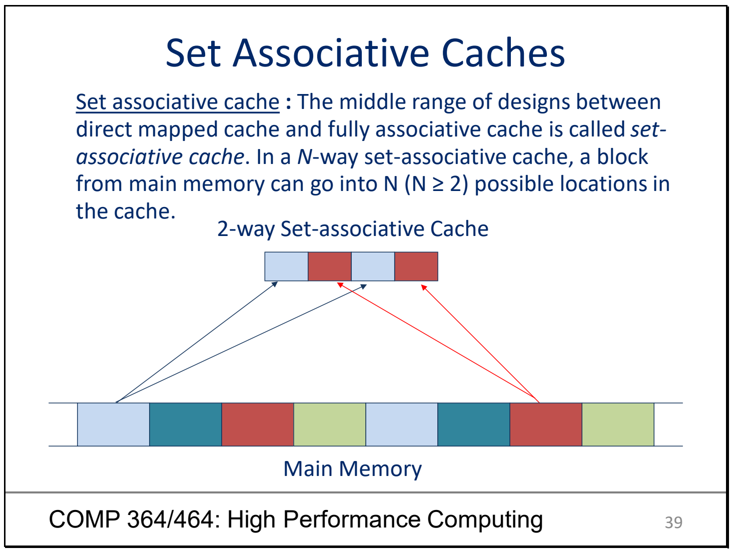
Slide 38



Fully Associative Caches

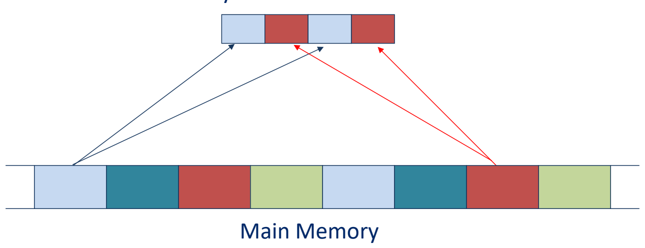
- Ideal situation
- Any memory location can be associated with any cache line
- Cost prohibitive

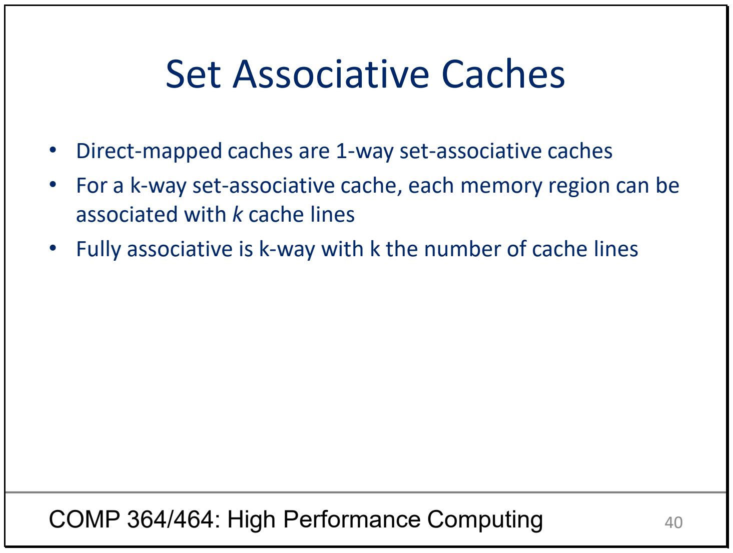
Slide 39



Set Associative Caches

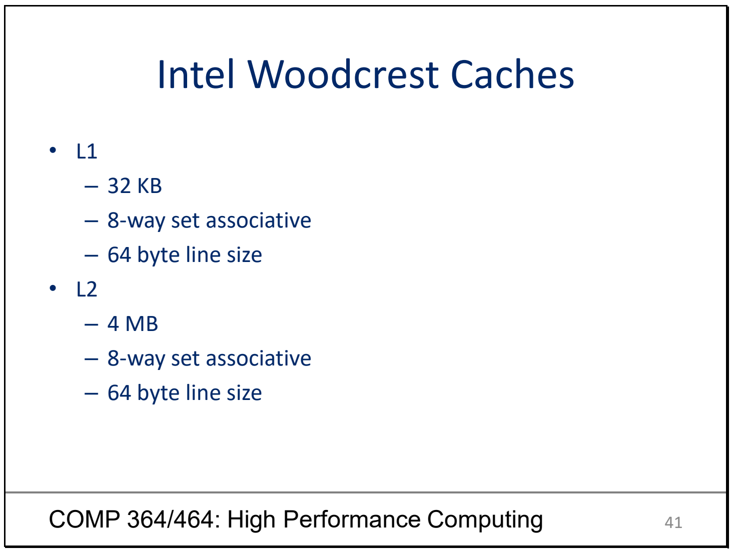
Set associative cache **:** The middle range of designs between direct mapped cache and fully associative cache is called *set-associative cache*. In a *N*-way set-associative cache, a block from main memory can go into N (N ≥ 2) possible locations in the cache.





Set Associative Caches

- Direct-mapped caches are 1-way set-associative caches
- For a k-way set-associative cache, each memory region can be associated with *k* cache lines
- Fully associative is k-way with k the number of cache lines



Intel Woodcrest Caches

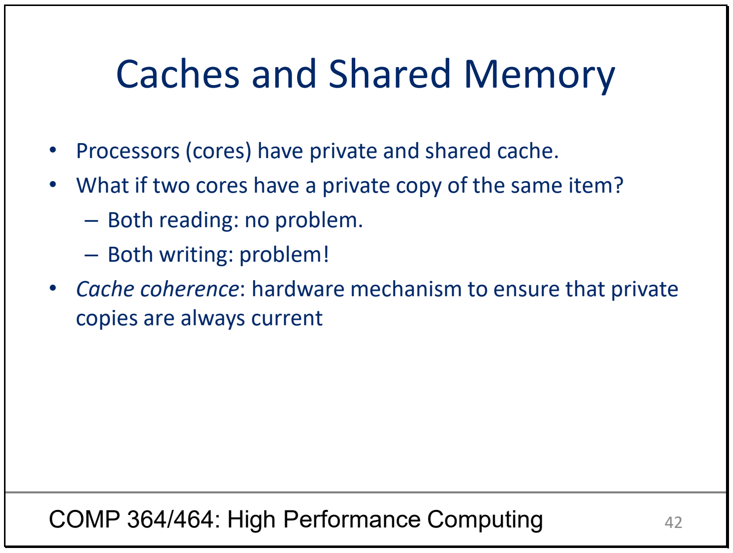
- L1
  - 32 KB
  - 8-way set associative
  - 64 byte line size
- L2
  - 4 MB
  - 8-way set associative
  - 64 byte line size

COMP 364/464: High Performance Computing 41

Really? Cache line size the same?

Technically, Woodcrest is the (former) product name of an early version of the Intel Xeon product line. Practically speaking "Woodcrest cache" is a plausible way to describe the cache technology in Stampede's Xeon E5 hosts.

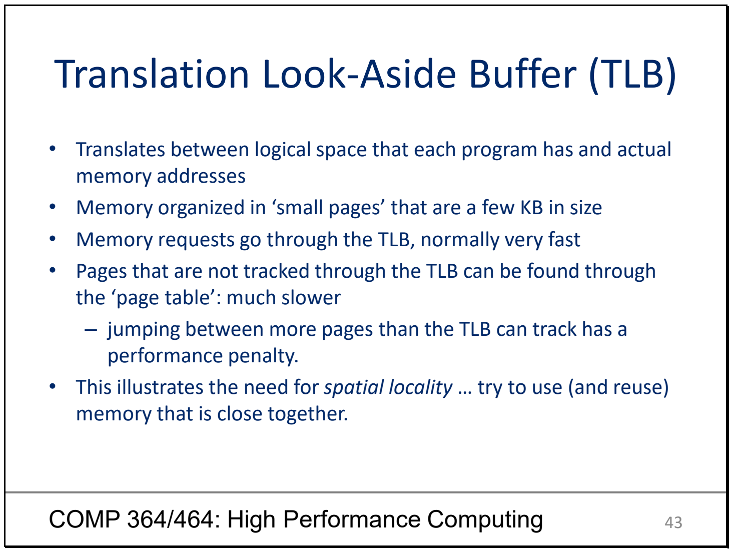
Slide 42



Caches and Shared Memory

- Processors (cores) have private and shared cache.
- What if two cores have a private copy of the same item?
  - Both reading: no problem.
  - Both writing: problem!
- *Cache coherence*: hardware mechanism to ensure that private copies are always current

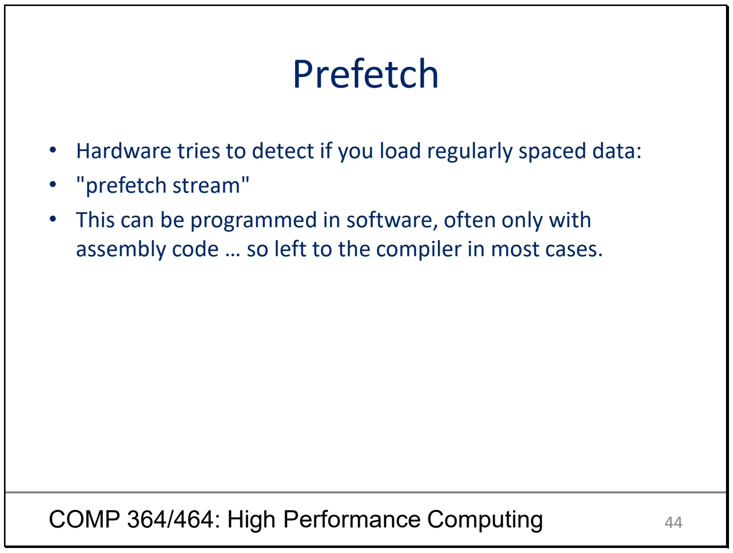
Slide 43



Translation Look-Aside Buffer (TLB)

- Translates between logical space that each program has and actual memory addresses
- Memory organized in ‘small pages’ that are a few KB in size
- Memory requests go through the TLB, normally very fast
- Pages that are not tracked through the TLB can be found through the ‘page table’: much slower
  - jumping between more pages than the TLB can track has a performance penalty.
- This illustrates the need for *spatial locality* … try to use (and reuse) memory that is close together.

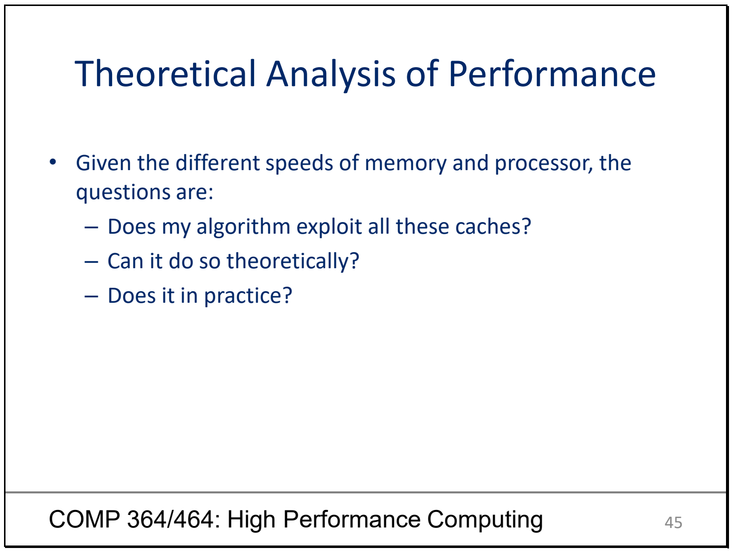
Slide 44



Prefetch

- Hardware tries to detect if you load regularly spaced data:
- "prefetch stream"
- This can be programmed in software, often only with assembly code … so left to the compiler in most cases.

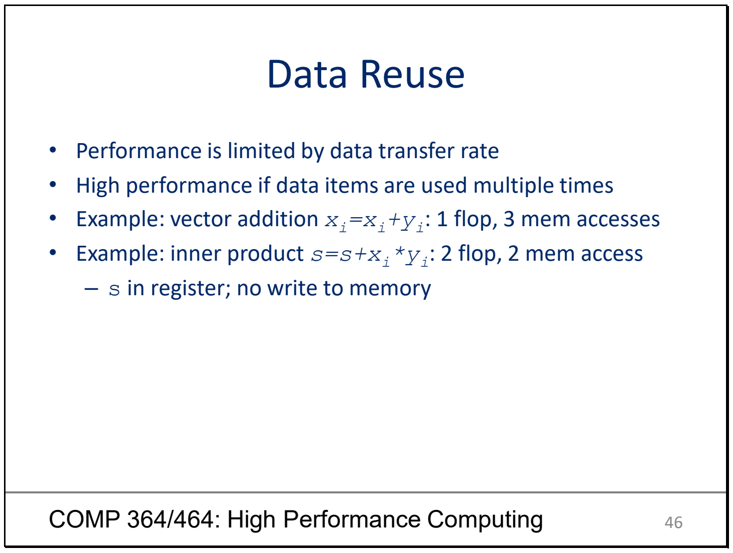
Slide 45



Theoretical Analysis of Performance

- Given the different speeds of memory and processor, the questions are:
  – Does my algorithm exploit all these caches?
  – Can it do so theoretically?
  – Does it in practice?

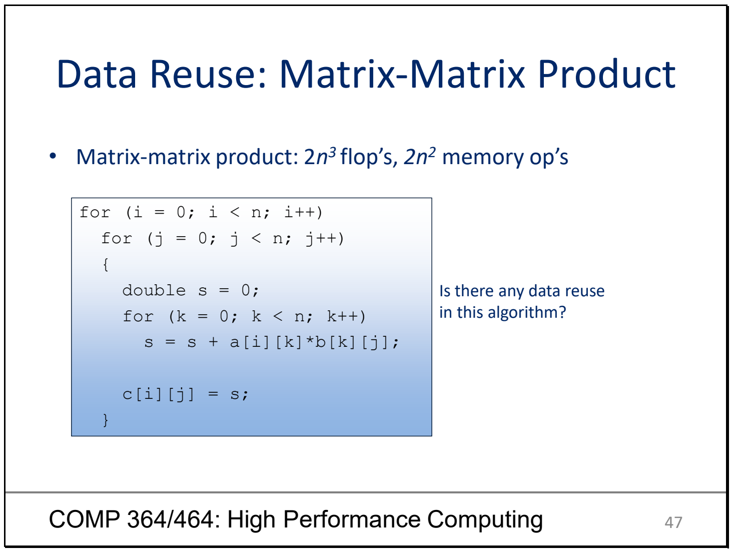
Slide 46



Data Reuse

- Performance is limited by data transfer rate
- High performance if data items are used multiple times
- Example: vector addition $x_i = x_i + y_i$: 1 flop, 3 mem accesses
- Example: inner product $s = s + x_i * y_i$: 2 flop, 2 mem access
  - s in register; no write to memory

Slide 47



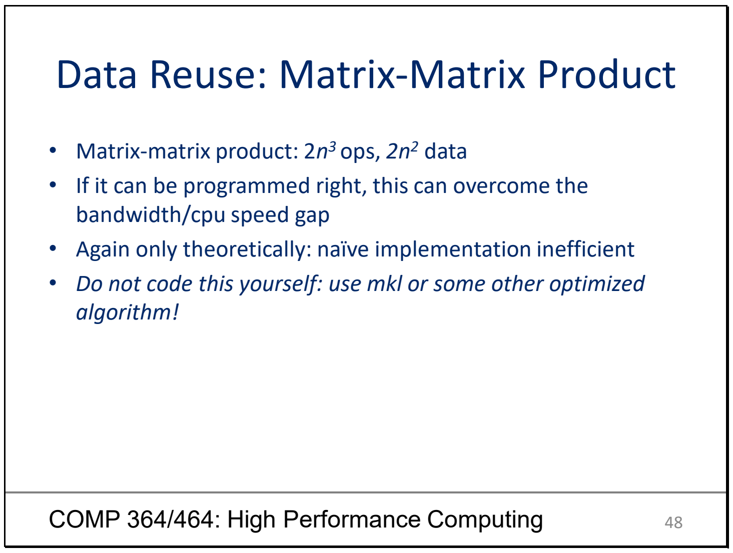
Data Reuse: Matrix-Matrix Product

- Matrix-matrix product: $2n^3$ flop's, $2n^2$ memory op's

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
  {
    double s = 0;
    for (k = 0; k < n; k++)
      s = s + a[i][k]*b[k][j];

    c[i][j] = s;
  }
```

Is there any data reuse in this algorithm?

Slide 48



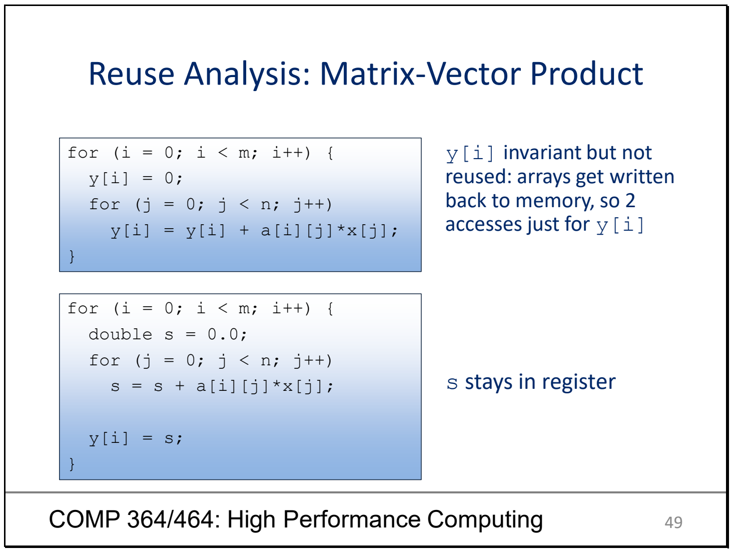
Data Reuse: Matrix-Matrix Product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data
- If it can be programmed right, this can overcome the bandwidth/cpu speed gap
- Again only theoretically: naïve implementation inefficient
- *Do not code this yourself: use mkl or some other optimized algorithm!*

"…use mkl…" -- MKL is Intel's Math Kernel Library (http://software.intel.com/en-us/intel-mkl). This library is especially important for Stamped because it supports the Xeon Phi MIC coprocessor in a number of important ways.

dgemm and sgemm are the names of the LAPACK-based routines that implement general matrix-matrix products in double and single precision respectively.

Slide 49



Reuse Analysis: Matrix-Vector Product

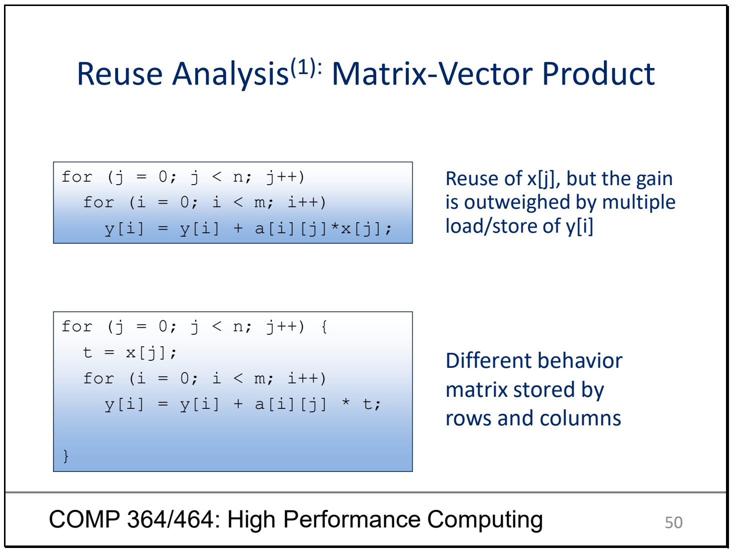
```
for (i = 0; i < m; i++) {
  y[i] = 0;
  for (j = 0; j < n; j++)
    y[i] = y[i] + a[i][j]*x[j];
}
```

`y[i]` invariant but not reused: arrays get written back to memory, so 2 accesses just for `y[i]`

```
for (i = 0; i < m; i++) {
  double s = 0.0;
  for (j = 0; j < n; j++)
    s = s + a[i][j]*x[j];

  y[i] = s;
}
```

`s` stays in register

Slide 50



Reuse Analysis(1): Matrix-Vector Product

```
for (j = 0; j < n; j++)
  for (i = 0; i < m; i++)
    y[i] = y[i] + a[i][j]*x[j];
```

Reuse of x[j], but the gain is outweighed by multiple load/store of y[i]

```
for (j = 0; j < n; j++) {
  t = x[j];
  for (i = 0; i < m; i++)
    y[i] = y[i] + a[i][j] * t;

}
```

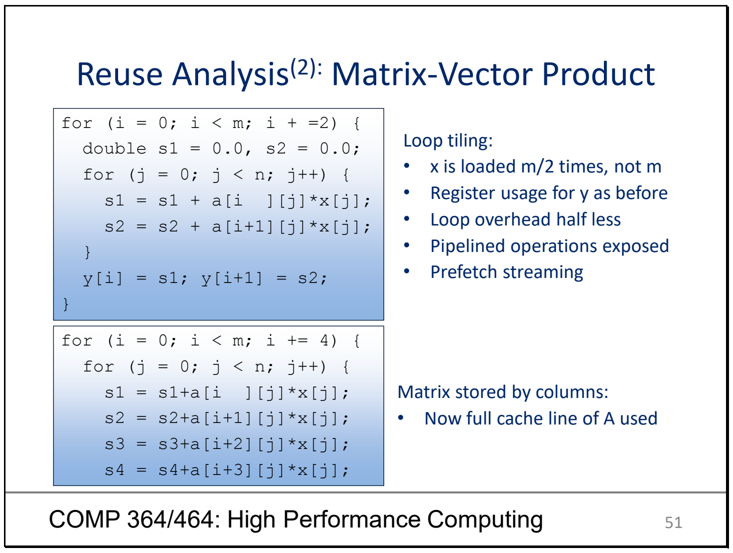
Different behavior matrix stored by rows and columns

COMP 364/464: High Performance Computing 50

1) Invert loop order: compiler will often change this if it can judge that the answer is the same and may be faster. What is fastest depends on matrix ordering. Row or column ordering …

2) Same inversion but forces the issue. And stores x[j] in register.

Slide 51



Reuse Analysis(2): Matrix-Vector Product

```
for (i = 0; i < m; i + =2) {
  double s1 = 0.0, s2 = 0.0;
  for (j = 0; j < n; j++) {
    s1 = s1 + a[i  ][j]*x[j];
    s2 = s2 + a[i+1][j]*x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```

Loop tiling:
- x is loaded m/2 times, not m
- Register usage for y as before
- Loop overhead half less
- Pipelined operations exposed
- Prefetch streaming

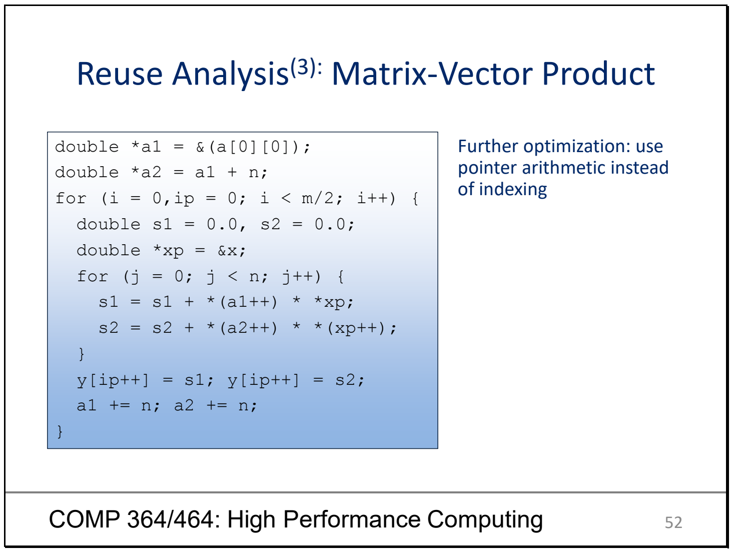
```
for (i = 0; i < m; i += 4) {
  for (j = 0; j < n; j++) {
    s1 = s1+a[i  ][j]*x[j];
    s2 = s2+a[i+1][j]*x[j];
    s3 = s3+a[i+2][j]*x[j];
    s4 = s4+a[i+3][j]*x[j];
```

Matrix stored by columns:
- Now full cache line of A used

Compiler often does the unrolling for you. And handles the extra at the end.

Other issues related to memory alignment not discussed here.

Slide 52



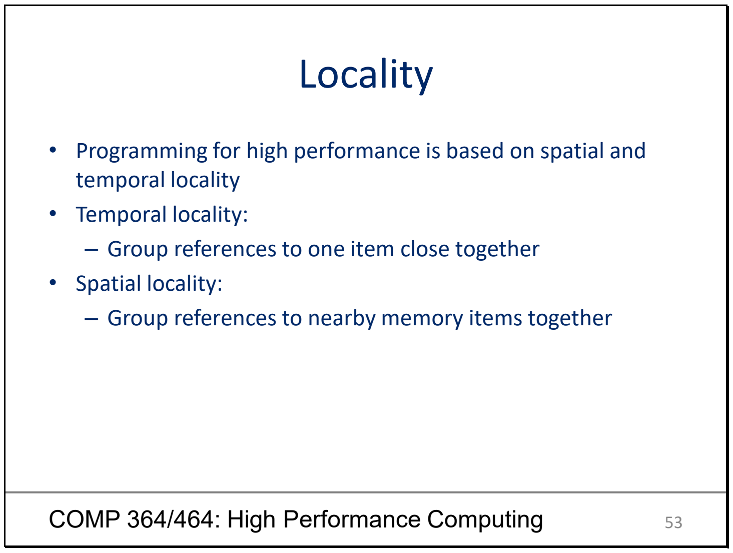
Reuse Analysis(3): Matrix-Vector Product

```
double *a1 = &(a[0][0]);
double *a2 = a1 + n;
for (i = 0,ip = 0; i < m/2; i++) {
  double s1 = 0.0, s2 = 0.0;
  double *xp = &x;
  for (j = 0; j < n; j++) {
    s1 = s1 + *(a1++) * *xp;
    s2 = s2 + *(a2++) * *(xp++);
  }
  y[ip++] = s1; y[ip++] = s2;
  a1 += n; a2 += n;
}
```

Further optimization: use pointer arithmetic instead of indexing

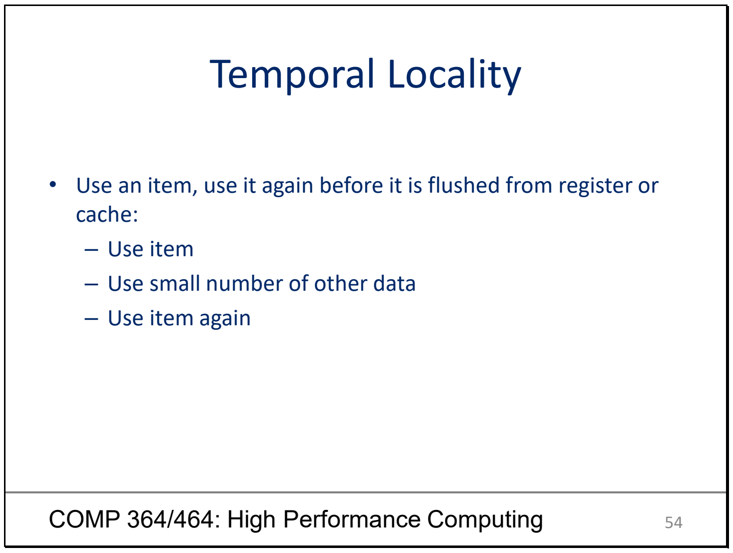
Performance improves but the code becomes hard to read. And some hand tuning can negate compiler optimizations!

Slide 53



Locality

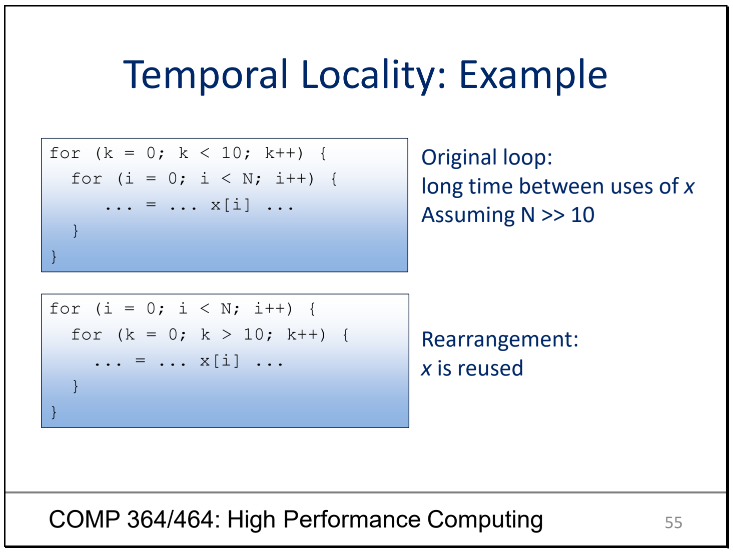
- Programming for high performance is based on spatial and temporal locality
- Temporal locality:
  - Group references to one item close together
- Spatial locality:
  - Group references to nearby memory items together



Temporal Locality

- Use an item, use it again before it is flushed from register or cache:
  - Use item
  - Use small number of other data
  - Use item again

Slide 55



Temporal Locality: Example

```
for (k = 0; k < 10; k++) {
  for (i = 0; i < N; i++) {
     ... = ... x[i] ...
  }
}
```

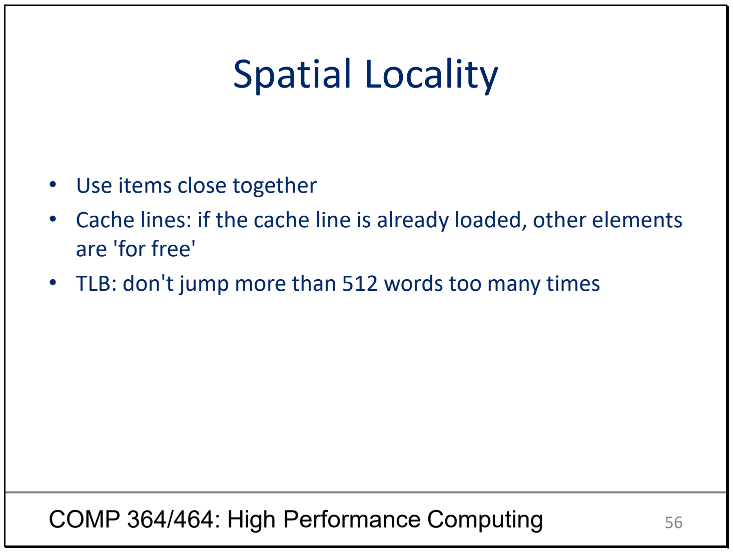
Original loop:
long time between uses of *x*
Assuming N >> 10

```
for (i = 0; i < N; i++) {
  for (k = 0; k > 10; k++) {
    ... = ... x[i] ...
  }
}
```

Rearrangement:
x is reused

Loop inversion common compiler optimization.

Slide 56

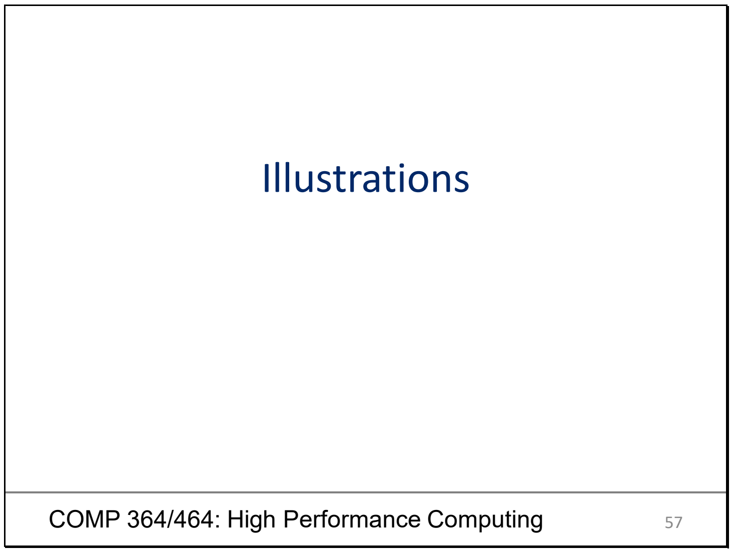


Spatial Locality

- Use items close together
- Cache lines: if the cache line is already loaded, other elements are 'for free'
- TLB: don't jump more than 512 words too many times

COMP 364/464: High Performance Computing 56

Pages are often 4K

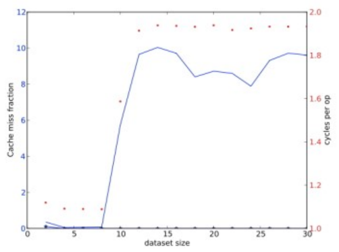


Illustrations

Slide 58

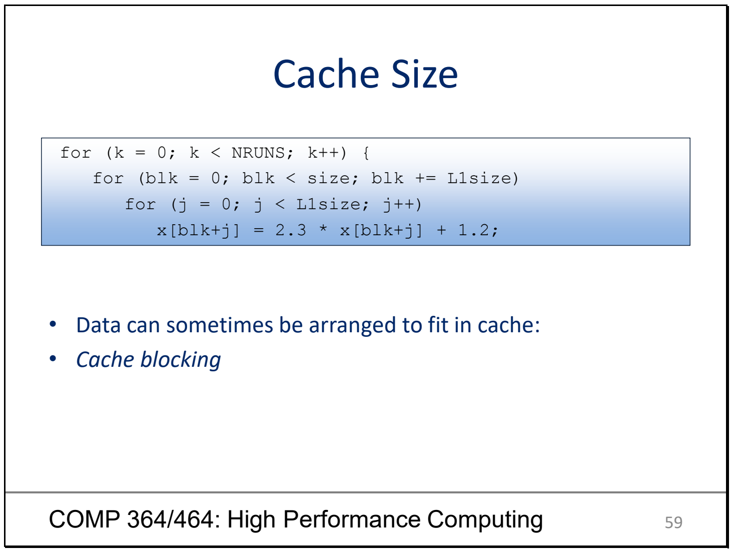
Cache Size

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



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

Slide 59



Cache Size

```
for (k = 0; k < NRUNS; k++) {
   for (blk = 0; blk < size; blk += L1size)
      for (j = 0; j < L1size; j++)
         x[blk+j] = 2.3 * x[blk+j] + 1.2;
```

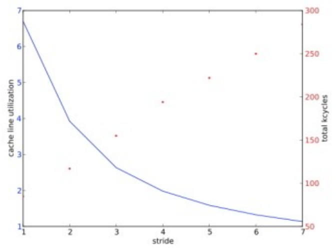
- Data can sometimes be arranged to fit in cache:
- *Cache blocking*

Slide 60

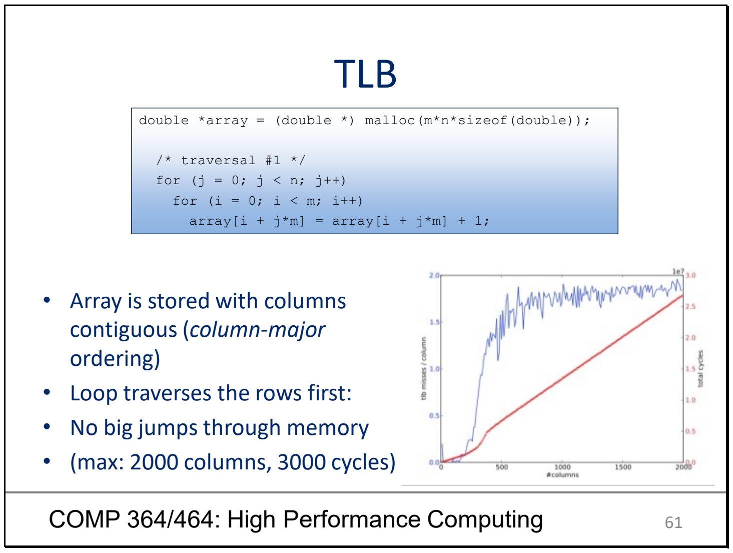
Cache Line Utilization

```
for (i = 0, n = 0; i < L1words; i++, n += Stride)
    array[n] = 2.3 * array[n] + 1.2;
```

- Same amount of data, but increasing stride
- Increasing stride: more cache lines loaded, slower execution



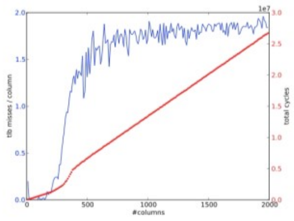
Slide 61



TLB

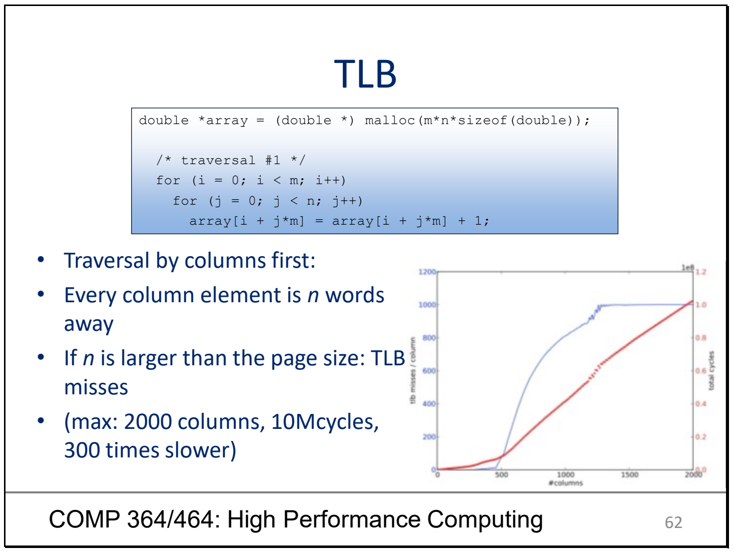
```
double *array = (double *) malloc(m*n*sizeof(double));

  /* traversal #1 */
  for (j = 0; j < n; j++)
    for (i = 0; i < m; i++)
      array[i + j*m] = array[i + j*m] + 1;
```

- Array is stored with columns contiguous (*column-major* ordering)
- Loop traverses the rows first:
- No big jumps through memory
- (max: 2000 columns, 3000 cycles)



Running down the columns. i = row index

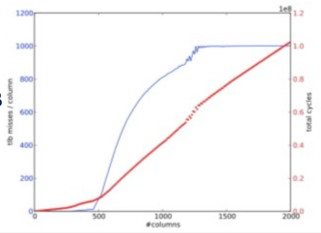
Slide 62



TLB

```
double *array = (double *) malloc(m*n*sizeof(double));

  /* traversal #1 */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      array[i + j*m] = array[i + j*m] + 1;
```

- Traversal by columns first:
- Every column element is *n* words away
- If *n* is larger than the page size: TLB misses
- (max: 2000 columns, 10Mcycles, 300 times slower)



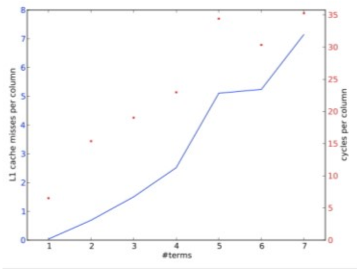
# of TLB misses 1000x higher.
# of cycles 10x higher!

Slide 63

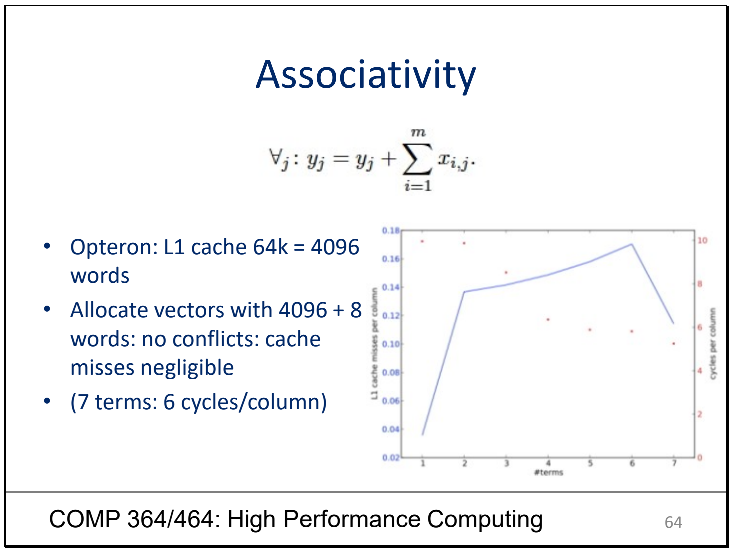
Associativity

$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}.$$

- Opteron: L1 DCache 64k = 4096 words
- Two-way associative, so m > 1 leads to conflicts:
- Cache misses/column goes up linearly
- (max: 7 terms, 35 cycles/column)



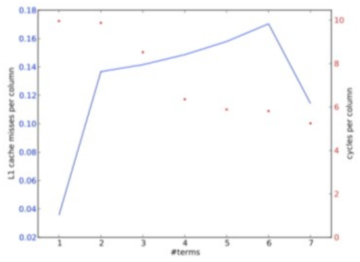
Slide 64



Associativity

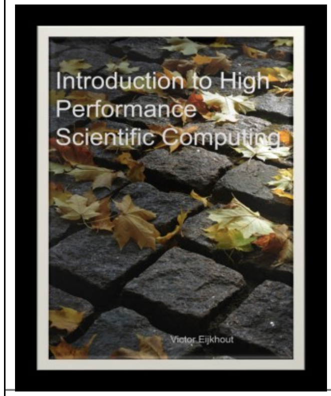
$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}.$$

- Opteron: L1 cache 64k = 4096 words
- Allocate vectors with 4096 + 8 words: no conflicts: cache misses negligible
- (7 terms: 6 cycles/column)



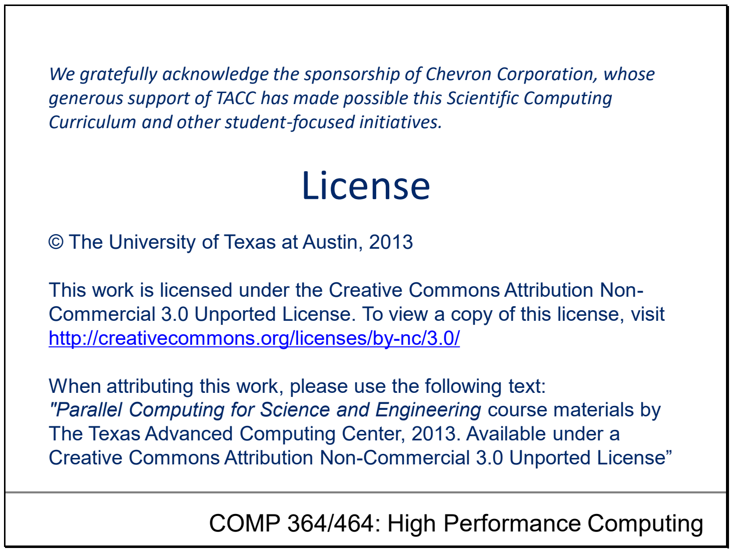
Slide 65

Further reading



- General page: http://www.tacc.utexas.edu/~eijkhout/istc/istc.html
- Direct download: http://tinyurl.com/EijkhoutHPC
- Content presented here is based on content from "Parallel Computing for Science and Engineering course materials by The Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License"

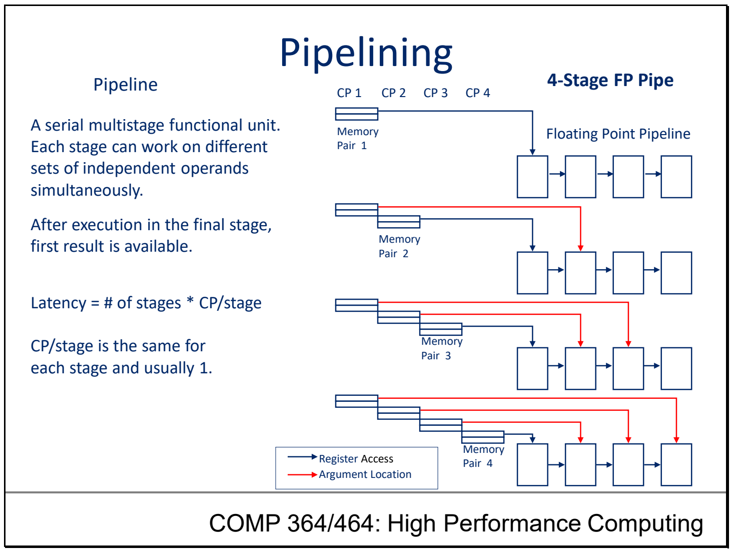
Slide 66



We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License

Slide 67



Pipelining

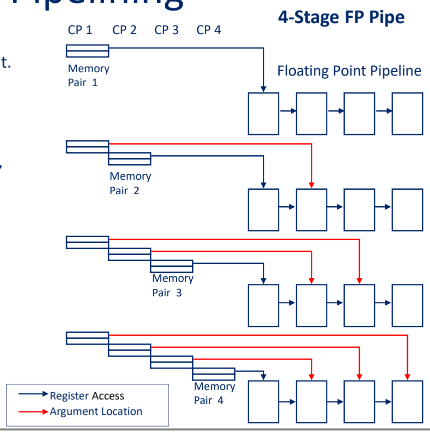
Pipeline

A serial multistage functional unit. Each stage can work on different sets of independent operands simultaneously.

After execution in the final stage, first result is available.

Latency = # of stages * CP/stage

CP/stage is the same for each stage and usually 1.



A = x * 2^q, B = y * 2^p: fetch A and B, align exponents (shift), subtract aligned mantissa, normalize result (shift), write back result.