

COMP 364 / 464

High Performance Computing

HPC CPU Architectures:

Pipelining, Cache and Memory hierarchy

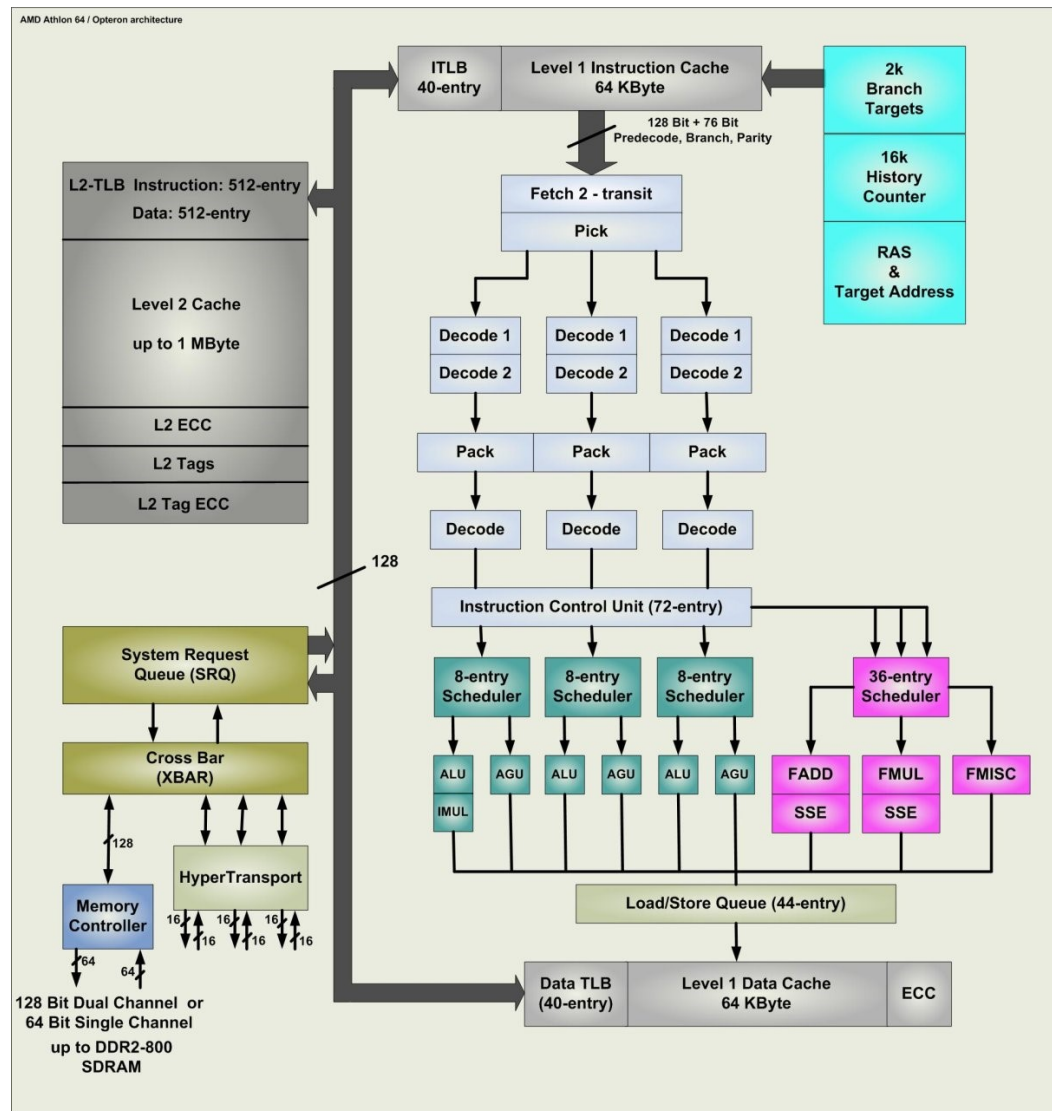
Von Neumann Architecture

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

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

What Does a CPU Look Like?



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

Functional Units

- Traditional: One instruction at a time
- Modern: Multiple floating point units
 - Example: fused multiply-add (FMA)
$$\mathbf{x[i]} \leftarrow \mathbf{c * x[i] + y[i]}$$
- 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!

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

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.*

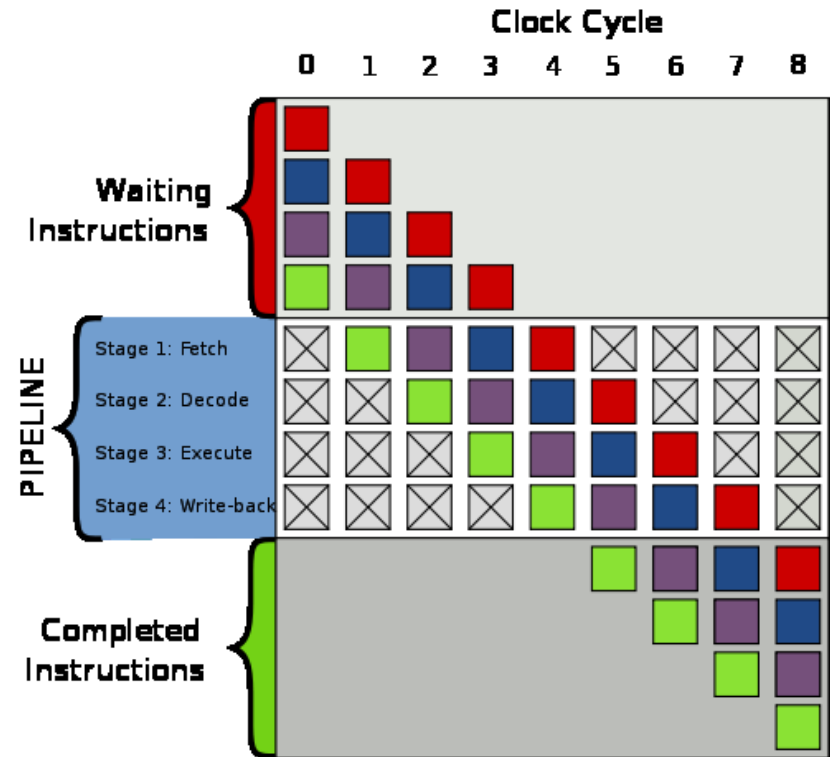
Pipeline (cont)

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

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.

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
---	---	---	---	---	---	---	---	---	----

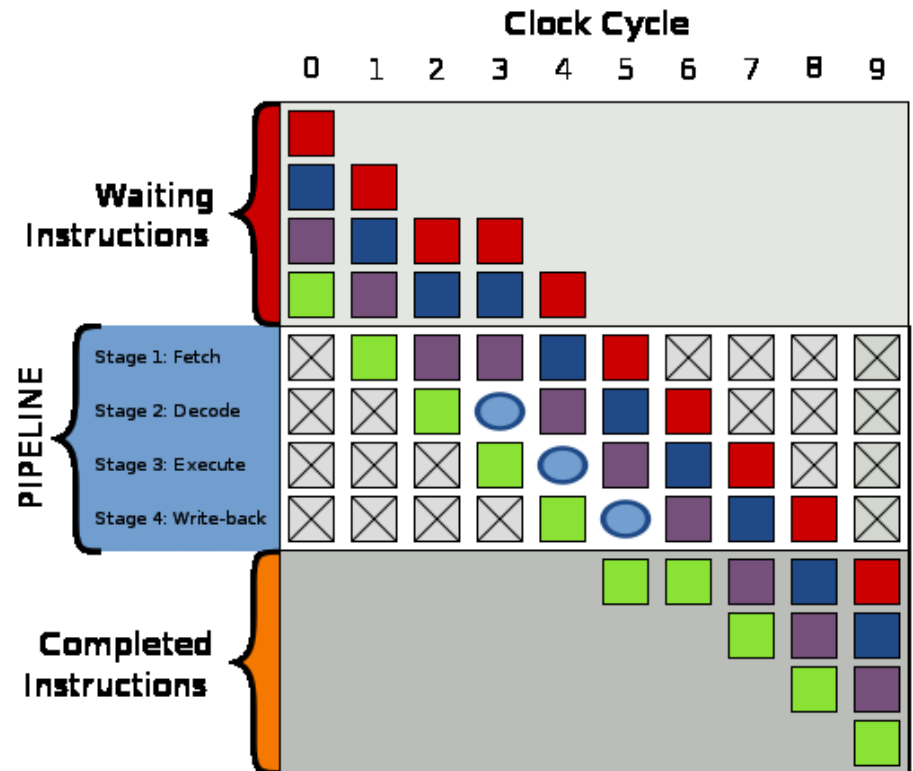
Pentium 4 processor pipeline

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.

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

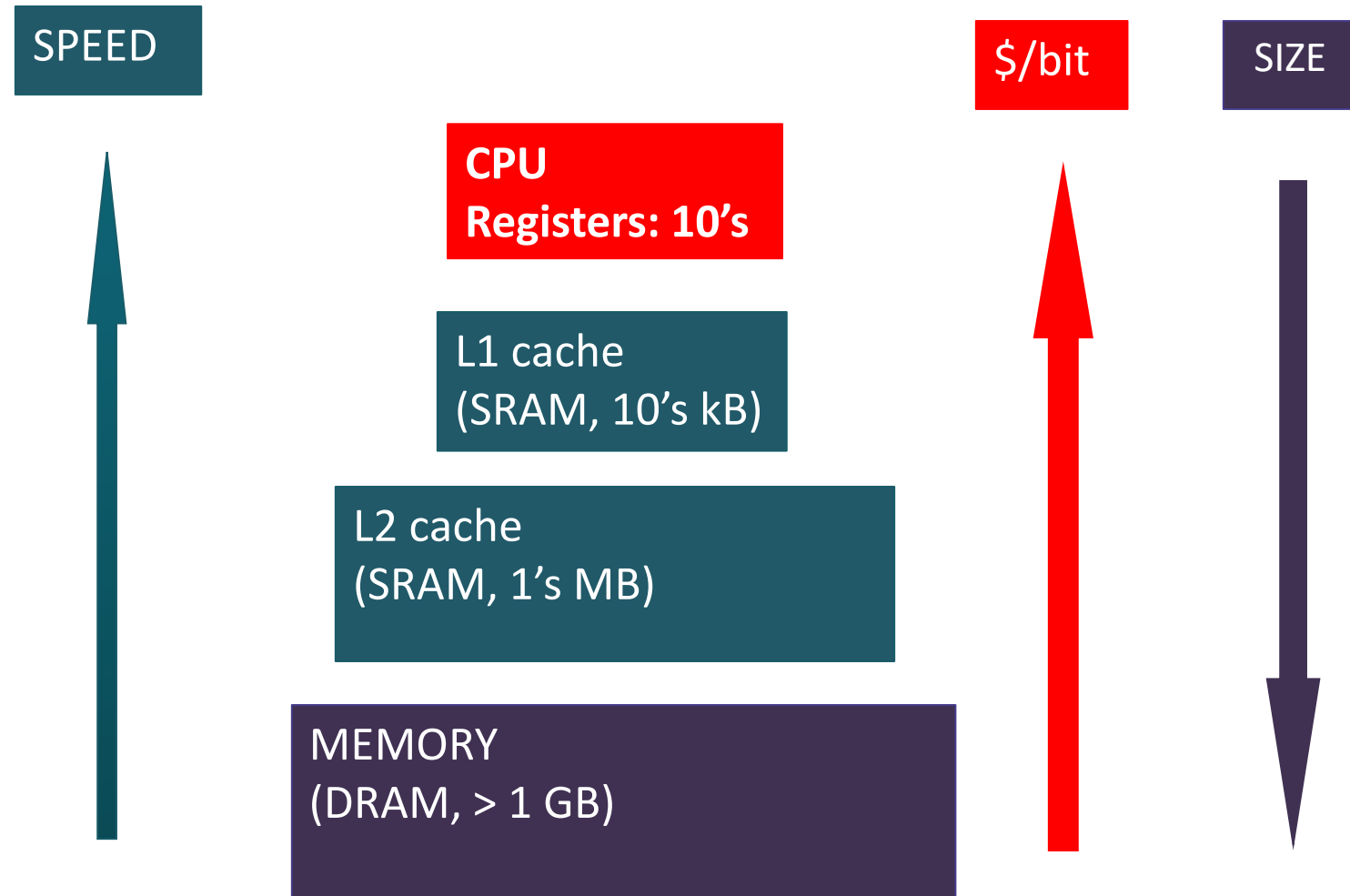
Memory Hierarchies

- 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 hierarchy.

Memory Hierarchies (cont)

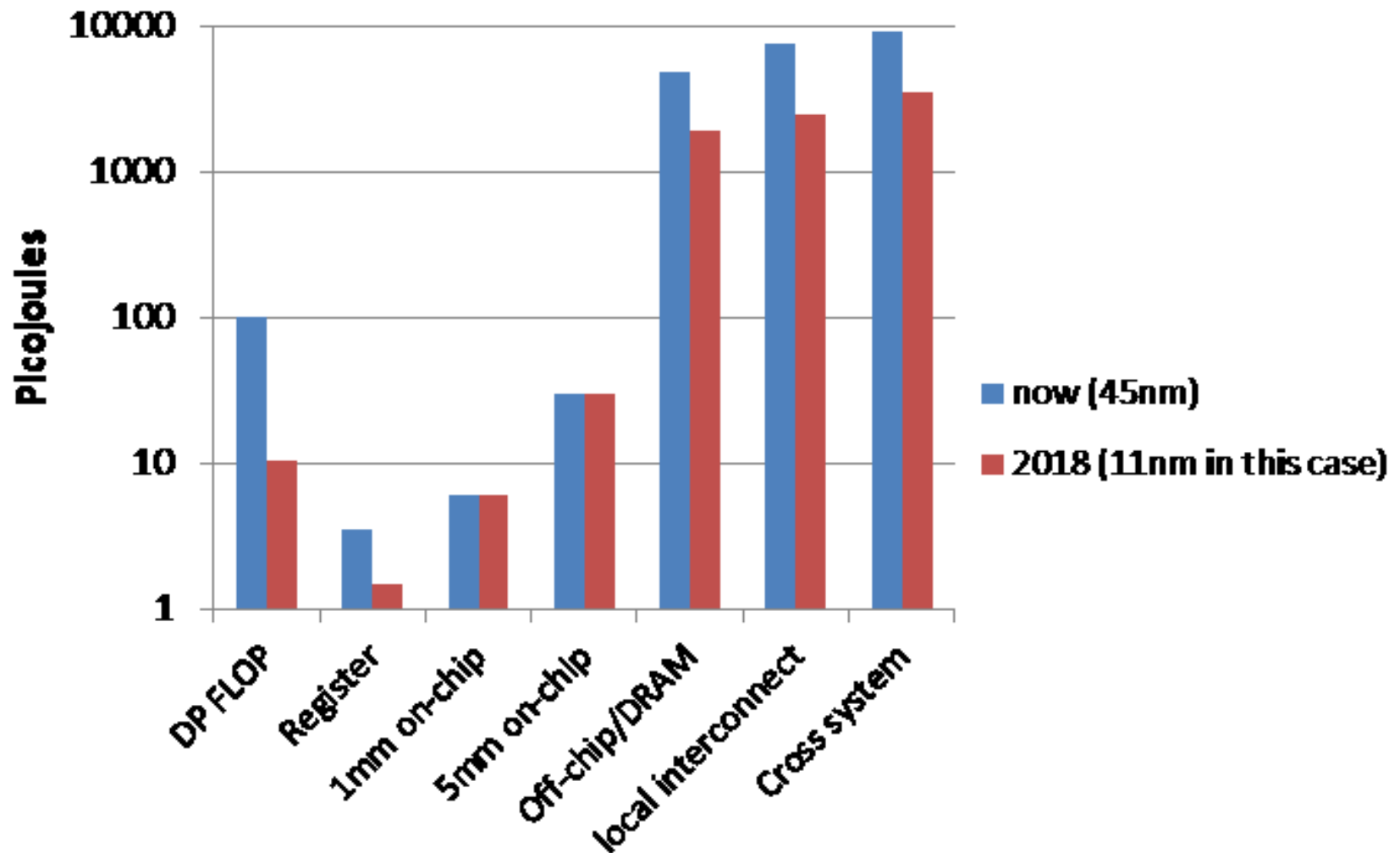
- Memory is divided into different levels:
 - Registers
 - Caches
 - Main Memory
- Memory is accessed through the hierarchy
 - registers when possible
 - ... then the caches
 - ... then main memory
- Closer to core is faster. Performance boosted by using data that's close (and fast).

Memory Relativity



Why Minimize Cache misses?

It's not always about speed



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.)

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*: $\text{Concurrency} = \text{Bandwidth} \times \text{Latency}$
 - We need this many items in flight to saturate the CPU.

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.

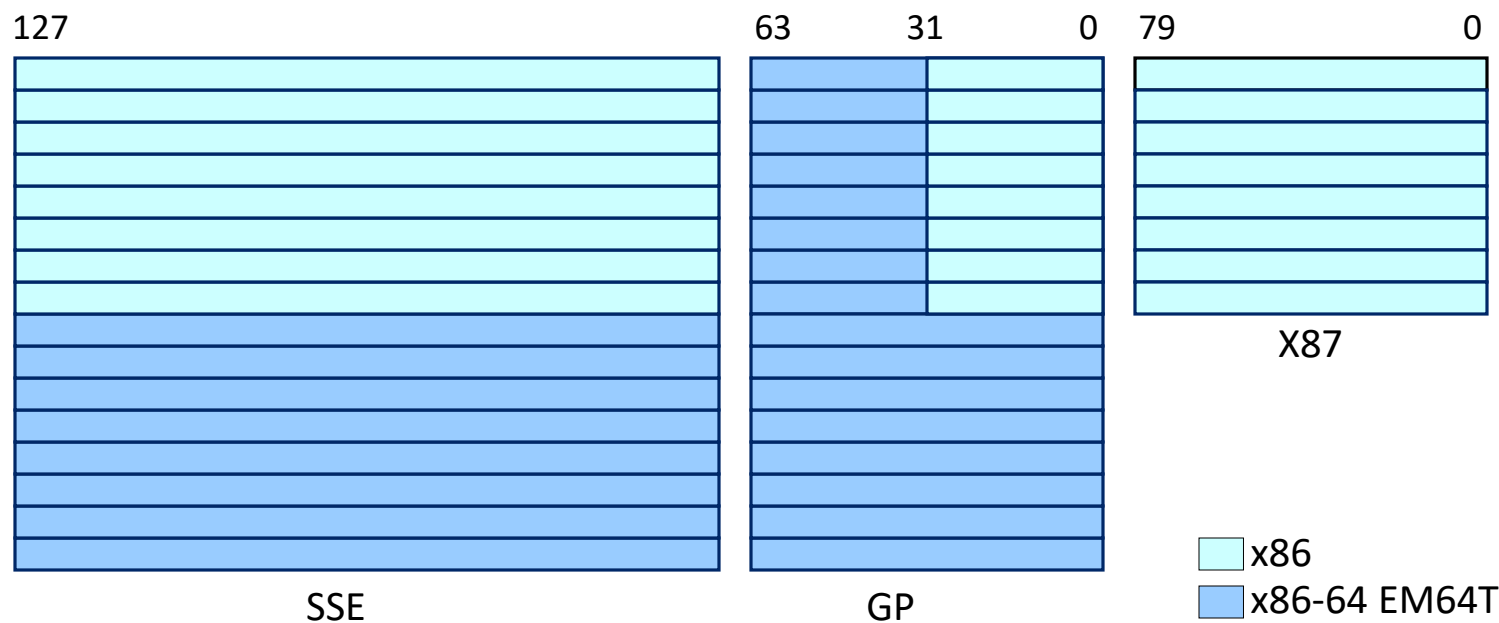
How Good Are GPUs?

- 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 $\sim 4x$
 - Use of multiple floating point units: another $\sim 2x$
 - Not using vector units: another $\sim 4x$
- 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.

The Memory Subsystem in Detail

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



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

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!

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.
- Data commonly allocated with 'word alignment' meaning a POD 64b value isn't split across a cacheline. But this doesn't apply to user-defined structs.

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 so we have to be careful with how we use it and the faster caches.
- Non-volatile memory (NVRAM) is starting to come around as a HPC hardware component.
 - Much faster than a spindle disk so could be useful to stage data between main memory and the disk when saving for later.
 - Also useful when you need a lot of memory. Store some things there and free up the (faster) DRAM.

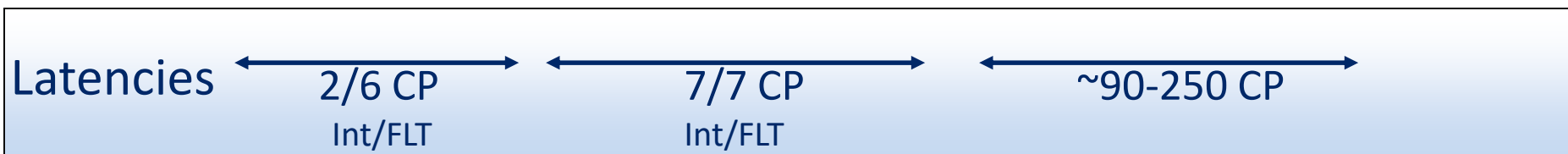
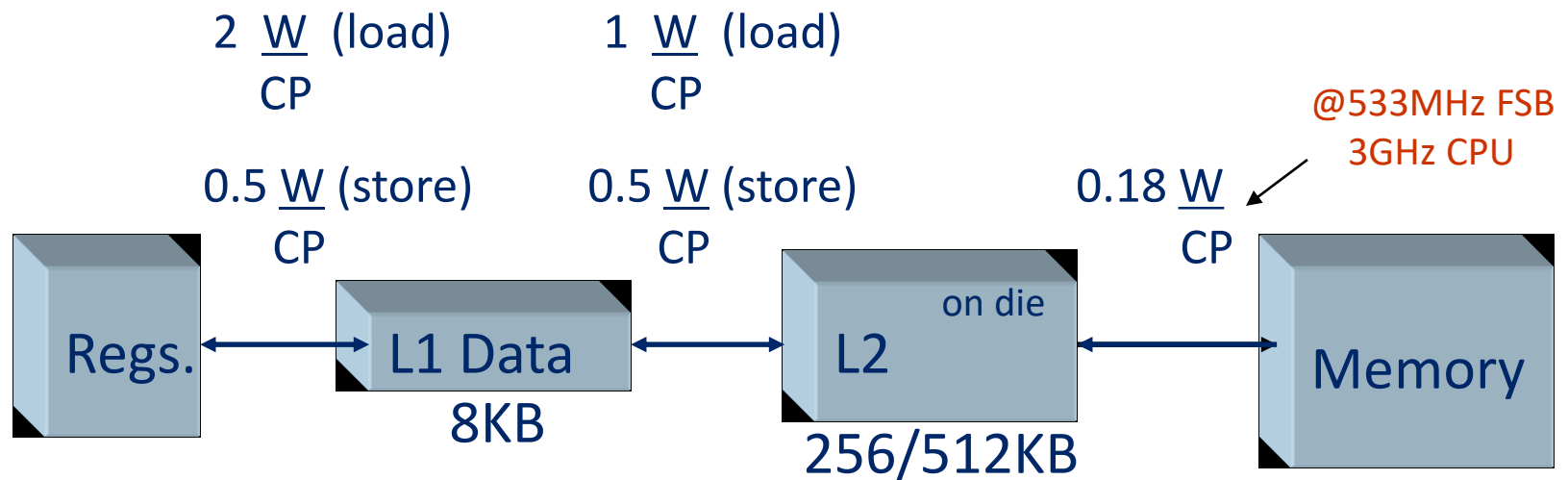
Multi-Core Chips

- 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.

Typical Latencies and Bandwidths in a Memory Hierarchy

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

Example: Pentium 4



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

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.)

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

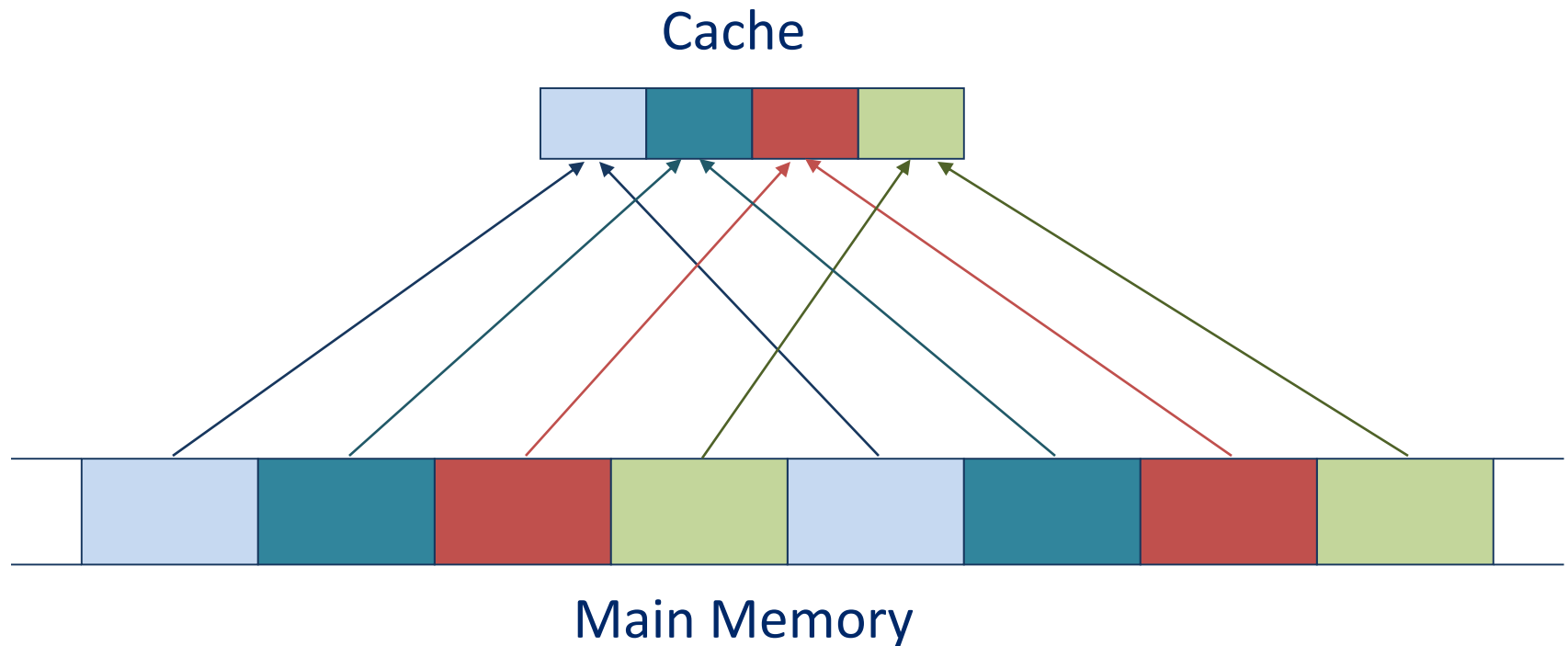
- Cache hit
 - location referenced is found in the cache
- Cache miss
 - location referenced is not found in cache
 - triggers access to the next higher cache or memory
- Cache thrashing
 - Two data elements can be mapped to the same cache line: loading the second “evicts” the first
 - Now what if this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

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

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).



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

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 ?

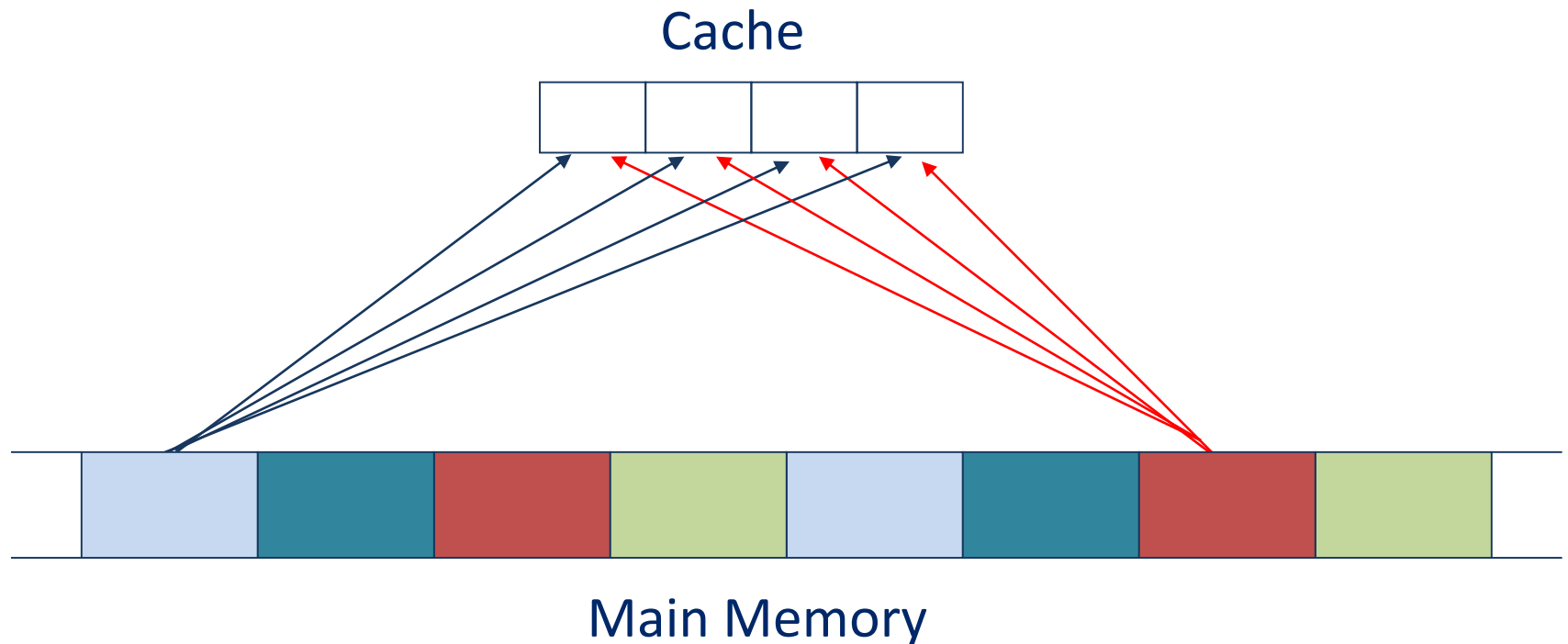
The Problem With Direct Mapping

- 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];
```

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.



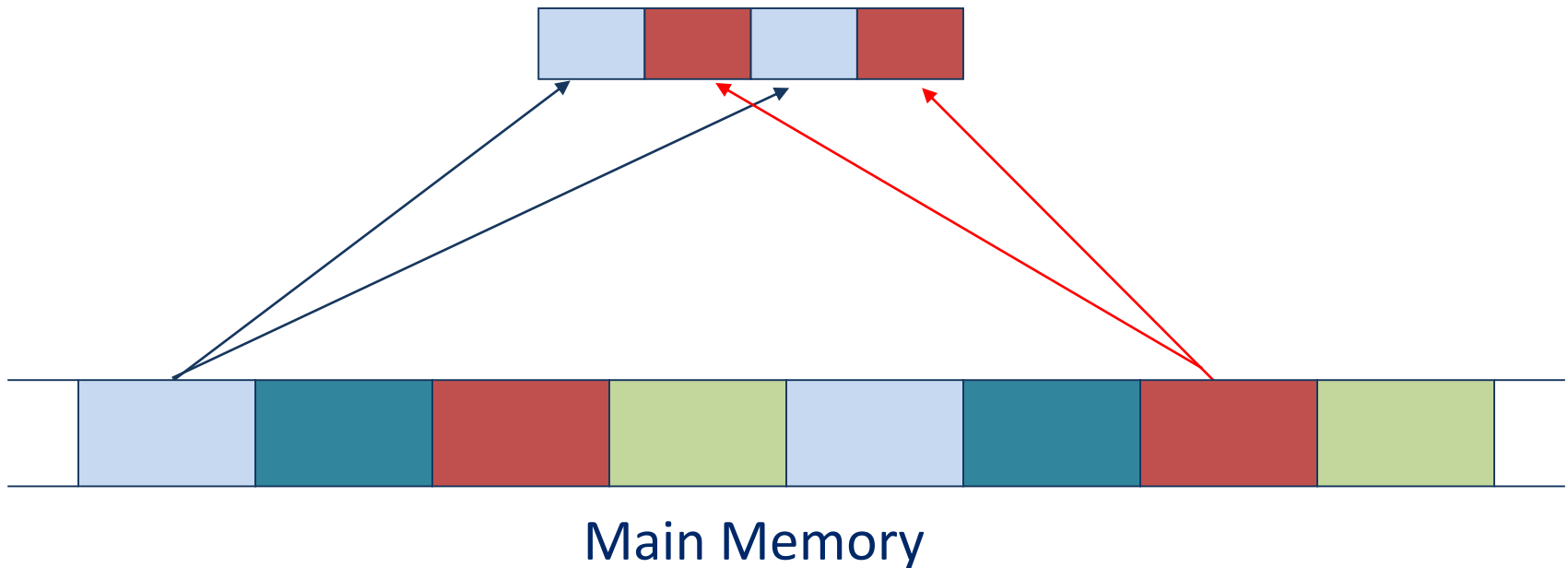
Fully Associative Caches

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

Set Associative Caches

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

2-way Set-associative 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

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

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.

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.

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?

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

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?

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!*

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

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

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;  
}
```

```
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];  
    }  
}
```

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

Matrix stored by columns:

- Now full cache line of A used

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

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

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 $N \gg 10$

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

Rearrangement:
 x is reused

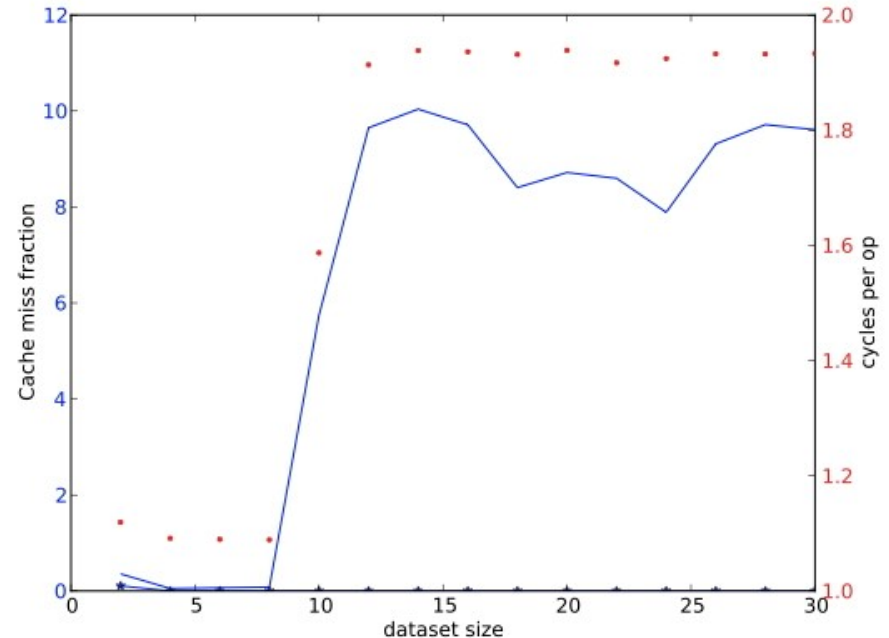
Spatial Locality

- 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

Illustrations

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

Cache Size

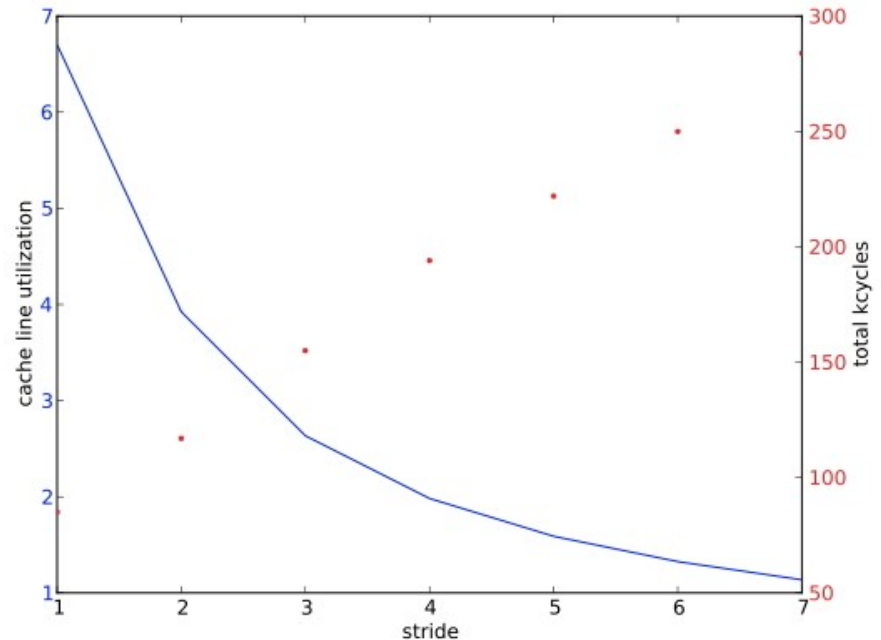
```
for (k = 0; k < NRUNS; k++) {  
    for (blk = 0; blk < size; blk += L1size)  
        for (j = 0; j < L1size; j++)  
            x[blk+j] = 2.3 * x[blk+j] + 1.2;
```

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

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

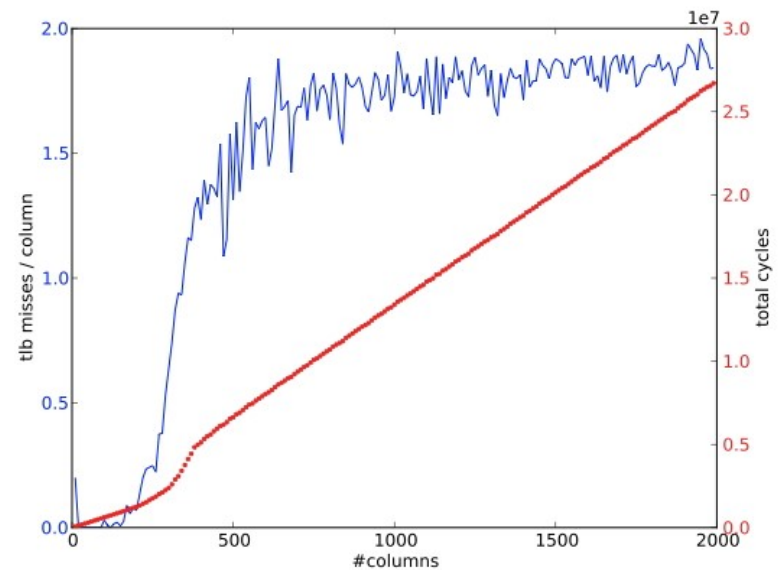


TLB

```
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)

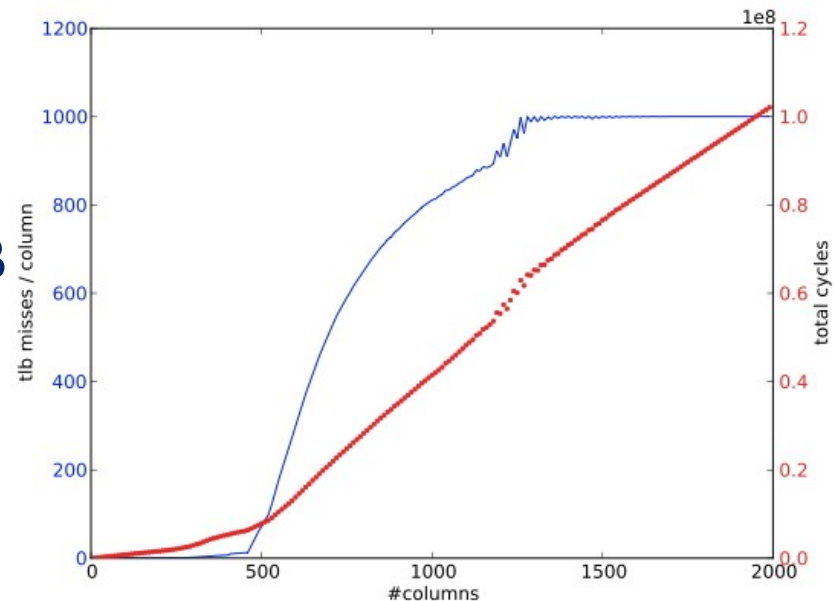


TLB

```
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;
```

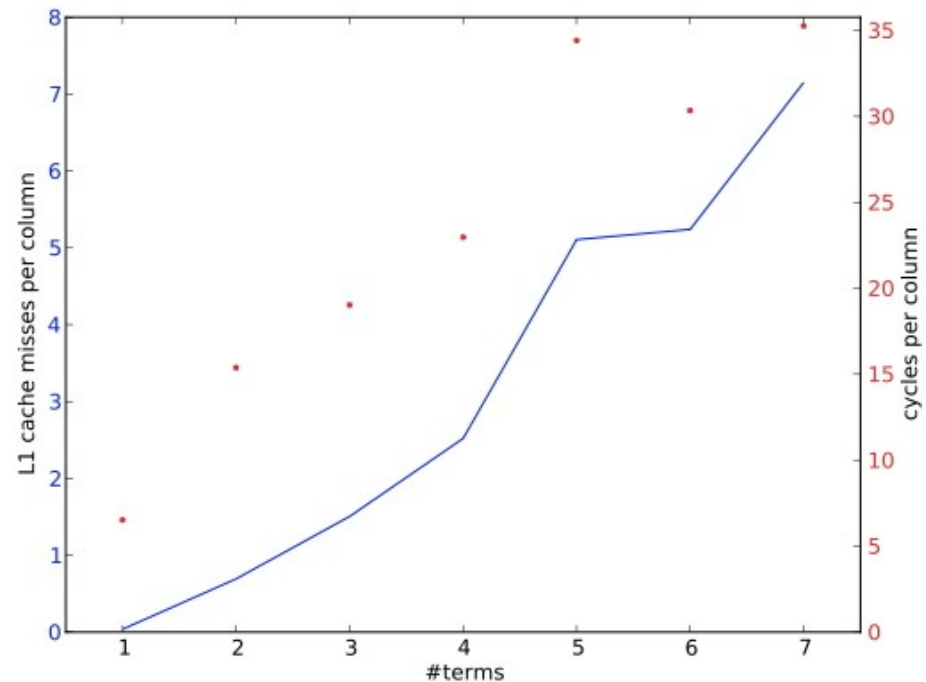
- Traversal by columns first:
- Every column element is n words away
- If n is larger than the page size: TLB misses
- (max: 2000 columns, 10Mcycles, 300 times slower)



Associativity

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

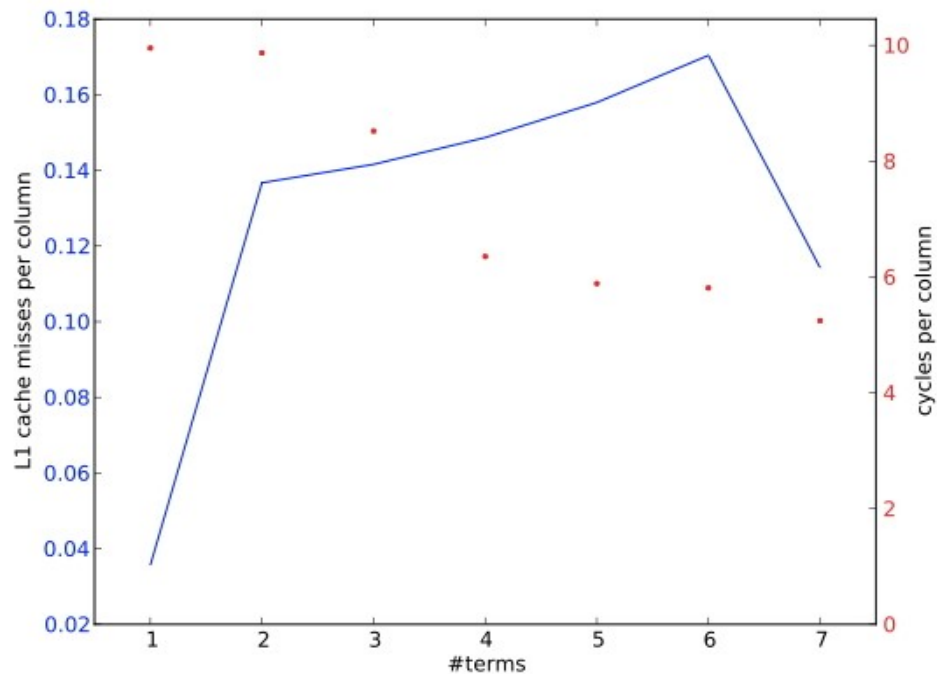
- Opteron: L1 DCache 64k = 4096 words
- Two-way associative, so $m > 1$ leads to conflicts:
- Cache misses/column goes up linearly
- (max: 7 terms, 35 cycles/column)



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)



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”

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

© The University of Texas at Austin, 2013

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/>

When attributing this work, please use the following text:
"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"

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.

4-Stage FP Pipe

