SSC 335/394: Scientific and Technical Computing

Computer Architectures
single CPU

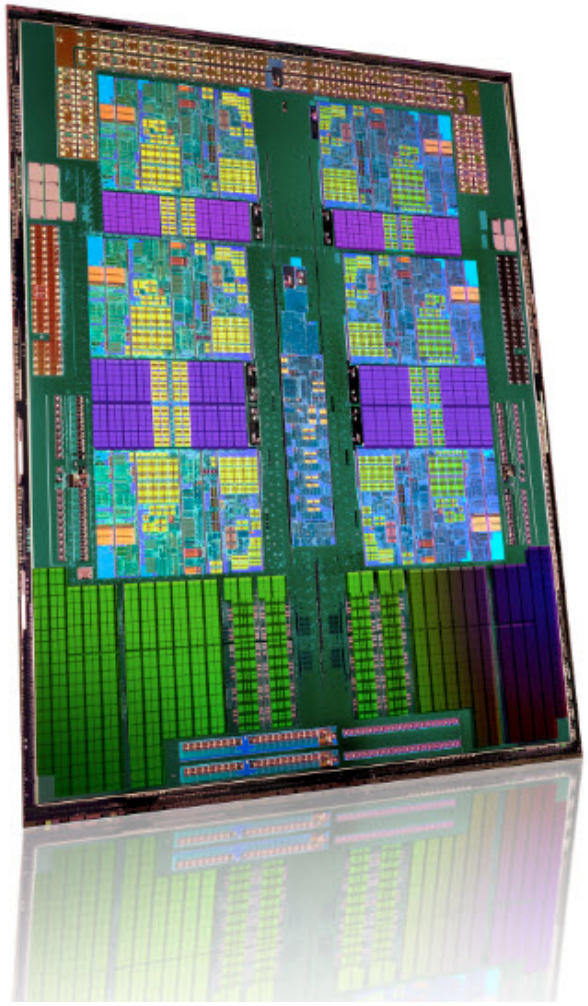
Von Neumann Architecture

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

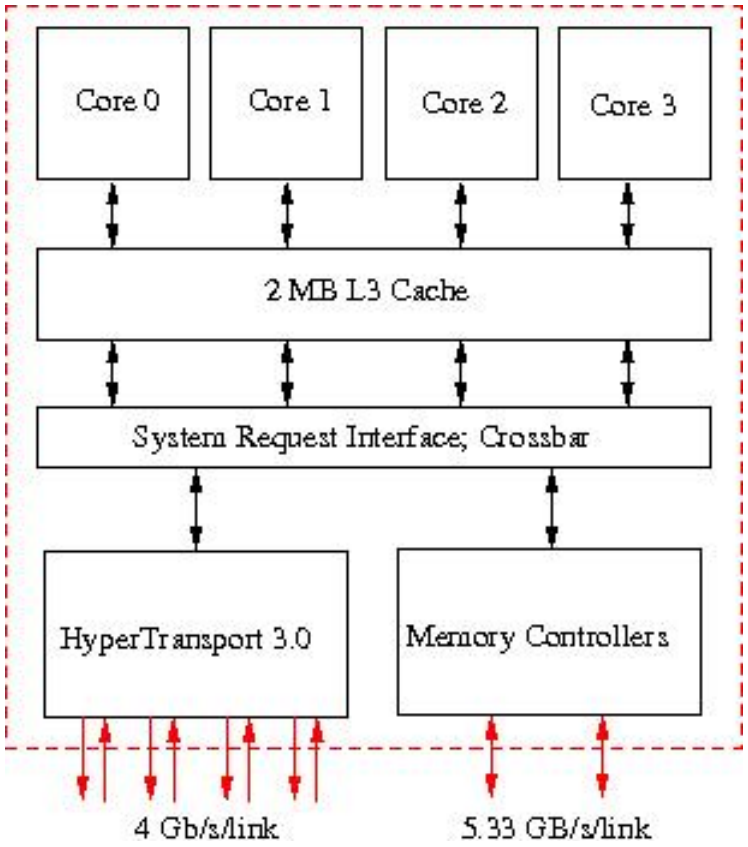
Contemporary Architecture

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively

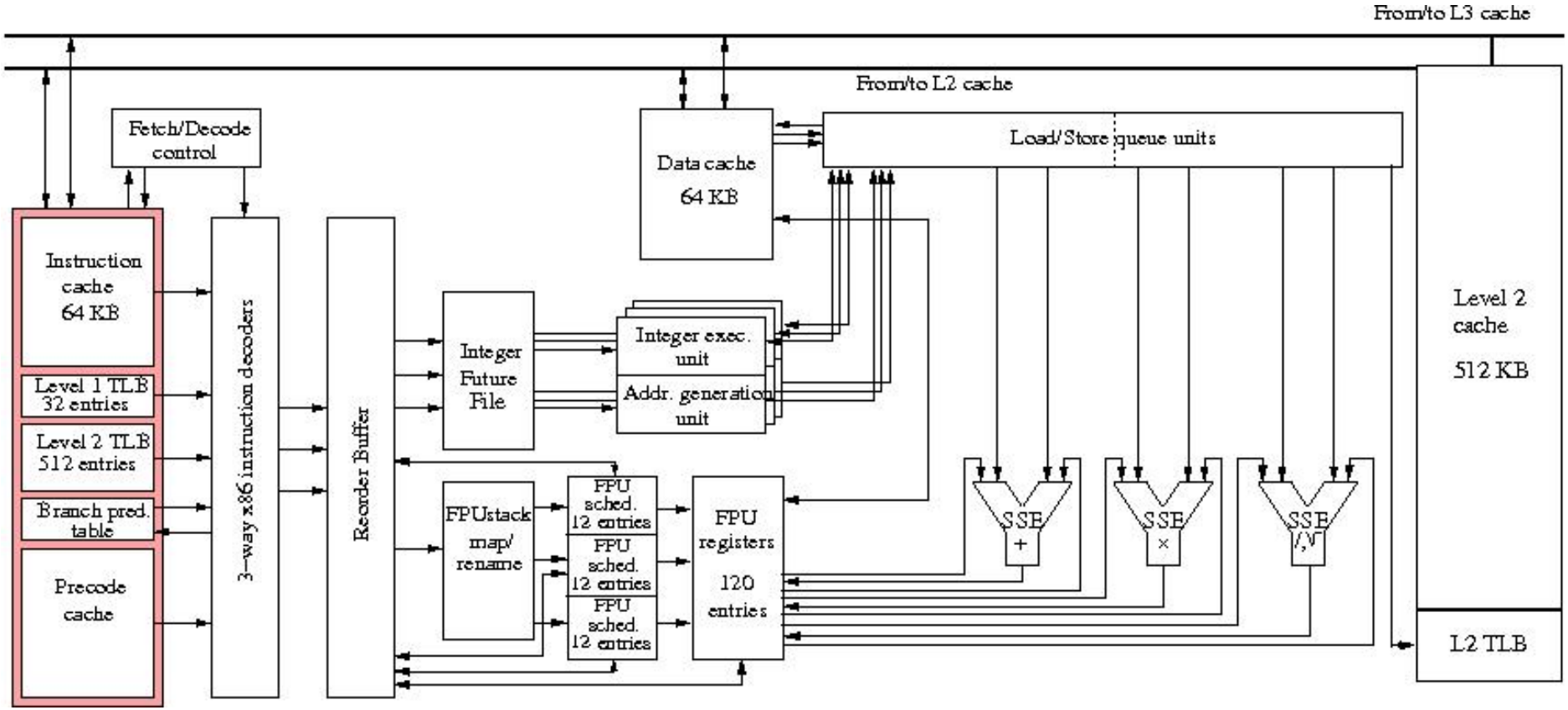
What does a CPU look like?



What does it mean?



What is in a core?



Functional units

- Traditionally: one instruction at a time
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA
  $x \leftarrow c*x+y$
- Peak performance is several ops/clock cycle (currently up to 4)
- This is usually very hard to obtain

Pipelining

- A single instruction takes several clock cycles to complete
- Subdivide an instruction:
  - Instruction decode
  - Operand exponent align
  - Actual operation
  - Normalize
- Pipeline: separate piece of hardware for each subdivision
- Compare to assembly line

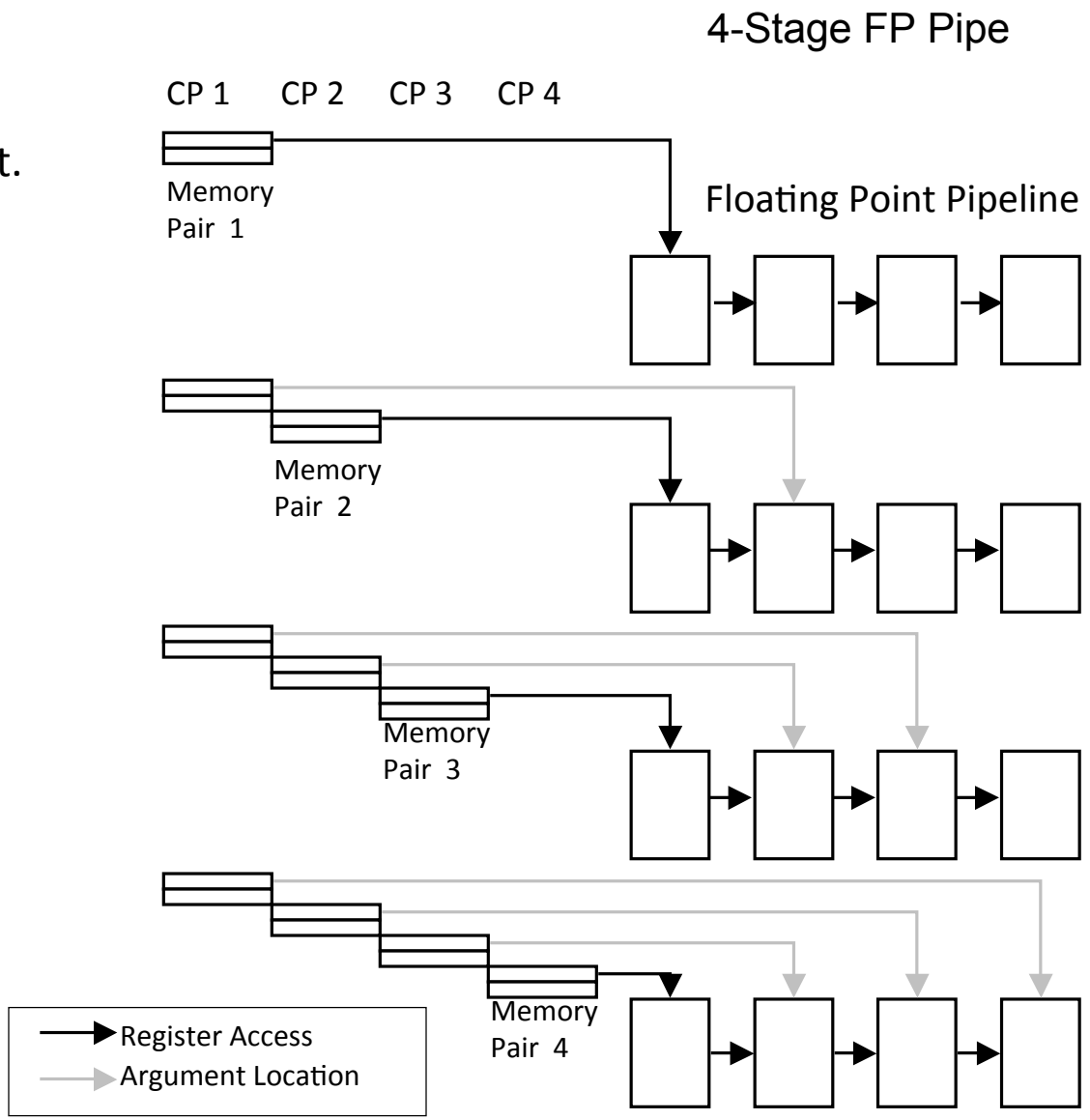
Pipeline

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.




TACC

Pipeline analysis: $n_{1/2}$

- With *s* segments and *n* operations, the time without pipelining is *sn*
- With pipelining it becomes *s+n-1+q* where *q* is some setup parameter, 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*

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

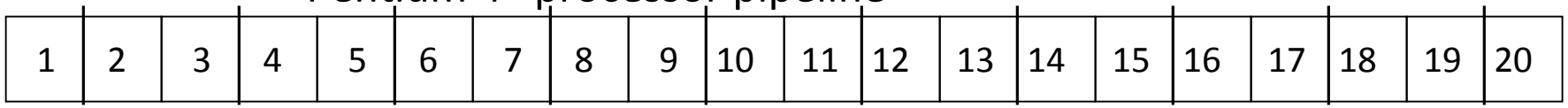
As long as instructions follow each other predictably everything is fine.

Branch Prediction

- The “instruction pipeline” is all of the 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 loop to branch route outcome, it is predicted.

Pentium III processor pipeline

Pentium 4 processor pipeline



Misprediction is more “expensive” on Pentium 4’s.

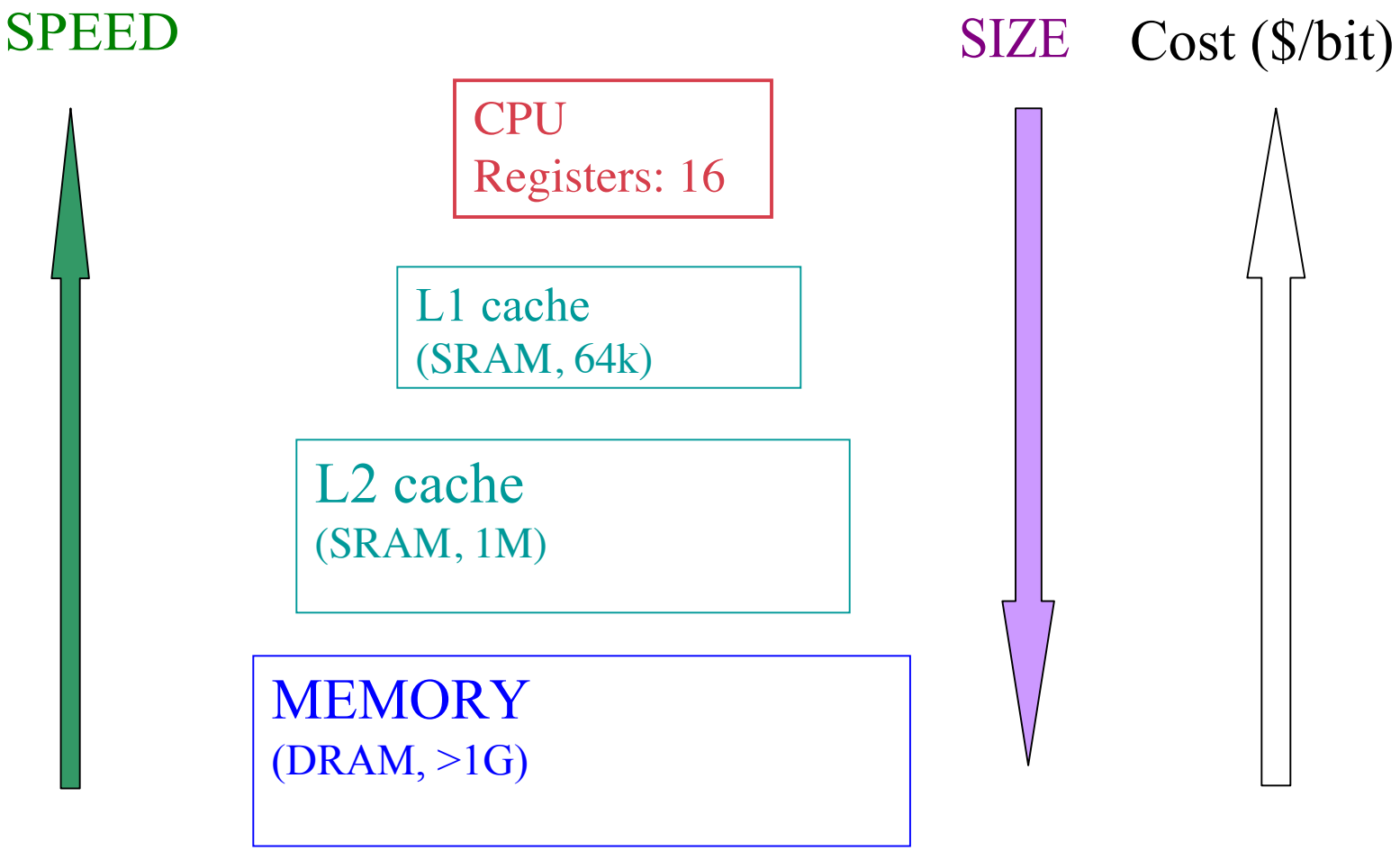
Memory Hierarchies

- Memory is too slow to keep up with the processor
  - 100--1000 cycles latency before data arrives
  - Data stream maybe 1/4 fp number/cycle; processor wants 2 or 3
- At considerable cost it's possible to build faster memory
- Cache is small amount of fast memory

Memory Hierarchies

- Memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory
- Memory is accessed through the hierarchy
  - registers where possible
  - ... then the caches
  - ... then main memory

Memory Relativity



Latency and Bandwidth

- The two most important terms related to performance for memory subsystems and for networks are:
  - Latency
    - How long does it take to retrieve a word of memory?
    - Units are generally nanoseconds (milliseconds for network latency) or clock periods (CP).
    - Sometimes addresses are predictable: compiler will schedule the fetch. Predictable code is good!
  - Bandwith
    - What data rate can be sustained once the message is started?
    - 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
- Two such loads have to be separated by at least the memory latency
- 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
- Multiply by bandwidth:

Little's law: Concurrency = Bandwidth x Latency

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*
- Requires fast context switch

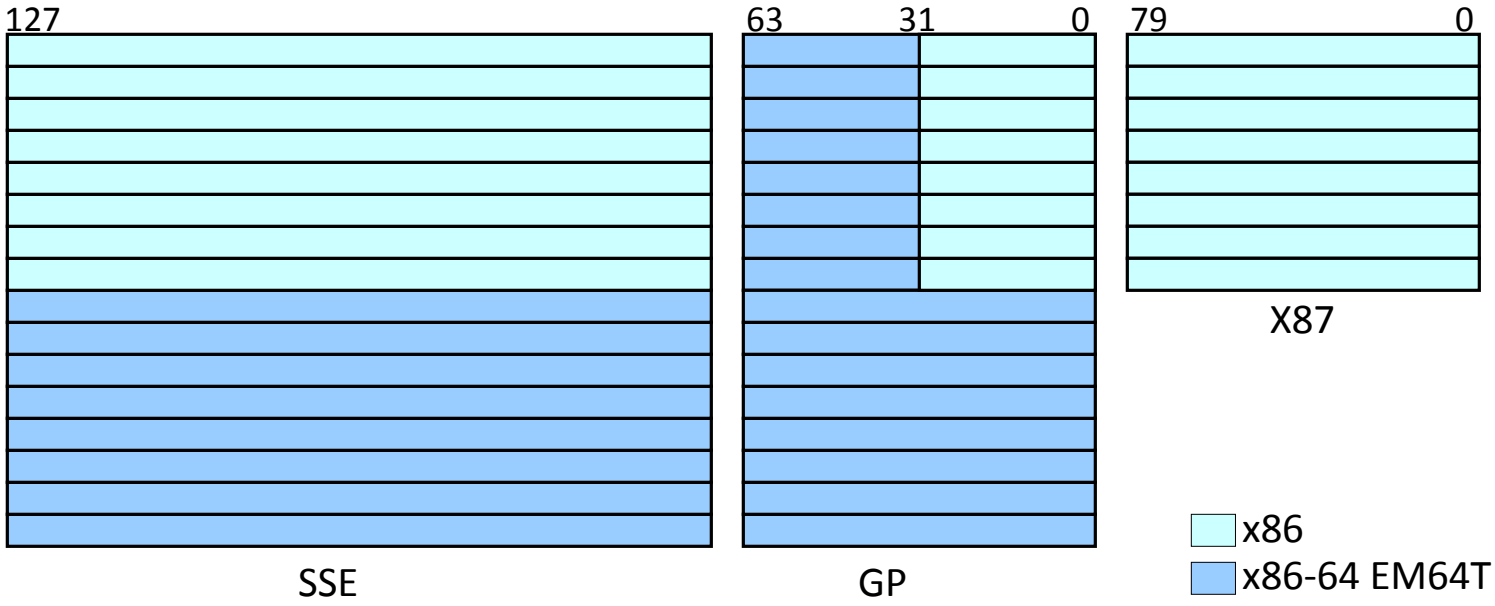
How good are GPUs?

- Reports of 400x speedup
- Memory bandwidth is about 6x better
- CPU peak speed hard to attain:
  - Multicores, lose factor 4
  - Failure to pipeline floating point unit: lose factor 4
  - Use of multiple floating point units: another 2

The memory subsystem in detail

Registers

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



Registers

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

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

- Separate instructions and registers for floating-point operations

Data Caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
  - Data from L2 has to go through L1 to registers
  - L2 is 10 to 100 times larger than L1
  - Some systems have an L3 cache, ~10x larger than L2
- Cache line
  - The smallest unit of data transferred between main memory and the caches (or between levels of cache)
  - *N* sequentially-stored, multi-byte words (usually N=8 or 16).

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 you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad

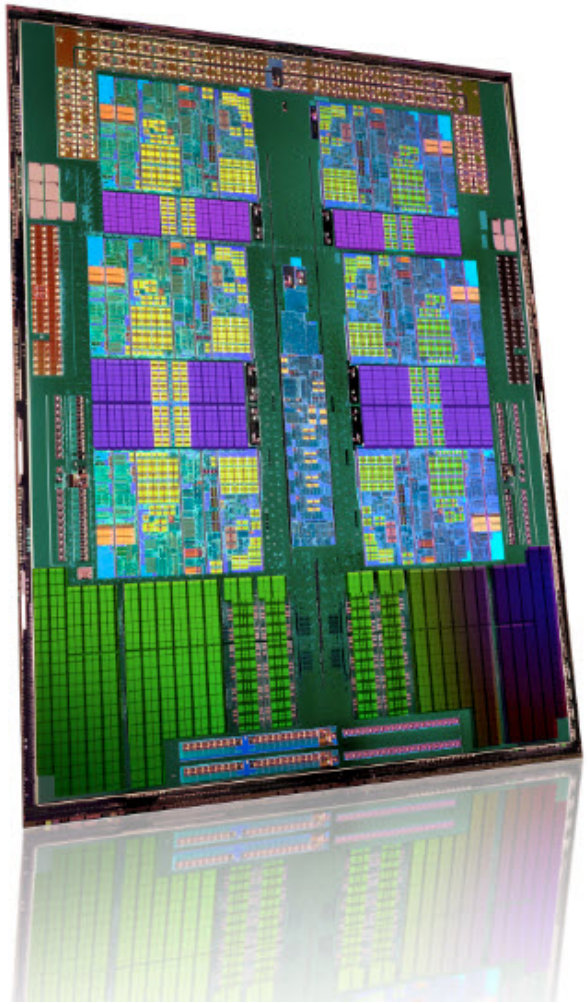
Main Memory

- Cheapest form of RAM
- Also the slowest
  - lowest bandwidth
  - highest latency
- Unfortunately most of our data lives out here

Multi-core chips

- What is a processor? Instead, talk of “socket” and “core”
- Cores have separate L1, shared L2 cache
  - Hybrid shared/distributed model
- Cache coherency problem: conflicting access to duplicated cache lines.

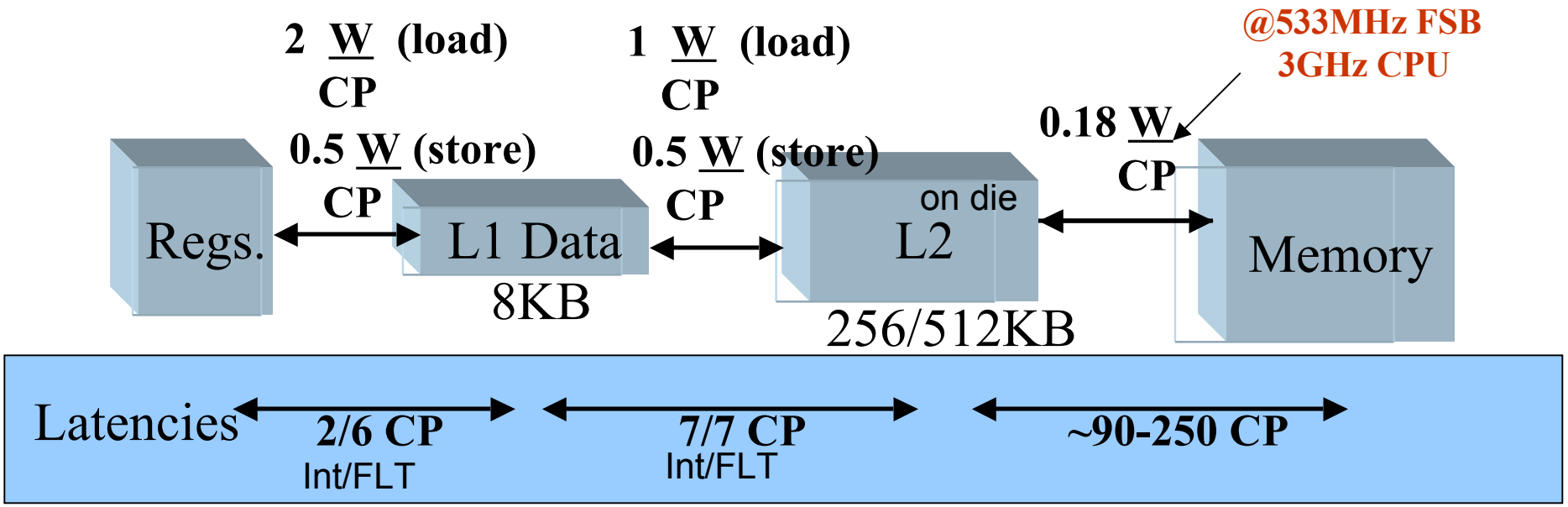
That Opteron again…



Approximate Latencies and Bandwidths in a Memory Hierarchy

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

Example: Pentium 4



Line size L1/L2 =8W/16W

Cache and register access

- Access is transparent to the programmer
  - data is in a register or in cache or in memory
  - Loaded from the highest level where it's found
  - processor/cache controller/MMU hides cache access from the programmer
- …but you can influence it:
  - Access *x* (that puts it in L1), 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
  - If you loop over data, try to take chunks of less than cache size
  - C declare register variable, only suggestion

Register use

- y[i] can be kept in register
- Declaration is only suggestion to the compiler
- Compiler can usually 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];
  }
}
```

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

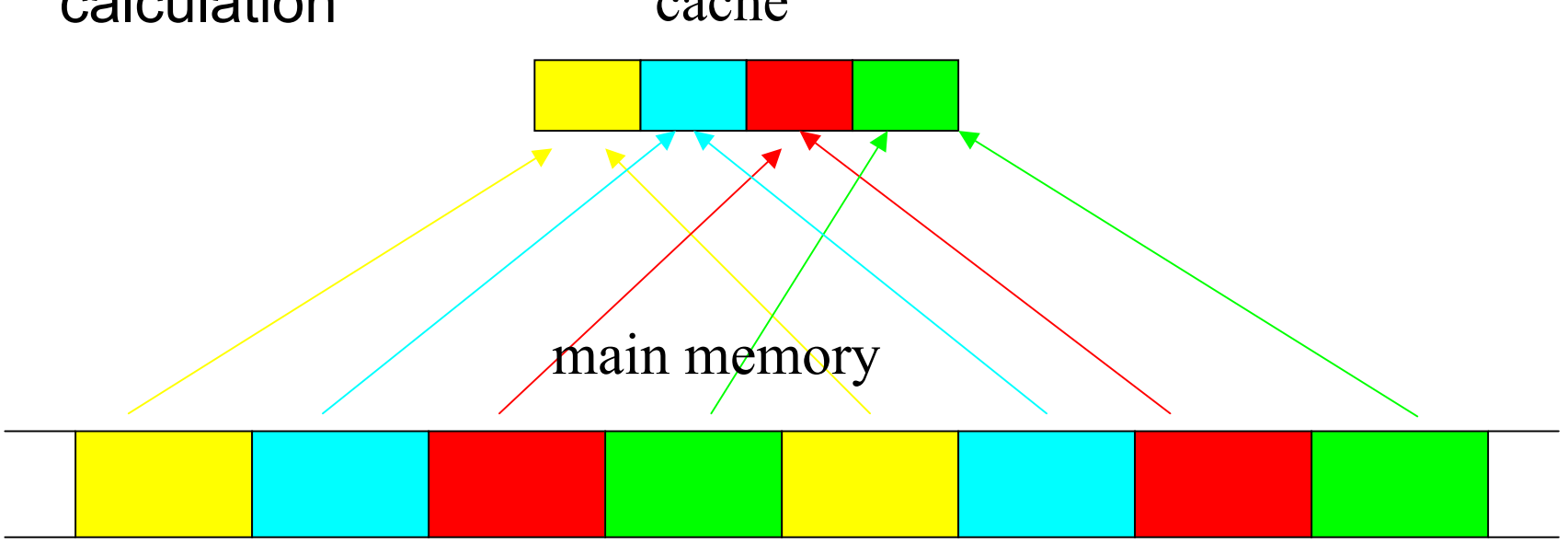
Cache Mapping

- 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



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 64K (or 8K words): 16 bits
- Map by taking last 16 bits
- (why last?)
- (how many different memory locations map to the same cache location?)
- (if you walk through a double precision array, *i* and *i*+*k* map to the same cache location. What is *k*?)

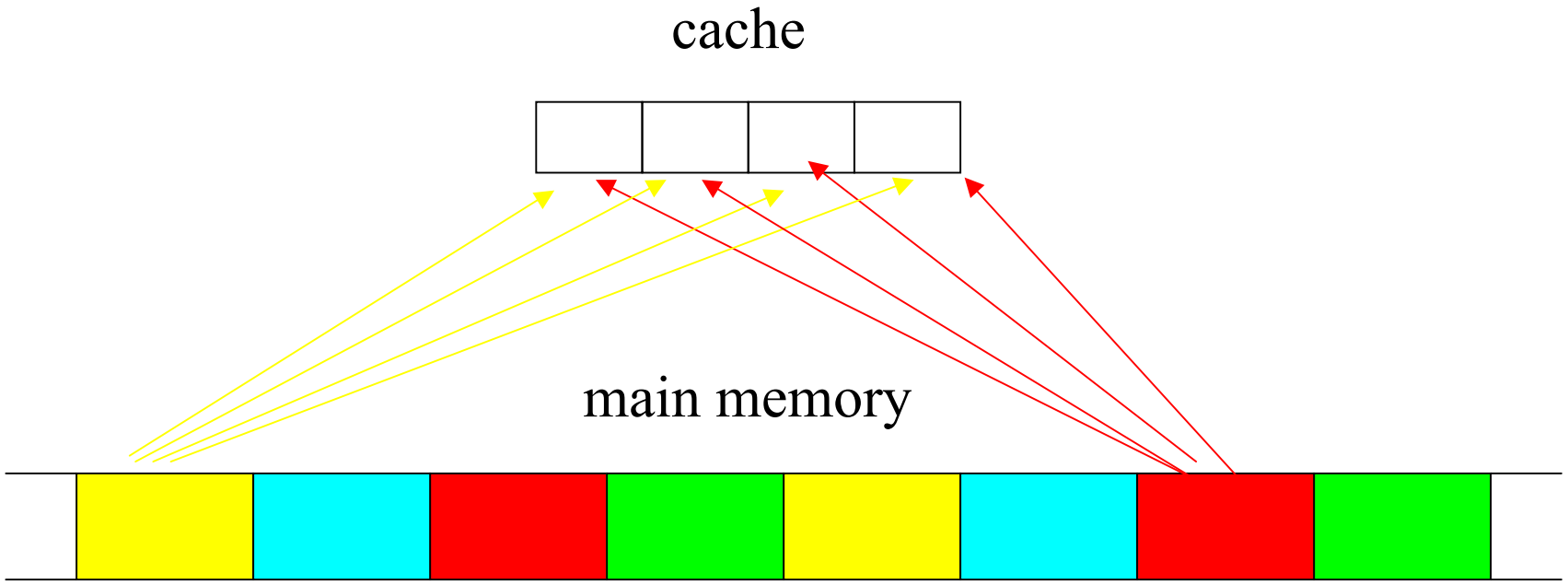
The problem with Direct Mapping

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

- 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*

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.

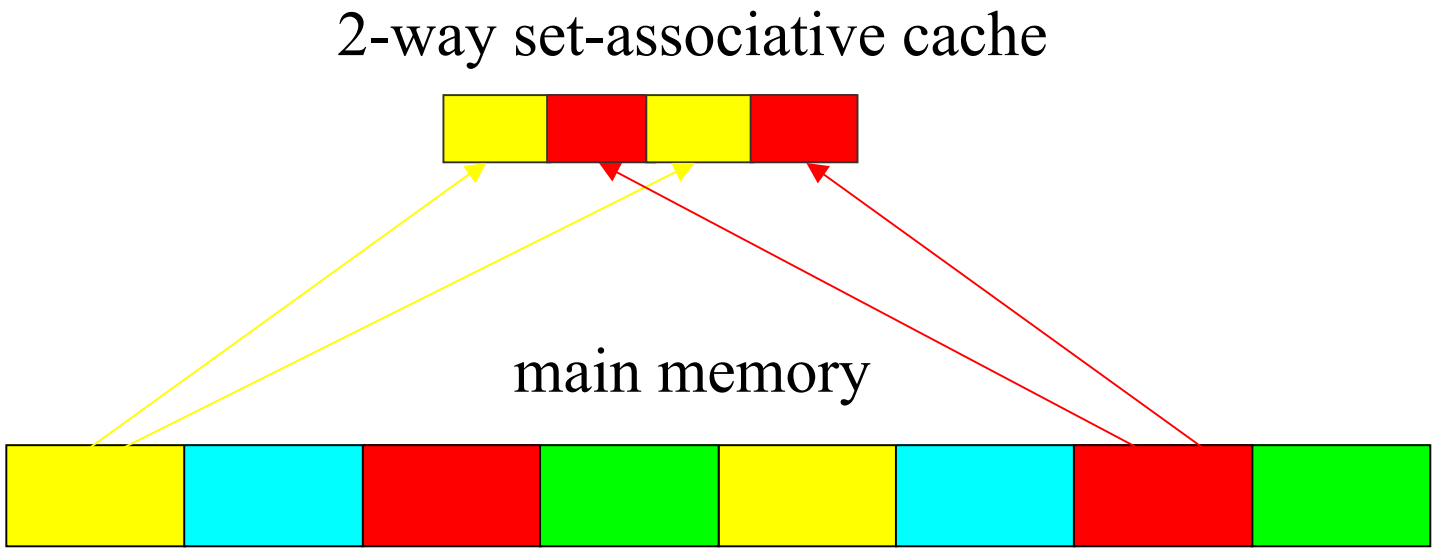


Fully Associative Caches

- 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 at least 2) 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

TLB

- Translation Look-aside Buffer
- Translates between logical space that each program has and actual memory addresses
- Memory organized in ‘small pages’, a few Kbyte 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.

Prefetch

- Hardware tries to detect if you load regularly spaced data:
- “prefetch stream”
- This can be programmed in software, often only in-line assembly.

Theoretical analysis of performance

- Given the different speeds of memory & processor, the question is: does my algorithm exploit all these caches? Can it 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 $x_i=x_i+y_i$: 1op, 3 mem accesses
- Example: inner product $s=s+x_i*y_i$: 2op, 2 mem access (s in register; also no writes)

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    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 so*
- (This is the important kernel in the Linpack benchmark.)

Reuse analysis: matrix-vector product

```
for (i=0; i<m; i++) {
  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++) {
  s = 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 behaviour matrix stored by rows and columns

Reuse analysis(2): matrix-vector product

```
for (i=0; i<m; i+=2) {
  s1 = 0.; s2 = 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

```
a1 = &(a[0][0]);
a2 = a1+n;
for (i=0,ip=0; i<m/2; i++) {
  s1 = 0.; s2 = 0.;
  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 (loop=0; loop<10; loop++) {
  for (i=0; i<N; i++) {
     ... = ... x[i] ...
  }
}

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

Original loop:
long time between uses of *x*,
Rearrangement:
x is reused

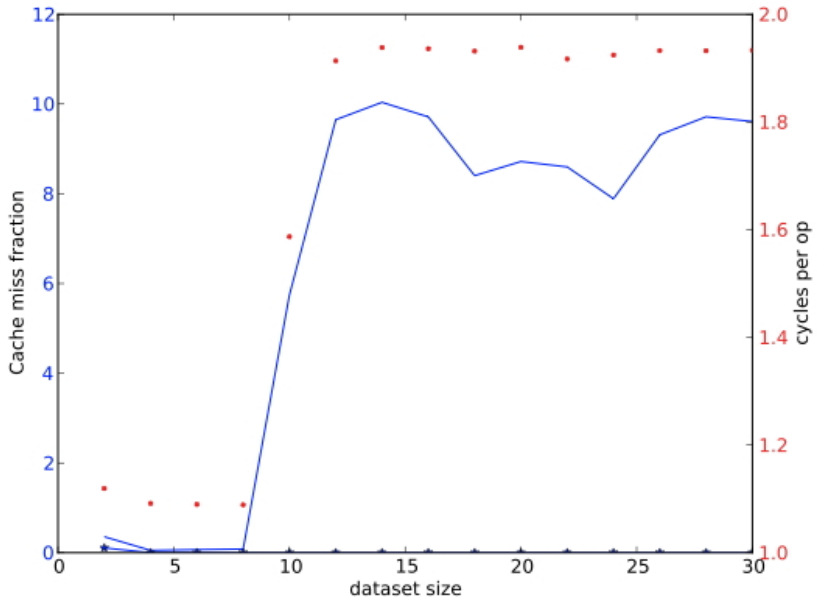
Spatial Locality

- 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 (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[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

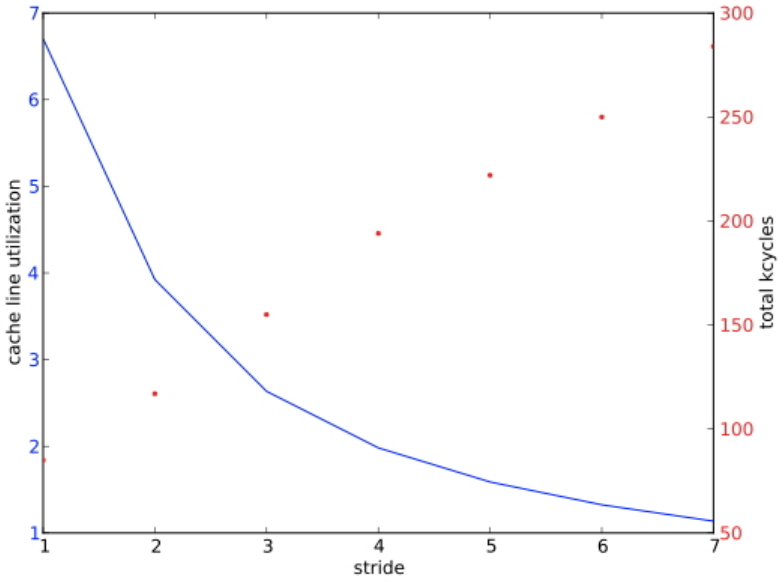
```
for (i=0; i<NRUNS; i++) {
   blockstart = 0;
   for (b=0; b<size/l1size; b++)
     for (j=0; j<l1size; j++)
       array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
 }
```

- 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;
```

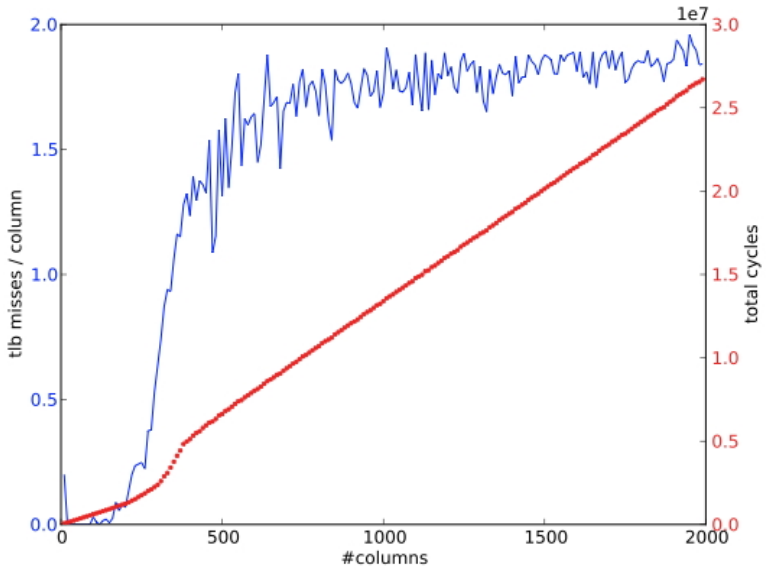
- Same amount of data, but increasing stride
- Increasing stride: more cachelines loaded, slower execution



TLB

```
#define INDEX(i,j,m,n) i+j*m
  array = (double*) malloc(m*n*sizeof(double));

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

- Array is stored with columns contiguous
- Loop traverses the columns:
- No big jumps through memory
- (max: 2000 columns, 3000 cycles)

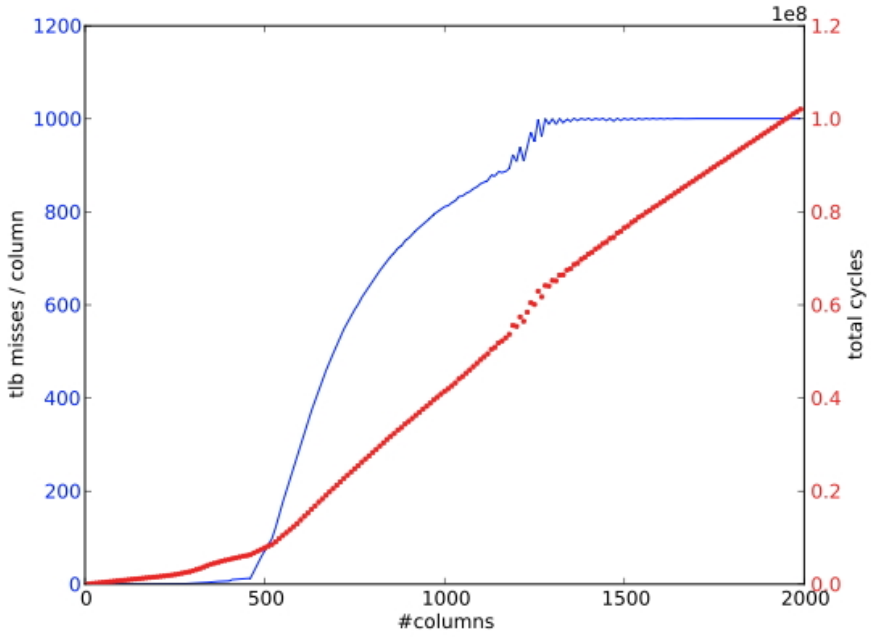


TACC

TLB

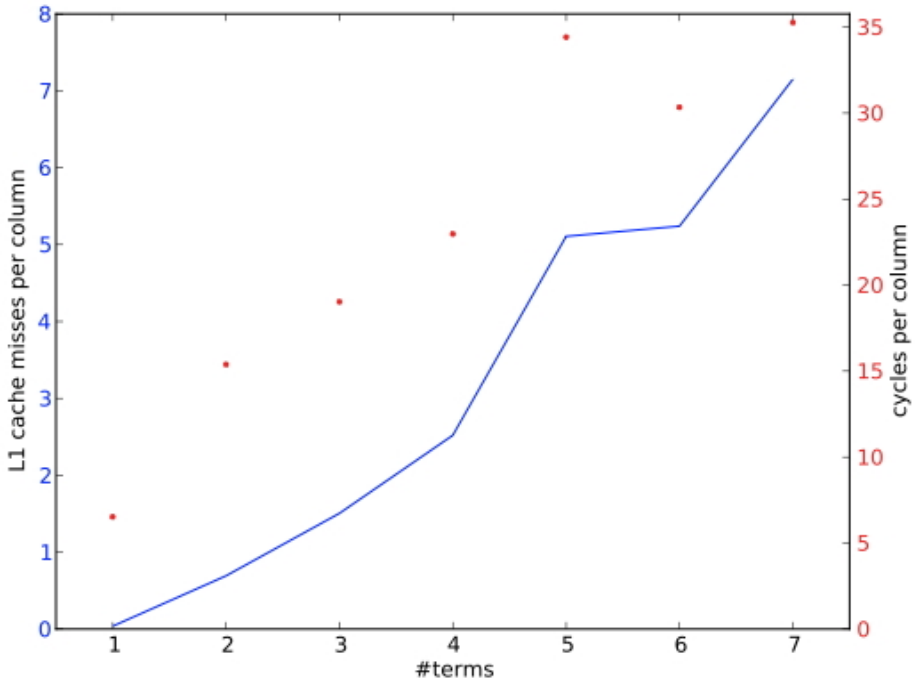
```
#define INDEX(i,j,m,n) i+j*m
  array = (double*) malloc(m*n*sizeof(double));

  /* traversal #2 */
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

- Traversal by columns:
- Every next column is *n* words away
- If *n* more than 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 cache 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)