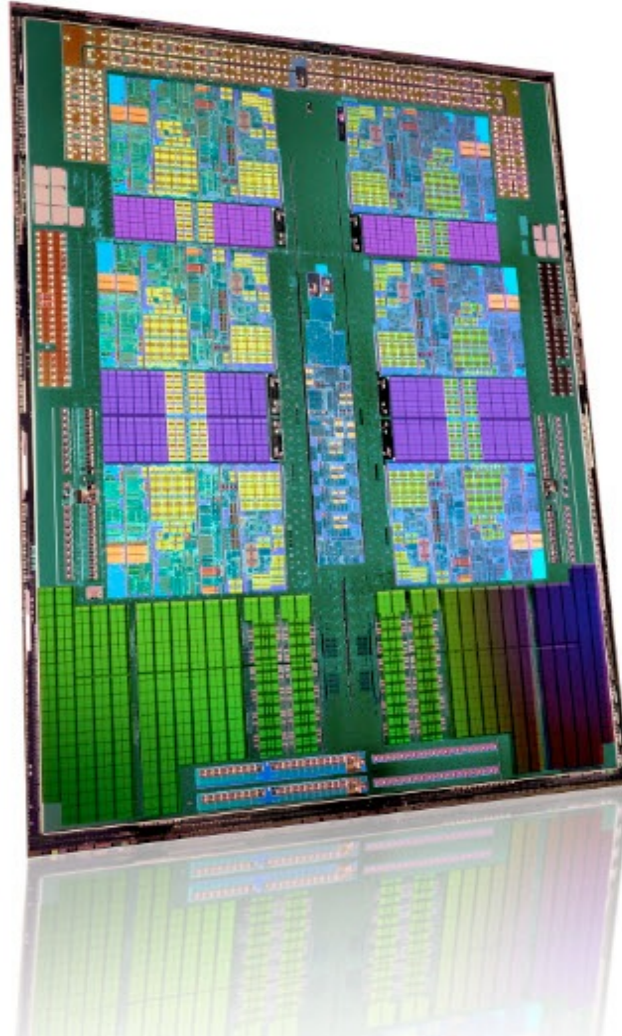
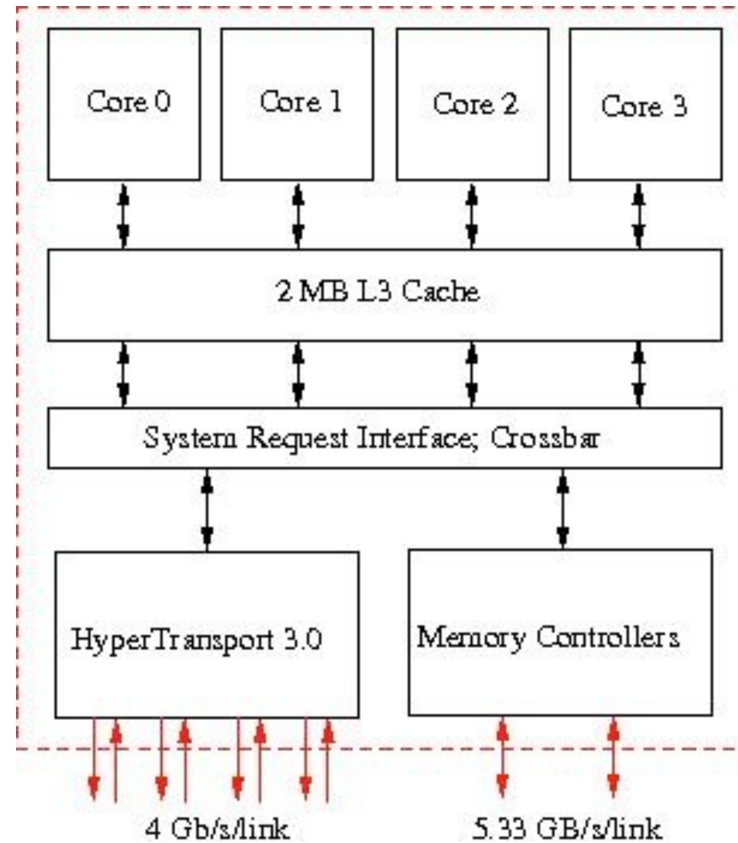
Computer Architecture
Single CPU

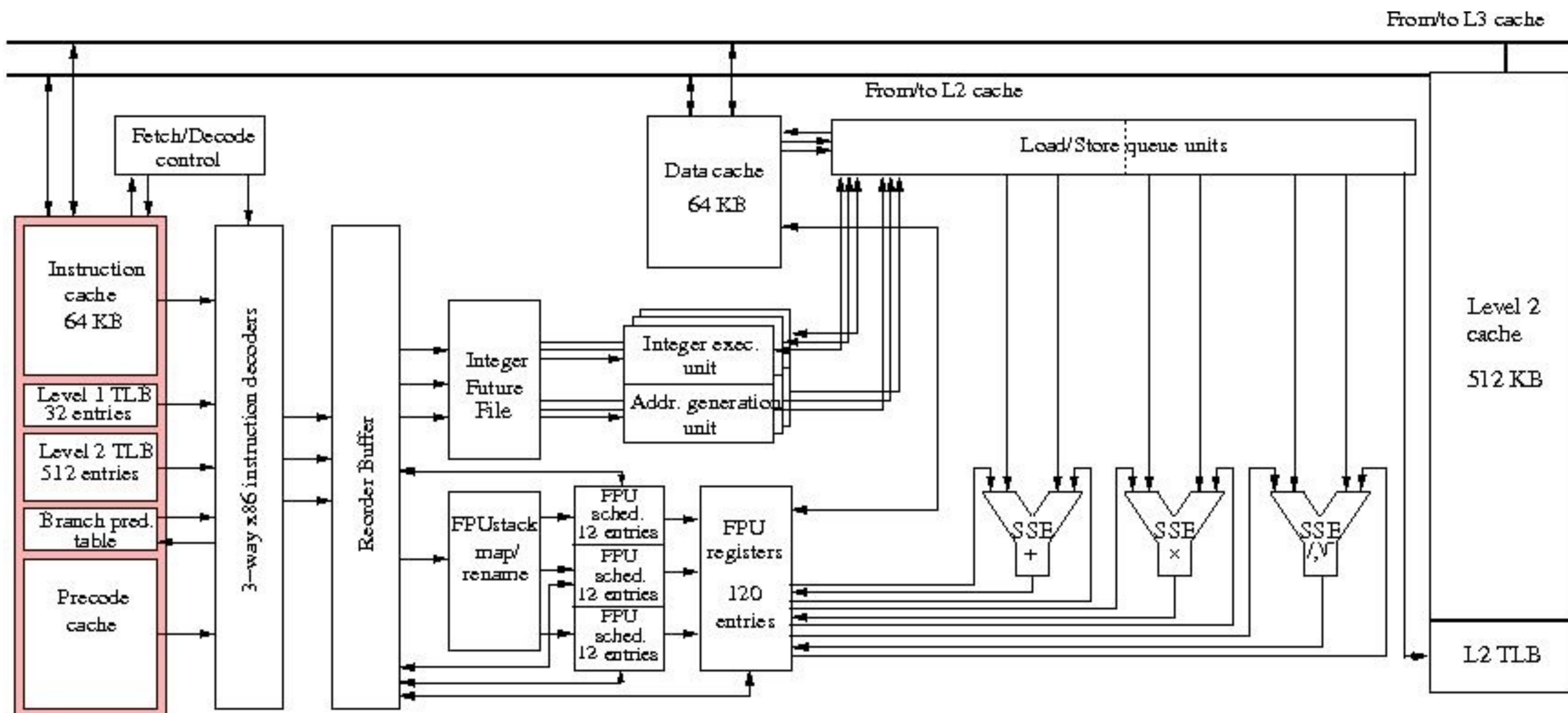
What Does a CPU Look Like?



What Does it Mean?



What is in a Core?



Von Neumann Architecture

Undivided memory that stores both program and data ('stored program') + processing unit that executes the instructions, operating on the data

- 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

Scientific Computing

- Some algorithms are “CPU-bound”
  - the speed of the processor is the most important factor
- Some algorithms are “memory-bound”
  - bus speed, cache size become important

“Memory-bound” becomes ever more prominent...

Simple “GHz” comparison does not tell the whole story!

Modern Floating Point Units

- Traditionally: one instruction at a time
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA (“Fused multiply-add”)

$$x \leftarrow ax + b$$

- Peak performance is several ops/clock cycle (currently up to 4); usually very hard to obtain
- Other operations not as optimized: a division requires 10 to 20 clock cycles

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

Pipelining

Example: addition of 2 fp numbers $.35 \times 10^{-1} + .6 \times 10^{-2}$ has the following subdivision ("components", "stages", "segments")

- Decoding the instruction, including finding the locations of the operands
- Copying the operands into registers ('data fetch').
- Aligning the exponents: $.35 \times 10^{-1} + .06 \times 10^{-1}$
- Executing the addition of the mantissas: $.41$
- Normalizing the result: $.41 \times 10^{-1}$
- Storing the result

Pipelining

Every component designed to finish in 1 clock cycle: the whole instruction takes 6 cycles

If each has its own hardware, one can execute two operations in less than 12 cycles:

- Execute the decode stage for the first operation;
- Do the data fetch for the first operation, and at the same time the decode for the second.
- Execute the third stage for the first operation and the second stage of the second operation simultaneously.
- ...

Pipelining

Analysis:

- First addition takes 6 clock cycles
- Second addition finishes a mere 1 cycle later

This idea can be extended to more than two operations: the first operation still takes the same amount of time as before, but after that one more result will be produced each cycle.

Executing n operations on a s-segment pipeline takes $(s + n - 1)$ cycles, as opposed to (ns) in the classical case.

This requires independent operations... One solution: multiple pipes

Pipelining

With pipelining, peak CPU performance =

(clock speed)

x

(number of independent floating point units)

The measure of floating point performance is ‘floating point operations per second’, abbreviated “flops”.

‘gigaflops’ = multiples of 10^9 flops

Pipelining Beyond Arithmetic

The whole CPU is pipelined, leading to "**Instruction Level Parallelism**" (ILP)

Facilitated by

- multiple issue (independent instructions can be started at the same time)
- branch prediction and speculative execution
- out-of-order execution

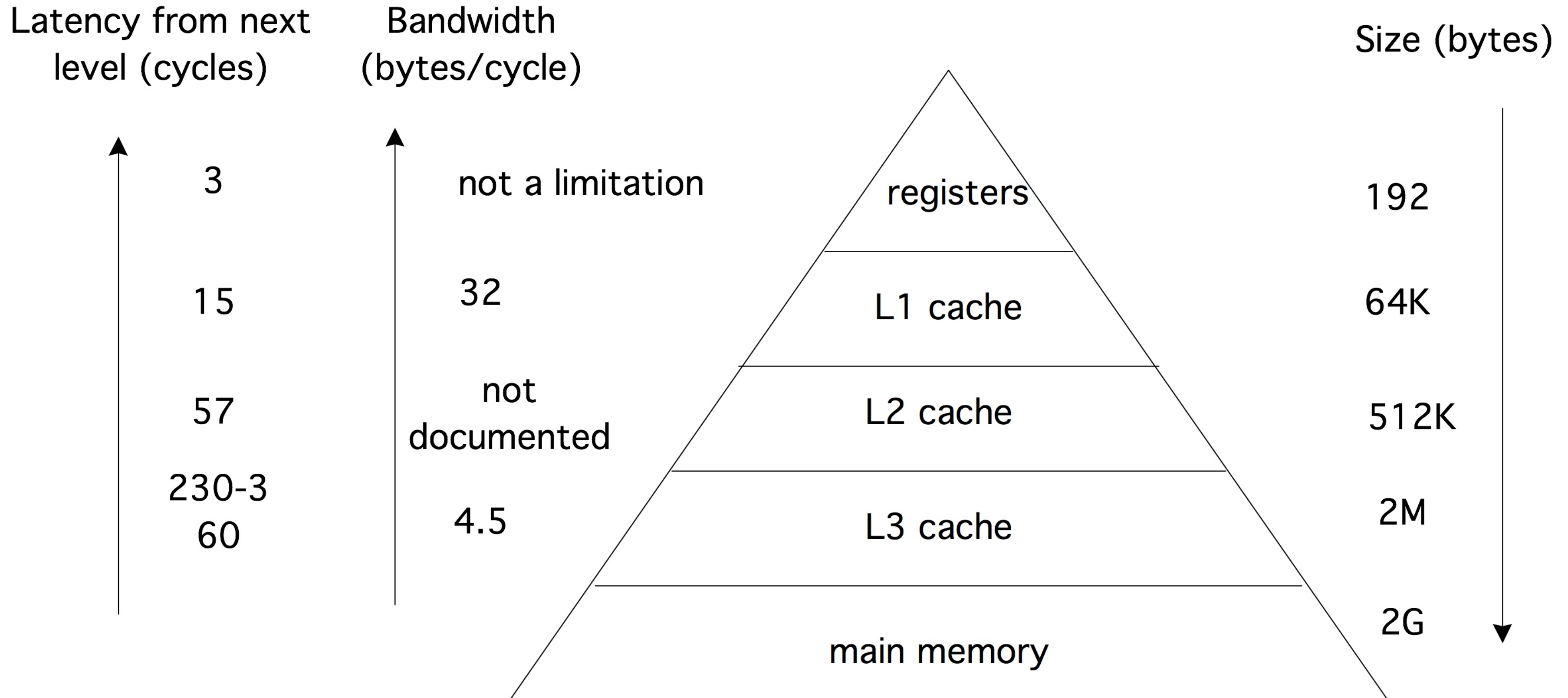
Memory Hierarchies

- 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
  - “Memory wall”
- 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 Hierarchies



AMD Opteron

Latency and Bandwidth

- The two most important terms related to performance for memory subsystems and for networks:
- **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!
- **Bandwidth**
  - What data rate can be sustained once the message is started?
  - Units are B/sec (MB/sec, GB/sec, etc.)

Latency and Bandwidth

- The time that a message takes from start to finish combines latency and bandwidth:

$$T(n) = \alpha + \beta n$$

- α latency
- β inverse of bandwidth (the time per byte)

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

PS: Latency Hiding and GPUs

- Finding parallelism is sometimes called `latency hiding': load data early to hide latency
- GPUs do latency hiding by spawning many threads
- Requires fast context switch

Registers

- Highest bandwidth, lowest latency memory that a modern processor can access; built into the CPU
- Often a scarce resource and not random access
- 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

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

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
  - In 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

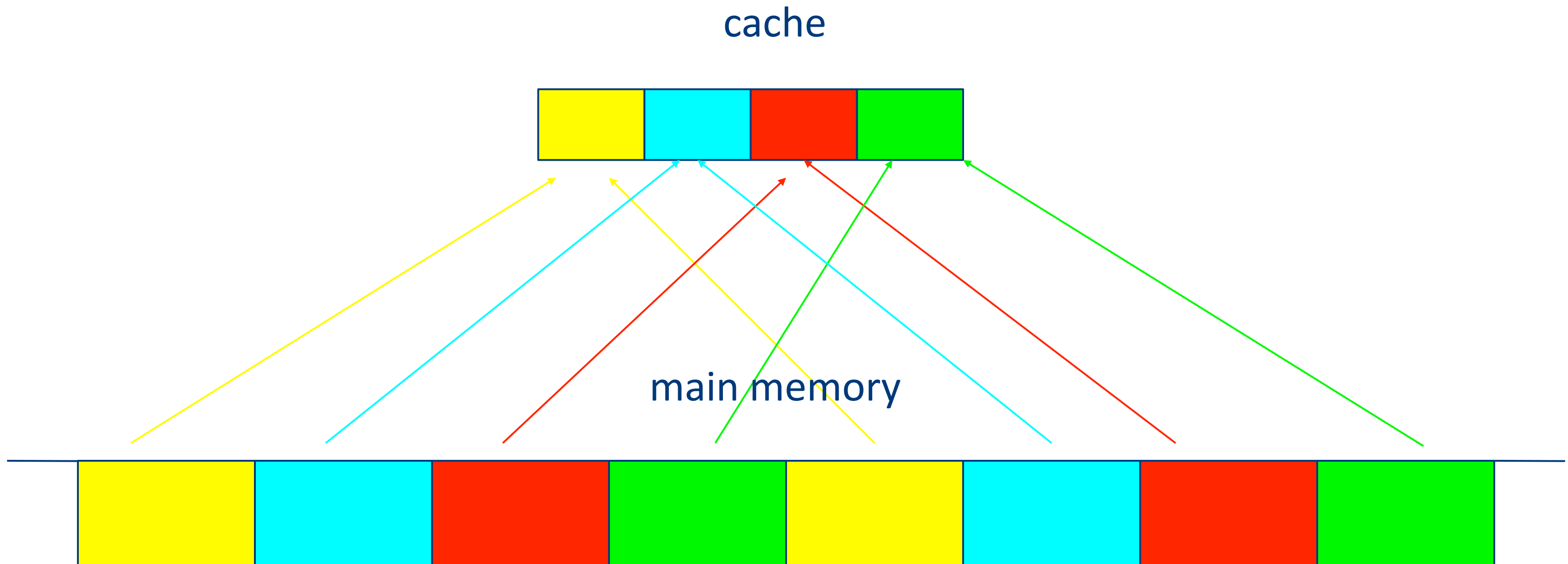
- Because each memory level is smaller than the next-closer level, data must be mapped

- Types of mapping
  - Direct
  - Set associative
  - Fully associative

Direct Mapped 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. keep 16 last bits of memory address)



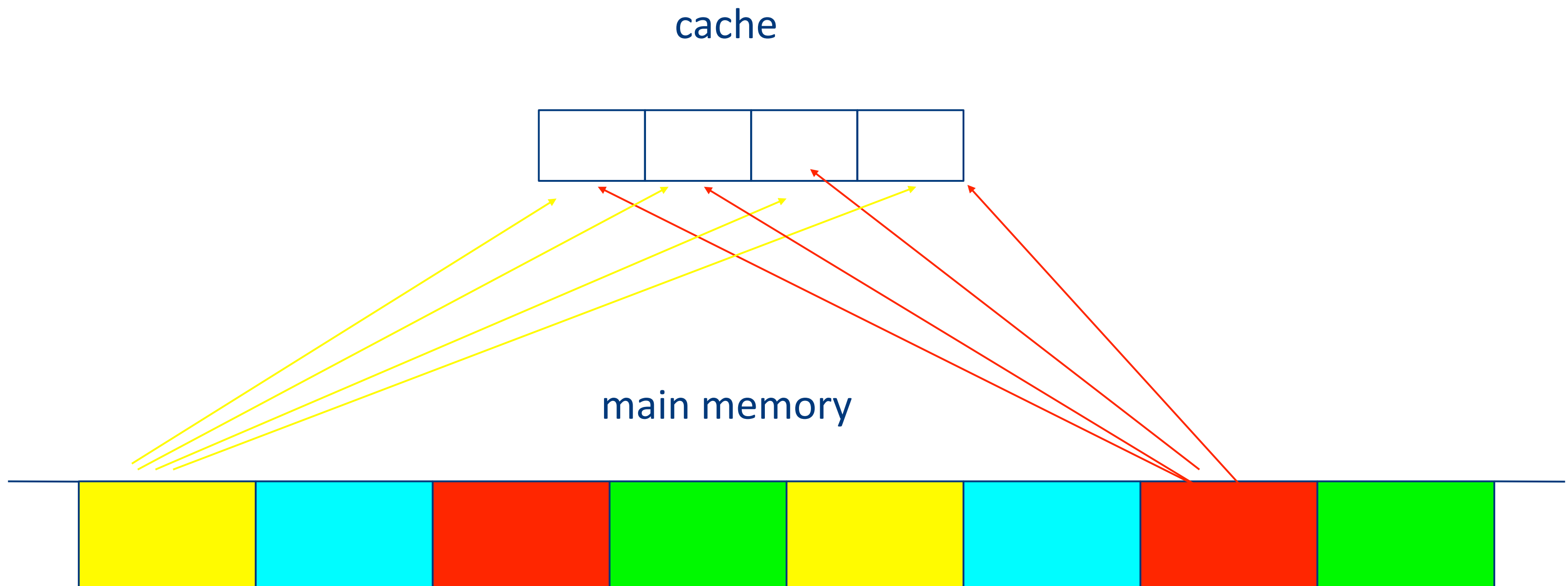
The Problem with Direct Mapping

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

- Example: cache size 64K, needs 16 bits to address
- a[0] and b[0] 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

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.

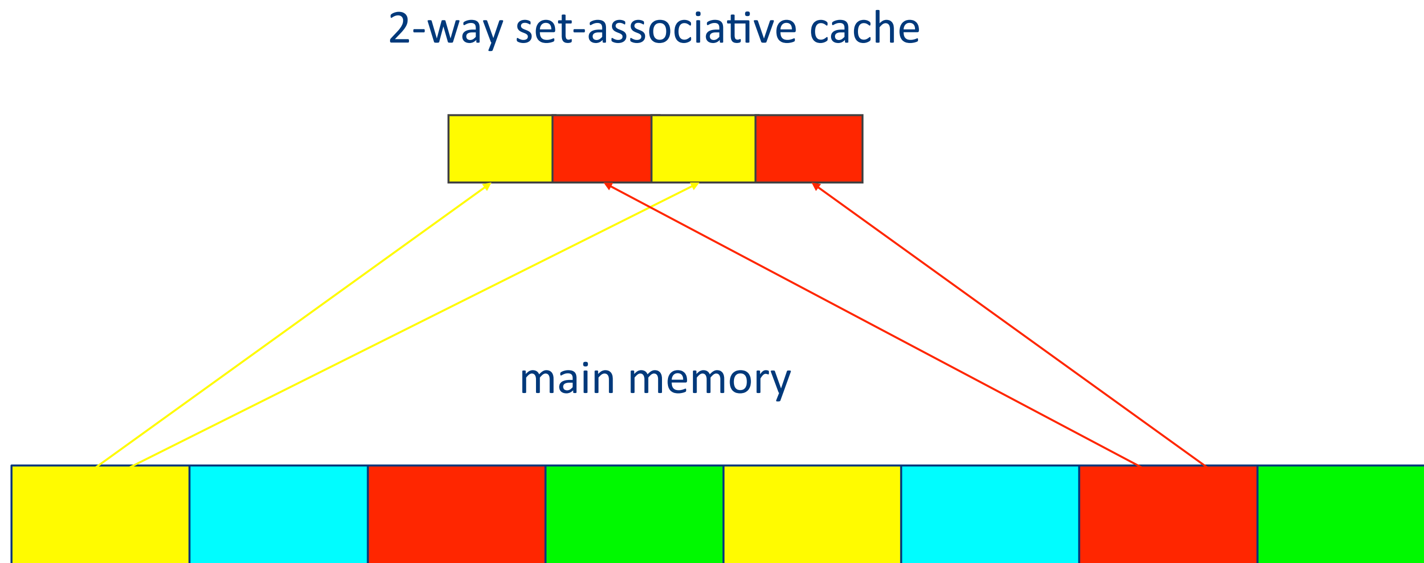


Fully Associative Caches

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

Set Associative Caches

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

Translation Look-Aside Buffer (TLB)

- 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 sometimes be programmed in software, often only in-line assembly

Data reuse

- Performance is limited by data transfer rate
- High performance if data items are used multiple times
- Examples:
  - vector addition $x_i=x_i+y_i$: 1op, 3 mem accesses
  - inner product $s=s+x_i*y_i$: 2op, 2 mem access (s in register; also no writes)

Data reuse: matrix-matrix product

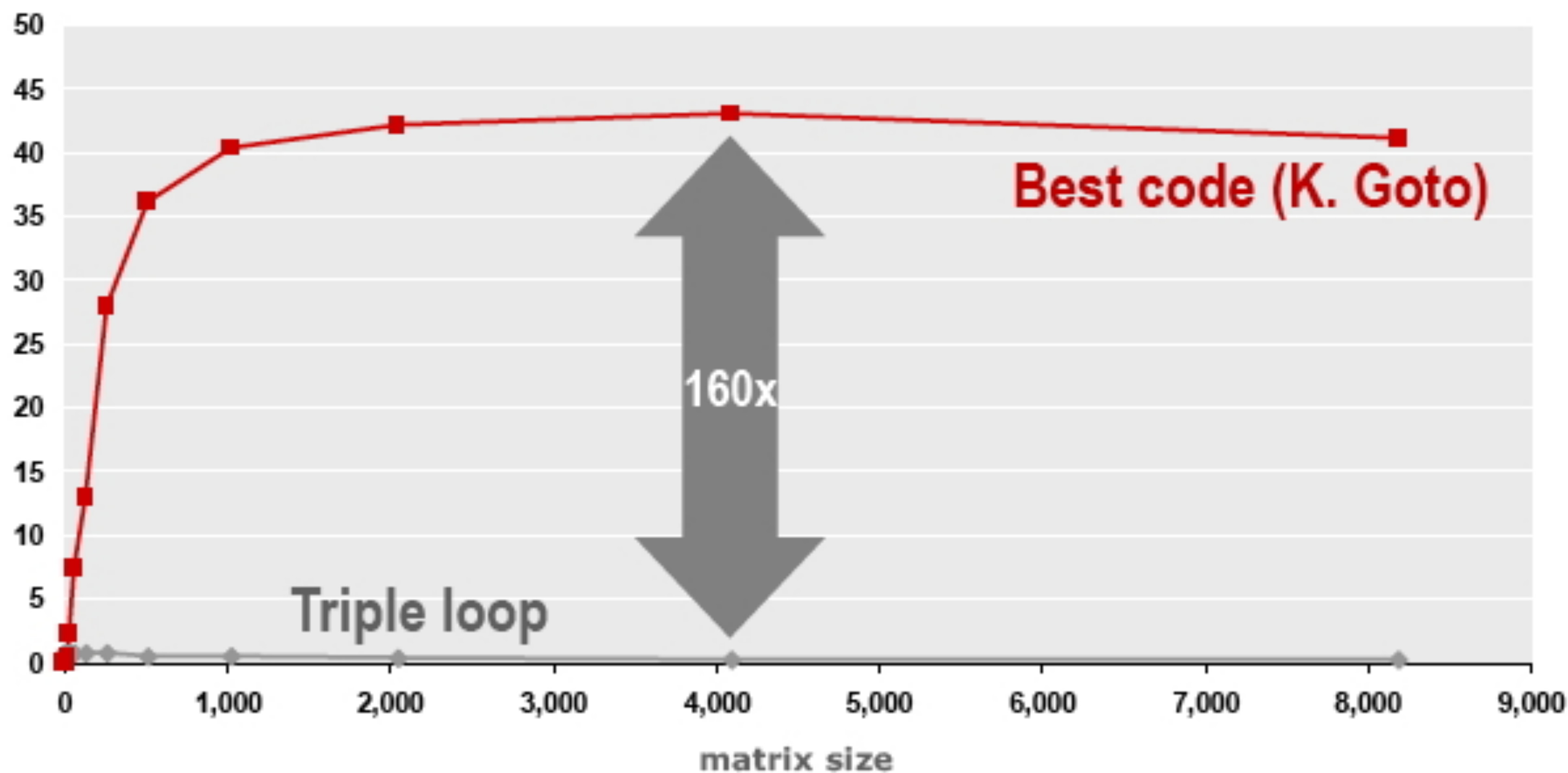
- Matrix-matrix product: $2n^3$ ops, $2n^2$ data

```
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
  - Data reuse is *O(n)*: every data item is used *O(n) times*
- If it can be programmed right, this can overcome the bandwidth/cpu speed gap
- Again only theoretically: naïve implementations are inefficient... *so do not code this yourself*: use BLAS (MKL, Atlas, etc.)
- (This is the important kernel in the Linpack benchmark: cf. Top500)


Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)
Gflop/s
50
45
40
35
30
25
20
15
10
5
0
Best code (K. Goto)
160x
Triple loop
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
9,000
matrix size

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] ...
  }
}
```

Original loop:
long time between uses of *x*,

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

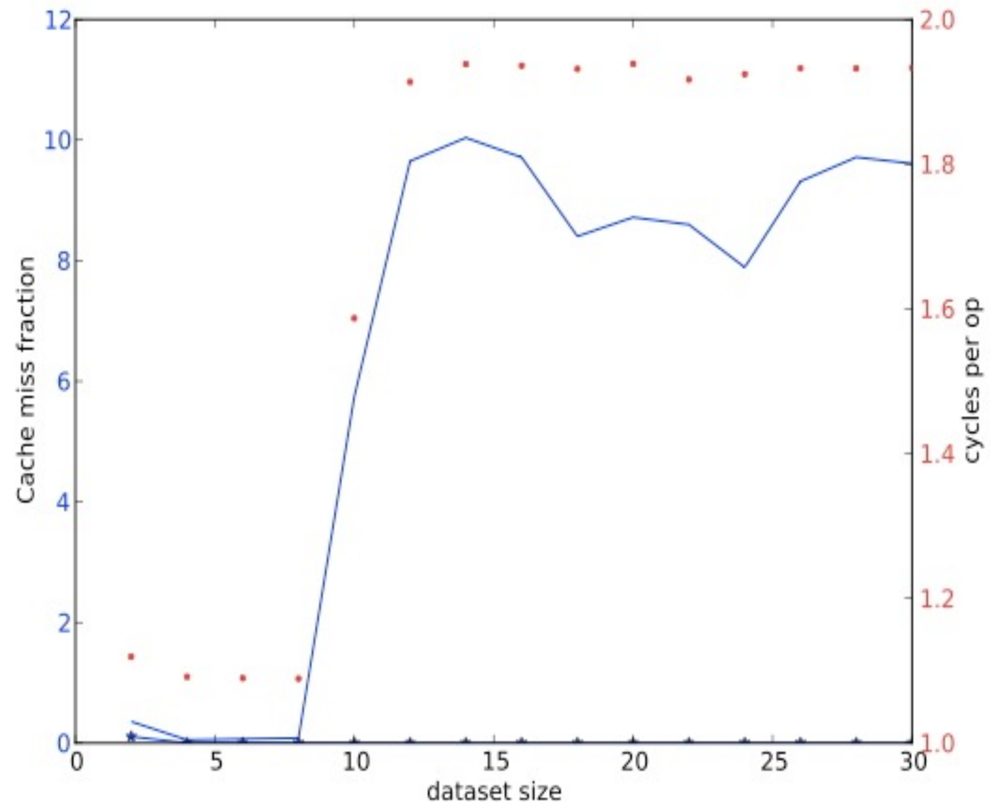
Rearrangement:
x is reused

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

Illustration: 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

Illustration: Cache size

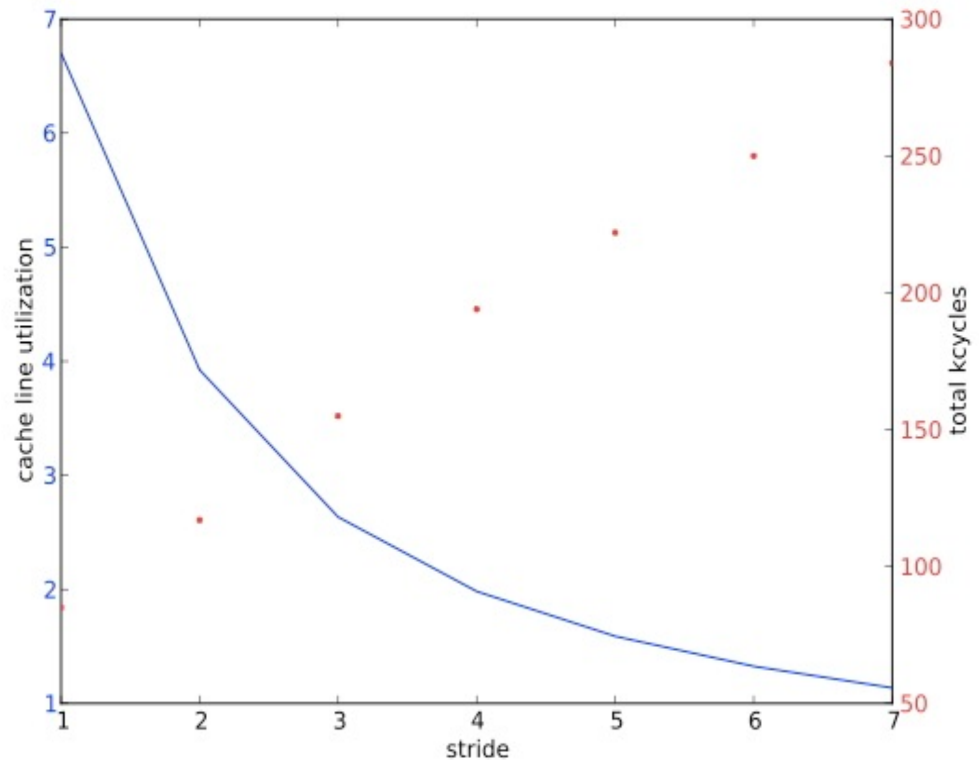
```
for (i=0; i<NRUNS; i++) {
   blockstart = 0;
   for (b=0; b<size/l1size; b++)
     for (j=0; j<l1size; j++)
       array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
 }
```

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

Illustration: Cache line utilization

```
 for (i=0,n=0; i<L1WORDS; i++,n+=stride)
    array[n] = 2.3*array[n]+1.2;
```

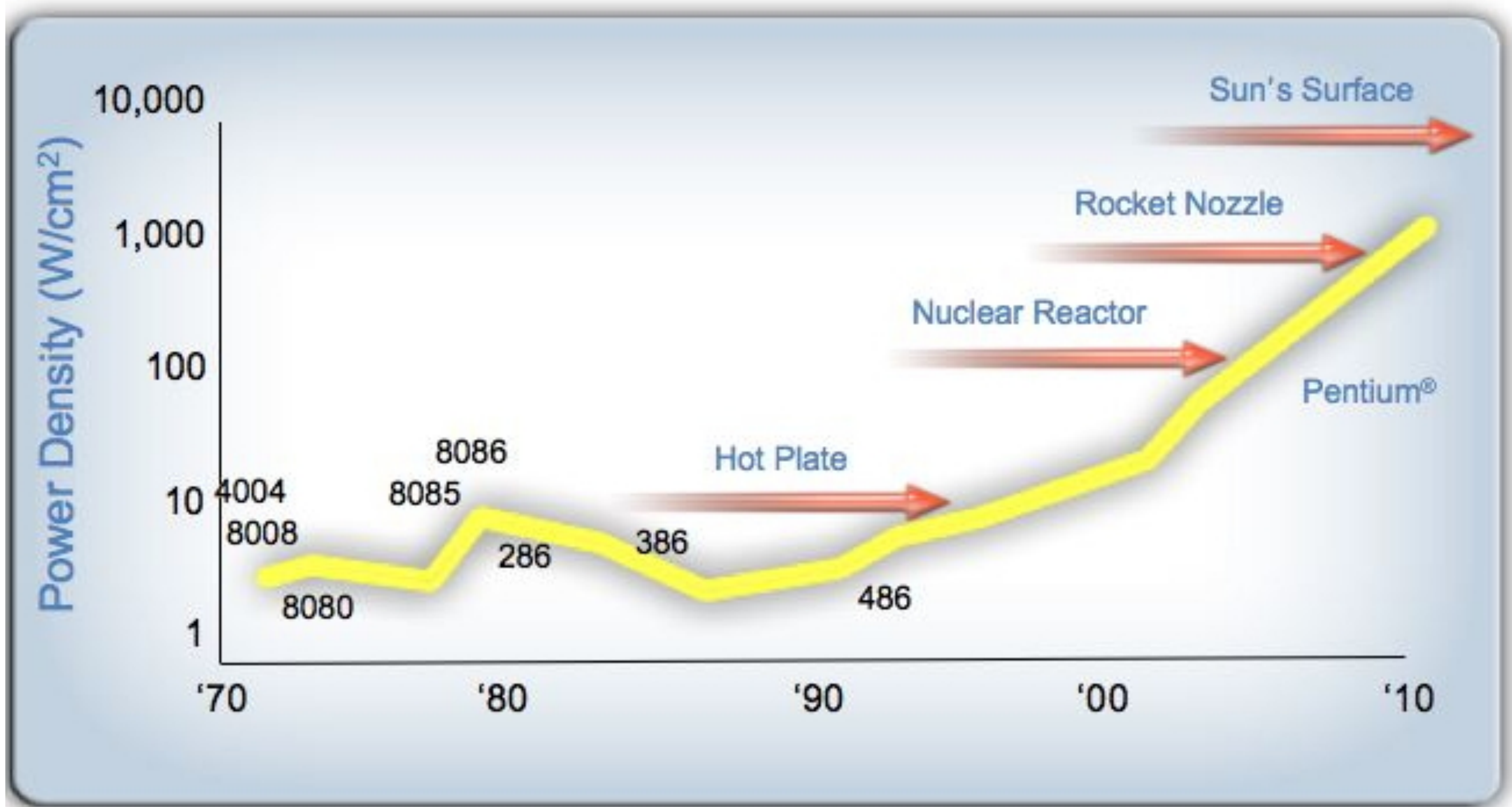
- Same amount of data, but increasing stride
- Increasing stride: more cachelines loaded, slower execution



Power Consumption

- Scale all geometrical features by s ($s < 1$):
  - dynamic power consumption P is scaled to s^2P
  - circuit delay T is scaled to sT
  - operating frequency F is changed to F/s
  - Energy consumption is scaled by s^3, and this gives us the space to put more components on a chip
- However, miniaturization of features is coming to a standstill due to laws of physics
- Increasing frequency would raise heat production
- -> “Power wall”

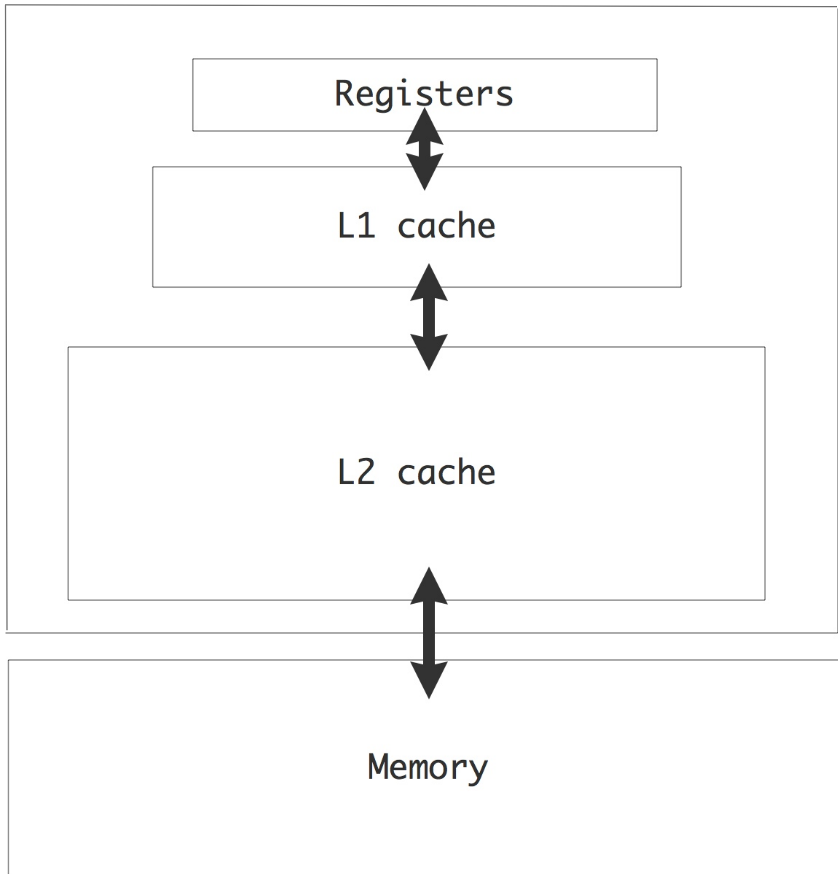
Power Consumption



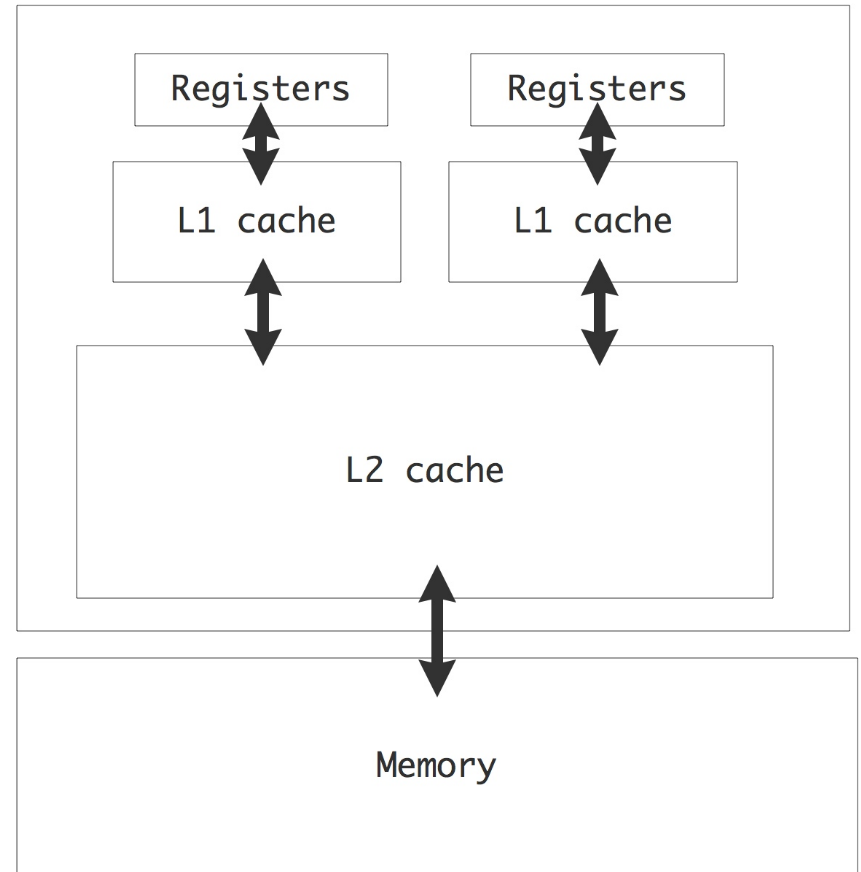
Multicore Architectures

- "Power wall" (clock frequency cannot be increased)
- Limits of Instruction Level Parallelism (ILP)
  - compiler limitations
  - limited amount of intrinsically available parallelism
  - branch prediction
- Solution: divide chip into multiple processing "cores":
  - 2 cores at lower frequency can have same throughput as 1 core at higher frequency (breaks power wall)
  - discovered ILP replaced by explicit task parallelism, managed by programmer

Multicore Architectures



Single core



Dual core

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

Need to study parallel architecture and programming...