Processor Architecture

Victor Eijkhout

PCSE 2015

- Structure of a modern processor
- Memory hierarchy: caches, register, TLB.
- Multicore issues
- Programming strategies for performance
- The power question

Justification

The performance of a parallel code has as one component the behaviour of the single processor or single-threaded code. In this section we discuss the basics of how a processor executes instructions, and how it handles the data these instructions operate on.

Von Neumann machine

The ideal processor:

- (Stored program)
- An instruction contains the operation and two operand locations
- Processor decodes instruction, gets operands, computes and writes back the result
- Repeat

The actual state of affairs

- Single instruction stream versus multiple cores / floating point units
- Single instruction stream versus Instruction Level Parallelism
- Unit-time-addressable memory versus large latencies

Modern processors contain lots of magic to make them seem like Von Neumann machines.

Complexity measures

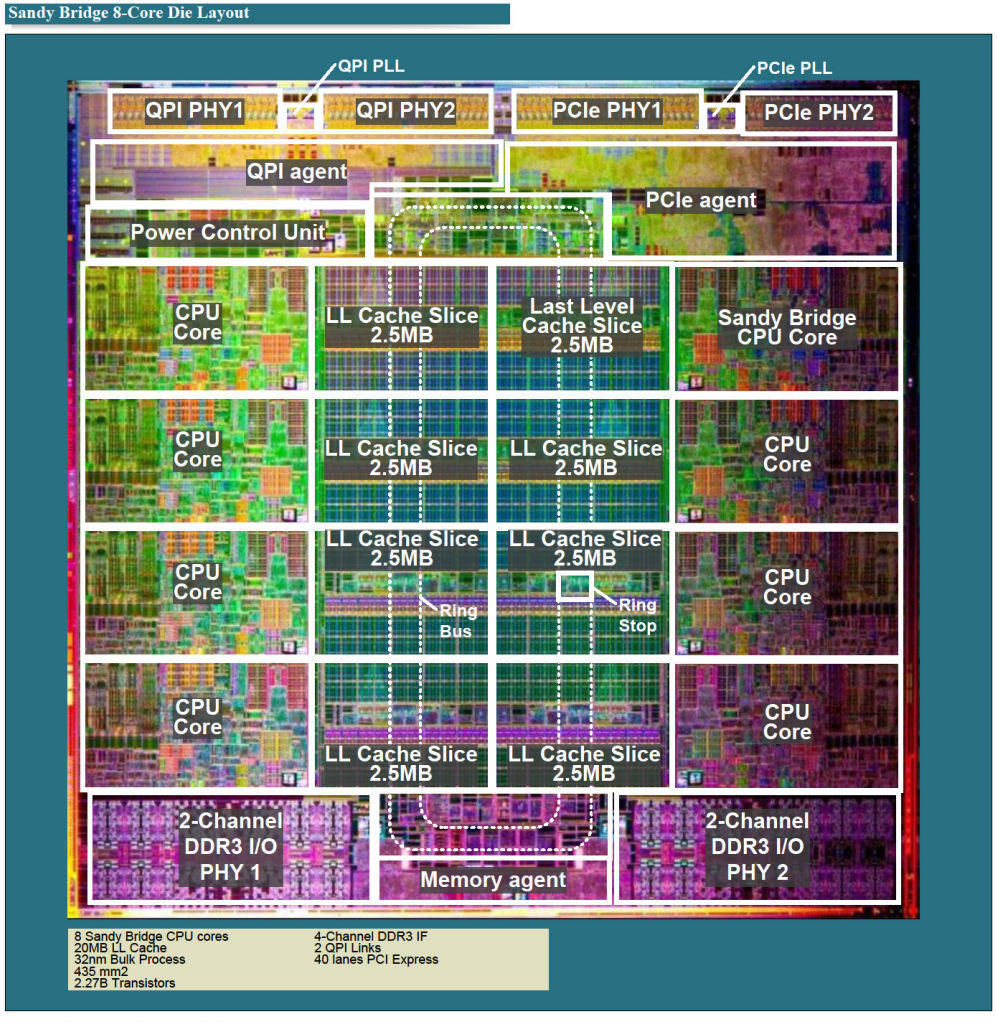
Traditional: processor speed was paramount. Operation counting.

Nowadays: memory is slower than processors (peak performance only out of register).

This course

Study data movement aspects
Dealing with latency
Algorithm design for processor reality

A first look at a processor



Structure of a core

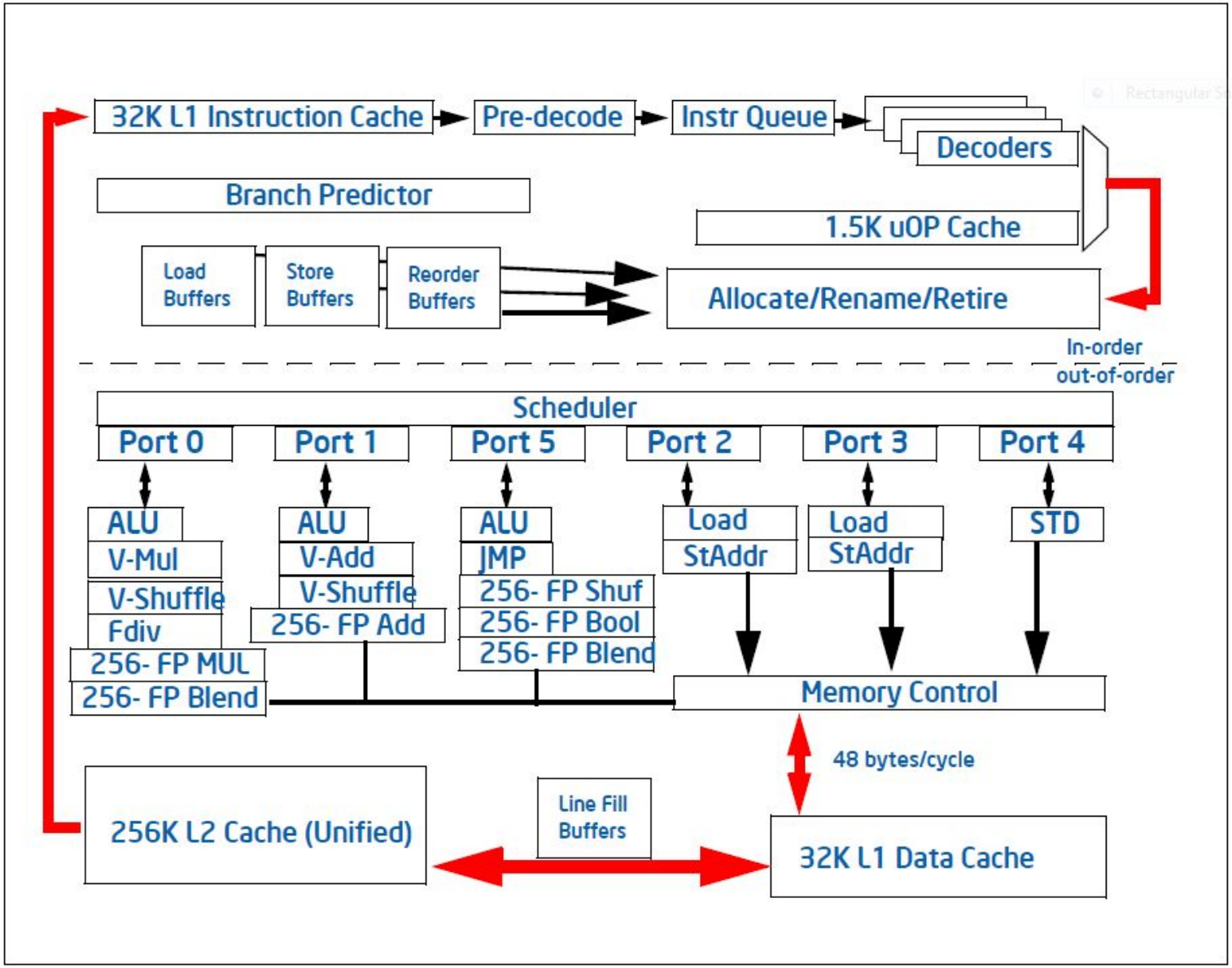


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Instruction-Level Parallelism

- multiple-issue of independent instructions
- branch prediction and speculative execution
- out-of-order execution
- prefetching

Problems: complicated circuitry, hard to maintain performance

Implications

- Long pipeline needs many independent instructions: demands on compiler
- Conditionals break the stream of independent instructions
  - Processor tries to predict branches
  - *branch misprediction penalty*: pipeline needs to be flushed and refilled
  - avoid conditionals in inner loops!

Peak performance

Performance is a function of

- Clock frequency,
- SIMD width
- Load/store unit behaviour

Behaviour (out of L1):

Processor	year	add/mult/fma units (count×width)	daxpy cycles (arith vs load/store)
MIPS R10000	1996	$1 \times 1 + 1 \times 1 + 0$	8/24
Alpha EV5	1996	$1 \times 1 + 1 \times 1 + 0$	8/12
IBM Power5	2004	$0 + 0 + 2 \times 1$	4/12
AMD Bulldozer	2011	$2 \times 2 + 2 \times 2 + 0$	2/4
Intel Sandy Bridge	2012	$1 \times 4 + 1 \times 4 + 0$	2/4
Intel Haswell	2014	$0 + 0 + 2 \times 4$	1/2

Floating point capabilities of several processor architectures, and DAXPY cycle number for 8 operands

Dirty secret

Processor design is sometimes optimized for certain algorithms

In particular: DGEMM/Linpack

Favourable property: one load per operation, no stores

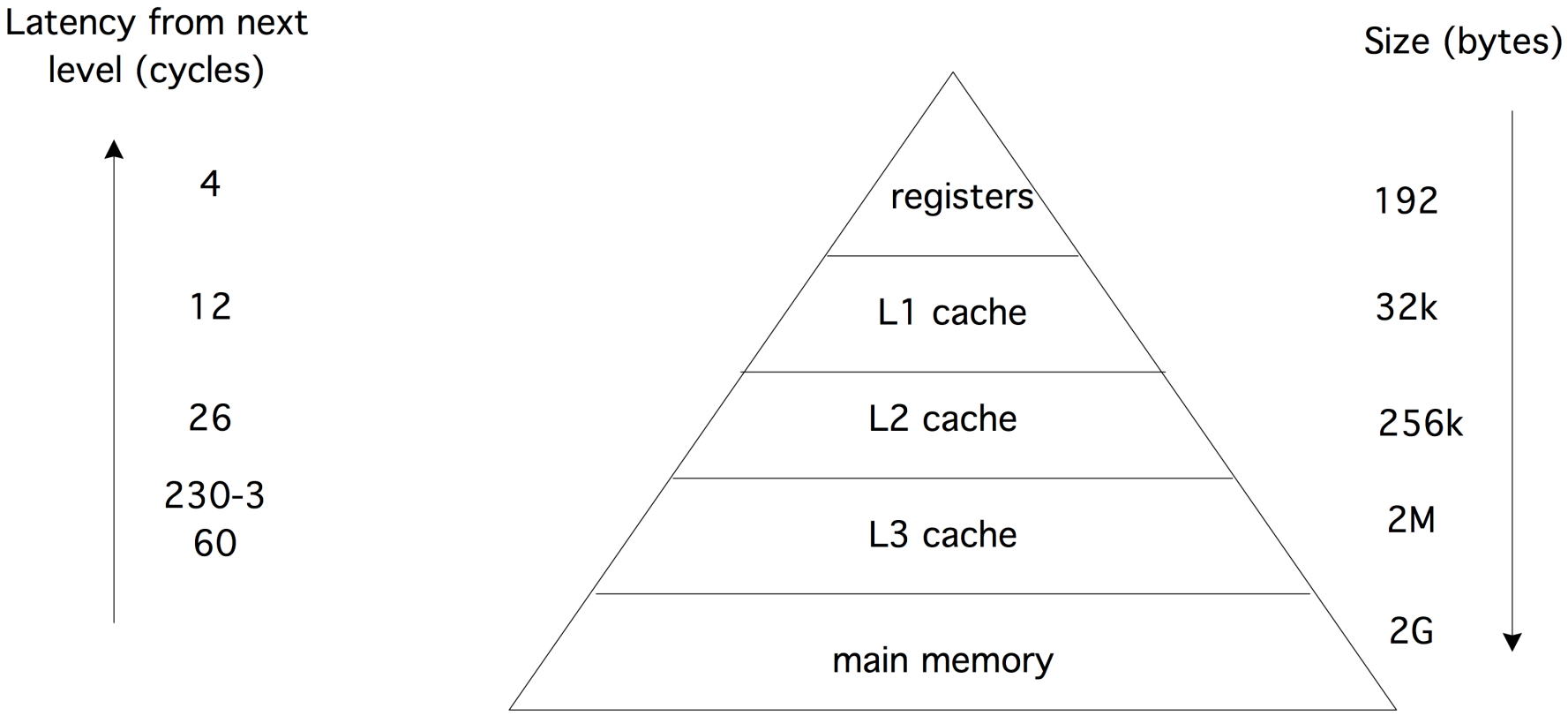
The Big Story

- DRAM memory is slow, so let's put small SRAM close to the processor
- This helps if data is reused
- Does the algorithm have reuse?
- Does the implementation reuse data?

Bandwidth and latency

Important theoretical concept:

- *latency* is delay between request for data and availability
- *bandwidth* is rate at which data arrives thereafter


Latency from next level (cycles)
4
12
26
230-360
Size (bytes)
registers
L1 cache
L2 cache
L3 cache
main memory
192
32k
256k
2M
2G

Registers

Computing out of registers

```
a := b + c
```

- load the value of `b` from memory into a *register*,
- load the value of `c` from memory into another register,
- compute the sum and write that into yet another register, and
- write the sum value back to the memory location of `a`.

Register usage

Assembly code

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

- Registers are named
- Can be explicitly addressed by the programmer
- . . . as opposed to caches.
- Assembly coding or inline assembly (compiler dependent)
- . . . but typically generated by compiler
- Very high bandwidth / low latency: peak performance only possible with data in register.

Examples of register usage

```
a := b + c
d := a + e
```

a stays resident in register, avoid store and load

```
t1 = sin(alpha) * x + cos(alpha) * y;
t2 = -cos(alpha) * x + sin(alpha) * y;
```

subexpression elimination:

```
s = sin(alpha); c = cos(alpha);
t1 = s * x + c * y;
t2 = -c * x + s * y
```

often done by compiler

Register variables

Hint to the compiler: declare *register variable*

```
register double t;
```

Declaring too many leads to *register spill*.

Caches

Cache basics

Fast SRAM in between memory and registers: mostly serves data reuse

```
... = ... x ..... // instruction using x
.........         // several instructions not involving x
... = ... x ..... // instruction using x
```

- load `x` from memory into cache, and from cache into register; operate on it;
- do the intervening instructions;
- request `x` from memory, but since it is still in the cache, load it from the cache into register; operate on it.
- essential concept: *data reuse*

Caches are associative

Cache levels

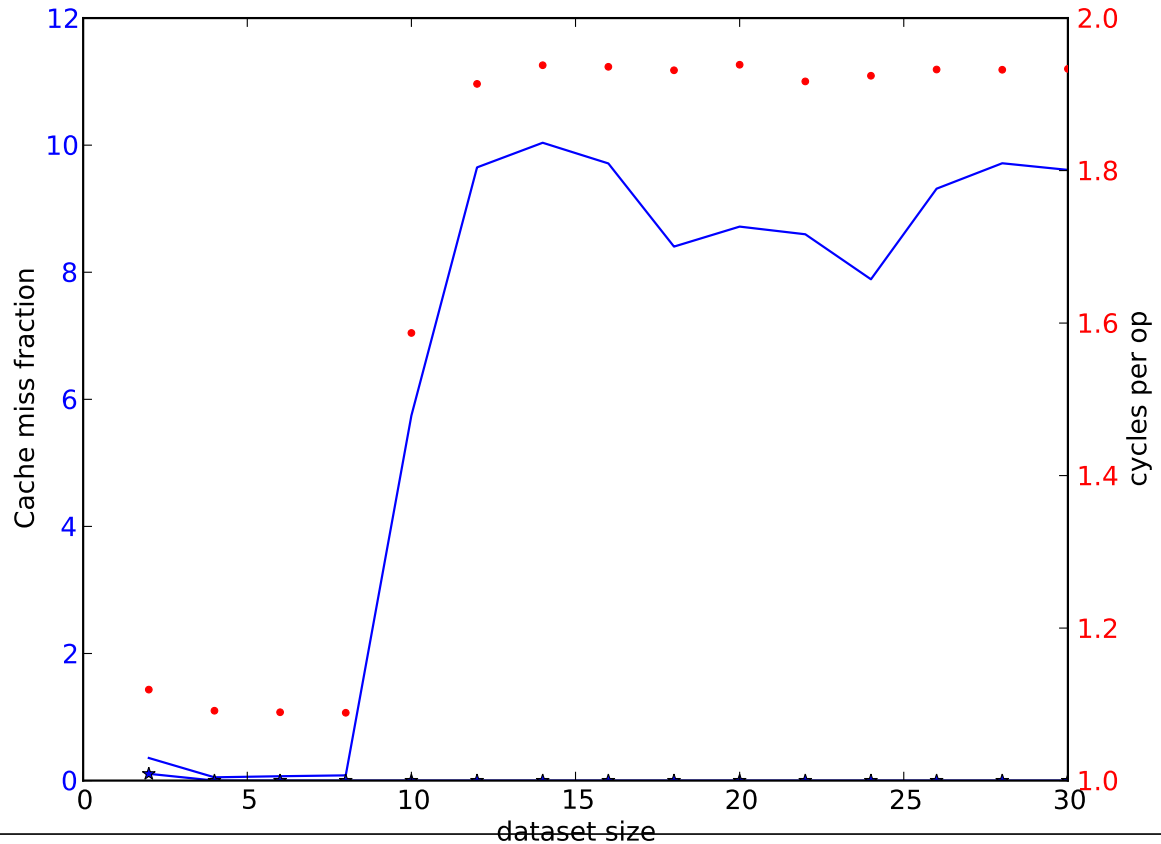
- Levels 1,2,3(,4): L1, L2, etc.
- Increasing size, increasing latency, increasing bandwidth
- (Note: L3/L4 can be fairly big; beware benchmarking)
- Cache hit / cache miss: one level is consulted, then the next
- L1 has separate data / instruction cache, other levels mixed
- Caches do not have enough bandwidth to serve the processor: coding for reuse on all levels.

Cache misses

- Compulsory miss: first time data is referenced
- Capacity miss: data was in cache, but has been flushed (overwritten) by LRU policy
- Conflict miss: two items get mapped to the same cache location, even if there are no capacity problems
- Invalidation miss: data becomes invalid because of activity of another core

Illustration of capacity

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

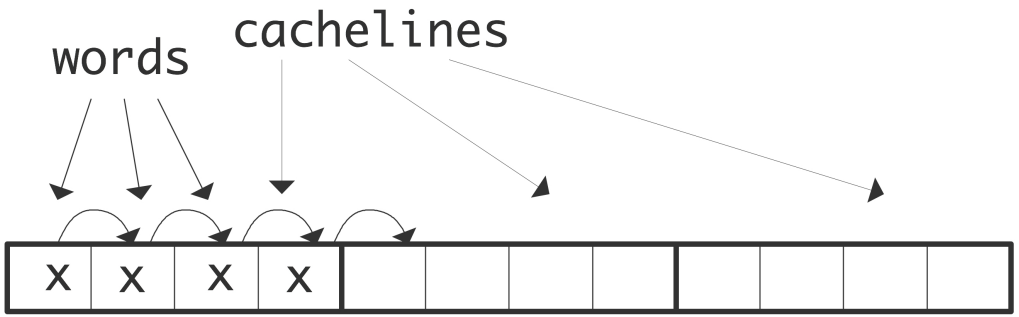


Cache lines

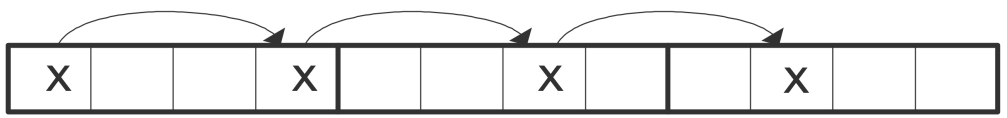
- Memory requests go by byte or word
- Memory transfers go by *cache line*:
  4 or 8 words
- Cache line transfer costs bandwidth
- $\Rightarrow$ important to use all elements

Cache line use

```
for (i=0; i<N; i++)
   ... = ... x[i] ...
```

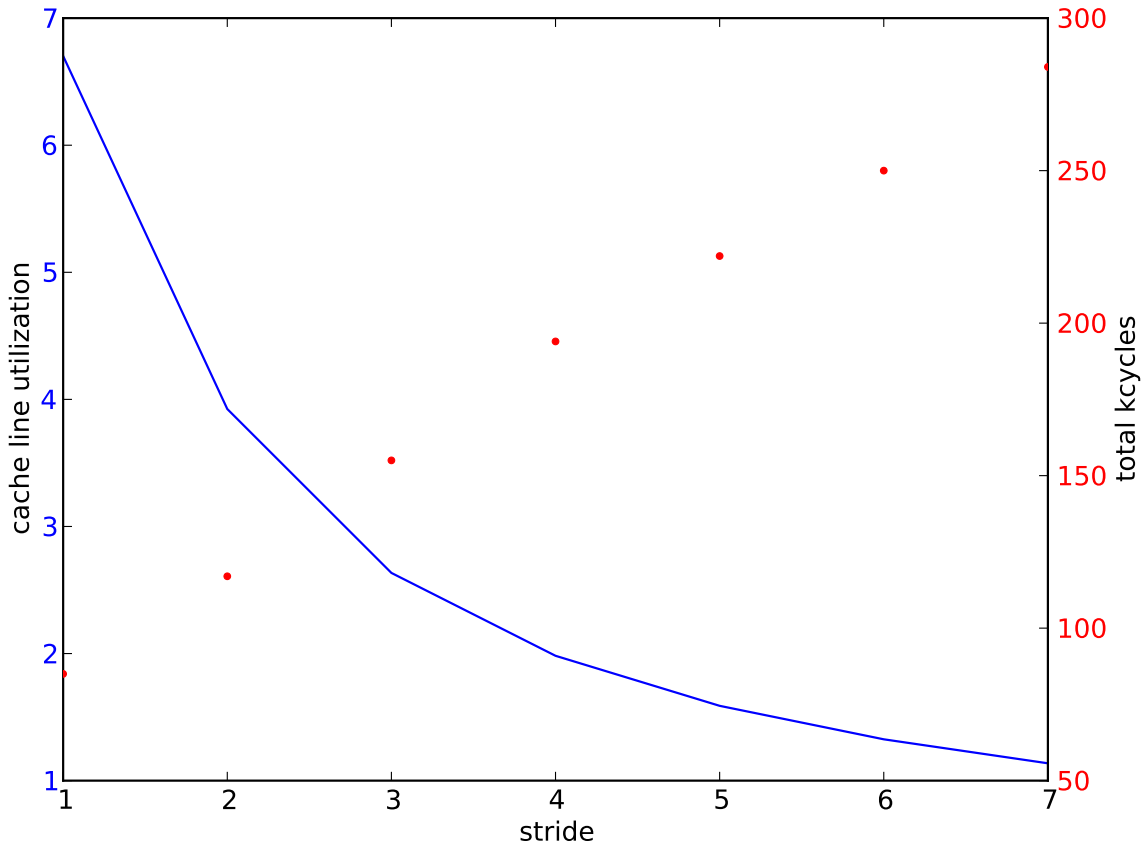



```
for (i=0; i<N; i+=stride)
   ... = ... x[i] ...
```



Stride effects

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



Spatial and temporal locality

Temporal locality: use an item, use it again but from cache
efficient because second transfer cheaper.

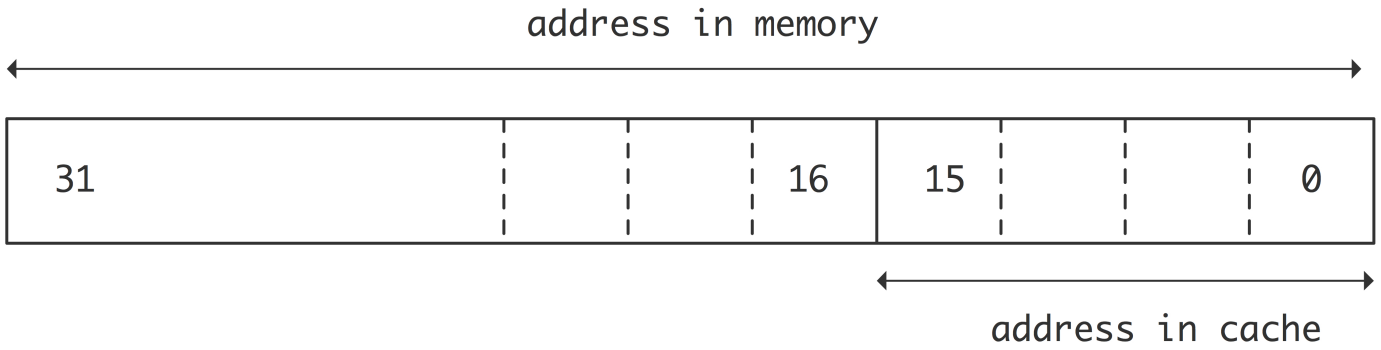
Spatial locality: use an item, then use one 'close to it'
(for instance from same cacheline)
efficient because item is already reachable even though not used before.

Cache mapping

Cache is smaller than memory, so we need a mapping scheme

- Ideal: any address can go anywhere; LRU policy for replacement
- pro: optimal; con: slow, expensive to manufacture
- Simple: direct mapping by truncating addresses
- pro: fast and cheap; con: I'll show you in a minute
- Practical: limited associativity; golden mean

Direct mapping



Direct mapping of 32-bit addresses into a 64K cache

- Use last number of bits to find cache address
- If (memory) addresses are cache size apart, they get mapped to the same cache location
- If you traverse an array, a contiguous chunk will be mapped to cache without conflict.

The problem with direct mapping

```
real*8 A(8192,3);
do i=1,512
  a(i,3) = ( a(i,1)+a(i,2) )/2
end do
```

In each iteration 3 elements map to the same cache location:
constant overwriting ('eviction', *cache thrasing*):
low performance

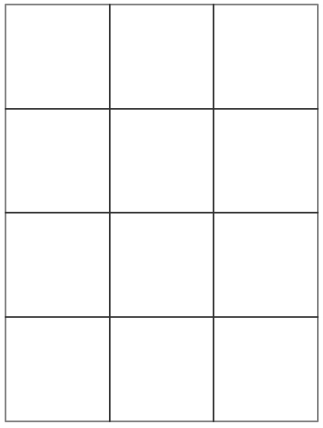
Associate cache mapping

- Allow each memory address to go to multiple (but not all) cache addresses; typically 2,4,8
- Prevents problems with multiple arrays
- Reasonable fast, still:
- Often lower associativity for L1 than L2, L3

Associativity	L1	L2
Intel (Woodcrest)	8	8
AMD (Bulldozer)	2	8

Illustration of associativity

{0, 12, 24, … }
{1, 13, 25, … }
{2, 14, 26, … }
{3, 15, 27, … }
{4, 16, 28, … }
{5, 17, 29, … }
{6, 18, 30, … }
{7, 19, 31, … }
{8, 20, 32, … }
{9, 21, 33, … }
{10, 22, 34, … }
{11, 23, 35, … }



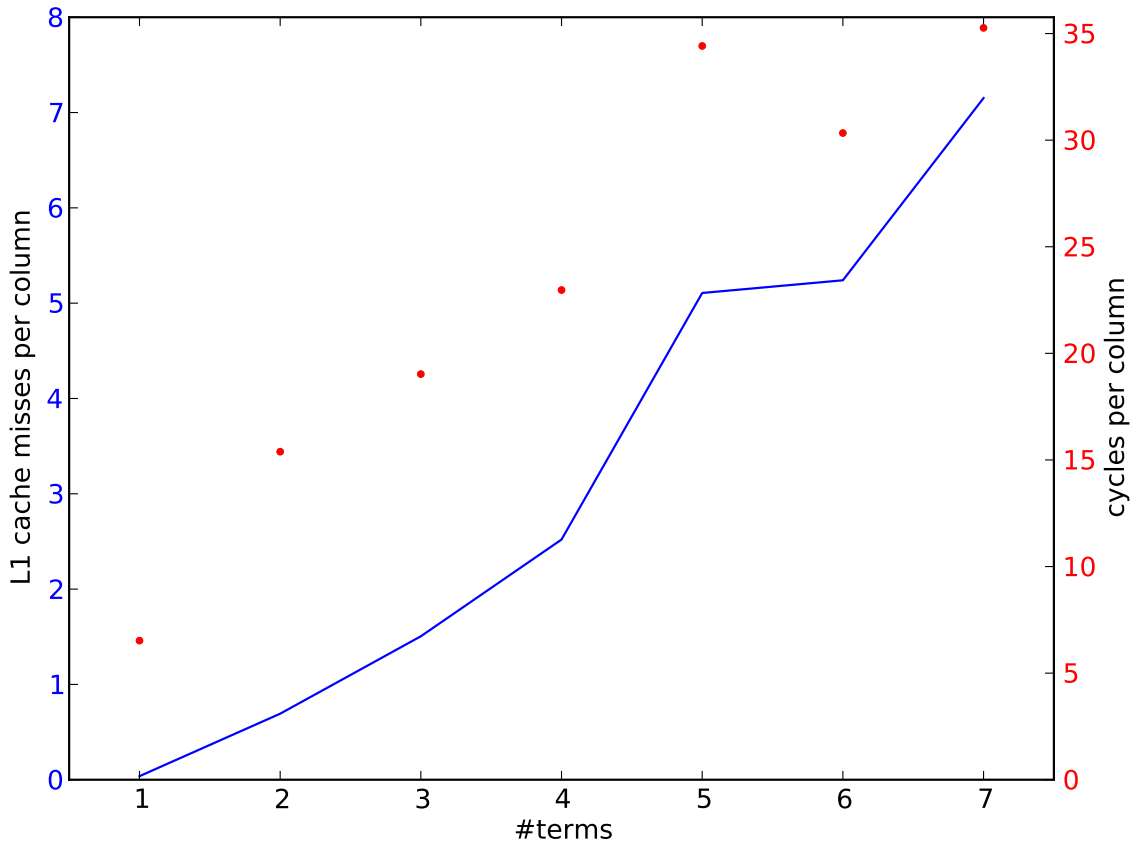
{0, 12, 24, … } {4, 16, 28, … } {8, 20, 32, … }
{1, 13, 25, … } {5, 17, 29, … } {9, 21, 33, … }
{2, 14, 26, … } {6, 18, 30, … } {10, 22, 34, … }
{3, 15, 27, … } {7, 19, 31, … } {11, 23, 35, … }

Two caches of 12 elements: direct mapped (left) and 3-way associative (right)

Direct map: 0–12 is conflict

Associativity in practice

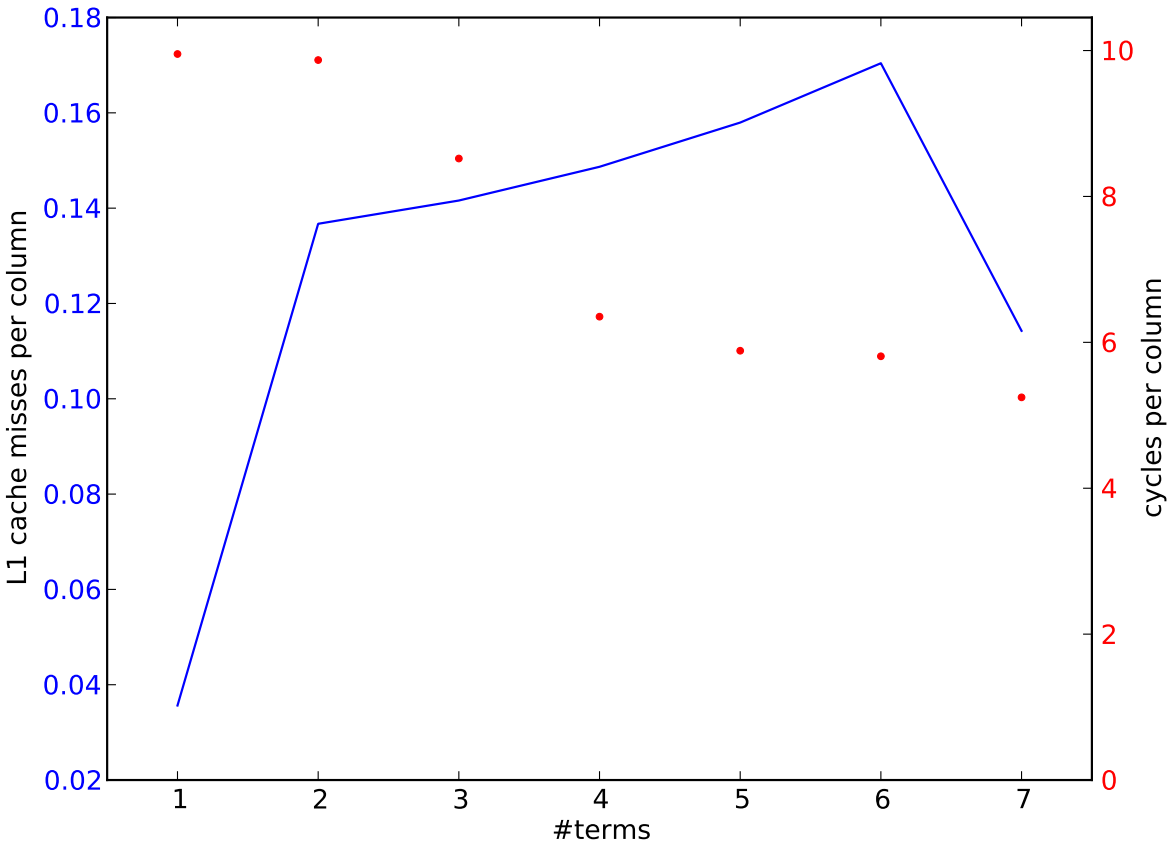
$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$



- The number of L1 cache misses and the number of cycles for each

One remedy

Do not user powers of 2.



The number of L1 cache misses and the number of cycles for each j column accumulation, vector length $4096 + 8$

More memory system topics

Bandwidth / latency

Simple model for sending n words:

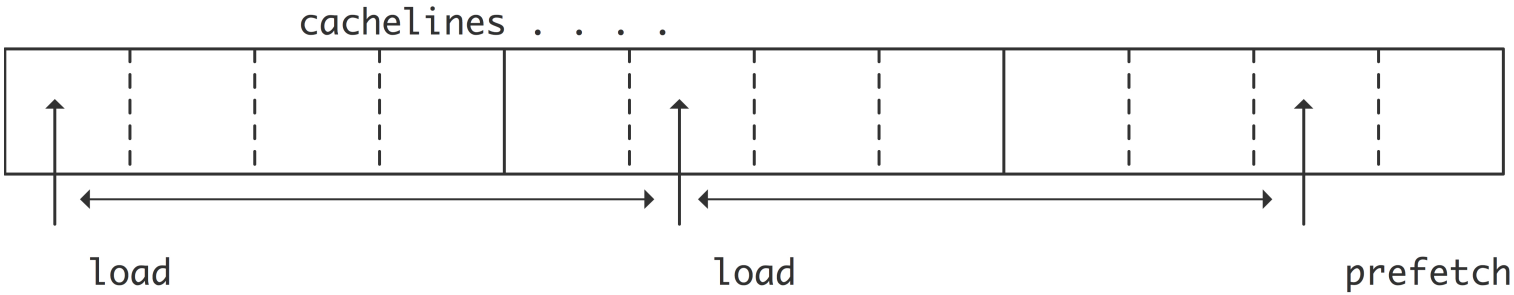
$$t = \alpha + \beta n$$

Quoted bandwidth figures are always optimistic:

- bandwidth shared between cores
- bandwidth wasted on coherence
- assumes optimal scheduling of DRAM banks

Prefetch

- Do you have to wait for every item from memory?
- Memory controller can infer streams: prefetch
- Sometimes controllable through assembly, directives, libraries (AltiVec)
- One form of latency hiding



Memory pages

Memory is organized in pages:

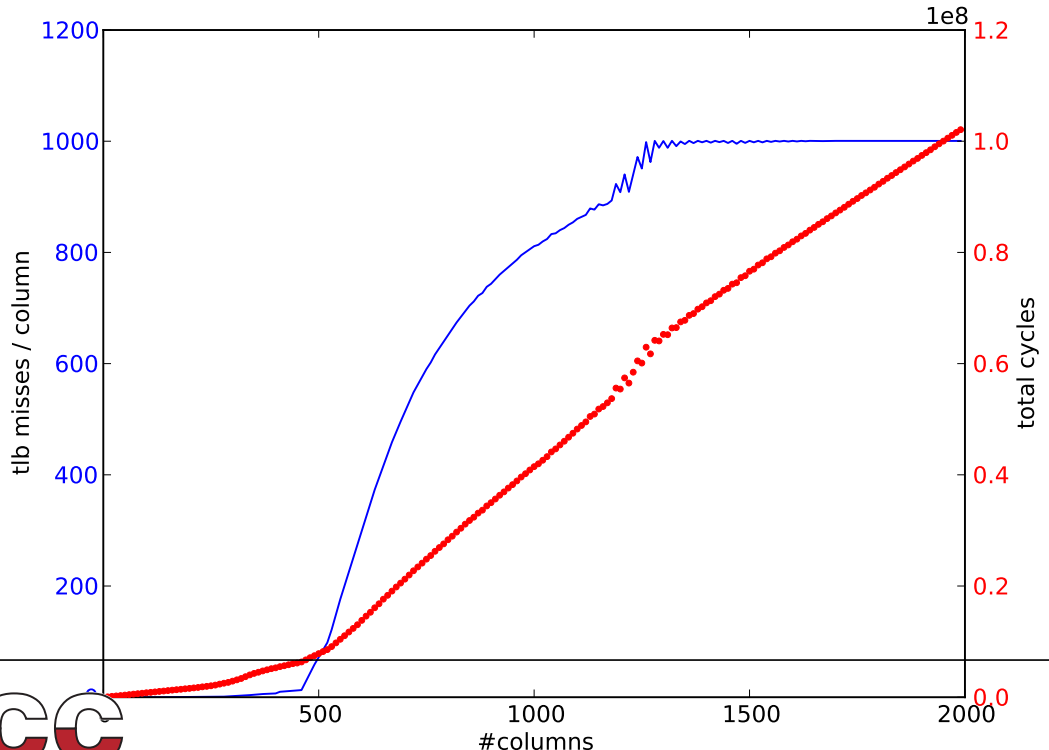
- Translation between logical address, as used by program, and physical in memory
- This serves virtual memory and relocatable code
- so we need another translation stage.

Page translation: TLB

- General page translation: slowish but extensive
- *Translation Look-aside Buffer (TLB)* is a small list of frequently used pages
- Example of spatial locality: items on an already referenced page are found faster

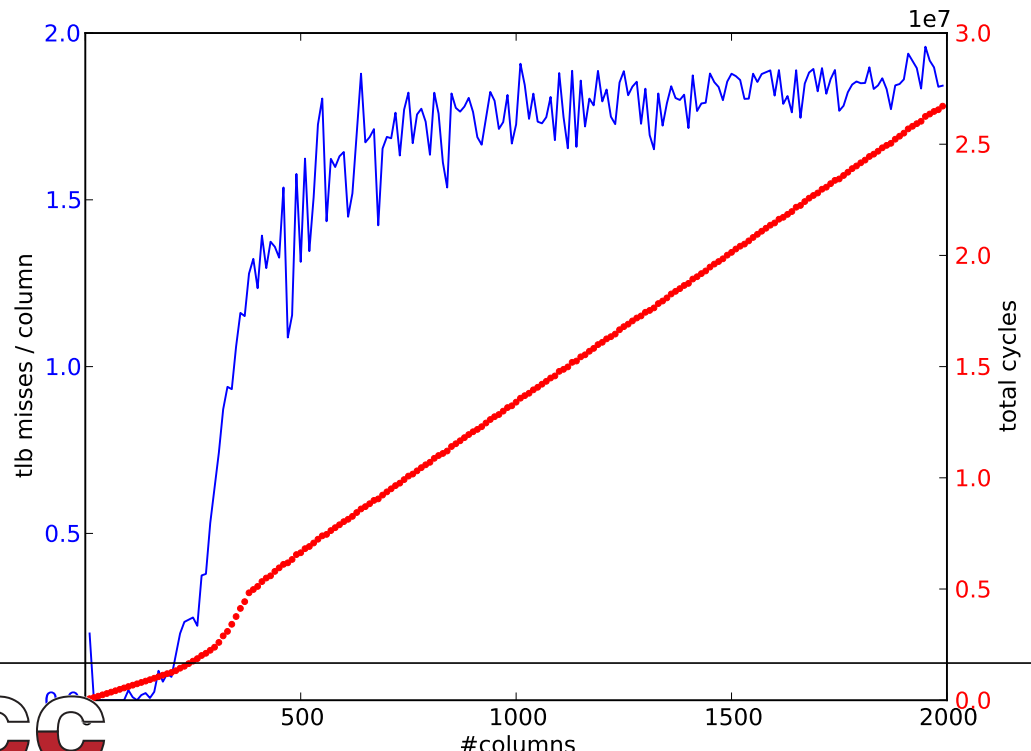
TLB misses

```
#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;
```



TLB hits

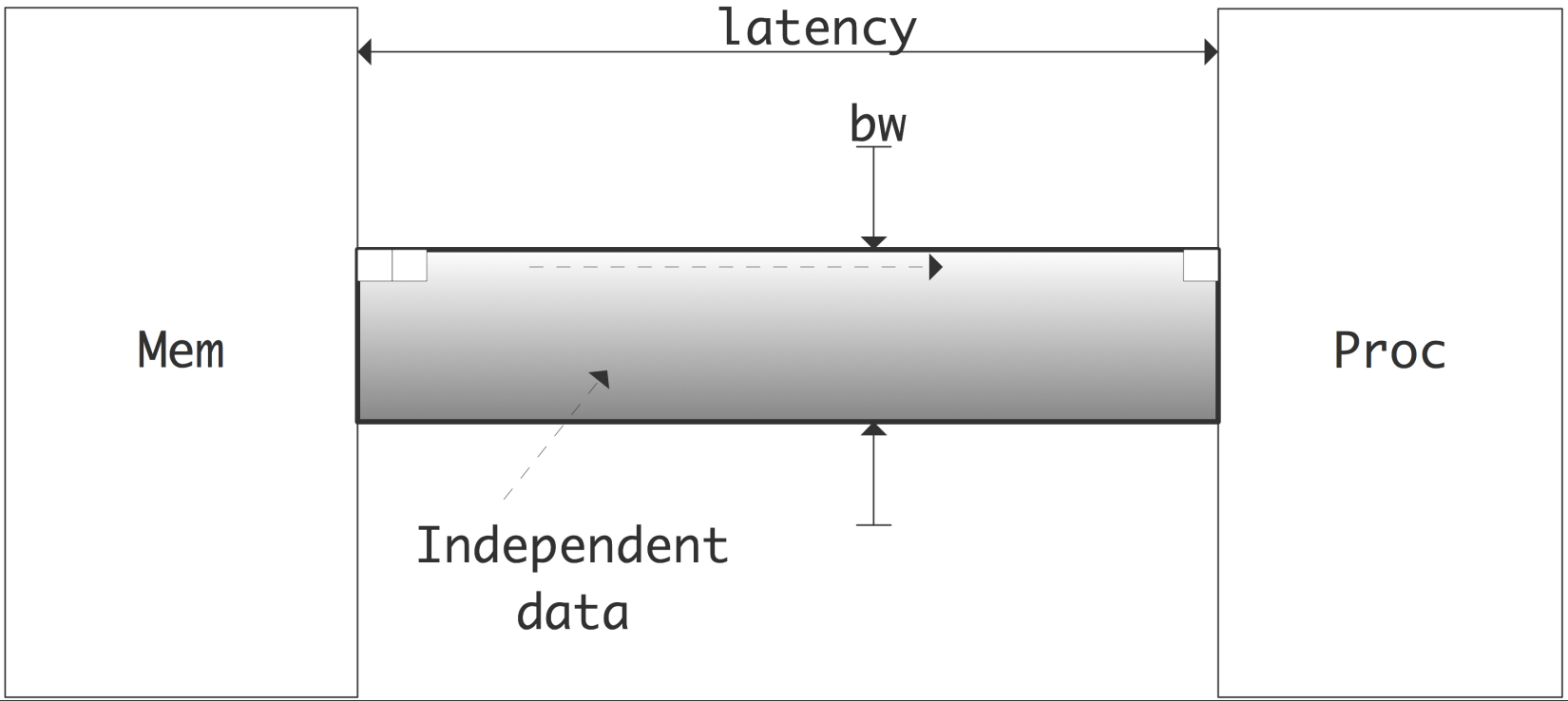
```
#define INDEX(i,j,m,n) i+j*m
array = (double*) malloc(m*n*sizeof(double));
/* traversal #1 */
for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```



Little's Law

- Item loaded from memory, processed, new item loaded in response
- But this can only happen after latency wait
- Items during latency are independent, therefore

$$\text{Concurrency} = \text{Bandwidth} \times \text{Latency}.$$



Why multicore

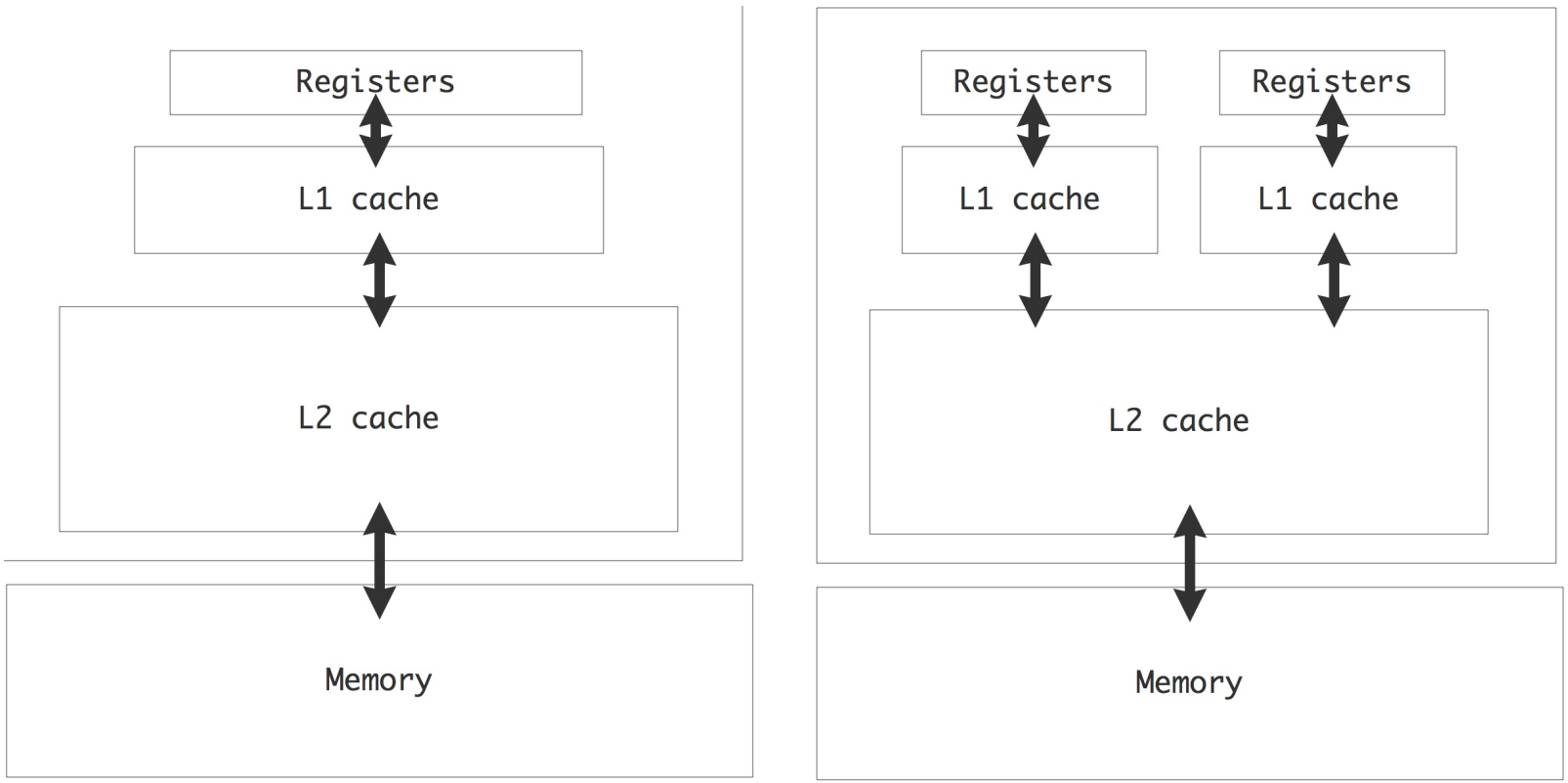
Quest for higher performance:

- Two cores at half speed more energy-efficient than one at full speed.
- Not enough instruction parallelism for long pipelines

Multicore solution:

- More theoretical performance
- Burden for parallelism is now on the programmer

Multicore caches



Cache coherence

Modified-Shared-Invalid (MSI) coherence protocol:

Modified:	the cacheline has been modified
Shared:	the line is present in at least one cache and is unmodified.
Invalid:	the line is not present, or it is present but a copy in another cache has been modified.

Coherence issues

- Coherence is automatic, so you don't have to worry about it...
- ...except when it saps performance
- Beware false sharing
  writes to different elements of a cache line

Coherence and (false) sharing

- Two cores access the same cacheline: Shared status in both
- One core modifies the cacheline: Modified there, Invalid in other core.
- Other core modifies the cacheline:
  - Find out that the other core has a copy
  - Get that copy, or load from memory
  - Modify ...
  - ... and then it's Invalid on the other core
- Concepts: coherence, snooping

Balance analysis

- Sandy Bridge core can aborb 300 GB/s
- 4 DDR3/1600 channels provide 51 GB/s, difference has to come from reuse
- It gets worse: latency 80ns, bandwidth 51 GB/s,
  Little's law: parallelism 64 cache lines
- However, each core only has 10 line fill buffers,
  so we need 6–7 cores to provide the data for one core
- Power: cores are 72%, uncore 17, DRAM 11.
- Core power goes 40% to instruction handling, not arithmetic
- Time for a redesign of processors and programming

How much performance is possible?

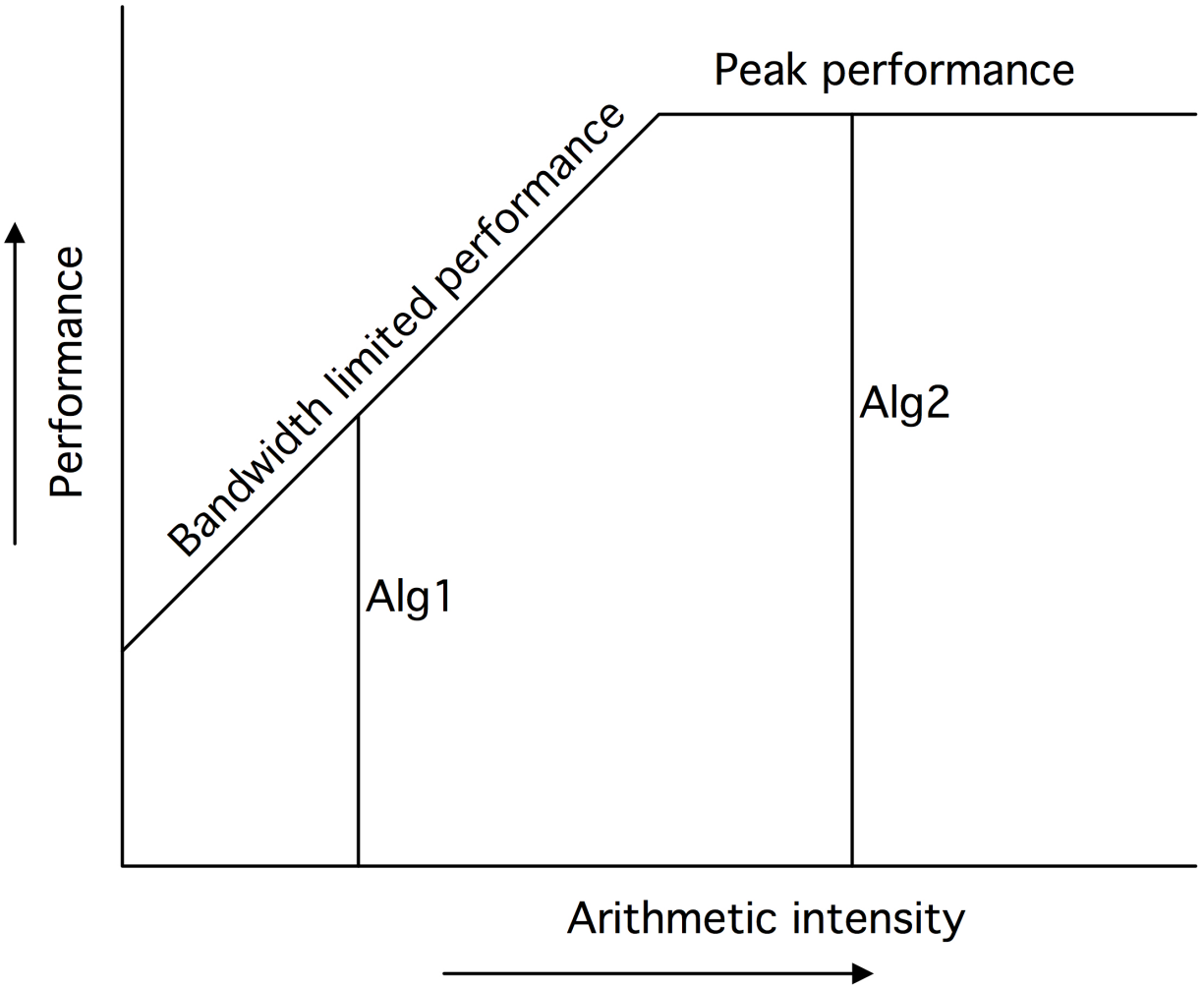
Performance limited by

- Processor peak performance: absolute limit
- Bandwidth: linear correlation with performance

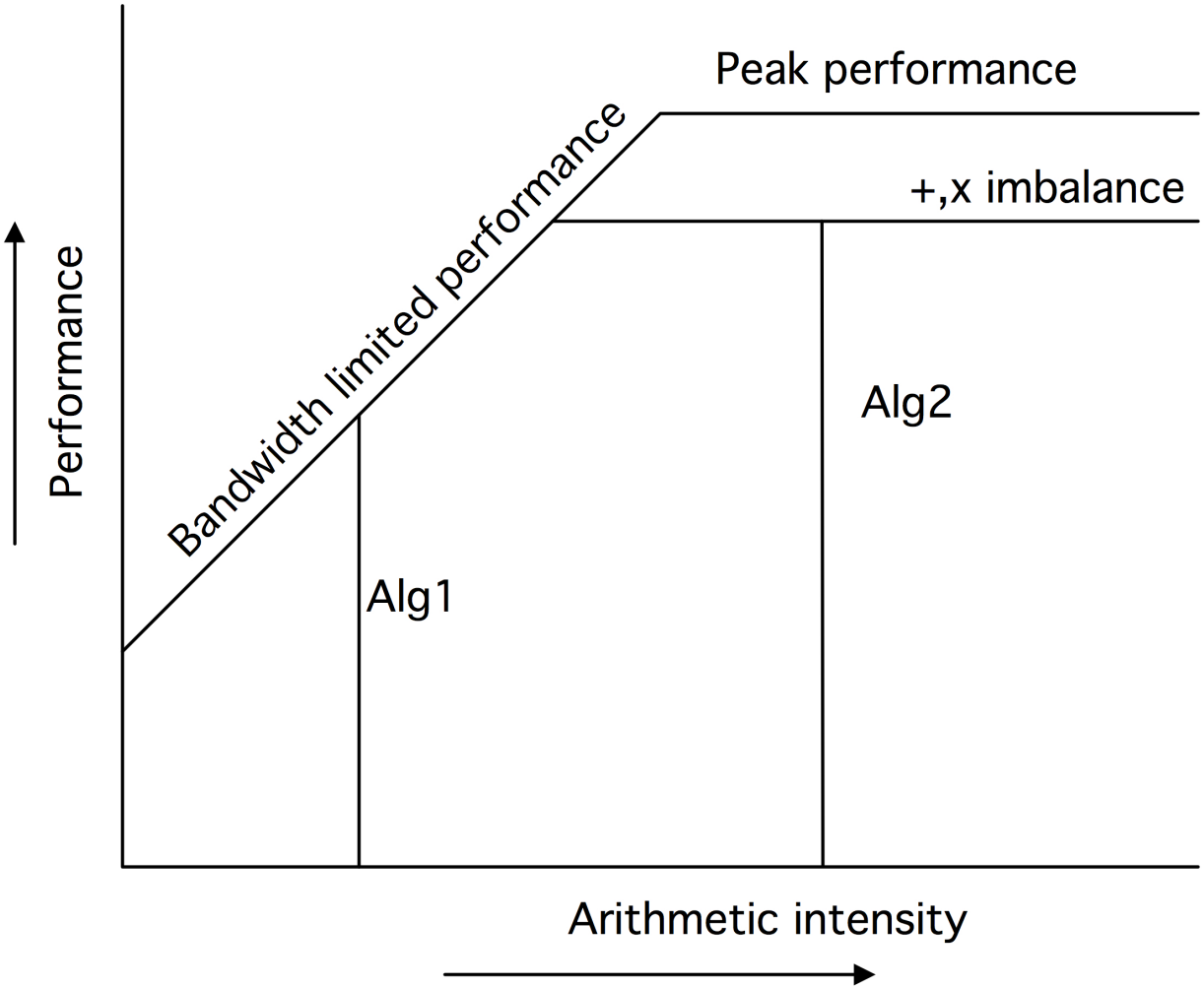
Arithmetic intensity: ratio of operations per transfer

If AI high enough: processor-limited
otherwise: bandwidth-limited

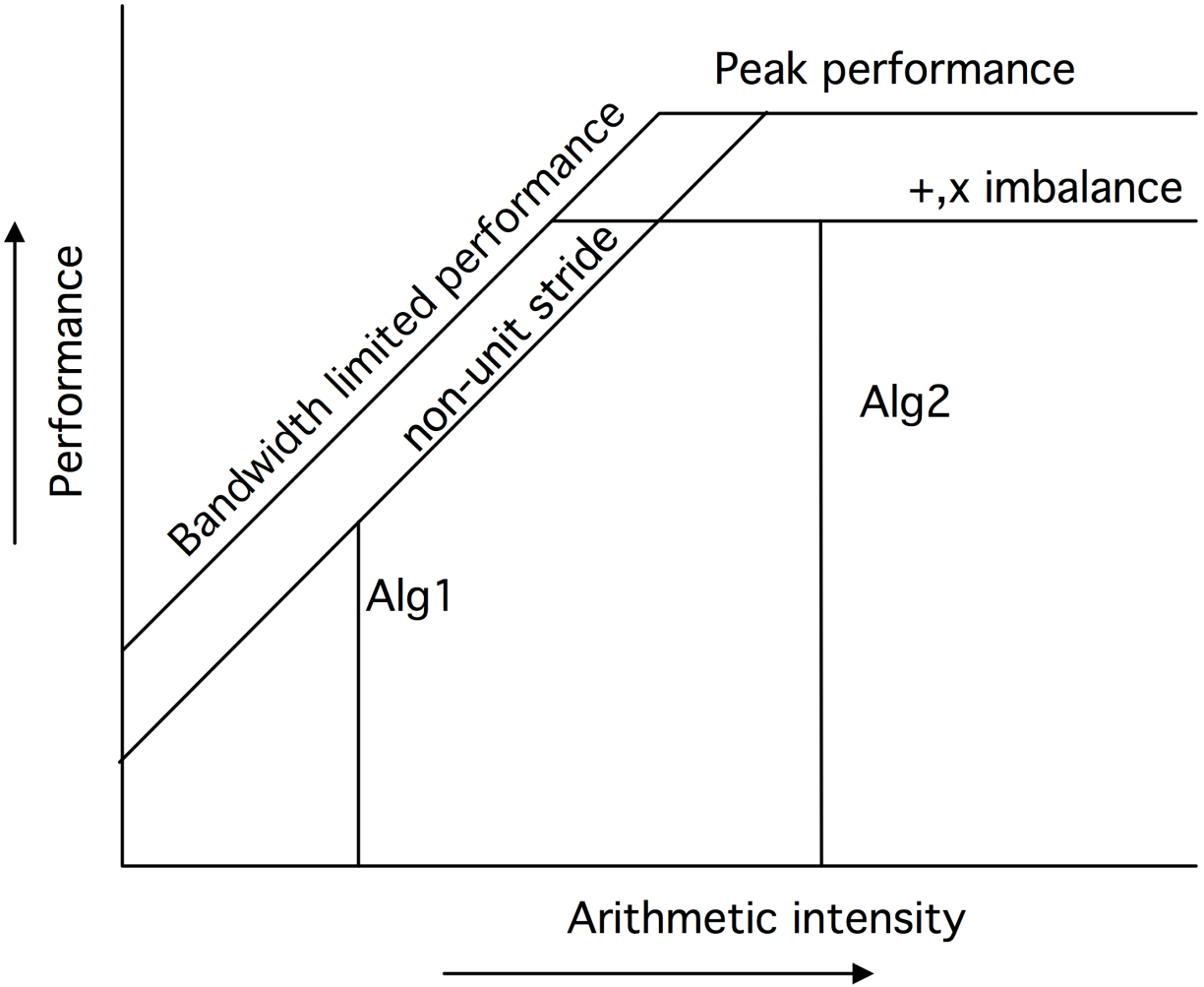
Performance depends on algorithm:



Insufficient utilization of functional units:



Imperfect data transfer:



Architecture aware programming

- Cache size: block loops
- pipelining and vector instructions: expose streams of instructions
- reuse: restructure code (both loop merge and splitting, unroll
- TLB: don't jump all over memory
- associativity: watch out for powers of 2

Loop blocking

Multiple passes over data

```
for ( k< small bound )
  for ( i < N )
    x[i] = f( x[i], k, .... )
```

Block to be cache contained

```
for ( ii < N; ii+= blocksize )
  for ( k< small bound )
    for ( i=ii; i<ii+blocksize; i++ )
      x[i] = f( x[i], k, .... )
```

This requires independence of operations

The ultimate in performance programming: DGEMM

Matrix-matrix product $C = A \cdot B$

$$\forall_i \forall_j \forall_k : c_{ij} += a_{ik} b_{kj}$$

- Three independent loop i, j, k
- all three blocked i', j', k'
- Many loop permutations, blocking factors to choose

DGEMM variant

Inner products

```
for ( i )
  for ( j )
    for ( k )
      c[i,j] += a[i,k] * b[k,j]
```

DGEMM variant

Outer product: updates with low-rank columns-times-vector

```
for ( k )
  for ( i )
    for ( j )
      c[i,j] += a[i,k] * b[k,j]
```

DGEMM variant

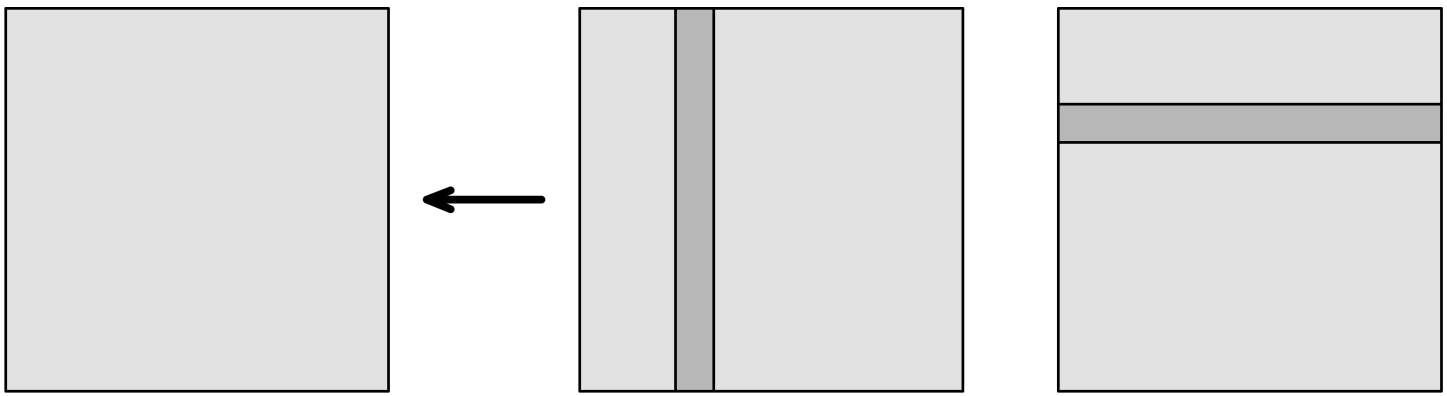
Building up rows by linear combinations

```
for ( i )
  for ( k )
    for ( j )
      c[i,j] += a[i,k] * b[k,j]
```

Exchanging i, j: building up columns

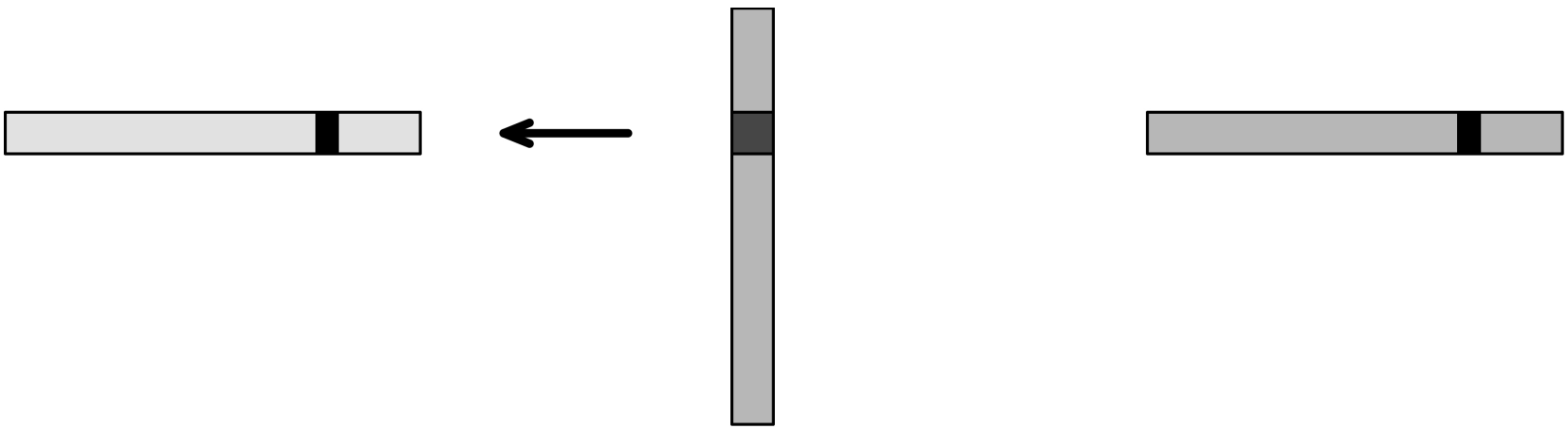
Rank 1 updates

$$C_{**} = \sum_k A_{*k} B_{k*}$$



Matrix-panel multiply

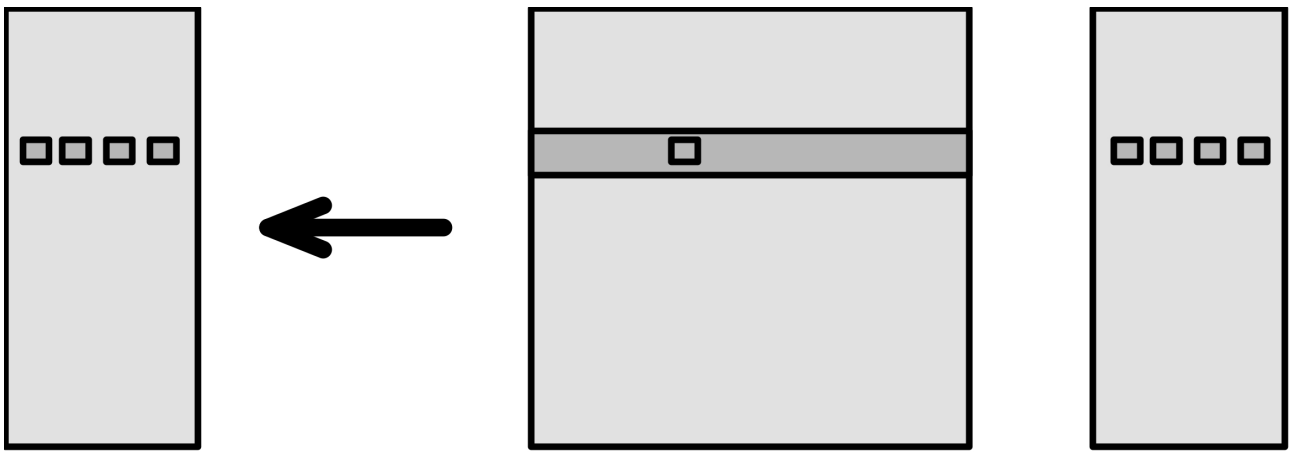
Block of A times 'sliver' of B



Inner algorithm

For inner *i*:

```
// compute C[i,*] :
for k:
   C[i,*] = A[i,k] * B[k,*]
```



Tuning

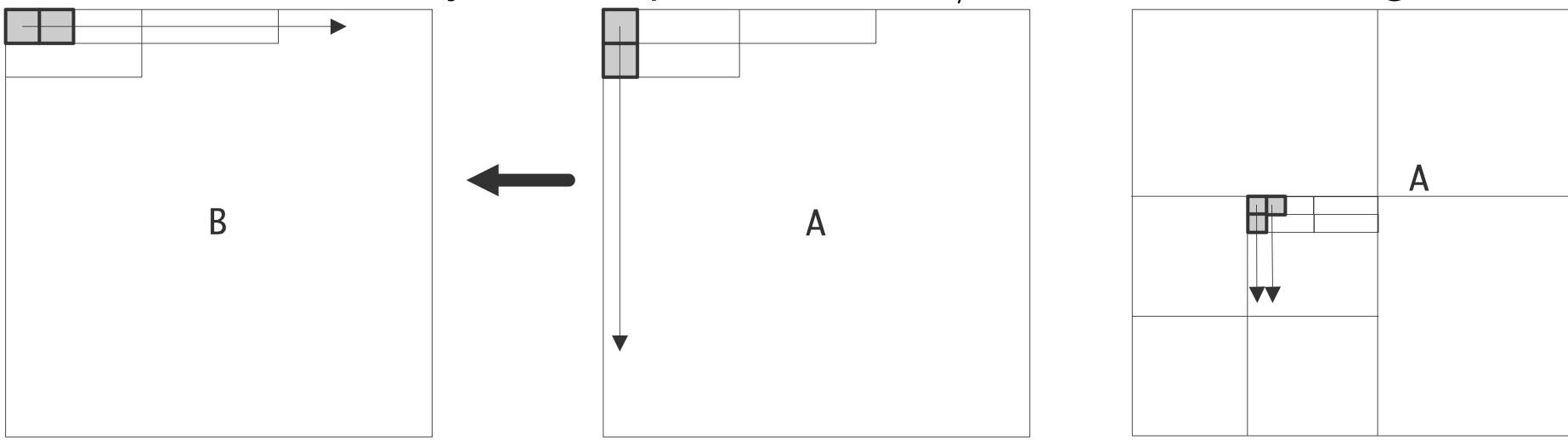
For inner i:

```
// compute C[i,*] :
for k:
   C[i,*] += A[i,k] * B[k,*]
```

- `C[i,*]` stays in register
- `A[i,k]` and `B[k,*]` stream from L1
- blocksize of A for L2 size
- A stored by rows to prevent TLB problems

Cache-oblivious programming

Observation: recursive sub-
division will ultimately make a problem small / well-behaved enough



Cache-oblivious matrix-matrix multiply

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

with $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Recursive approach will be cache contained.
Not as high performance as being cache-aware...

Dennard scaling

Scale down feature size by s:

Feature size	$\sim s$
Voltage	$\sim s$
Current	$\sim s$
Frequency	$\sim s^{-1}$

Miracle conclusion:

$$\text{Power} = V \cdot I \sim s^2; \text{Power density} \sim 1$$

Everything gets better, cooling problem stays the same

Opportunity for more components, higher frequency

Dynamic power

Charge	$q = CV$
Work	$W = qV = CV^2$
Power	$W/\text{time} = WF = CV^2F$

(1)

Two cores at half frequency:

$$\left.\begin{aligned} C_{\text{multi}} &= 2C \\ F_{\text{multi}} &= F/2 \\ V_{\text{multi}} &= V/2 \end{aligned}\right\} \Rightarrow P_{\text{multi}} = P/4.$$

Same computation, less power