Processor Architecture

Victor Eijkhout

Fall 2025

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.

Structure of a modern processor

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

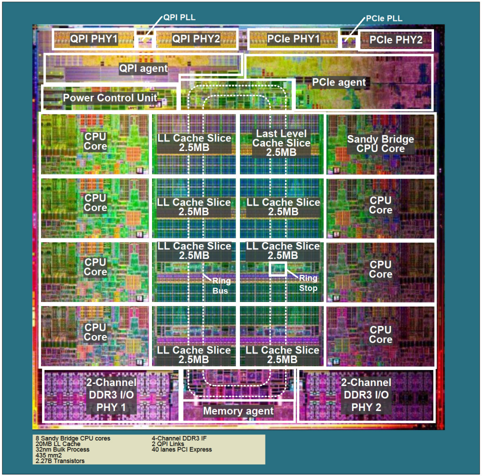
This lecture:

Study data movement aspects

Algorithm design for processor reality

A first look at a processor

Sandy Bridge 8-Core Die Layout



Structure of a core

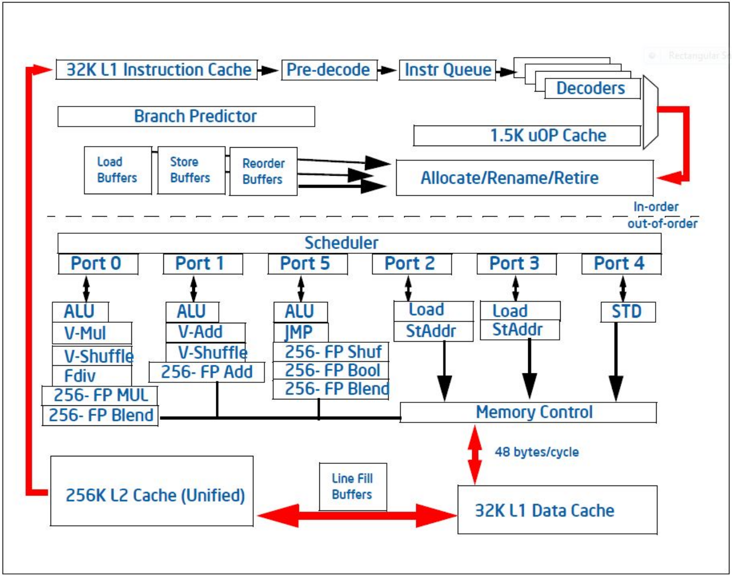


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Motivation for pipelining

An operation consists of several stages.
Addition:

- Decoding the instruction operands.
- Data fetch into register
- Aligning the exponents:

$$.35 \times 10^{-1} + .6 \times 10^{-2} \qquad \text{becomes}$$
$$.35 \times 10^{-1} + .06 \times 10^{-1}.$$

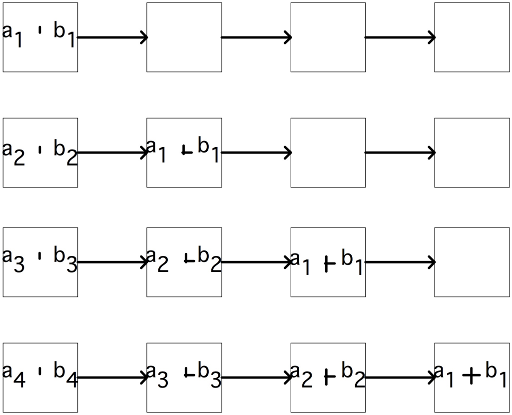
- Adding mantissas, giving .41.
- Normalizing the result, giving $.41 \times 10^{-1}$.
- Storing the result.

pipeline stages

Pipelining, pictorially

Discrete hardware for each stage:

$c_i \leftarrow a_i + b_i$



Analysis

Operation timing:

$$\begin{cases} n & \text{operations} \\ \ell & \text{number of stages} \\ \tau & \text{clock cycle} \end{cases} \Rightarrow t(n) = n\ell\tau$$

With pipelining:

$$t(n) = [s + \ell + n - 1]\tau$$

where s is a setup cost
$\Rightarrow$ Asymptotic speedup is ℓ
$n_{1/2}$: value for which speedup is $\ell/2$

Applicability of pipelining

Pipelining works for:

vector addition/multiplication

Division/square root maybe pipelined, but much slower

Recurrences

Pipelining does not immediately work:

```
for (i) {
  x[i+1] = a[i]*x[i] + b[i];
}
```

Transform:

$$\begin{aligned} x_{n+2} = \quad & a_{n+1}x_{n+1} + b_{n+1} \\ & = a_{n+1}(a_n x_n + b_n) + b_{n+1} \\ & = a_{n+1}a_n x_n + a_{n+1}b_n + b_{n+1} \end{aligned}$$

Instruction pipeline

- Instruction-Level Parallelism: more general notion of independent instructions
- Requires independent instructions
- As frequency goes up, pipeline gets longer: more demands on compiler

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!

Instructions

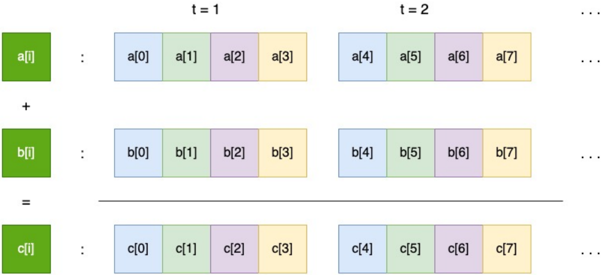
- Addition/multiplication: pipelined
- Division (and square root): much slower

  ```
  for ( i )
     a[i] = b[i] / c
  ```

  Can you improve on this?
- Fused Multiply-Add (FMA) `s += a*b`
  where can you use this?

Vector instructions

- Multiple (independent) operands per cycle
- Similar to pipelining but not the same
- ‘Vector width’ is 2,4,8.



DAXPY

BLAS operation ‘A-x plus y’:

```
for ( i )
    y[i] = a[i]*x + y[i]
```

- *x* stays in register
- two loads and one store per operation
- candidate for pipelining or vector instructions

DAXPY performance

Performance is a function of

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

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

Processor	year	add/mult/fma units (count$\times$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

Memory hierarchy: caches, register, TLB.

The Big Story

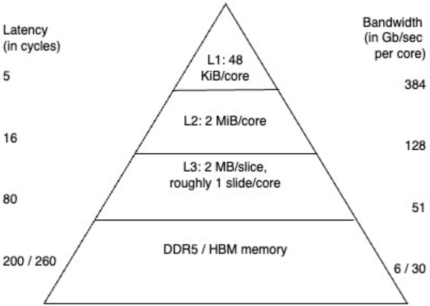
- DRAM memory is slow, so let's put small SRAM close to the processor
- This helps if data is reused: second access is faster
- Does the algorithm have reuse?
- Does the implementation reuse data?

Bandwidth and latency

Important theoretical concept:

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

Memory hierarchy



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
(note: Intel two-operand syntax)

```
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

Examples of register usage

1. Resident in register

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

`a` stays resident in register, avoid store and load

2. subexpression elimination:

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

becomes:

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

often done by compiler

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

Cache levels

- Levels 1,2,3(,4): L1, L2, etc.
- Increasing size, increasing latency, decreasing 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.

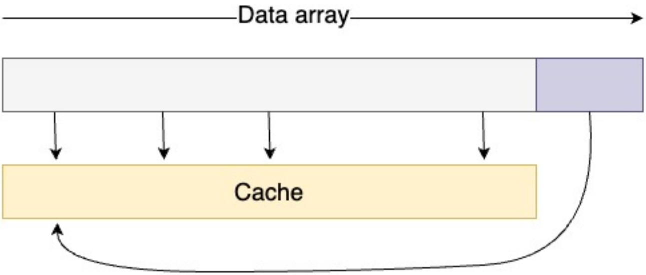
Cache misses

- 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

Cache hits

- Data has been requested, used a second time: temporal locality
- $\Rightarrow$ Can't wait too long between uses
- (Data can be loaded because it's close to data requested: spatial locality. Later.)

Capacity miss



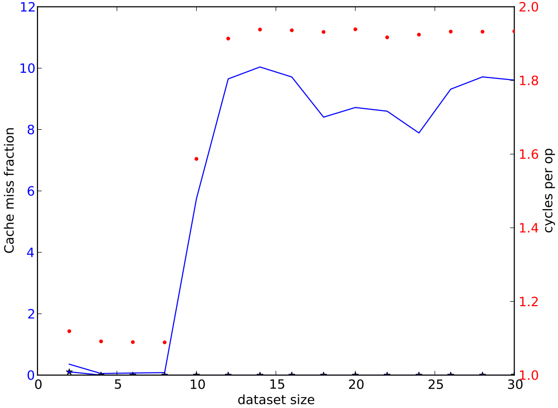
(Why is that last block going where it is going?)

Cache capacity

- Loading data multiple times
- LRU: oldest item evicted if needed
- Reuse if not too much data

```
for ( lots of times ) // sequential loop
   load and process data // probably parallel loop
```

Illustration of capacity



Replacement policies

What determines where new data goes /
what old data is overwritten?

- Least Recently Used (LRU): most common
- First-in-first-out (FIFO): IBM Power4. Not a good idea.
- Random Replacement. Sometimes used.

It's actually more subtle than pure LRU . . .

Cache lines

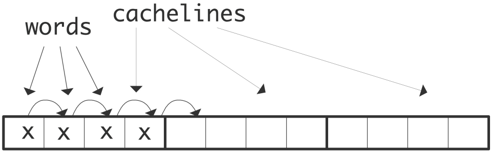
- Memory requests go by byte or word
- Memory transfers go by cache line:
  typically 64 bytes / 8 double precision numbers
- Cache line transfer costs bandwidth
- $\Rightarrow$ important to use all elements

Effects of striding

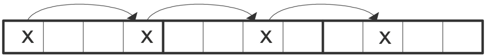
- Always 8 numbers transferred
- With stride $s > 1$: $8/s$ elements used
- Loss of efficiency if bandwidth-limited

Cache line use

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

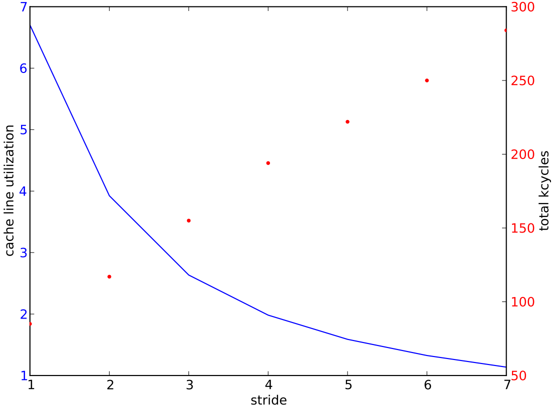



```
for (i=0; i<N; i+=stride)
  ... = ... x[i] ...
```



Stride effects

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

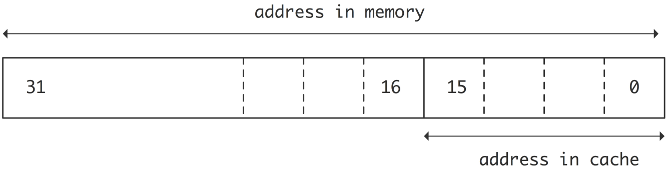


Cache mapping

Cache is smaller than memory, so we need a mapping scheme
memory address $\mapsto$ cache address

- 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
  'enough but not too much'

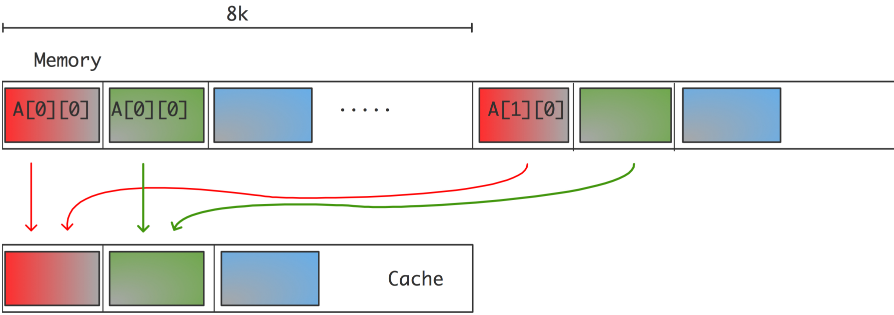
Direct mapping



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

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

Conflicts



Mapping conflicts in a direct-mapped cache.

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
```

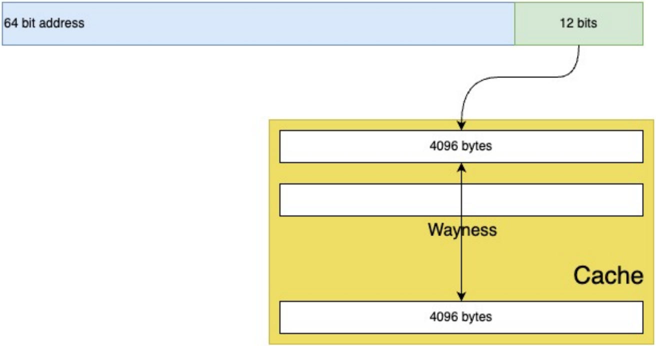
In each iteration 3 elements map to the same cache location:
constant overwriting ('eviction', cache thrasing):
low performance

Associative 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
- Often lower associativity for L1 than L2, L3

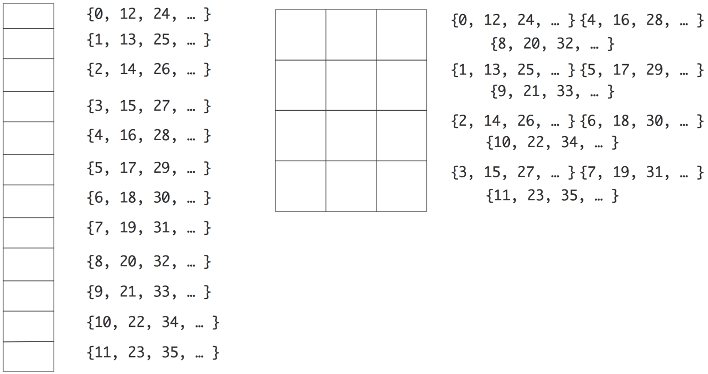
Associativity	L1	L2
Intel (Woodcrest)	8	8
AMD (Bulldozer)	2	8

Associativity



Associative cache structure of Intel Ice Lake.

Illustration of associativity



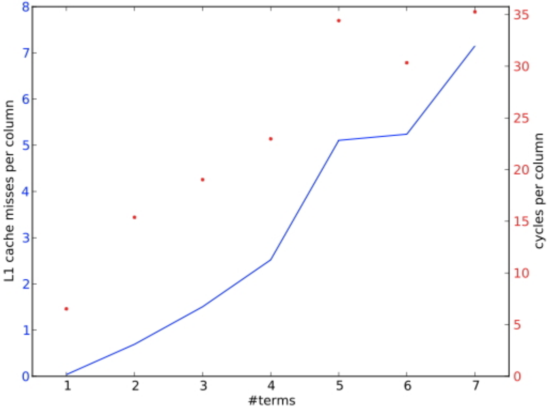
Two caches of 12 elements: direct mapped (left) and 3-way associative (right)

Direct map: 0–12 is conflict

Associative: no conflict

Associativity in practice

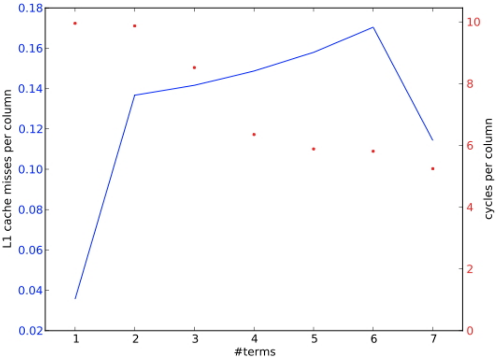
$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$



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

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 $4096 + 8$

Exercise

Write a small cache simulator in your favorite language. Assume a k-way associative cache of 32 entries and an architecture with 16 bit addresses.

Exercise: vectorsum

- Compare sequential performance to single-threaded OMP
- For some problem sizes observe a difference in performance
- Use Intel option `-qopt-report=3` and inspect the report.
- Compare different compilers: Intel 19 behaves differently from 24! Also gcc13.

```
for ( int iloop=0; iloop<nloops; ++iloop ) {
  for ( int i=0; i<vectorsize; ++i ) {
    outvec[i] += invec[i]*loopcoeff[iloop];
  }
}
```

Analyze and report

More memory system topics

Bandwidth / latency

Simple model for sending n words:

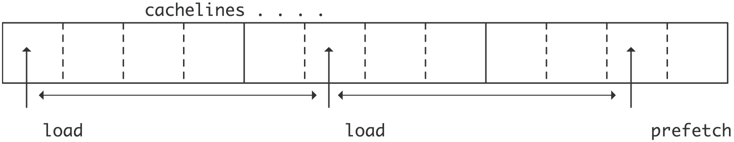
$$t = \alpha + \beta n$$

Quoted bandwidth figures are always optimistic:

- bandwidth shared between cores
  not enough bandwidth for all cores:
  $\Rightarrow$ speedup less than linear
- bandwidth wasted on coherence
- NUMA: pulling data from other socket
- 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:

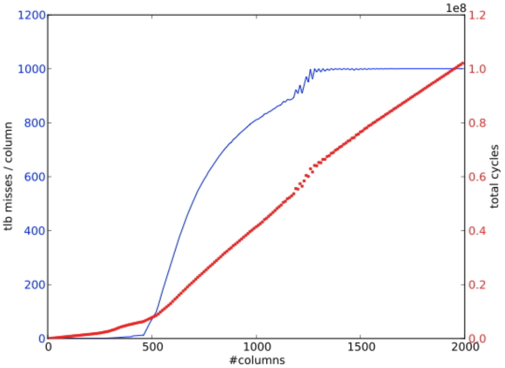
- Translation between logical address, as used by program, and physical in memory
- This serves virtual memory and relocatable code
- so we need another translation stage.

Page translation: TLB

- General page translation: slowish and expensive
- 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

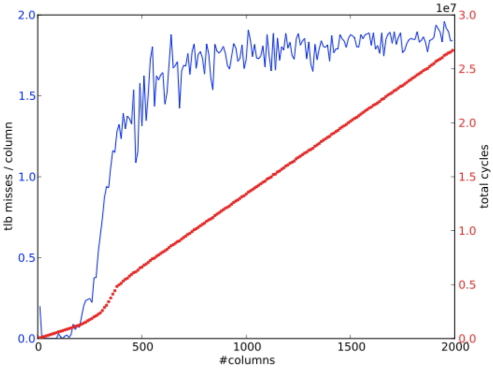
TLB misses

```
#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

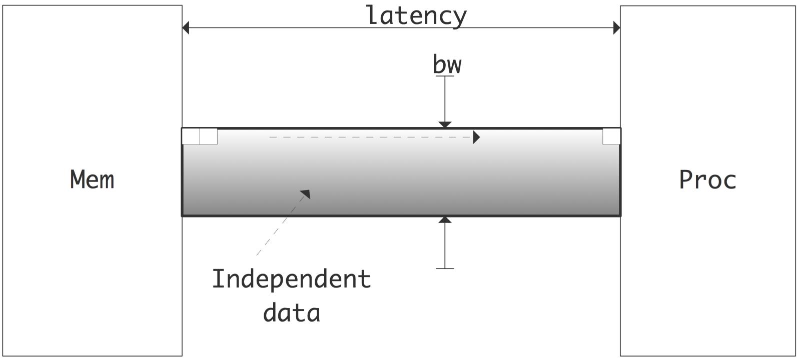
```
#define INDEX(i,j,m,n) i+j*m
array = (double*) malloc(m*n*sizeof(double));
/* traversal #1 */
for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```



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

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



Multicore issues

Why multicore

Quest for higher performance:

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

Multicore solution:

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

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

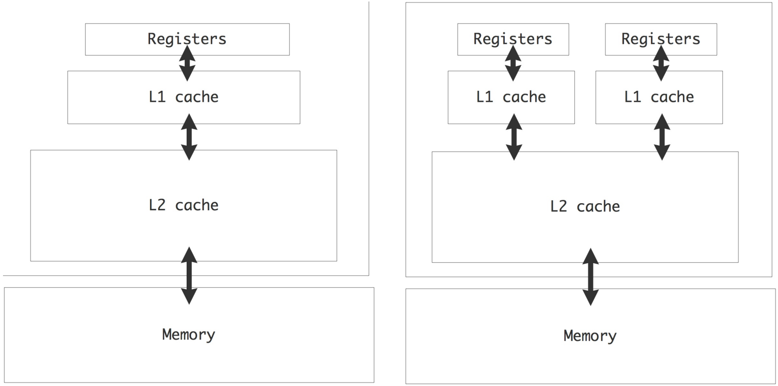
$$
\begin{array}{|l|l|}
\hline
\text{Charge} & q = CV \\
\text{Work} & W = qV = CV^2 \\
\text{Power} & W/\text{time} = WF = CV^2F \\
\hline
\end{array}
\tag{1}
$$

Two cores at half frequency:

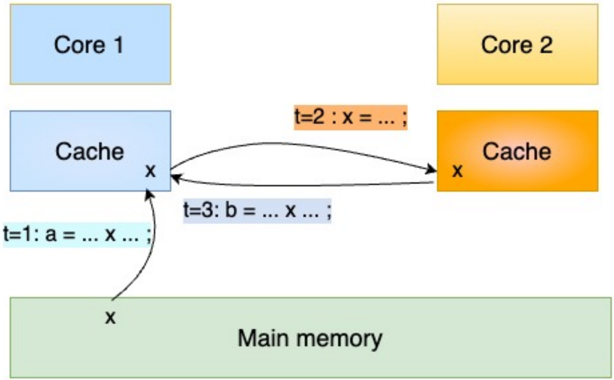
$$
\left.
\begin{array}{c}
C_{\text{multi}} = 2C \\
F_{\text{multi}} = F/2 \\
V_{\text{multi}} = V/2
\end{array}
\right\} \Rightarrow P_{\text{multi}} = P/4.
$$

Same computation, less power

Multicore caches



The coherence problem



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

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; see my research presentation

Programming strategies for performance

How much performance is possible?

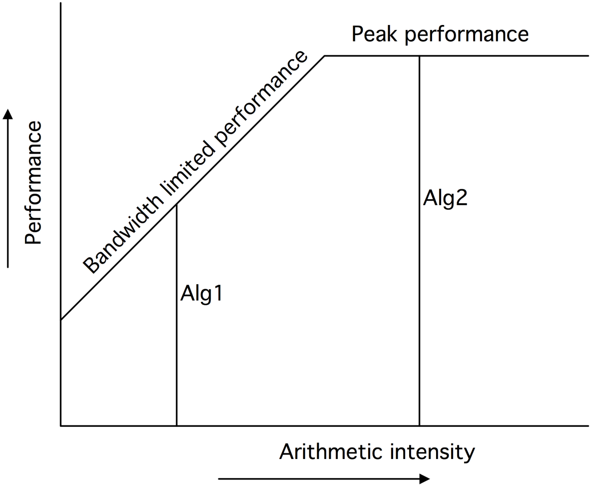
Performance limited by

- 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

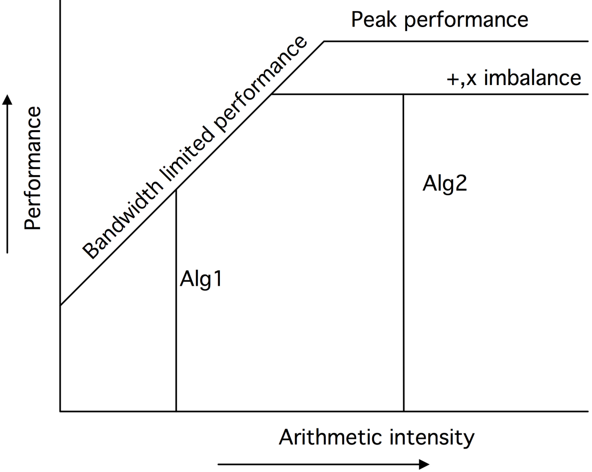
Roofline plot

Performance depends on algorithm:



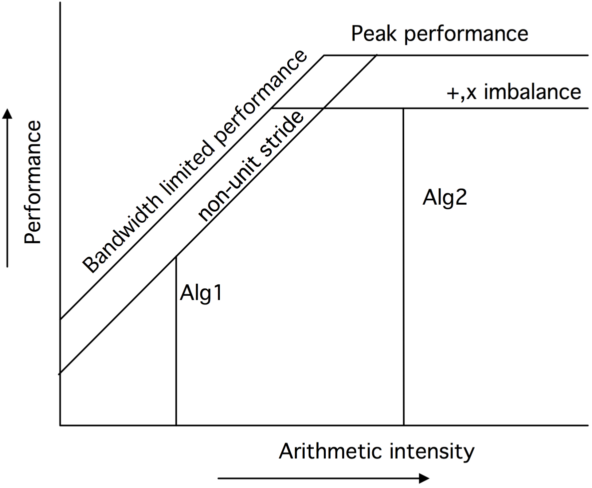
Lowering the roof, 1

Insufficient utilization of functional units:



Lowering the roof, 2

Imperfect data transfer:



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.

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

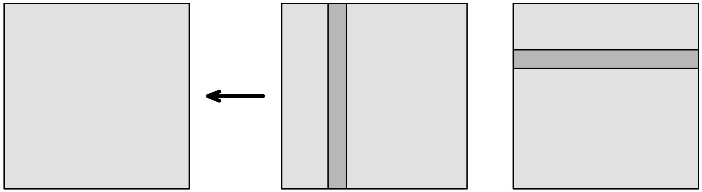
Building up rows by linear combinations

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

Exchanging i, j: building up columns

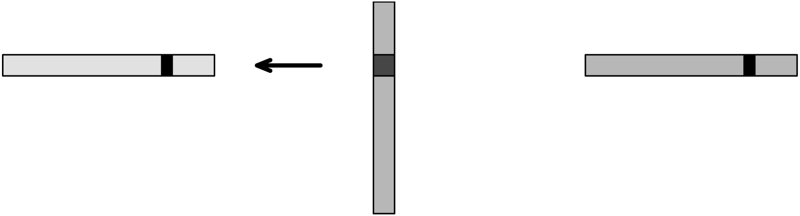
Rank 1 updates

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



Matrix-panel multiply

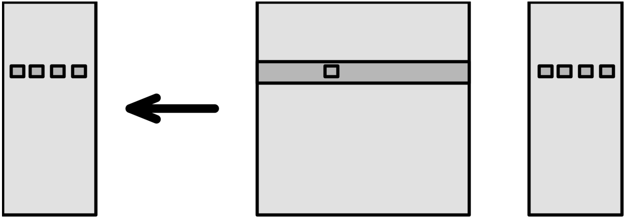
Block of A times ‘sliver’ of B



Inner algorithm

For inner *i*:

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



Tuning

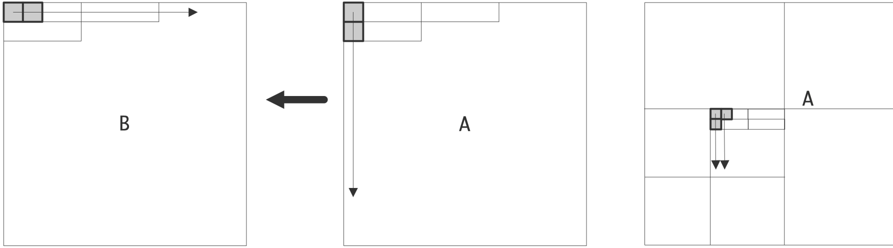
For inner *i*:

```
// 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 subdivision 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...