Taming the inner loop

Louis J. M. Aslett (aslett@stats.ox.ac.uk)

Department of Statistics, University of Oxford

Young Researchers' Meeting
8 November 2016: University of Warwick

www.louisaslett.com

Background

Thank-you for having me back!

October 2015 talk for Young Researchers' Meeting entitled 'Background on HPC for Statistics'.

Discussed fundamental importance of parallelism, giving a mostly code-free introduction to GPUs and cloud computing.

Thank-you for having me back!

October 2015 talk for Young Researchers' Meeting entitled 'Background on HPC for Statistics'.

Discussed fundamental importance of parallelism, giving a mostly code-free introduction to GPUs and cloud computing.

This year, would like to talk more generally than the multi-core, many-core and cluster parallelism buzz. We'll dive a little deeper on getting the most from the machine that probably sits on your desk.

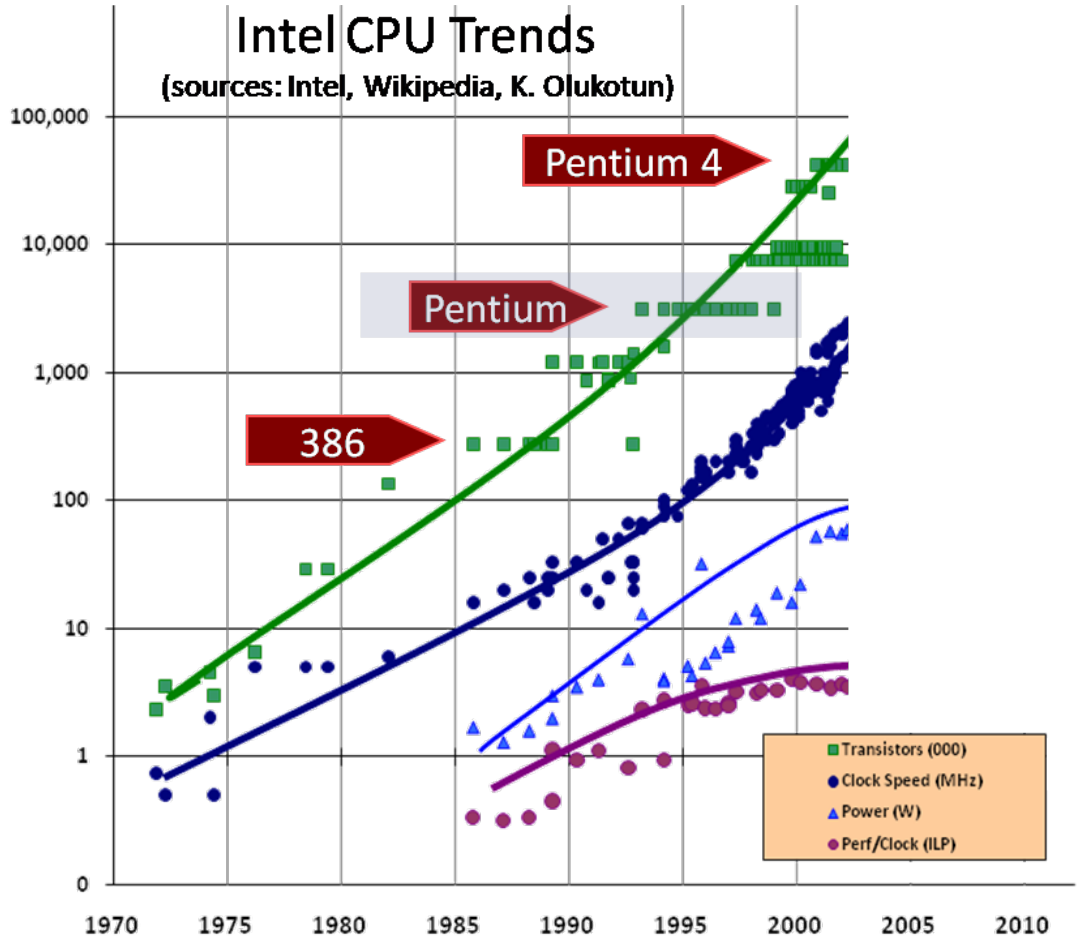
- Modern CPU architectures
- Implications for programming
- Importance of memory

Bayesian statistics & computing

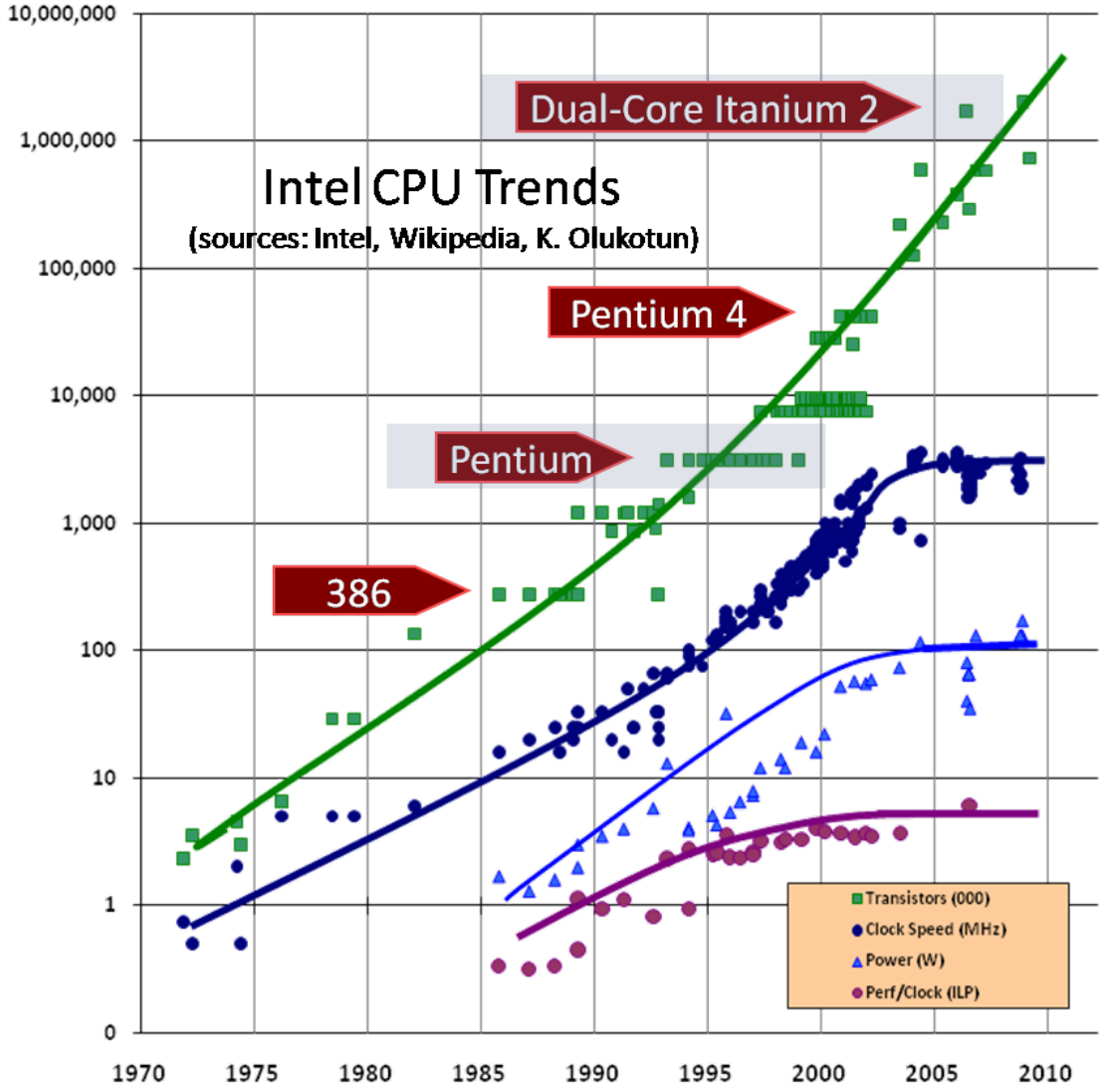
- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)

Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)



‘The free lunch is over’ — Herb Sutter



‘The free lunch is over’ — Herb Sutter

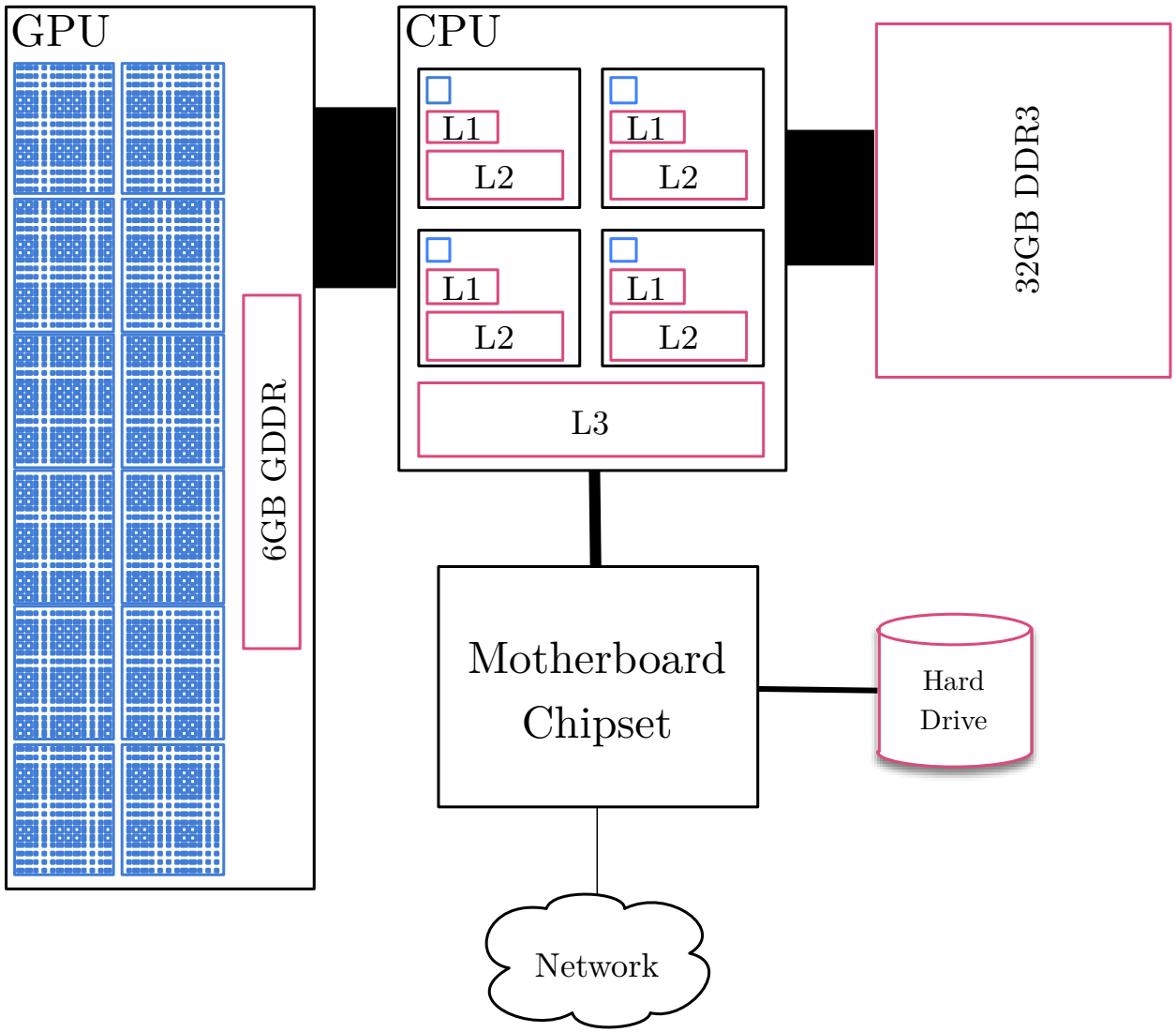
Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)

Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)
- 2010s trying to make MCMC parallel friendly firmly embedded as an important research direction
  - July 2006 Intel ship first desktop class dual core CPU
  - August 2006 Amazon EC2 launches as a public beta (production in 2008)
  - November 2006 nVidia announce CUDA, first ever C development environment for GPUs
  - $\approx$ 9 years later:
    - 12 core Intel Xeon CPUs (/w 30MB cache)
    - $2 \times 2,496$ core Tesla K80 GPUs (/w 12GB GDDR)
    - $\geq 50,000$ core EC2 clusters launched (/w 29TB RAM)

Massively simplified architecture



But ...

... let's take a step back.

Many core parallelism **isn't** the *whole* picture.

Modern CPUs are:

1. *super-scalar*;
2. *pipelined*;
3. with *dynamic execution*;
4. *speculative execution* (including branch prediction);
5. per core *hyper-threading*;
6. *vector instruction* units;
7. and large hierarchical caches.

But ...

... let's take a step back.

Many core parallelism **isn't** the *whole* picture.

Modern CPUs are:

1. *super-scalar*;
2. *pipelined*;
3. with *dynamic execution*;
4. *speculative execution* (including branch prediction);
5. per core *hyper-threading*;
6. *vector instruction* units;
7. and large hierarchical caches.

We'll dig into these points today, which are not directly related to many-core processing.

Aim today

- Hopefully most of you find it interesting in itself!

[1] vs 2.5 years

Aim today

- Hopefully most of you find it interesting in itself!
- Perhaps useful to a few people with short tight inner looped code to speed up.

i-like.org.uk

[1] vs 2.5 years

Aim today

- Hopefully most of you find it interesting in itself!
- Perhaps useful to a few people with short tight inner looped code to speed up.
- There are few recipies … hence deep understanding is key.
- **But**, whatever you do, *please* don't do this for all your code. Profile and then optimise only what's slow!
- Typically, a few hotspots account for most of your runtime.

[1] vs 2.5 years

Aim today

- Hopefully most of you find it interesting in itself!
- Perhaps useful to a few people with short tight inner looped code to speed up.
- There are few recipies … hence deep understanding is key.
- **But**, whatever you do, *please* don't do this for all your code. Profile and then optimise only what's slow!
- Typically, a few hotspots account for most of your runtime.
- e.g. Genetics application:
  - half dozen lines of code account for $>$ 99% of runtime
  - Chromopainter $< 5,000,000$ elements of HMM forward equation per second.
  - our problem specific optimized code $\approx 2,000,000,000$ elements of HMM forward equation per second on a single laptop CPU core $\implies$ 55 hours[1] for full forward sweep of Malaria data ($\approx 20,000$ individuals, $\approx 1,000,000$ loci) on one core.

[1] vs 2.5 years

x86-64

Instruction Sets and Assembly Language

A CPU has a set of fundamental operations it can perform called the *instruction set*. They correspond closely (though not injectively) to physical circuitry.

The compiler's job is to translate code into the instruction set of the CPU you want to run on. Assembly language is the (often bijective) mapping of the intruction set to a human readable form, so looking at assembler tells you exactly what the CPU 'sees' during execution.

For example, when I compiled `a+b+a*b` it became:

```
movsd xmm0, qword ptr [rbp-0x18]
movsd xmm1, qword ptr [rbp-0x18]
addsd xmm0, qword ptr [rbp-0x20]
mulsd xmm1, qword ptr [rbp-0x20]
addsd xmm0, xmm1
movsd qword ptr [rbp-0x10], xmm0
```

Hugely simplified CPU model

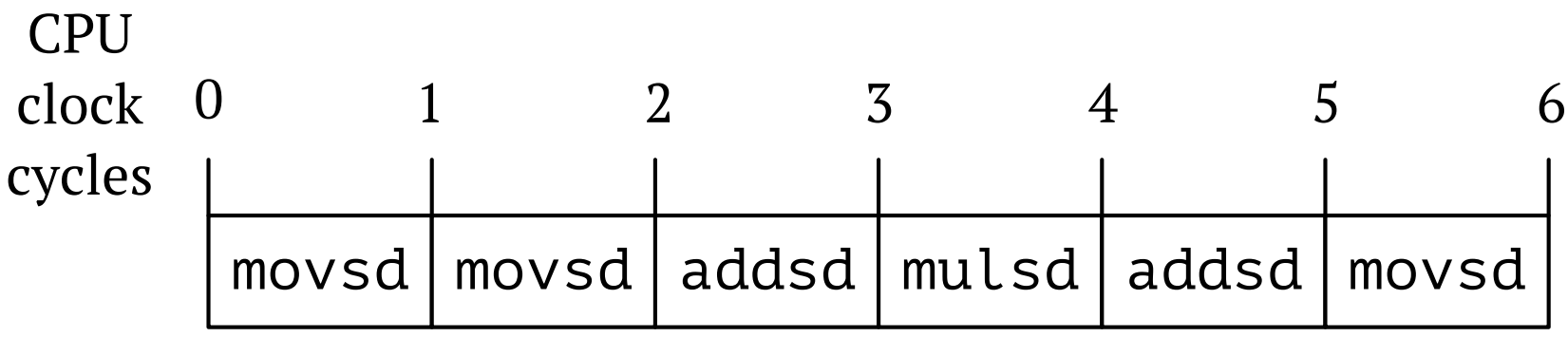
- Instructions in your code are loaded into the CPU
  - program counter points to first instruction, loads to instruction register
  - must then be ‘decoded’ before execution (takes time)
  - e.g. load from/store to memory
  - e.g. $\cdot + \cdot \rightarrow$ `add` $\cdot, \cdot$
  - CISC -vs- RISC battle of yore
- Non-memory instructions require at least one part in ‘registers’
  - `xmm0` and `xmm1` on previous slide were register names
  - the tiny, lightning fast working memory of the CPU
  - very scarce resource
  - CPU spends it’s life shuffling round memory & registers
- There are caches to speed up the management of memory operations
  - cache is *very* expensive and limited, but importance can’t be overstated

CPU nomenclature

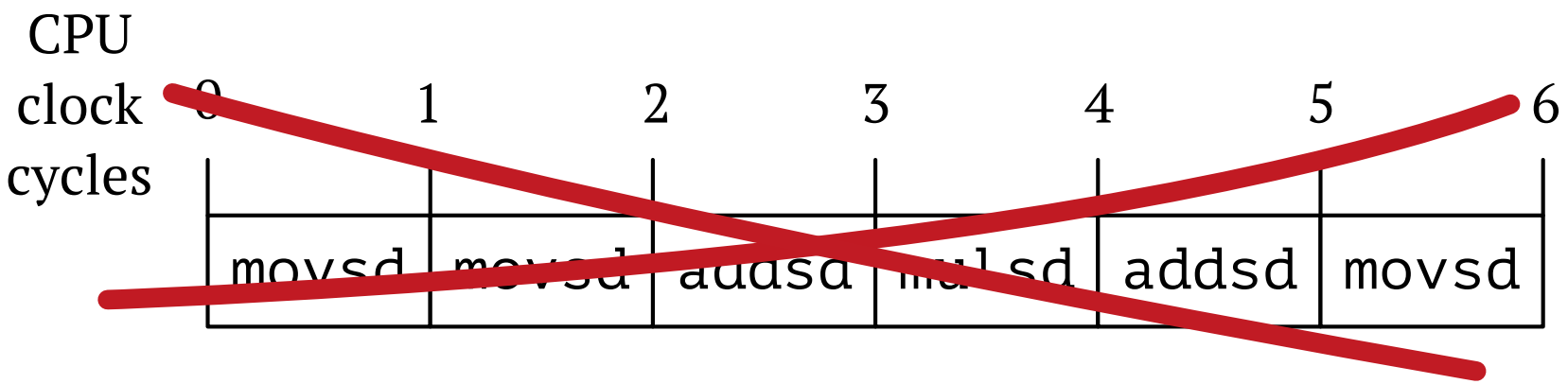
- x86, x86-64, ARMv6, ARMv7, ARMv8 ...
  - instruction sets defining fundamental capabilities of CPU
  - usually only compatible within architectures
    - x86 $\subset$ x86-64
    - ARMv6 $\subset$ ARMv7
- The high level instruction sets change very slowly ...
  - x86 (1978/1985), x86-64 (2003)
  - ARMv6 (2001), ARMv7 (2004), ARMv8 (2013)
- ... but many extension instruction sets glued on providing specific high performance (often SIMD) instructions
  - MMX (1996)
  - SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2006), SSE4 (2006)
  - AVX (2008), AVX2 (2013), AVX-512 (2017?)
  - NEON (2004)
- `less /proc/cpuinfo` for `flags:` line showing supported ISAs

Super-scalar pipelined architectures

One might imagine ...



One might imagine ...



But in fact, there are two crucial concepts for CPU instructions:

- **Throughput**: maximum number of the same kind of instruction which can be executed per clock cycle.
- **Latency**: how long does it take for result of instruction to be available after it is started.

We usually think about reciprocal throughput so that it is comparable with latency.

For the Intel Haswell architecture (my MacBook Pro)

Instruction	Port	1/Throughput	Latency
movsd (mem $\rightarrow$ reg)	2/3	0.5	6
movsd (reg $\rightarrow$ mem)*	2/3/7 + 4	1	5
movsd (reg $\rightarrow$ reg)	5	1	1
addsd (reg + reg)	1	1	3
addsd (reg + mem)*	1 + 2/3	1	9
mulsd (reg + reg)	0/1	0.5	5
mulsd (reg + mem)*	0/1 + 2/3	0.5	11
divsd (reg + reg)	0/1	14	20
divsd (reg + mem)*	0/1 + 2/3	14	26

* these are actually just fused micro-ops. In other words,

addsd (reg + mem) $\equiv$ movsd (mem $\rightarrow$ reg) & addsd (reg + reg).

i. Super-scalar architectures

$$\text{Throughput} > 1 \implies \text{more than 1 instruction in a clock cycle?}$$

i. Super-scalar architectures

Throughput $> 1 \implies$ more than 1 instruction in a clock cycle?

- Modern processors have many *execution units* within a single core.
- This means (for example) the circuitry for doing multiplication is printed twice on the same core.
- Dispatch to these execution units happens through *execution ports* (a potential bottleneck).
- $\therefore$ independent instructions can execute simultaneously on a single core!

Instruction-level parallelism

i. Super-scalar architectures

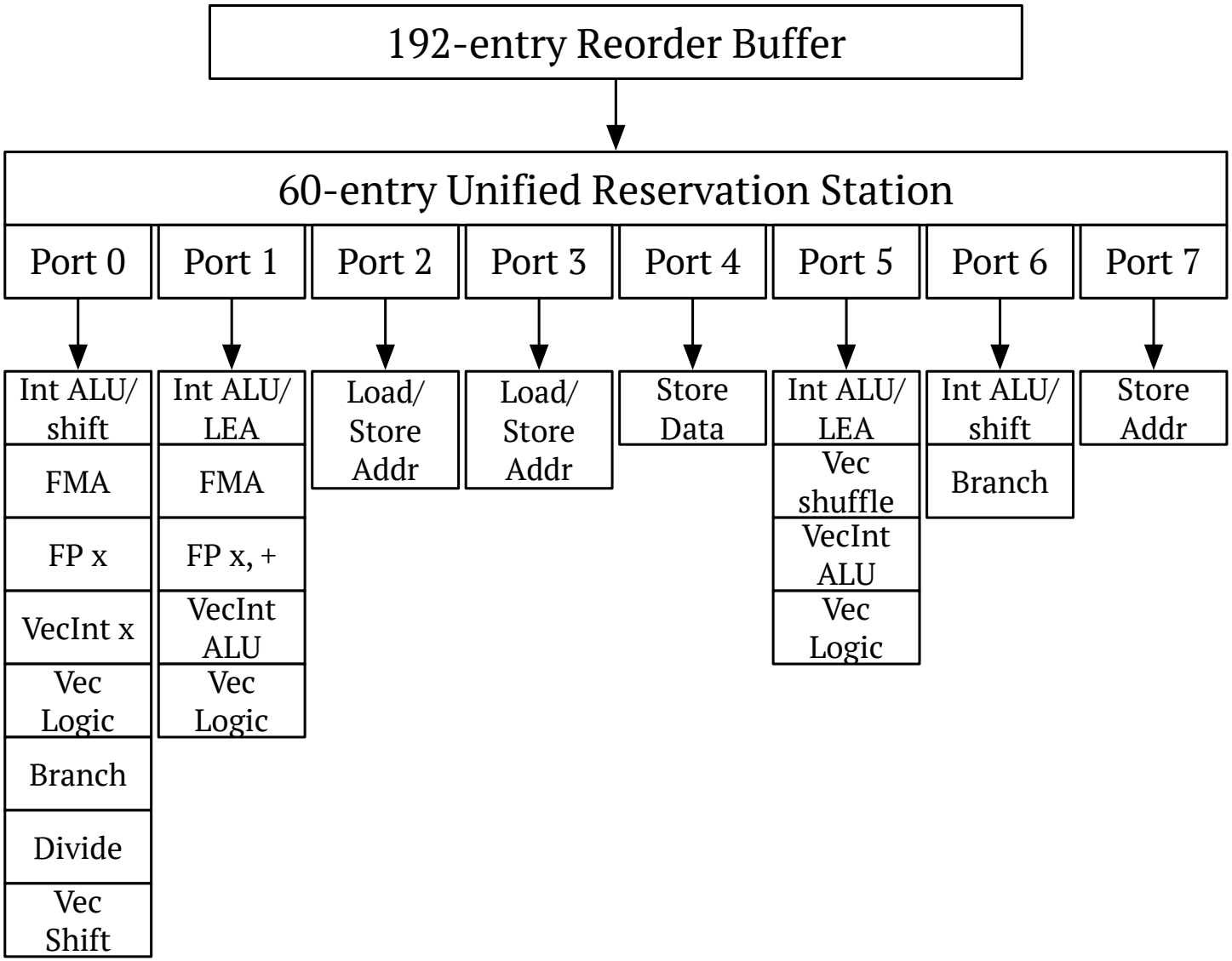
Throughput $> 1 \implies$ more than 1 instruction in a clock cycle?

- Modern processors have many *execution units* within a single core.
- This means (for example) the circuitry for doing multiplication is printed twice on the same core.
- Dispatch to these execution units happens through *execution ports* (a potential bottleneck).
- $\therefore$ independent instructions can execute simultaneously on a single core!

Instruction-level parallelism

Indeed, not just the same instruction and not just 2 ...

i. Super-scalar architectures (Haswell execution ports)

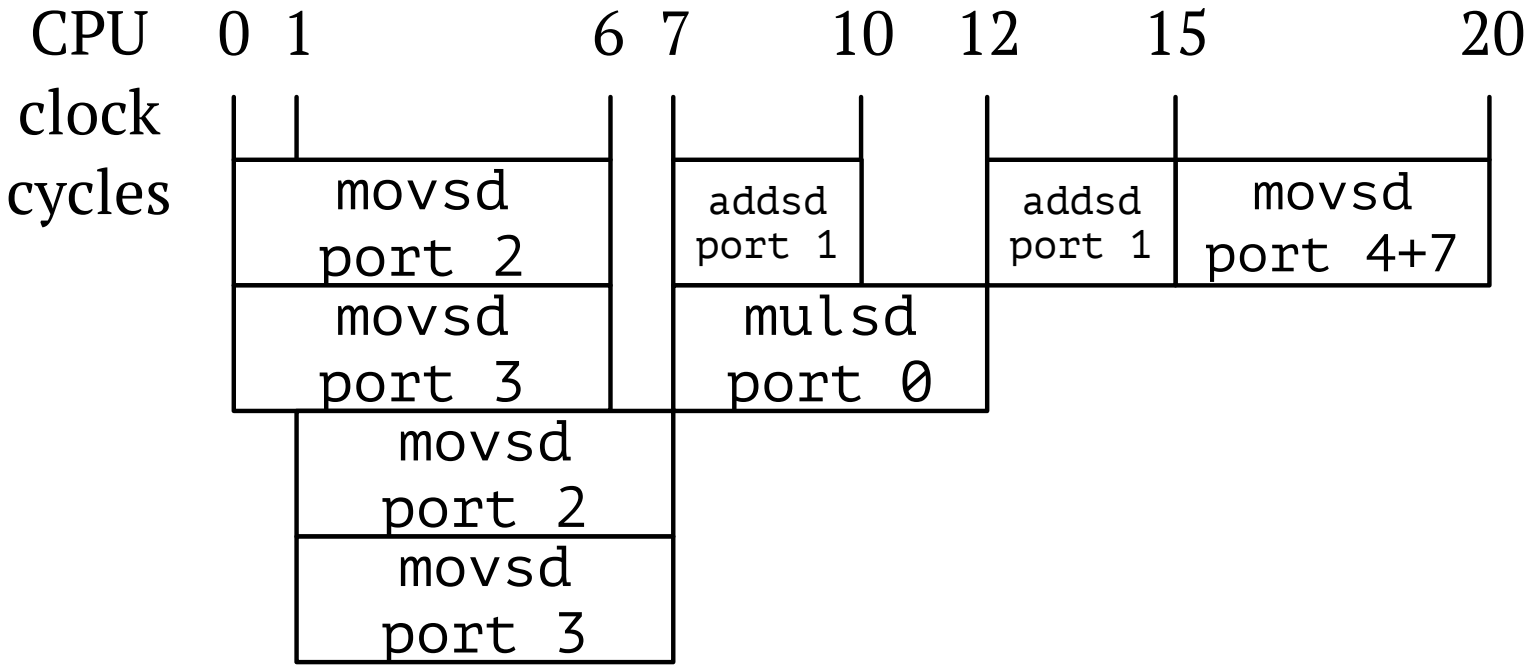


ii. Pipelined

- Great, so can dispatch two multiplies, but then what?
  - is execution port 0 out of action for 5 cycles?
  - or just the multiply execution unit?

ii. Pipelined

- Great, so can dispatch two multiplies, but then what?
  - is execution port 0 out of action for 5 cycles?
  - or just the multiply execution unit?
- Neither! Modern CPUs pipeline the circuitry so that a new instruction can start executing on the next cycle if it is independent of the 'in-flight' instructions.



Woah! Time-out!

Ok, all very nice … but it sounds like the CPU is pretty sophisticated and has this covered. Why should I care?

Woah! Time-out!

Ok, all very nice … but it sounds like the CPU is pretty sophisticated and has this covered. Why should I care?

Quiz: can this code be speeded up?

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = b * c * d * e * f * g * h * i;
```

… back to assembly

Well, what's wrong with this assembly?

```
movsd xmm0, qword ptr [rbp-0x18]
mulsd xmm0, qword ptr [rbp-0x20]
mulsd xmm0, qword ptr [rbp-0x28]
mulsd xmm0, qword ptr [rbp-0x30]
mulsd xmm0, qword ptr [rbp-0x38]
mulsd xmm0, qword ptr [rbp-0x40]
mulsd xmm0, qword ptr [rbp-0x48]
mulsd xmm0, qword ptr [rbp-0x50]
movsd qword ptr [rbp-0x10], xmm0
```

Runtime: 42 CPU cycles

… back to C

Version 1

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = b * c * d * e * f * g * h * i;
```

… back to C

Version 1

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = b * c * d * e * f * g * h * i;
```

Version 2

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = (b * c) * (d * e) * (f * g) * (h * i);
```

… back to assembly

```
movsd xmm0, qword ptr [rbp-0x18]
mulsd xmm0, qword ptr [rbp-0x20]
movsd xmm1, qword ptr [rbp-0x28]
mulsd xmm1, qword ptr [rbp-0x30]
mulsd xmm0, xmm1
movsd xmm1, qword ptr [rbp-0x38]
mulsd xmm1, qword ptr [rbp-0x40]
mulsd xmm0, xmm1
movsd xmm1, qword ptr [rbp-0x48]
mulsd xmm1, qword ptr [rbp-0x50]
mulsd xmm0, xmm1
movsd qword ptr [rbp-0x10], xmm0
```

… back to assembly

```
movsd xmm0, qword ptr [rbp-0x18]
mulsd xmm0, qword ptr [rbp-0x20]
movsd xmm1, qword ptr [rbp-0x28]
mulsd xmm1, qword ptr [rbp-0x30]
mulsd xmm0, xmm1
movsd xmm1, qword ptr [rbp-0x38]
mulsd xmm1, qword ptr [rbp-0x40]
mulsd xmm0, xmm1
movsd xmm1, qword ptr [rbp-0x48]
mulsd xmm1, qword ptr [rbp-0x50]
mulsd xmm0, xmm1
movsd qword ptr [rbp-0x10], xmm0
```

Runtime: 28 CPU cycles

… and the winner?

Version 3

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = ((b * c) * (d * e)) * ((f * g) * (h * i));
```

Runtime: 25 CPU cycles

In other words, Version 1 takes 92% longer to run![2] Don't want that in an inner loop.

[2] assuming cached accesses

… and the winner?

Version 3

```
double a, b, c, d, e, f, g, h, i;
// Values assigned
a = ((b * c) * (d * e)) * ((f * g) * (h * i));
```

Runtime: 25 CPU cycles

In other words, Version 1 takes 92% longer to run![2] Don't want that in an inner loop.

Another ridiculously simple example?

`array[i++]` versus `array[++i]`

[2] assuming cached accesses

A slightly less trivial example

Careful of dependency across loop boundaries (Fog, 2014):

```
const int size = 100;
float list[size], sum = 0; int i;
for (i = 0; i < size; i++) sum += list[i];
```

Latency: 300 cycles

A slightly less trivial example

Careful of dependency across loop boundaries (Fog, 2014):

```
const int size = 100;
float list[size], sum = 0; int i;
for (i = 0; i < size; i++) sum += list[i];
```

Latency: 300 cycles

```
const int size = 100;
float list[size], sum1 = 0, sum2 = 0; int i;
for (i = 0; i < size; i += 2) {
 sum1 += list[i];
 sum2 += list[i+1];}
sum1 += sum2;
```

Latency: 154 cycles

Take home message ...

... is to explicitly eliminate dependency chains as far as possible. The compiler is very conservative, **do not** assume it can figure it out!

Dynamic & speculative execution

Dynamic execution

- Some good news!

Dynamic execution

- Some good news!
- Modern CPUs don't dumbly execute machine instructions sequentially.
- Seen importance of latency and reciprocal throughput.
- CPU will 'park' instructions it can't execute yet due to dependency and look ahead for more work it can do.
    - this happens in the re-order buffer
        - 168 instructions for Sandy Bridge
        - 192 instructions for Haswell
        - 224 instructions for SkyLake

```
addsd xmm0, xmm1
mulsd xmm0, xmm2
addsd xmm1, xmm2
```

Speculative execution

- Indeed, CPU will even do simple branch prediction and start executing bits of code it isn't sure about yet.
  - if prediction was right, compute speed is as though branch wasn't there;
  - if prediction was wrong …

Speculative execution

- Indeed, CPU will even do simple branch prediction and start executing bits of code it isn't sure about yet.
  - if prediction was right, compute speed is as though branch wasn't there;
  - if prediction was wrong … really bad news!
- The entire pipeline of instructions needs to be flushed and refilled.
  - cost can be 20+ cycles

```
b = 0
if(a == 0) {
  b = 1;
}
c = 3.141 * b;
if(b == 0)
  ++b;
c = exp(b);
```

Speculative execution

- The CPU is ‘smart’ in that it will learn frequently revisited if/for/while/switch statements.
- Branch Target Buffer (BTB) holds a record of previous results used to predict future hits on the branch
  - not a simple majority vote
  - works right down to perfectly predicting the end of fixed size loops
- But … branching on pseudo-random outcomes as we are prone to do in statistics wreaks havoc, making branch prediction pretty useless
- Assume that you’ll pay the branch penalty for branches on random outcomes

An unrealistic example (don't do this!)

It's not the costly step, so this is unrealistic, but an example everyone can relate to is Metropolis-Hastings. Branch elimination here would sacrifice some additional compute:

```
if( pi(theta) * q(theta, thetanew) * u
     < pi(thetanew) * q(theta, thetanew) ) {
  theta = thetanew;
}
```

becomes

```
double increment = theta - thetanew;
*(int64_t*) &u =
     -( pi(theta) * q(theta, thetanew) *
        u < pi(thetanew) * q(theta, thetanew) );
*(int64_t*) &increment &= *(int64_t*) &u;
theta += increment;
```

Take home ...

... is to mathematically reformulate to eliminate unpredictable branches within tightly run code where possible.

But, caution required for this one: it's not an automatic win if the new code is much more expensive.

Vector units

CPU SIMD vector units

All modern CPUs have the capability to perform SIMD operations on *each* core. On Intel these are termed MMX, SSE and AVX.

These are special execution units which are much wider (bit width) and can perform the *same* instruction on multiple values at once. For example, AVX2 allows you to add 8 pairs of standard integers together in just 1 clock cycle (`vaddpd ymm, ymm, ymm`). This is where a lot of the reported modern speedup has come from since clock speeds stopped rising.

$$(x_1, x_2, \ldots, x_8) + (y_1, y_2, \ldots, y_8) \rightarrow (x_1 + y_1, x_2 + y_2, \ldots, x_8 + y_8)$$

The easy way

The first thing to be very careful about is expending effort completely unnecessarily. Sometimes all it takes is a few compiler switches and you get up to 4× speedup from vector units … but only works in simple cases!

First thing is to make sure you are specifying the architecture of the CPU you're working on. See:

`http://gcc.gnu.org/onlinedocs/gcc/i386-and-x86-64-Options.html`

Usually just use `gcc -march=native -mtune=native ...` and to see what this detects/activates check output of `gcc -march=native -Q --help=target`

Full compile & diagnose: `gcc -march=native -mtune=native -O3 -ftree-vectorizer-verbose=2`

Diving deep — is SEE/AVX being used?

See instructions of compiled object (say `test.o`) using:

```
objdump -d -M intel -S test.o
```

and check area of interest for vector calls. Can run on final executable, but hard to see what you want. Recommend splitting code of interest into small C file.

Even better, use free Intel Architecture Code Analyzer (demo).

Note: make sure *packed* operations. e.g. `vaddsd` = AVX scalar add, `vaddpd` = AVX vector add.

Or, use cool interactive compiler (C++):
`http://gcc.godbolt.org`

Advanced reading

Recommended advanced reading:

- `http://locklessinc.com/articles/vectorize/`
- `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`

When GCC can't figure out what you're doing ...

If loop structure too complicated for GCC to figure out automatically, can resort to *intrinsics*.

An 'intrinsic' is a C function which maps directly to an assembly language construct. This enables forcing the compiler to use the vector instruction sets (SSE/AVX).

However, means you have more housekeeping to do, roughly corresponding to three steps where there used to be one. Not onerous, but some people find it looks scary!

1. map your variables into vector types, correctly aligned
2. use intrinsic functions rather than usual functions or arithmetic symbols (+, ∗, etc)
3. where necessary unpack results from vector units

Intrinsics — Step 1a, memory alignment

For AVX must align to 32-byte boundary, for SSE to 16-byte boundary.

To align regular array of, say, 100 `doubles`:

```
double a[100] __attribute__ ((aligned (32)));
```

This is GCC specific.

To align dynamic memory for 100 `doubles`, simply replace `malloc` call with:

```
double *b = aligned_alloc(32, 100 * sizeof(double));
```

This is an ISO C11 standard function declared in `stdlib.h`.

Intrinsics — Step 1b, map variables

The compiler needs to know that you want to treat the array as a packed vector for intrinsics. The packed variable types are:

ISA	Normal type	Packed type	Size
SSE	4×`float`	`__m128`	128-bit
SSE2	4×`int`	`__m128i`	128-bit
	2×`double`	`__m128d`	128-bit
AVX	8×`float`	`__m256`	256-bit
	4×`double`	`__m256d`	256-bit
AVX2	8×`int`	`__m256i`	256-bit

Note: Integer vector types can also hold more `char` or `short`, or fewer `long long int`

Intrinsics — Step 1b, map variables

The easiest (but not only) way is to create a pointer of the packed type which points to the aligned regular type array.

```
double a[100] __attribute__ ((aligned (32)));
__m256d *aV = (__m256d*) a;
```

Now `aV[0]` points to a vector of elements 0–3 of `a` and `aV[1]` points to a vector of elements 4–7 of `a`. AVX instructions executed on `aV` therefore operate on 4 elements of `a` at a time.

Since it's a pointer, if `aV[i]` is assigned to then we can access the results through `a[i*4]` to `a[i*4+3]`.

Note: The header to include is `immintrin.h` for both the types and intrinsic functions.

Intrinsics — Step 2, use intrinsic functions

All operations now take place on the packed type and we switch to intrinsics. Thus:

```
for(size_t i=0; i<100; ++i)
  a[i] = a[i]+b[i];
```

becomes

```
for(size_t i=0; i<100/4; ++i)
  aV[i] = _mm256_add_pd(aV[i], bV[i]);
```

The best documentation to find the instrinsic you need is provided by Intel:

`http://software.intel.com/sites/landingpage/IntrinsicsGuide/`

Intrinsics — Step 3, unpack results

If we had instead wanted to sum the vector for example

```
__m256d sumV = _mm256_setzero_pd();
for(size_t i=0; i<100/4; ++i)
  sumV = _mm256_add_pd(sumV, aV[i]);
```

Then the partial sums of `a` would be in the 4 elements of `sum`. We could copy out to a regular aligned `double` and sum:

```
double total;
double sum[4] __attribute__ ((aligned (32)));
_mm256_store_pd(sum, sumV);
total = sum[0]+sum[1]+sum[2]+sum[3];
```

Intrinsics — the upshot

It may be worth learning intrinsics if you want to be able to have low level control to guarantee you're getting use of vector units.

All the indications are it's only going to get more important: ultimately there is a limit to what compilers can detect and auto-vectorise, yet Intel already have a roadmap for 1024-bit vector units.

That's 16 `double` instructions per clock tick … would be almost like the CDT machines being $16 \times 48 = 768$ core! GPU territory.

Cache

The CPU cache

- By default the CPU always loads (and stores) data through the cache, triggering the load of a full ‘cache line’
  - A cache line is the atomic unit size of memory the cache deals with
  - Loading a single byte will trigger loading the whole aligned cache line: organise your data to take advantage!
- Cache retains this data until
  - it is stale (another core updates)
  - it hasn’t been accessed for a long time and more data needs to be cached
- We essentially have no direct control of the cache
- Modern CPUs will auto-detect common patterns of access and trigger prefetching
  - sequential reads
  - strided reads

Approximate Intel Haswell memory characteristics

Memory Access	Size	Latency
Registers (per core)	168 physical	0 clocks
L1 cache (per core)	0.03MB	≈ 4 clocks
L2 cache (per core)	0.25MB	≈ 12 clocks
L3 cache (shared)	2 – 30MB	≈ 36 clocks
Main memory	up to $32,768$ MB	≈ 212 clocks
Hard drive	Terabytes	can be $> 10^6$ clocks

Cache line: 64 bytes

Considerations

We can't explicitly control the caches, but indirectly can have an effect:

- If you need multiple passes over memory, try to block the passes together to fit in an appropriate cache
- Where possible, ensure temporally local accesses are also spatially local
  - e.g. operating on matrices
- For uncommon but predictable (to you) access patterns, software prefetching may help in rare cases.
  - CPU will only detect fairly obvious patterns as candidates for hardware prefetch.

Conclusion

For that small bit of code in a tight inner loop:

- avoid long dependency chains, so that the CPU can always be doing something useful;
- keep dependency chains well below the re-order buffer size (192 on Haswell);
- steer clear of divison operations where possible;
- avoid branching, especially randomly, anywhere that is possible — perhaps even if it means more compute (profile!);
- make use of the vector capabilities of each core: except in trivial cases, you're going to have to be explicit;
- be careful about memory layout and access patterns to make best use of cache.