

CS250B: Modern Computer Systems

Performance Profiling with PerfTools



Sang-Woo Jun

How To Evaluate Our Approaches?

- ❑ Say, we made a performance engineering change in our program
 - ...And performance decreased by 10%
 - Why? Can we know?
- ❑ Many tools provide profiling capabilities
 - gprof, OProfile, Valgrind, VTune, PIN, ...
- ❑ We will talk about perf, part of perf tools
 - Native support in the Linux kernel
 - Straightforward PMC (Performance Monitoring Counter) support

Aside:

Performance Monitoring Counters (PMC)

- ❑ Problem: How can we measure architectural events?
 - L1 cache miss rates, branch mis-predicts, total cycle count, instruction count, ...
 - No way for software to know
 - Events happen too often for software to be counting them
- ❑ Solution: PMCs (Sometimes called Hardware Performance Counters)
 - Dozens of special registers that can each be programmed to count an event
 - Privileged registers, only accessible by kernel
 - Supported PMCs differ across models and designs
- ❑ Usage
 - Program PMC, read PMC, run piece of code, read PMC, compare read values

Linux Perf

- ❑ Performance analysis tool in Linux
 - Natively supported by kernel
 - Supports profiling a VERY wide range of events: PMC to kernel events
 - Note: needs sudo to do most things
- ❑ Many operation modes: top, stat, record, report, ...
 - Supported events found in “sudo perf list”

```
List of pre-defined events (to be used in -e):
```

```
branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]
bus-cycles [Hardware event]
cache-misses [Hardware event]
cache-references [Hardware event]
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
ref-cycles [Hardware event]
```

```
⋮
page-faults OR faults [Software event]
task-clock [Software event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-loads [Hardware cache event]
L1-dcache-stores [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
LLC-load-misses [Hardware cache event]
LLC-loads [Hardware cache event]
```

Linux Perf: Stat

❑ Default command prints some useful information

○ “sudo perf stat ls”

❑ More events can be traced using -e

○ sudo perf stat -e task-clock,page-faults,cycles,instructions,branches,branch-misses,LLC-loads,LLC-load-misses ls

Performance counter stats for 'ls':

0.652008	task-clock (msec)	#	0.805 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
104	page-faults	#	0.160 M/sec
2,797,861	cycles	#	4.291 GHz
2,245,082	instructions	#	0.80 insn per cycle
444,095	branches	#	681.119 M/sec
16,749	branch-misses	#	3.77% of all branches

0.000810402 seconds time elapsed

Performance counter stats for 'ls':

0.681485	task-clock (msec)	#	0.781 CPUs utilized
102	page-faults	#	0.150 M/sec
2,921,152	cycles	#	4.286 GHz
2,217,325	instructions	#	0.76 insn per cycle
439,589	branches	#	645.046 M/sec
16,608	branch-misses	#	3.78% of all branches
9,736	LLC-loads	#	14.286 M/sec
3,269	LLC-load-misses	#	33.58% of all LL-cache hits

0.000872088 seconds time elapsed

Linux Perf: Record, Report

- ❑ Log events with “record”, interactively analyze it with “report”
 - `sudo perf record -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses [...]`
 - Creates “perf.data”
- ❑ “sudo perf report” reads “perf.data”

```
Available samples
8 cycles
8 instructions
8 L1-dcache-loads
7 L1-dcache-load-misses
```



```
Samples: 8 of event 'cycles', Event count (approx.): 2476964
Overhead Command Shared Object Symbol
96.20% tail [kernel.kallsyms] [k] memcpy_erms
3.80% perf [kernel.kallsyms] [k] perf_event_addr_filters_exec
0.00% perf [kernel.kallsyms] [k] native_write_msr
0.00% tail [kernel.kallsyms] [k] native_write_msr
```

This is where most cycles are spent!

```
Samples: 7 of event 'L1-dcache-load-misses', Event count (approx.): 36818
Overhead Command Shared Object Symbol
86.85% tail [kernel.kallsyms] [k] copy_page
12.36% perf [kernel.kallsyms] [k] perf_iterate_ctx
0.74% perf [kernel.kallsyms] [k] perf_event_addr_filters_exec
0.04% perf [kernel.kallsyms] [k] native_write_msr
0.00% tail [kernel.kallsyms] [k] native_write_msr
```

This is where most L1 cache misses are!

CS 250B: Modern Computer Systems

Modern Processors – Handling Branches

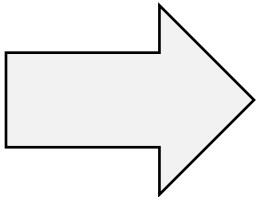


Sang-Woo Jun

What Do Conditionals Compile To?

- ❑ Conditionals (sometimes) compile to branch instructions in assembly
 - Compiler optimizations may replace branch instructions with something else
 - But not always
- ❑ Branch instructions take cycles(s)
 - At least one cycle, perhaps more
 - Obvious!

```
int bar(int v) {  
    if ( v == 0 ) return 1;  
    else return 0;  
}
```

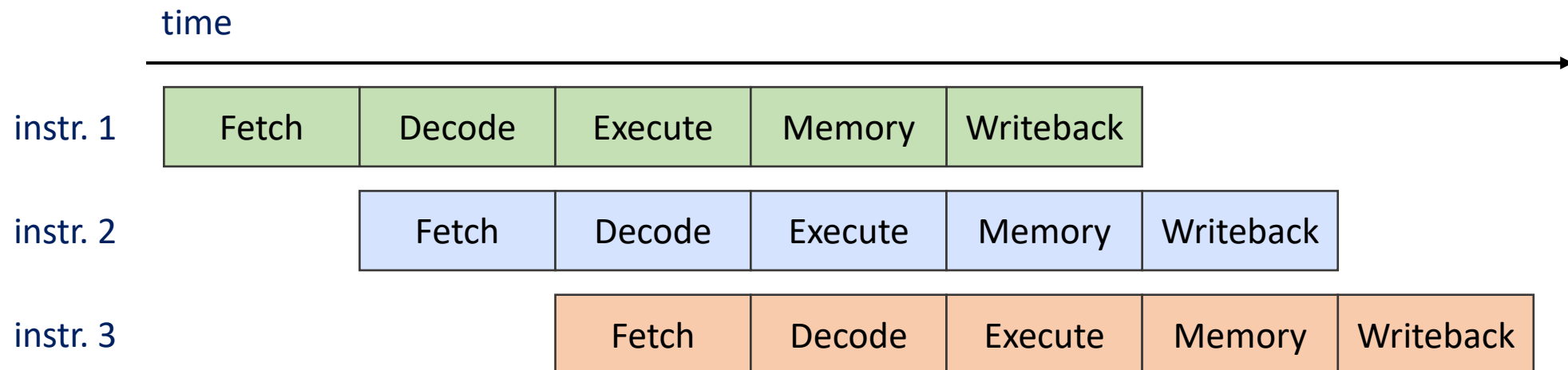

gcc, x86-64,
no optimizations

```
bar(int):  
    push    rbp  
    mov     rbp, rsp  
    mov     DWORD PTR [rbp-4], edi  
    cmp     DWORD PTR [rbp-4], 0  
    jne     .L2  
    mov     eax, 1  
    jmp     .L3  
.L2:  
    mov     eax, 0  
.L3:  
    pop     rbp  
    ret
```


Remember:

Pipelined Processors and Hazards

- ❑ Modern, pipelined processors handle multiple instructions at once
 - Ideally, N-stage pipeline processes N instructions at a given cycle
 - But, sometimes future instructions depend on results of earlier ones (“Hazard”)
 - Many types of hazards were introduced in undergrad architecture class
- ❑ Today, we look at the impact of handling “Control hazards”



Handling Control Hazards

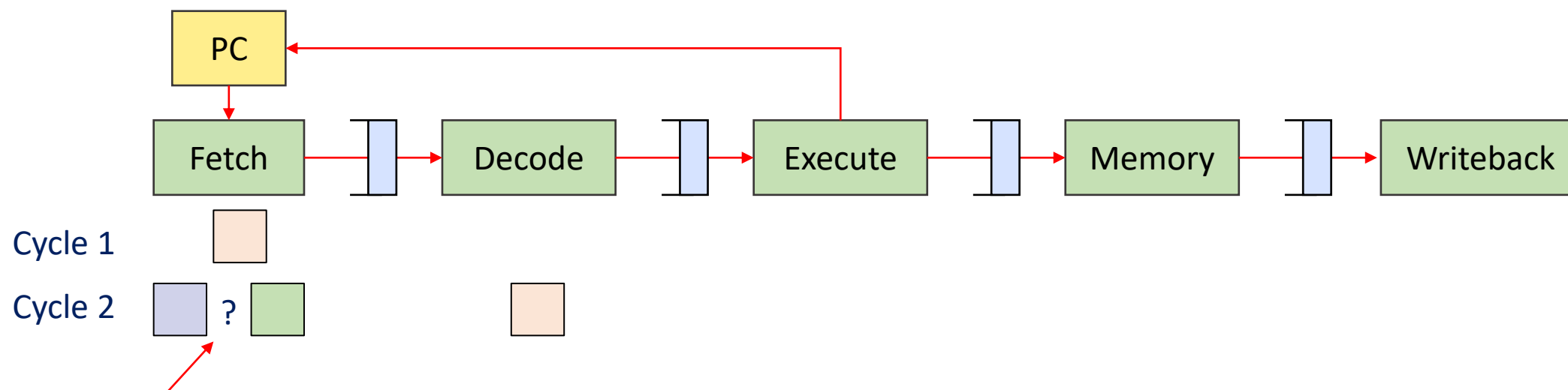
- ❑ Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - e.g., Still working on decode stage of branch

i1: beq s0, zero, elsewhere

i2: addi s1, s0, 1

elsewhere:

i3: addi s1, s0, 2



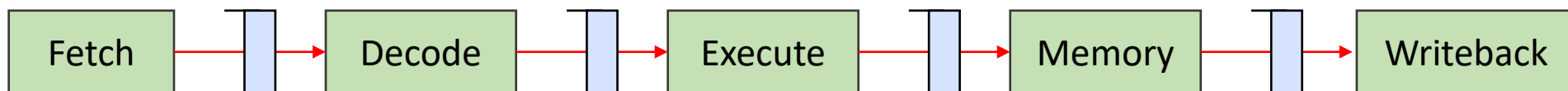
Which one should I load?

Stalling until we know the correct answer results in multi-cycle overhead

Control Hazard (Partial) Solution: Branch Prediction

- ❑ Processor will try to predict whether branch is taken or not
 - If prediction is correct, great!
 - Single cycle overhead
 - If not, we do not apply the effects of mis-predicted instructions
 - Effectively same performance penalty as stalling in this case
 - Can be many cycles of overhead depending on pipeline depth

Simple Branch Predictor Example



```
addi t1, zero, 3
addi t2, zero, 3
beq t1, t2, skip
sw t3, 0(t0)
ret
skip:
sw t2, 0(t0)
ret
```



Mispredict detected!

Pipeline bubbles

Fetch correct branch

No state update before Execute stage can detect misprediction
(Fetch and Decode stages don't write to register)

Some Classes Of Branch Predictors

❑ Static branch prediction

- Based on typical branch behavior
- Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken

❑ Dynamic branch prediction

- Hardware measures actual branch behavior
 - e.g., record recent history (1-bit “taken” or “not taken”) of each branch in a fixed size “branch history table”
- **Assume future behavior will continue the trend**
 - When wrong, stall while re-fetching, and update history

Many many different methods, Lots of research, some even using neural networks!

Branch prediction and performance

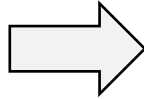
- ❑ Effectiveness of branch predictors is crucial for performance
 - Spoilers: On SPEC benchmarks, modern predictors routinely have 98+% accuracy
 - Of course, less-optimized code may have much worse behavior
- ❑ Branch-heavy software performance depends on good match between software pattern and branch prediction
 - Some high-performance software optimized for branch predictors in target hardware
 - Or, avoid branches altogether! (Branchless code)

Recap: Loop unrolling

A Compiler Solution To Branch Hazards

```
for ( i = 0 to 15 ) foo();
```

Loop unrolling



```
for ( i = 0 to 3 ) {  
    foo();  
    foo();  
    foo();  
    foo();  
}
```

Potentially **16** branch mispredicts
Even without mispredicts,
branch instruction consume **16** cycles

Potentially **4** branch mis-predicts
Without mis-predicts,
branch instruction consume **4** cycles

We can do this manually, or tell the compiler to do its best

- GCC flags `-funroll-loops`, `-funroll-all-loops`
- How much to unroll depends on heuristics within compiler

Code Example: Counting Numbers

□ How fast is the following code?

- a and b are initialized to `rand()%256`
- cnt is 100,000,000
- Compiled with GCC `-O3`

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) cnt++;  
}
```

□ This code takes 0.44s on my desktop (i5 @ 3 GHz)

- Each loop takes 13.2 cycles ($3 \text{ GHz} * 0.44 / 100,000,000$)
- Can we do better? *My x86 is 4-way superscalar!*

Optimization Attempt #1: Loop Unrolling

- ❑ There are three potential branch instruction locations
 - “i < cnt”, “a[i] < 128”, and b[i] < 128”
- ❑ Is the bottleneck the “for” loop?
 - Let’s try giving -funroll-all-loops

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) lcnt++;  
}
```

- ❑ Performance increased from 0.44s to ~0.43s.
 - Better, but not by much

Identifying The Bottleneck

- ❑ We predict the “if” statements are the bottlenecks
 - Each of the two branch instructions has a 50% chance of being taken
 - Branch prediction very inefficient!

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) lcnt++;  
}
```

- ❑ Performance improves when comparison becomes skewed
 - 0.44s when comparing against 128 (50%)
 - 0.27s when comparing against 64 (25%), 0.17s with 32

Optimization Attempt #2: Branchless Code

- ❑ Let's try getting rid of the "if" statement. How?
- ❑ Some knowledge of architectural treatment of numbers is required
 - x86 represents negative numbers via two's complement
 - "1" == 0x1, "-1" == 0xffffffff
 - "1>>31" == 0x0, "-1>>31" == 0xffffffff
- ❑ "(v-128)>>31"
 - if v >= 128: 0x0
 - v < 128: 0xffffffff

So many more instructions! Will this be faster?

```
for ( int i = 0; i < cnt; i++ ) {  
>  lcnt += ( (((a[i] - 128)>>31)&1) * (((b[i] - 128)>>31)&1) );  
}
```

Comparing Performance Numbers

Name	Elapsed (s)
Vanilla	0.44 s
Branchless	0.06 s

Branch predictor is almost always correct



Vanilla: Total misses: 57 M out of 3,623 M

Overhead	Command	Shared Object	Symbol
87.38%	a.out	a.out	[.] main
9.80%	a.out	libc-2.27.so	[.] __random
1.48%	a.out	libc-2.27.so	[.] __random_r
0.36%	a.out	[kernel.kallsyms]	[k] __pagevec_lru_add_fn
0.29%	a.out	[kernel.kallsyms]	[k] get_page_from_freelist

~2 cycles per loop! 8 Operations with 4 way superscalar...

Branchless: Total misses: 7 M out of 3,514 M

Over 7x performance!

Overhead	Command	Shared Object	Symbol
77.47%	a.out	libc-2.27.so	[.] __random
10.13%	a.out	libc-2.27.so	[.] __random_r
3.12%	a.out	[kernel.kallsyms]	[k] get_page_from_freelist
2.86%	a.out	[kernel.kallsyms]	[k] __pagevec_lru_add_fn
1.78%	a.out	[kernel.kallsyms]	[k] __handle_mm_fault
0.74%	a.out	a.out	[.] main
0.70%	a.out	libc-2.27.so	[.] rand

Interestingly, loop with only one comparator is automatically optimized by compiler

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 ) lcnt ++;  
}
```

Shows same performance as the branchless one

Aside: Spectre (Simplified)



- ❑ Branch prediction is supposed to be transparent
- ❑ But not all aspects of it are!

```
int array[256];
char forbidden = (*forbidden_ptr);
if ( a != a ) {      Never executes, so never caught by sandbox (e.g., JVM, javascript)
    int ignore = array[forbidden] ; But mispredicted branch still affects cache!
}
for ( int i = 0; i < 256; i++ ) is_in_cache(array[i]);
Attacker can time the cache to discover contents
```

Slightly Deeper into Speculative Execution

- ❑ Branch prediction is one example of Speculative Execution
- ❑ Modern processors speculatively execute many things!
 - e.g., Does this Virtual Memory page belong to the Kernel?
 - Assume we have access to everything, and roll back state if turns out to be false
 - Always waiting for checks is too much overhead!

Aside: Meltdown (Simplified)

- ❑ Caches are supposed to be transparent
- ❑ But not all aspects of it are!

```
int array[256];  
char forbidden = (*forbidden_ptr);  
int ignore = array[forbidden] ;  
for ( int i = 0; i < 256; i++ ) is_in_cache(array[i]);
```

Assuming `(*forbidden_ptr)` returns zeros in the correct operation, instead of protection fault

Always being safe is too much overhead. How do we fix this?



Questions?

CS 250B: Modern Computer Systems

Modern Processors – SIMD Extensions



Sang-Woo Jun

Modern Processor Topics

❑ Transparent Performance Improvements

- Pipelining, Caches
- Superscalar, Out-of-Order, Branch Prediction, Speculation, ...
- Covered in CS250A and others

❑ Explicit Performance Improvements

- SIMD extensions, AES extensions, ...
- ...

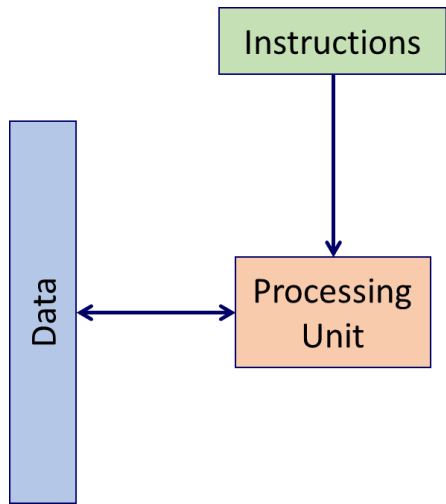
❑ Non-Performance Topics

- Virtualization extensions, secure enclaves, transactional memory, ...

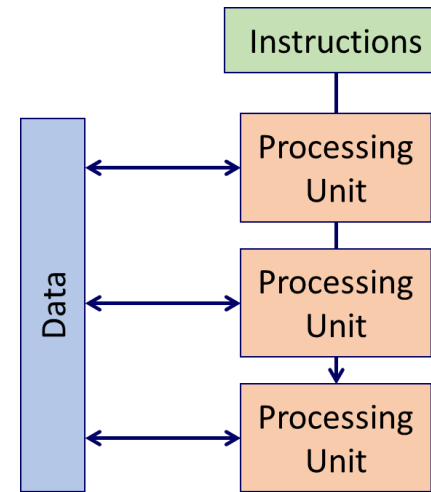
Flynn Taxonomy (1966) Recap

		Data Stream	
		Single	Multi
Instruction Stream	Single	SISD (Single-Core Processors)	SIMD (GPUs, Intel SSE/AVX extensions, ...)
	Multi	MISD (Systolic Arrays, ...)	MIMD (VLIW, Parallel Computers)

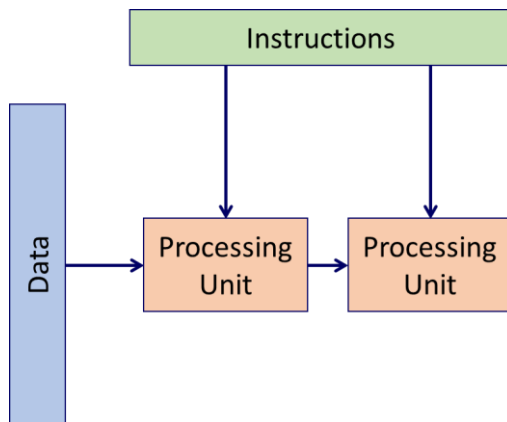
Flynn Taxonomy Recap



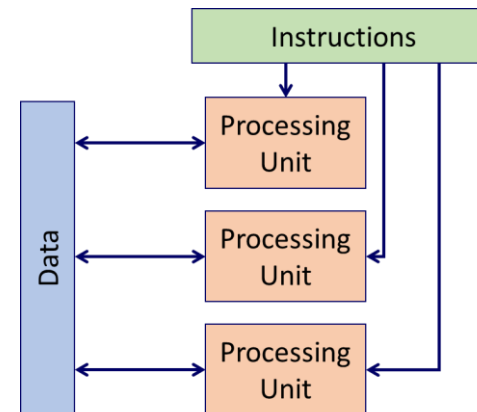
Single-Instruction
Single-Data
(Single-Core Processors)



Single-Instruction
Multi-Data
(GPUs, SIMD Extensions)



Multi-Instruction
Single-Data
(Systolic Arrays,...)

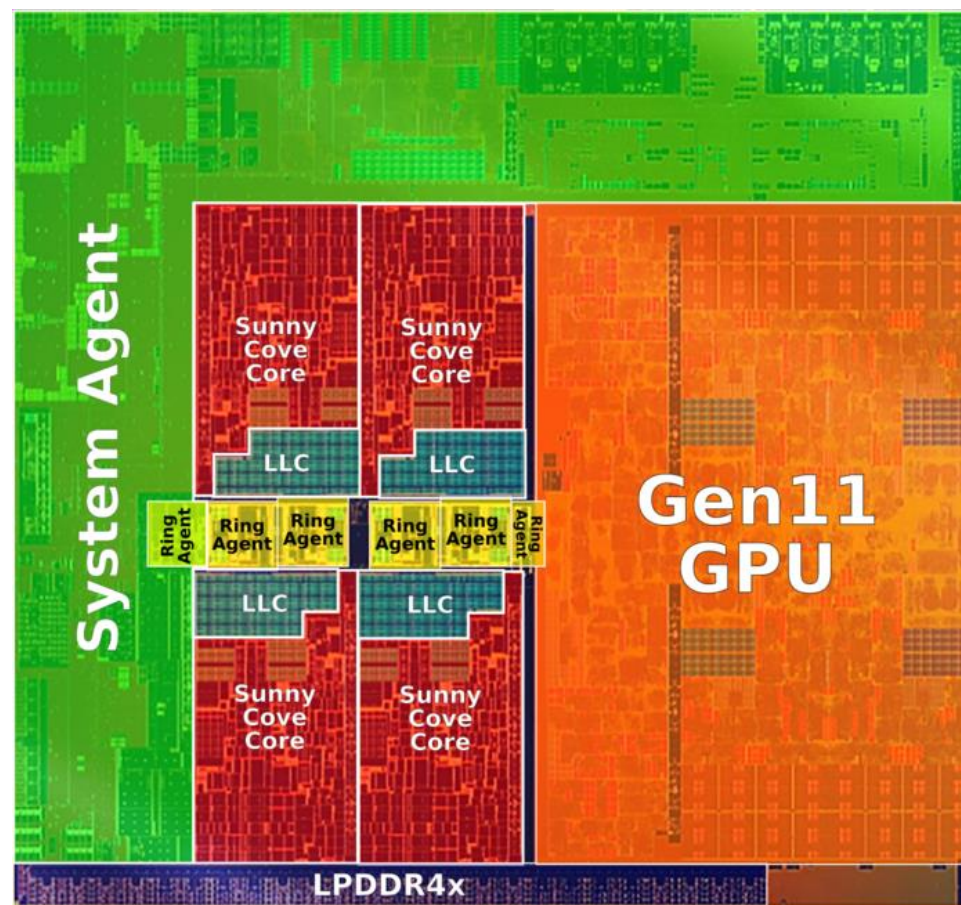
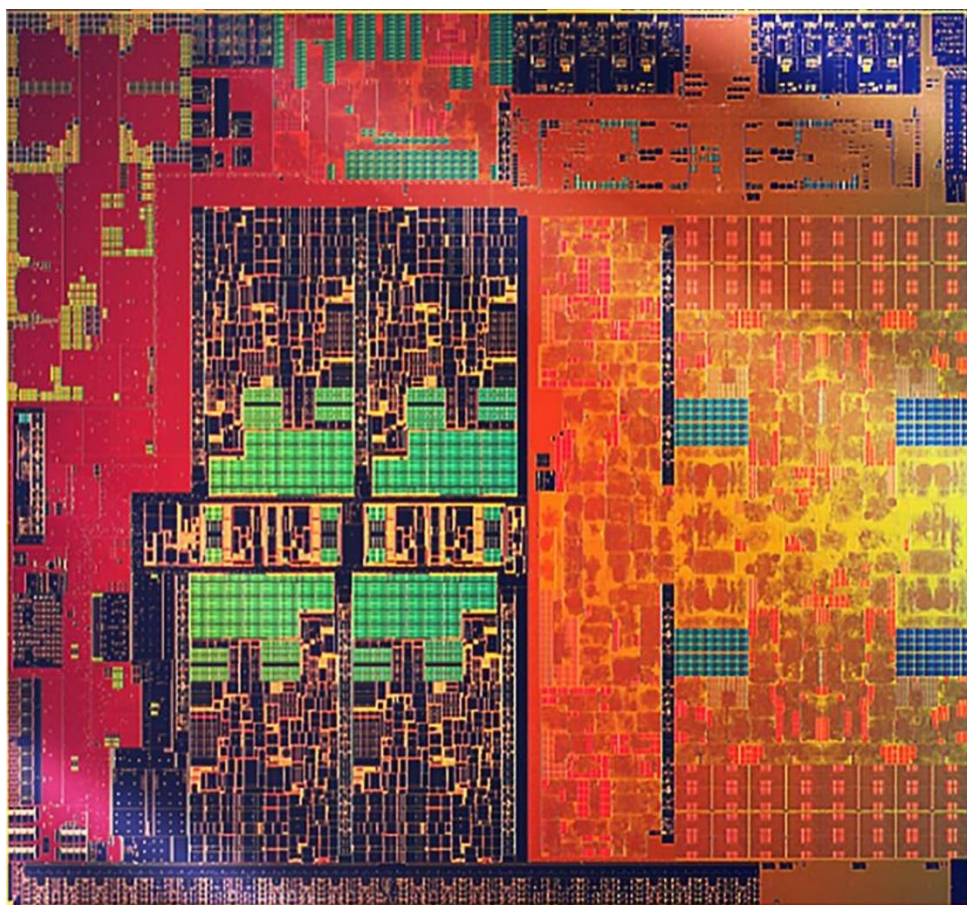


Multi-Instruction
Multi-Data
(Parallel Computers)

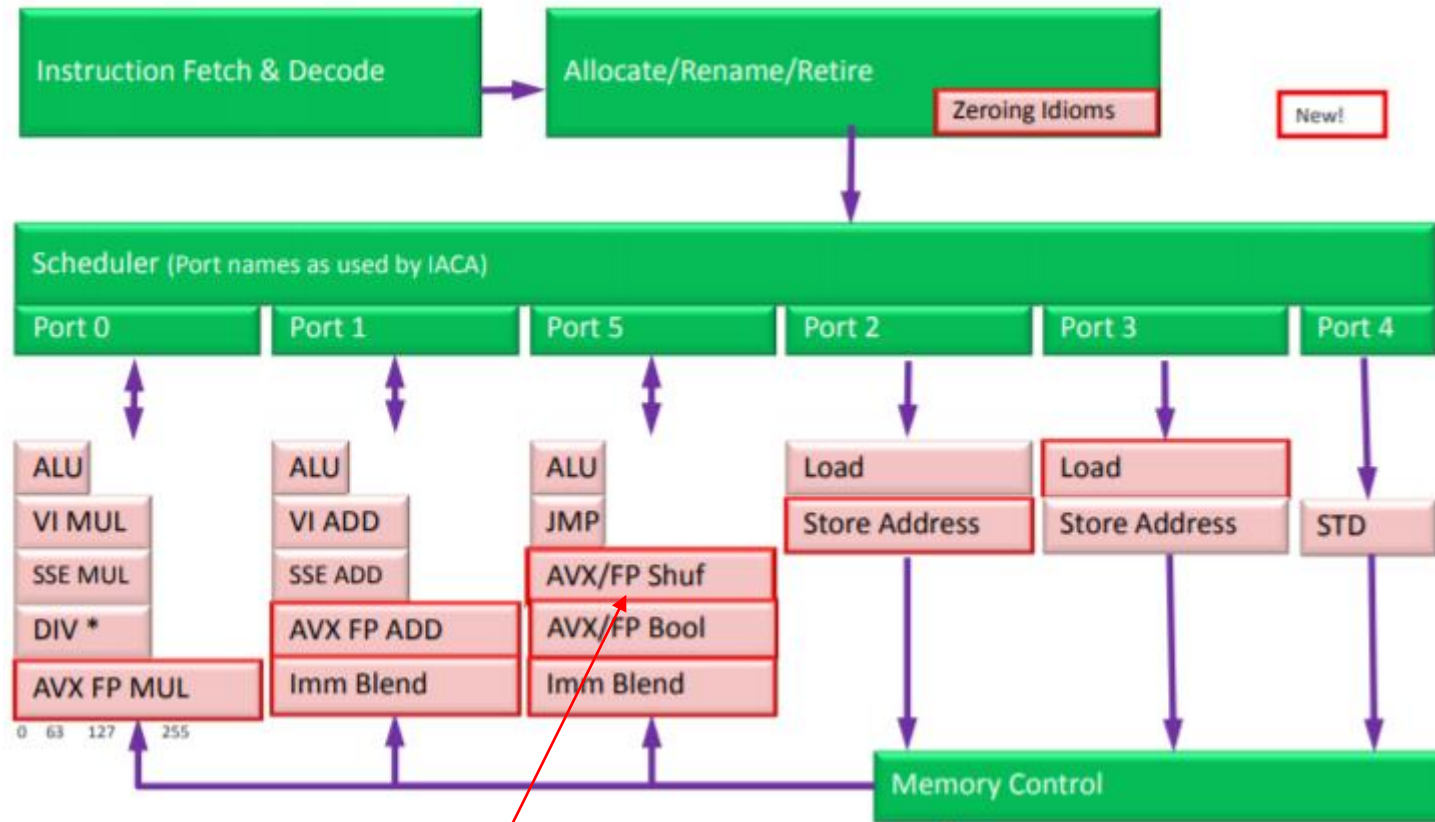
Intel SIMD Extensions

- ❑ New instructions, new registers
- ❑ Introduced in phases/groups of functionality
 - SSE – SSE4 (1999 – 2006)
 - 128 bit width operations
 - AVX, FMA, AVX2, AVX-512 (2008 – 2015)
 - 256 – 512 bit width operations
- ❑ F16C, and more to come?

Ice Lake Annotated Die (2019)

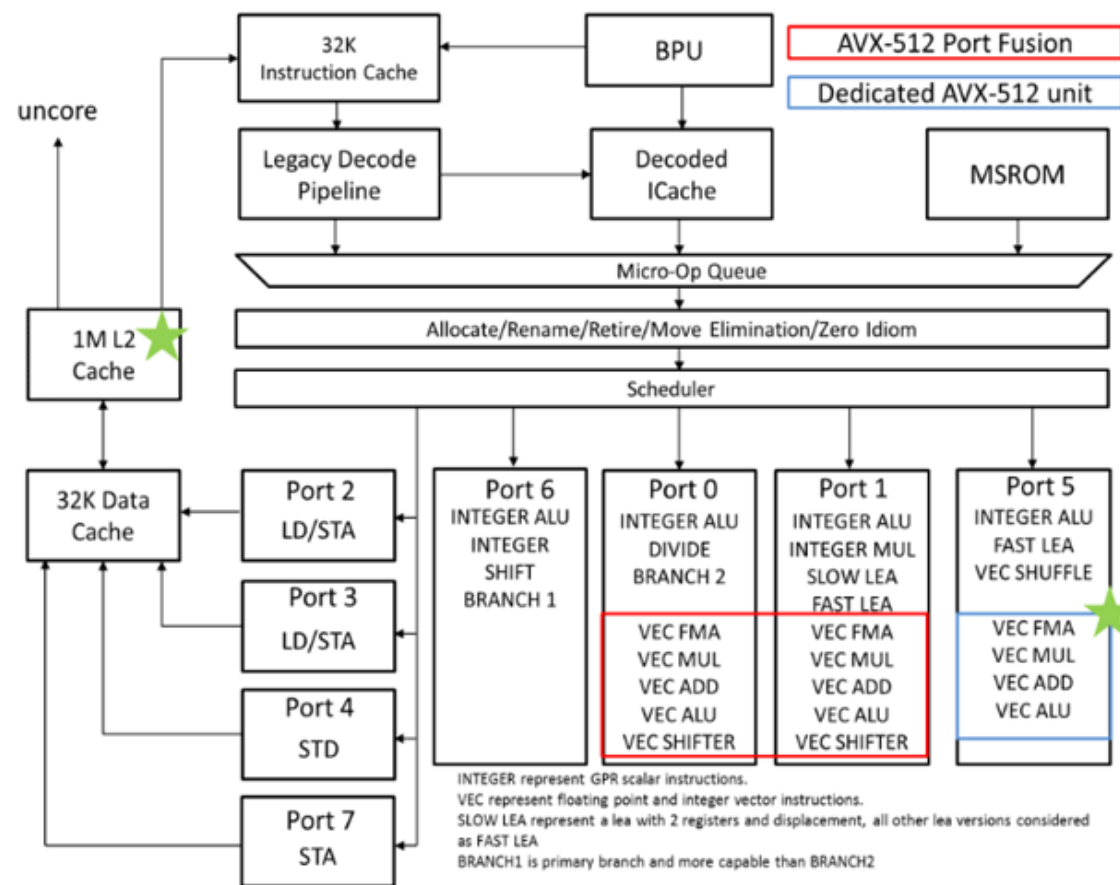


Sandy Bridge Microarchitecture (2011)

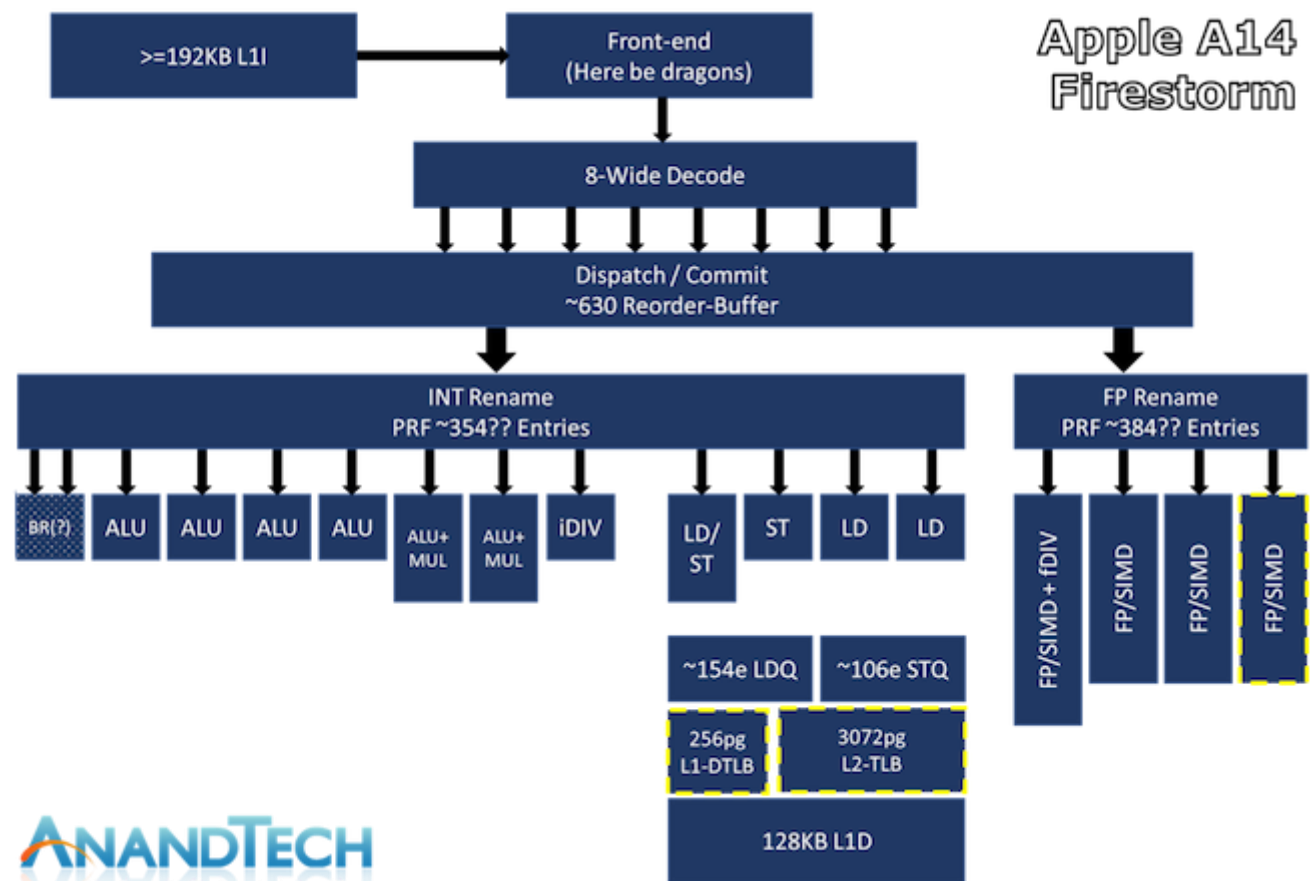


e.g., “Port 5 pressure” when code uses too much shuffle operations

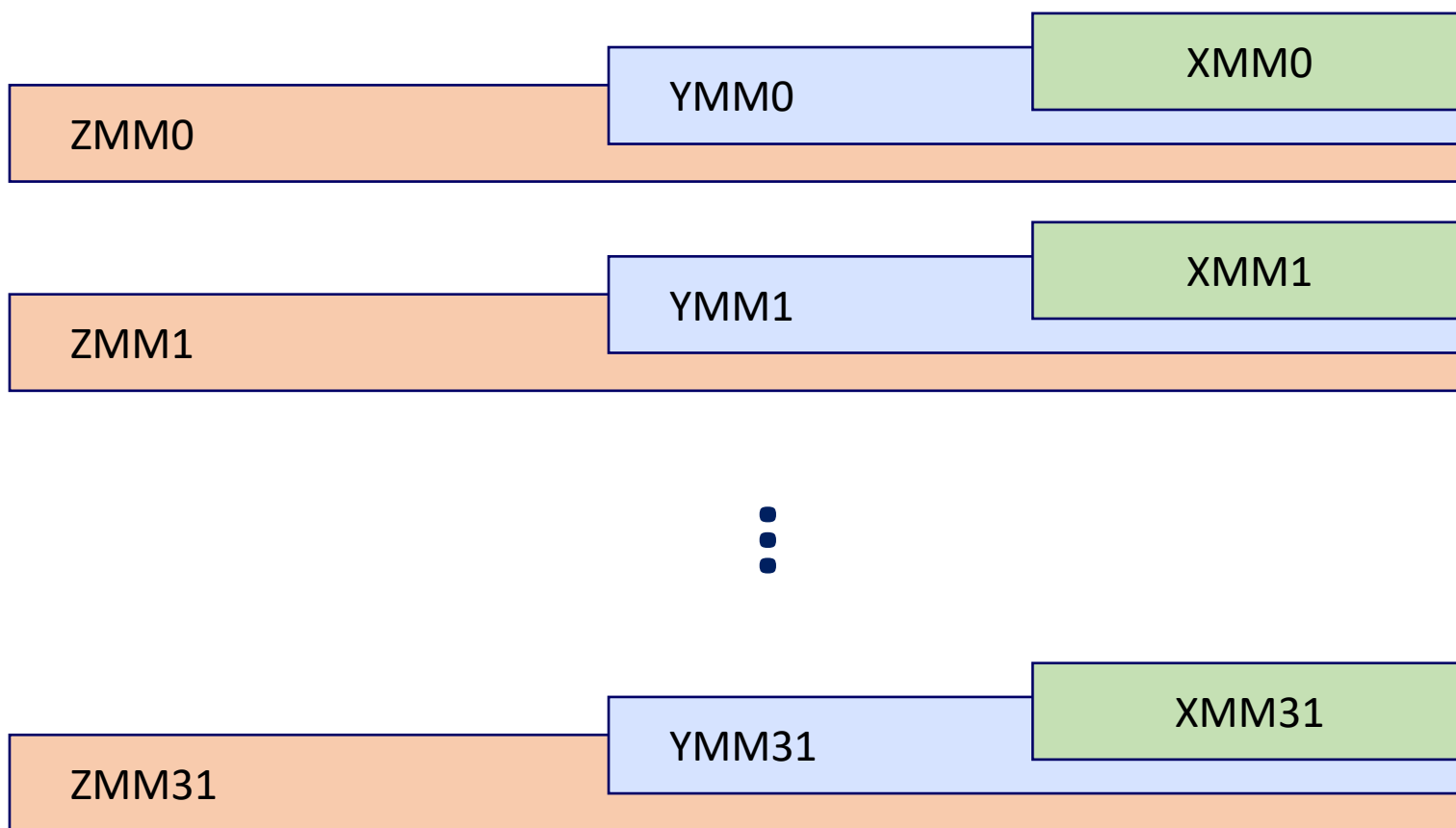
Skylake-X Microarchitecture (2019)



Apple M1 Microarchitecture (2020)



Intel SIMD Registers (AVX-512)



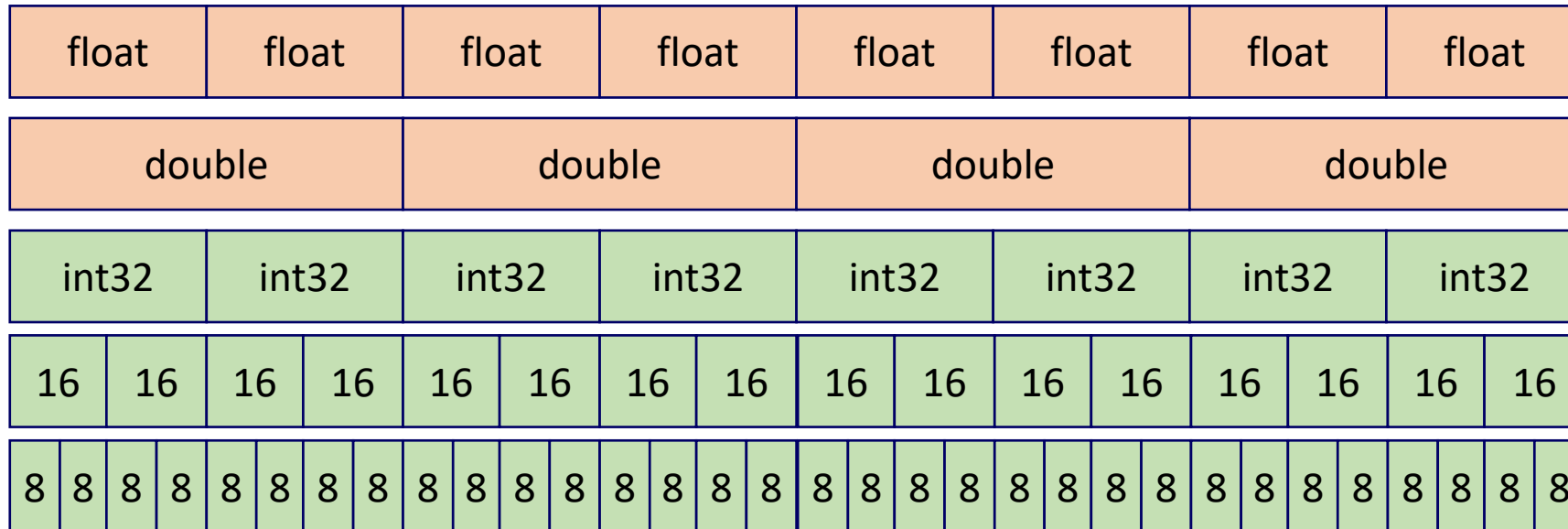
- ❑ XMM0 – XMM15
 - 128-bit registers
 - SSE
- ❑ YMM0 – YMM15
 - 256-bit registers
 - AVX, AVX2
- ❑ ZMM0 – ZMM31
 - 512-bit registers
 - AVX-512

SSE/AVX Data Types

255

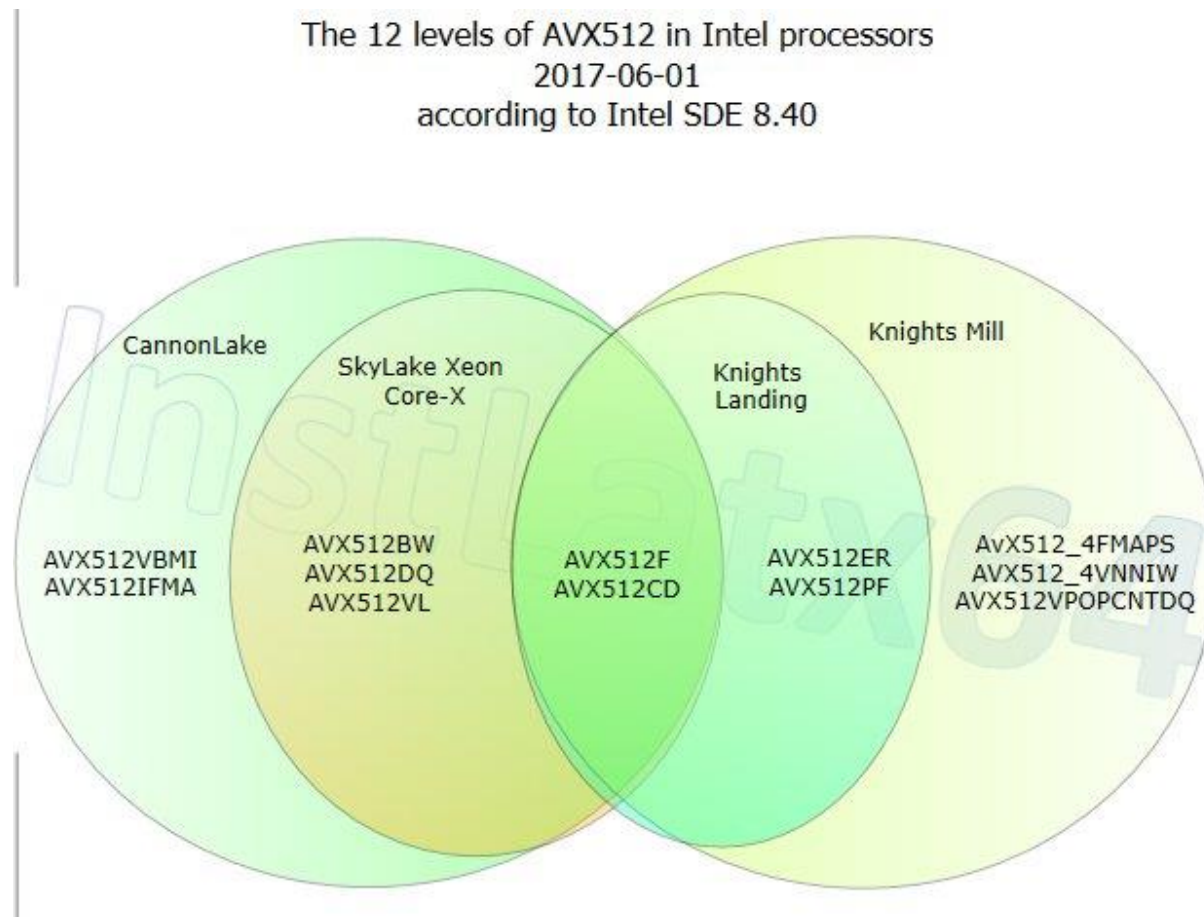
0

YMM0



Operation on
32 8-bit values
in one instruction!

Complexity of AVX-512



Aside: Do I Have SIMD Capabilities?

❑ `less /proc/cpuinfo`

```
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm con
stant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
erf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f
16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibp
b stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

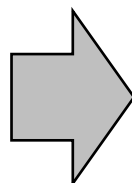
Processor Microarchitectural Effects on Power Efficiency

- ❑ The majority of power consumption of a CPU is not from the ALU
 - Cache management, data movement, decoding, and other infrastructure
 - Adding a few more ALUs should not impact power consumption
- ❑ Indeed, 4X performance via AVX does not add 4X power consumption
 - From i7 4770K measurements with matrix multiplication:
 - Idle: 40 W
 - Under load : 117 W
 - Under AVX load : 128 W

Compiler Automatic Vectorization

- ❑ In gcc, flags “-O3 -mavx -mavx2” attempts automatic vectorization
- ❑ Works pretty well for simple loops

```
int a[256], b[256], c[256];  
void foo () {  
    for (int i=0; i<256; i++) a[i] = b[i] * c[i];  
}
```



.L2:

```
vmovdqa xmm1, XMMWORD PTR b[rax]  
add     rax, 16  
vpmulld xmm0, xmm1, XMMWORD PTR c[rax-16]  
vmovaps XMMWORD PTR a[rax-16], xmm0  
cmp     rax, 1024  
jne     .L2
```

Generated using GCC explorer: <https://gcc.godbolt.org/>

- ❑ But not for anything complex
 - E.g., naïve bubblesort code not parallelized at all

Intel SIMD Intrinsics

- ❑ Use C functions instead of inline assembly to call AVX instructions
- ❑ Compiler manages registers, etc
- ❑ Intel Intrinsics Guide
 - <https://software.intel.com/sites/landingpage/IntrinsicsGuide>
 - One of my most-visited pages...

e.g.,

```
__m256 a, b, c;
```

```
__m256 d = _mm256_fmadd_ps(a, b, c); // d[i] = a[i]*b[i]+c[i] for i = 0 ...7
```


Data Types in AVX/AVX2

Type	Description
__m128	128-bit vector containing 4 floats
__m128d	128-bit vector containing 2 doubles
__m128i	128-bit vector containing integers
__m256	256-bit vector containing 8 floats
__m256d	256-bit vector containing 4 doubles
__m256i	256-bit vector containing integers

1~16 signed/unsigned integers

1~32 signed/unsigned integers

__m512 variants also for AVX-512

Intrinsic Naming Convention

□ `_mm<width>_[function]_[type]`

- E.g., `_mm256_fmadd_ps` :
perform `fmadd` (fused multiply-add) on
`256 bits` of
`packed single-precision` floating point values (8 of them)

Width	Prefix
128	<code>_mm_</code>
256	<code>_mm256_</code>
512	<code>_mm512_</code>

Type	Postfix
Single precision	<code>_ps</code>
Double precision	<code>_pd</code>
Packed signed integer	<code>_epiNNN</code> (e.g., <code>epi256</code>)
Packed unsigned integer	<code>_epuNNN</code> (e.g., <code>epu256</code>)
Scalar integer	<code>_siNNN</code> (e.g., <code>si256</code>)

Not all permutations exist! Check guide

Load/Store/Initialization Operations

❑ Initialization

- `_mm256_setzero_ps/pd/epi32/...`
- `_mm256_set_...`
- ...

❑ Load/Store : Variants for addresses aligned/unaligned by 256-bit

- `_mm256_load_...` `_mm256_loadu_...`
- `_mm256_store_...` `_mm256_storeu_...`

❑ And many more! (Masked read/write, strided reads, etc...)

e.g.,

```
__mm256d t = _mm256_load_pd(double const * mem); // loads 4 double values from mem to t  
__mm256i v = _mm256_set_epi32(h,g,f,e,d,c,b,a); // loads 8 integer values to v
```

Vertical Vector Instructions

❑ Add/Subtract/Multiply

- `_mm256_add/sub/mul/div_ps/pd/epi`
 - Mul only supported for `epi32/epu32/ps/pd`
 - Div only supported for `ps/pd`
 - Consult the guide!

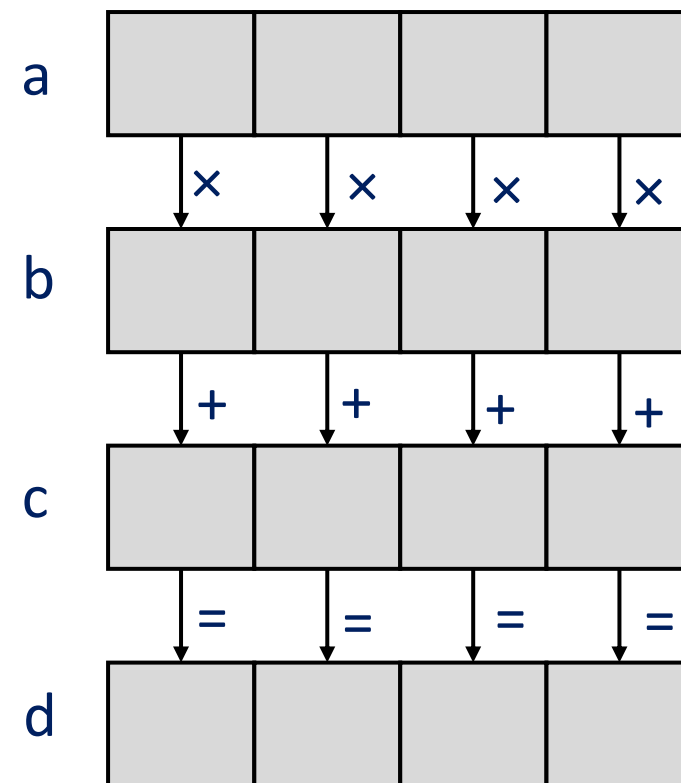
❑ Max/Min/GreaterThan/Equals

❑ Sqrt, Reciprocal, Shift, etc...

❑ FMA (Fused Multiply-Add)

- $(a*b)+c$, $-(a*b)-c$, $-(a*b)+c$, and other permutations!
- Consult the guide!

❑ ...



```
__m256 a, b, c;
```

```
__m256 d = _mm256_fmadd_pd(a, b, c);
```

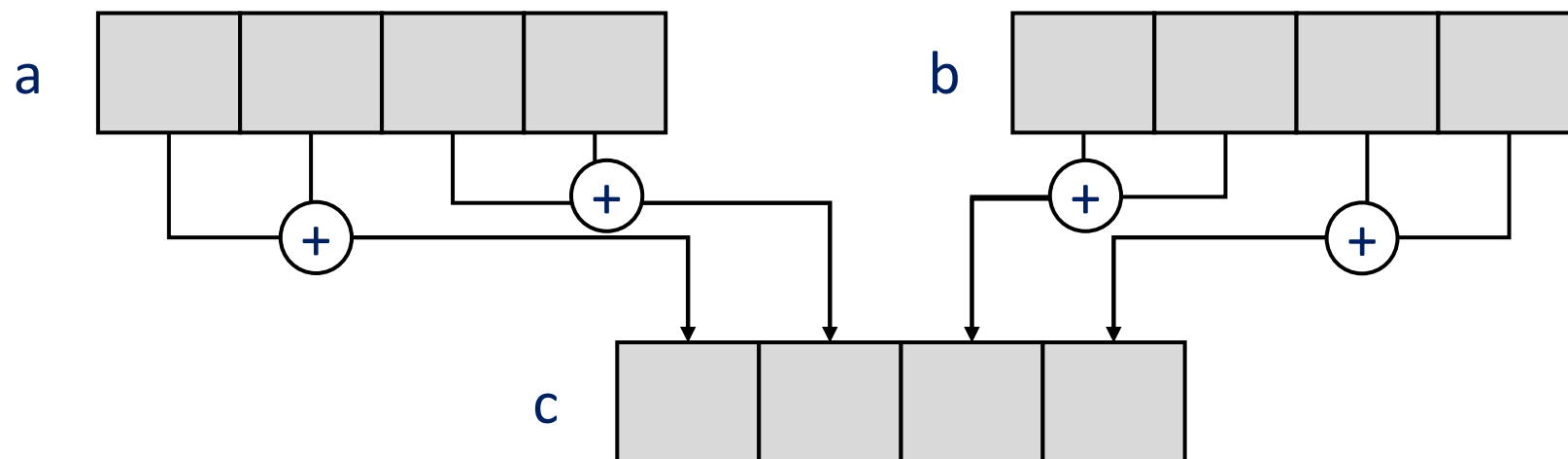
Integer Multiplication Caveat

- ❑ Integer multiplication of two N bit values require 2N bits
- ❑ E.g., `__mm256_mul_epi32` and `__mm256_mul_epu32`
 - Only use the lower 4 32 bit values
 - Result has 4 64 bit values
- ❑ E.g., `__mm256_mullo_epi32` and `__mm256_mullo_epu32`
 - Uses all 8 32 bit values
 - Result has 8 truncated 32 bit values
- ❑ And more options!
 - Consult the guide...

Horizontal Vector Instructions

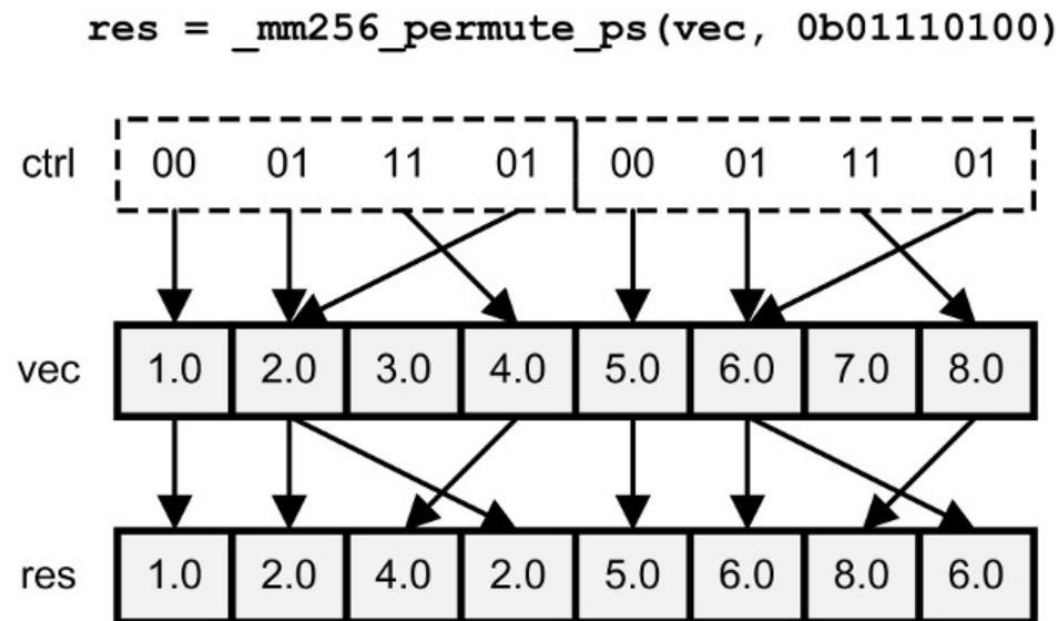
□ Horizontal add/subtraction

- Adds adjacent pairs of values
- E.g., `__m256d _mm256_hadd_pd (__m256d a, __m256d b)`



Shuffling/Permutation

- ❑ Within 128-bit lanes
 - `_mm256_shuffle_ps/pd/...` (a,b, imm8)
 - `_mm256_permute_ps/pd`
 - `_mm256_permutevar_ps/...`
- ❑ Across 128-bit lanes
 - `_mm256_permute2x128/4x64` : Uses 8 bit control
 - `_mm256_permutevar8x32/...` : Uses 256 bit control
- ❑ Not all type permutations exist for each type, but variables can be cast back and forth between types

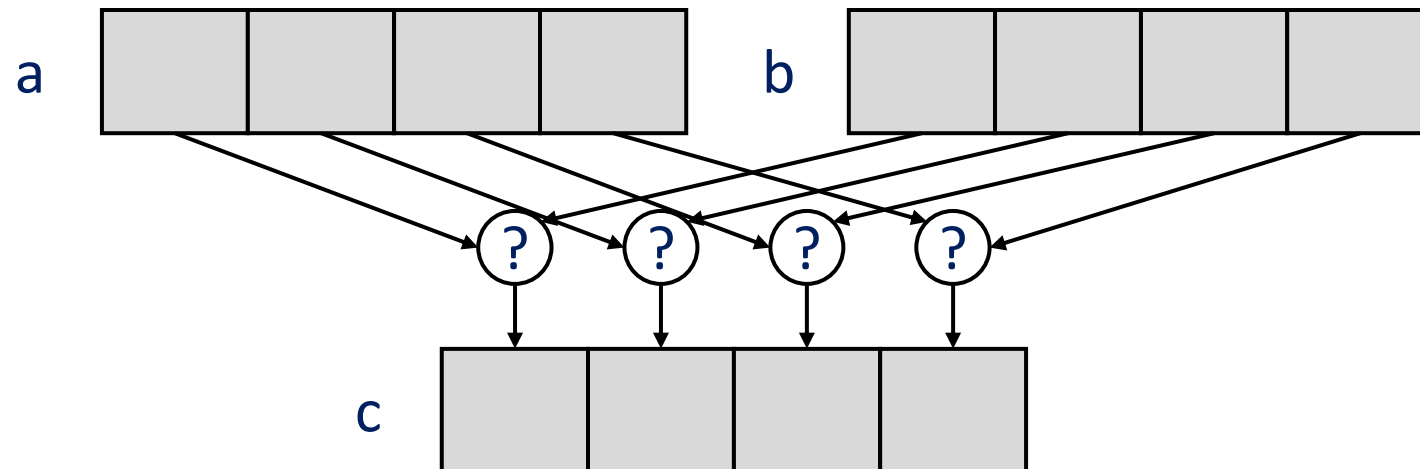


Matt Scarpino, "Crunching Numbers with AVX and AVX2," 2016

Blend

□ Merges two vectors using a control

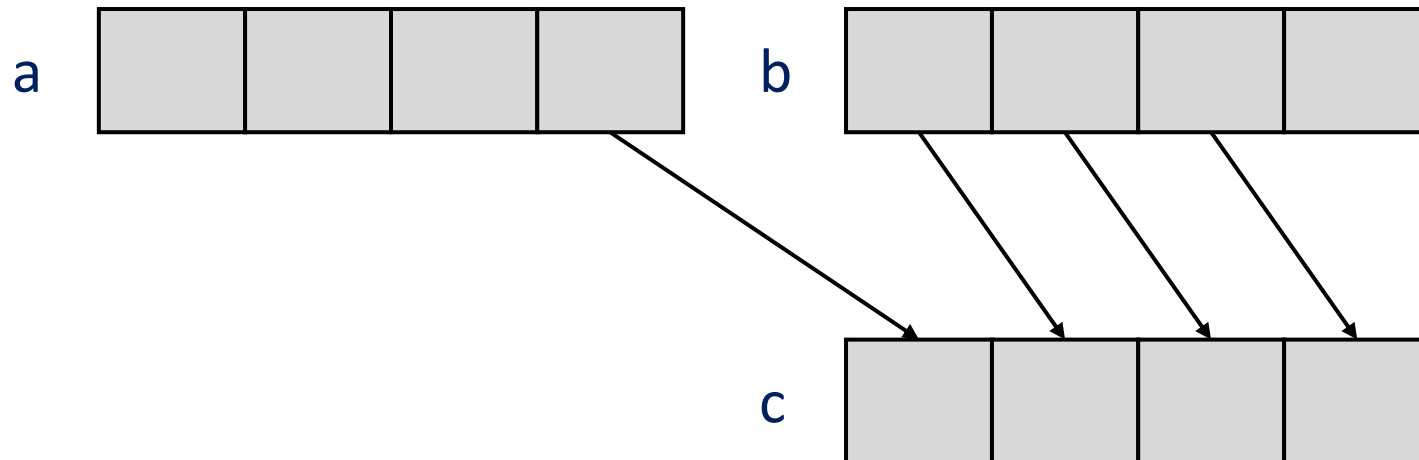
- `_mm256_blend_...` : Uses 8 bit control
 - e.g., `_mm256_blend_epi32`
- `_mm256_blendv_...` : Uses 256 bit control
 - e.g., `_mm256_blendv_epi8`



Alignr

- Right-shifts concatenated value of two registers, by byte
 - Often used to implement circular shift by using two same register inputs
 - `_mm256_alignr_epi8 (a, b, count)`

Example of 64-bit values being shifted by 8



Helper Instructions

❑ Cast

- `__mm256i <-> __mm256`, etc...
- Syntactic sugar `--` does not spend cycles

❑ Convert

- 4 floats `<->` 4 doubles, etc...

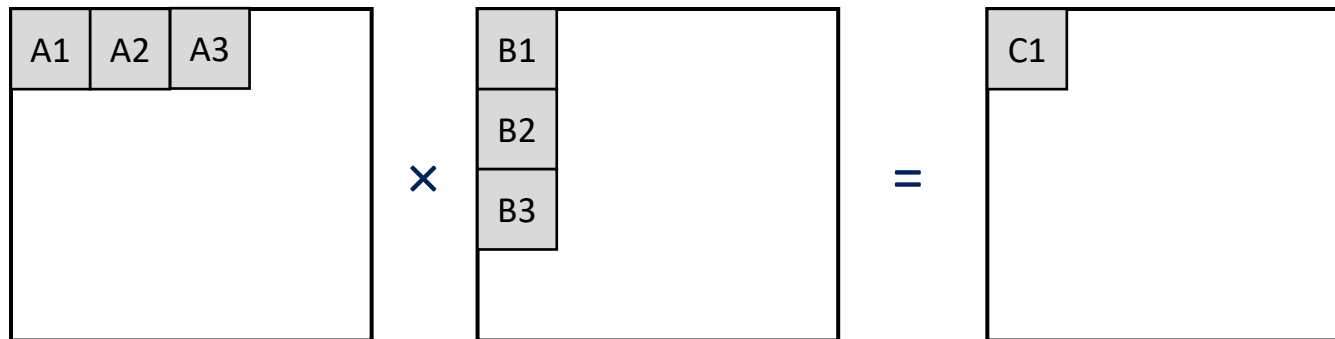
❑ Movemask

- `__mm256 mask to -> int imm8`

❑ And many more...

Our Current State Of Matrix Multiply: Blocked Multiplication

- ❑ Performance is best when working set fits into cache
 - But as shown, even 2048×2048 doesn't fit in cache
 - $\rightarrow 2048 * 2048 * 2048$ elements read from memory for matrix B
- ❑ Solution: Divide and conquer! – Blocked matrix multiply
 - For block size $32 \times 32 \rightarrow 2048 * 2048 * (2048/32)$ reads



$$C1 \text{ sub-matrix} = A1 \times B1 + A2 \times B2 + A3 \times B3 \dots$$

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked (32)	7.35	8.60

Bottlenecked by computation

Bottlenecked by memory

Bottlenecked by processor

Bottlenecked by memory (Not scaling!)

- ❑ AVX Transposed reading from DRAM at 14.55 GB/s
 - $2048^3 * 4 \text{ (Bytes)} / 2.20 \text{ (s)} = 14.55 \text{ GB/s}$
 - 1x DDR4 2400 MHz on machine -> 18.75 GB/s peak
 - Pretty close! Considering DRAM also used for other things (OS, etc)
- ❑ Multithreaded getting 32 GB/s effective bandwidth
 - Cache effects with small chunks

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked (32)	7.35	8.60
AVX Transposed	2.20	28.72
Blocked (32) AVX	1.50	42.13
4 Thread Blocked (32) AVX	1.09	57.97

- ❑ Using FMA SIMD, Cache-Oblivious AVX gets 19 GFLOPS
 - Theoretical peak is 3 GHz x 8 way SIMD == 24 GFLOPS... Close!

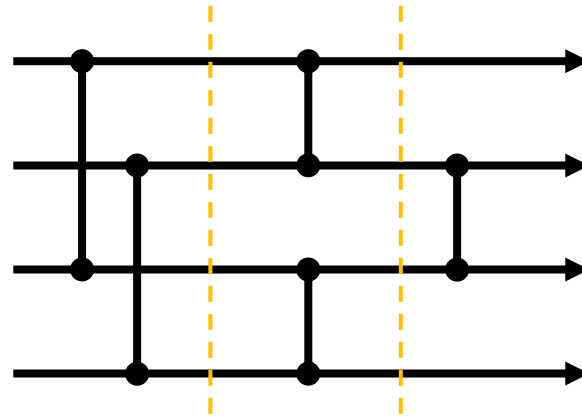
140x performance increase compared to the baseline!

Case Study: Sorting

- ❑ Important, fundamental application!
- ❑ Can be parallelized via divide-and-conquer
- ❑ How can SIMD help?

Reminder: Sorting Network

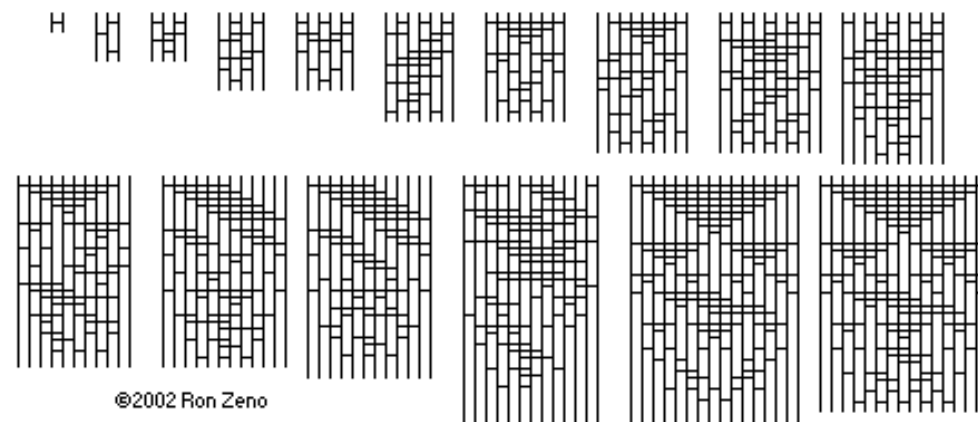
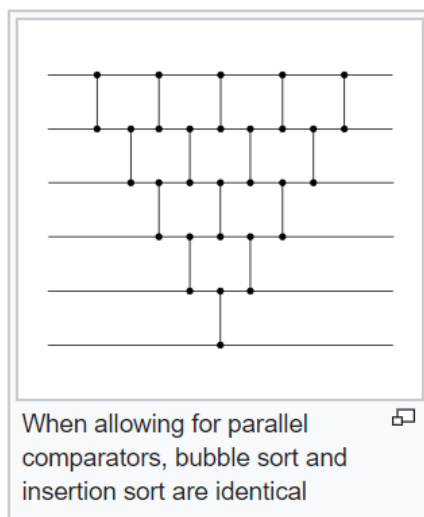
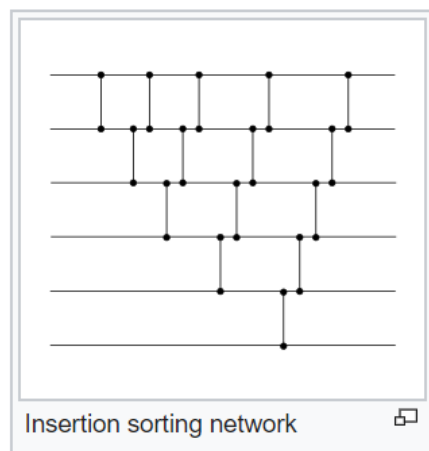
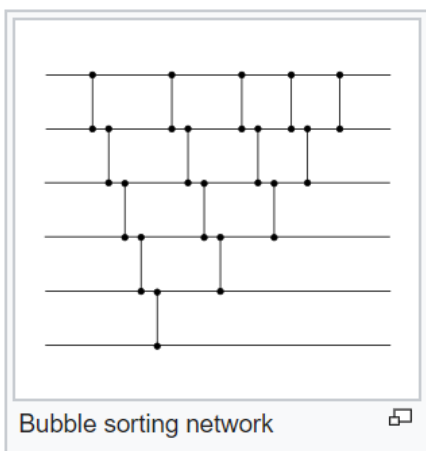
- ❑ Network structure for sorting fixed number of values
- ❑ Type of a “comparator network”
 - comparators perform compare-and-swap
- ❑ Easily pipelined and parallelized



Example 4-element sorting network
5 comparators, 3 cycle pipelined

Reminder: Sorting Network

- Simple to generate correct sorting networks, but optimal structures are not well-known

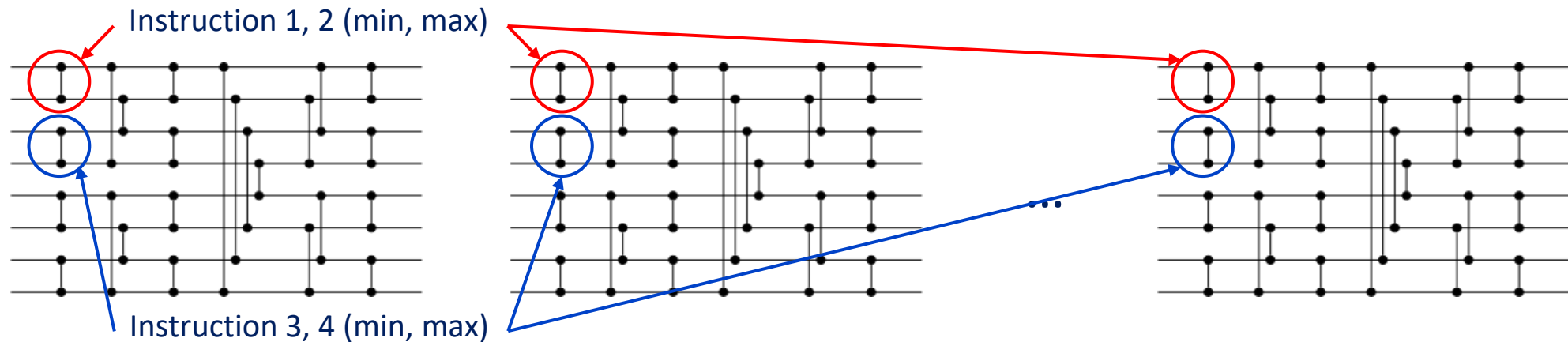


Source: Wikipedia (Sorting Network)

Some known optimal sorting networks

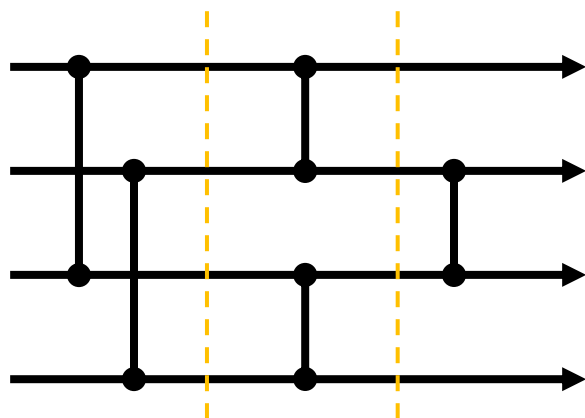
SIMD And Sorting Networks

- ❑ Typically, we are sorting more than one set of tuples
 - If we have multiple tasks, we can have task-level parallelism – Optimized networks!
 - Sort multiple tuples at the same time
- ❑ We first need to transpose the 8 8-element variables
 - Each variable has a value for each sorting network instance
 - Non-SIMD works, or a string of unpackhi/unpacklo/blend



SIMD And Sorting Networks

- ❑ Some SIMD instructions have high throughput, but high latency
 - Data dependency between two consecutive max instructions can take 8 cycles on Skylake
 - If each parallel stage has less than 4 operations, pipeline may stall
 - Solution: Interleave two sets of parallel 8-tuple sorting
 - In reality, min/max means even for 4-tuples, pipeline is still filled



`__m256d _mm256_max_pd (__m256d a, __m256d b)`

Performance

Source: Intrinsic guide

Architecture	Latency	Throughput (CPI)
Skylake	4	0.5
Broadwell	3	1
Haswell	3	1
Ivy Bridge	3	1

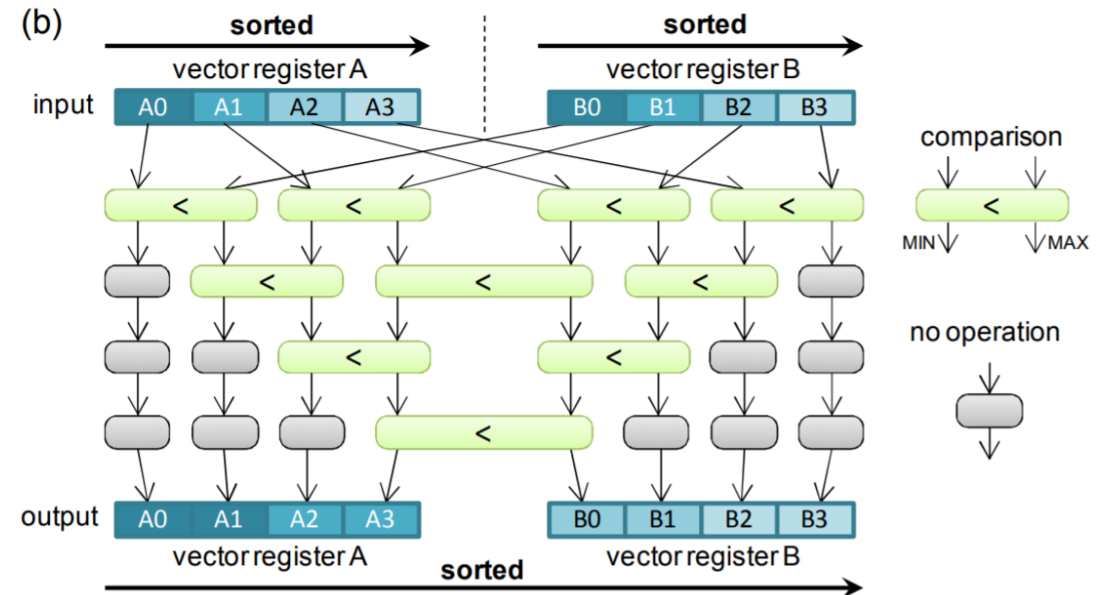
The Two Register Merge

□ Sort units of two pre-sorted registers, K elements

○ $\text{minv} = A, \text{maxv} = B$

○ // Repeat K times

- $\text{minv} = \min(\text{minv}, \text{maxv})$
- $\text{maxv} = \max(\text{minv}, \text{maxv})$
- // circular shift one value down
- $\text{minv} = \text{alignr}(\text{minv}, \text{minv}, \text{sizeof}(\text{int}))$



SIMD And Merge Sort

- ❑ Hierarchically merged sorted subsections
- ❑ Using the SIMD merger for sorting
 - `vector_merge` is the two-register sorter from before

```
aPos = bPos = outPos = 0;
vMin = va[aPos++];
vMax = vb[bPos++];
while (aPos < aEnd && bPos < bEnd) {
    /* merge vMin and vMax */
    vector_merge(vMin, vMax);

    /* store the smaller vector as output*/
    vMergedArray[outPos++] = vMin;

    /* load next vector and advance pointer */
    /* a[aPos*4] is first element of va[aPos] */
    /* and b[bPos*4] is that of vb[bPos] */
    if (a[aPos*4] < b[bPos*4])
        vMin = va[aPos++];
    else
        vMin = vb[bPos++];
}
```

Inoue et.al., "SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures," VLDB 2015

Topic Under Active Research!

- ❑ Papers being written about...
 - Architecture-optimized matrix transposition
 - Register-level sorting algorithm
 - Merge-sort
 - ... and more!
- ❑ Good find can accelerate your application kernel N_x

Questions?