Presented at Handmade Seattle
www.handmade-seattle.com/

SMACNI to AVX512 the life cycle of an instruction set

Why 29%* of x86 is my** fault***

Tom Forsyth
November 2019

* Dubious accounting methods detected!
** And a whole bunch of other people of course
*** #UD if CR4.OSXSAVE=0

Caveats

Focusing on the Larrabee-derived instruction set, not the device.

All this is from memory, so may not be 100% accurate!

Lots and lots of people involved – far too many to name.

(this is the Director's Cut Extended Edition of the slides)

Not even remotely an official Intel document/guide/spec sheet.

Levels of hardware

- User-level architecture
    - Register count & size
    - Instruction set, encoding
- OS-level architecture
    - Supervisor states and faulting
    - Virtual memory table structures
    - Hyperthreading, non-uniform memory arch
- Micro-architecture (“uarch”)
    - Cache size/ways/tags, branch prediction
    - Number & type of pipeline stages, latency
    - In/out-of-order, number & type of ALUs
- Design
    - Physical layout, timing, power & clock gating, clock trees, etc

Innovation in ISA (instruction set architecture)

A mix of both history and technology.

Design constraints drive uarch.

New uarch usually demands new ISA to drive it (e.g. wider SIMD).

ISA is always in the context of the mechanical function of the machine you are building it for (uarch), and the two interact tightly.

And sometimes design drives effects all the way up to arch.

Innovation becomes legacy!

Future machines with different design & uarch then need to cope with the “legacy” architecture that made sense for the old design.

This is not a unique problem for x86!

- Branch delay slot in MIPS
- Register stack/window in SPARC
- ARM predication and “free” shifter

Innovation becomes legacy!

Future machines with different design & uarch then need to cope with the “legacy” architecture that made sense for the old design.

This is not a unique problem for x86!

- Branch delay slot in MIPS
- Register stack/window in SPARC
- ARM predication and “free” shifter

Before rolling your eyes at an instruction or feature, consider it may have made perfect sense when it was invented, and that reason may **never** have been visible to you as a programmer.

Pixomatic (~2004)

- Software rast by Michael Abrash & Mike Sartain, RAD Game Tools
  - Standard MMX/SSE, JIT-compiled from DX7-style render states
  - 2 textures, 3 blend stages
  - All integer shading
- Planning Pixomatic 2
  - Wanted FMA instruction in x86
  - Talked to Dean Macri of Intel at GDC…

SMCA (~2005)

- A large array of simple, power-efficient x86 cores
  - SMCA = “Symmetric Multi-Core Architecture”
- Concept from Doug Carmean and Eric Sprangle of Intel
  - Original idea from ~2003
- Assumed (correctly!) that future would be limited by power, not area
- But where do they find “embarrassingly parallel” workloads to give it?
  - GPGPU and/or multicore wasn’t a common “thing” yet

SMCA (~2005)

- A large array of simple, power-efficient x86 cores
  - SMCA = “Symmetric Multi-Core Architecture”
- Concept from Doug Carmean and Eric Sprangle of Intel
  - Original idea from ~2003
- Assumed (correctly!) that future would be limited by power, not area
- But where do they find “embarrassingly parallel” workloads to give it?
  - GPGPU and/or multicore wasn’t a common “thing” yet
- Answer – graphics?
  - Michael Abrash + Mike Sartain started on the “fixed function” pipeline
  - I started writing a DX shader -> SSE shader compiler

SMCA New Instructions

- Quickly realized that just adding FMA to SSE wasn’t enough
- 128 bits wide was inefficient – not enough FMA per core
- Sod it – we’re making a whole new ISA – “SMCA New Instructions”
- BUT – still has to be x86-like
    - Remember job #1 is general-purpose computing – graphics is just a workload
    - C, Fortran, etc, (not just shaders)
    - Run FreeBSD/Linux and multitasking
    - Virtual memory, page faults, etc
    - User/supervisor levels
    - x86 memory ordering model

SMCANI early decisions (~2007)

- FMA is obviously good
- As wide as possible – couldn’t build 1024 bits, so 512 it was
  - 16 lanes of float32
  - Also matched x86 cache-line size
- “Ternary” encoding – needed for FMA, but also generally useful
  - Removes a lot of extra copy inst, compared to SSE-style destructive binary
- Load-op: vaddps v0, v1, [rax]
  - Used by approx. 50% of maths instructions
  - Removes a lot of separate load instructions

How to develop an ISA

- Gather shader workloads from games
- Compile to SMCANI with compiler
    - Add new instruction or change architecture
    - Change compiler to use new thing
    - Run through simulators to gauge performance & power
    - Accept/reject new thing
    - Iterate like crazy
- Hugely powerful
    - Typically managed a week for a new architectural feature
    - A new instruction was a day
    - Tried a massive number of features and combos
    - Lots of “interesting ideas” the compiler couldn’t deal with – rejected!
- This also informs the design of the surrounding “fixed function” pipe

SMCA core choice

- Which core do we start with?
  - Both need major surgery to support 512-bit SIMD units + 4 threads
- P54C – version of the original Pentium
  - Last Intel in-order core
  - Needs to be expanded to 64-bit
  - No existing MMX/SSE
  - But… Ed Grochowski
- Bonnell (Atom 1)
  - New, modern x86 ISA, 64-bit, already has MMX/SSE
  - But that team was heads-down trying to ship
- P54C was judged the lowest risk (in retrospect – correct decision)

P54C pairing and “free” memory

- P54C pairing: two decode+execute pipes
  - “Fat” U pipe can execute any instruction. The SIMD ALU hangs off this pipe
  - “Thin” V pipe can execute scalar instructions, and SIMD store
- Compiler high goals
  - Keep the U pipe full of SIMD math instructions all the time
  - Use the V pipe for “life support” scalar instructions and vector stores
    - Address computation
    - Loop counters
    - Branches
    - Vector stores
  - Load-op on U and store on V means memory is often “free”

P54C details shaped the ISA

- Example of microarchitecture driving architecture and ISA
  - Dual-issue fat+thin pipes
  - Requires load-op in the ISA to support it
  - Vector stores remain cheap
- Before committing to this, we HAD to prove the compiler can cope
  - And indeed it could
  - Lots of other interesting ideas rejected because compiler couldn’t cope
  - When designing an ISA, write the compiler first!
- Looking forwards, these limits help all architectures
  - But be careful of painting yourself into a corner when the uarch changes!

P54C pairing and load-op

	Pipe	RF reads	RF writes	Total per clock
vmadd v0, v1 , v2	U	v0, v1, v2	v0	3R, 2W
vload v3, [rax]	V		v3	
vmadd v0, v4, v3	U	v0, v4, v3	v0	4R, 1W
vstore [rbx], v5	V	v5		

Req: 4R, 2W

P54C pairing and load-op

	Pipe	RF reads	RF writes	Total per clock
vmadd v0, v1 , v2	U	v0, v1, v2	v0	3R, 2W
vload v3, [rax]	V		v3	
vmadd v0, v4, v3	U	v0, v4, v3	v0	4R, 1W
vstore [rbx], v5	V	v5		

Req: 4R, 2W

Load-op

	Pipe	RF reads	RF writes	Total per clock
vmadd v0, v1 , v2	U	v0, v1, v2	v0	3R, 1W
vmadd v0, v4, [rax]	U	v0, v4	v0	3R, 1W
vstore [rbx], v5	V	v5		

Req: 3R, 1W

P54C pairing and load-op

	Pipe	RF reads	RF writes	Total per clock
vmadd v0, v1 , v2	U	v0, v1, v2	v0	3R, 2W
vload v3, [rax]	V		v3	
vmadd v0, v4, v3	U	v0, v4, v3	v0	4R, 1W
vstore [rbx], v5	V	v5		

Req: 4R, 2W

Load-op

	Pipe	RF reads	RF writes	Total per clock
vmadd v0, v1 , v2	U	v0, v1, v2	v0	3R, 1W
vmadd v0, v4, [rax]	U	v0, v4	v0	3R, 1W
vstore [rbx], v5	V	v5		

Req: 3R, 1W

Significant reduction in area and power for the register file

Encoding

- Three different and sadly incompatible encodings
- Constant tension between the needs of a simple in-order core and a complex out-of-order Big Core
- KNF: D6 (SALC) and 62 (BOUND) prefixes – only free in 64-bit mode
- KNC: 62 + 3 byte “MVEX” prefix used for all instructions
- KNL/AVX512: MVEX tweaked to become current “EVEX” prefix

Encoding

- Three different and sadly incompatible encodings
- Constant tension between the needs of a simple in-order core and a complex out-of-order Big Core
- KNF: D6 (SALC) and 62 (BOUND) prefixes – only free in 64-bit mode
- KNC: 62 + 3 byte “MVEX” prefix used for all instructions
- KNL/AVX512: MVEX tweaked to become current “EVEX” prefix
- Convergence was extremely painful
  - The REAL cost of x86 legacy is not gates, it’s lots and lots of meetings
  - Cunning-but-complex encoding tricks used to avoid existing x86 instructions

Memory faults

- From a programming viewpoint they:
    - Are esoteric
    - Get in your way
    - Never happen
    - Why do I even care?

Memory faults

- From a programming viewpoint they:
  - Are esoteric
  - Get in your way
  - Never happen
  - Why do I even care?
- From an OS viewpoint they:
  - Are how virtual and demand-paged memory works at all
  - Happen constantly
  - Are incredibly important to get right
  - Very subtle – small changes in HW behavior can cause deadlocks/livelocks

Memory faults

- From a programming viewpoint they:
  - Are esoteric
  - Get in your way
  - Never happen
  - Why do I even care?
- From an OS viewpoint they:
  - Are how virtual and demand-paged memory works at all
  - Happen constantly
  - Are incredibly important to get right
  - Very subtle – small changes in HW behavior can cause deadlocks/livelocks
  - **P54C WASN'T BROKEN, SO DON'T TRY TO FIX IT!**

Memory faults

- P54C pipeline was simple, small, short, but simple:
  - Faults must be taken immediately after TLB lookup
  - Go/nogo point is at the same time as the cache read
  - Once an instruction has passed that point, CANNOT fail or fault or stop
- Therefore, must take/eliminate the fault before any “work”
  - Can’t do 4 clocks of work THEN find a problem and take a fault
  - Entire vector-maths pipeline is after go/nogo point
- Cannot look up TLB more than once
  - Otherwise you need slow microcode or sequencing
  - Therefore can only hit one cache line per instruction
  - This affects a lot of the user-level instruction set

Memory faults

vaddps v0{k1}, v1, [rax+128]

- Stages of the P54C virtual memory pipeline:
- Compute (rax+128) – “linear address”
- Look up in TLB to find “physical address” and permissions
- If TLB hit:

Memory faults

vaddps v0{k1}, v1, [rax+128]

- Stages of the P54C virtual memory pipeline:
- Compute (rax+128) – “linear address”
- Look up in TLB to find “physical address” and permissions
- If TLB hit:
- Check permissions, look up physical address in cache
- Do vector addition
- Write result to register file
- Retire operation

Memory faults

vaddps v0{k1}, v1, [rax+128]

- Stages of the P54C virtual memory pipeline:
- Compute (rax+128) – “linear address”
- Look up in TLB to find “physical address” and permissions
- If TLB miss:

Memory faults

vaddps v0{k1}, v1, [rax+128]

- Stages of the P54C virtual memory pipeline:
- Compute (rax+128) – “linear address”
- Look up in TLB to find “physical address” and permissions
- If TLB miss:
- Put thread to sleep
- Walk page tables to resolve virtual-to-physical mapping
- If no valid entry in page table, signal page fault to OS
- Otherwise, fill in TLB entry, retry instruction

Other early decisions – effort/risk reduction

- SMCANI is how we do all floating-point maths
    - Do not add MMX, SSE
    - Deprecate x87 (allow it to stay slow)
- 64-bit address mode only
    - Real, protected and virtual modes work for boot & testing, but don’t have SMACNI
- No native 8+16 bit SIMD support
    - Upconvert to 32 bit on load, downconvert on store
    - Built-in conversions to/from float16/11/10, int8/int16, 10:10:10:2, other gfx formats
- No native SIMD int64 support – use add-with-carry instructions instead
- Limited native float64 support
    - Would be improved in KNC

Other early decisions – effort/risk reduction

- Unaligned memory access forbidden
  - Was 1/3rd speed in P54C already – would never use it deliberately!
  - HW to support it was prohibitive
  - Nasty corner cases e.g. spanning cache lines
- Denormals read as zero (DAZ) and flushed to zero (FTZ)
- FP rounding fixed at Round-To-Nearest,Even
- Floating-point exceptions just set sticky bits, do not cause faults
- Transcendental functions composed from code sequences
  - P54C ran microcode quite slowly
  - Typically a low-precision lookup table, then Newton-Raphson iteration

SMCA to Larrabee

- Around 2005, official codename chosen
  - Codenames are landmarks – Larrabee is a mountain & a state park
  - Got funding & team to make a product, not just a research project
- Steal with pride!
  - Core from P54C
  - L2$ from a Pentium4
  - Texture sampler from Gen
  - Ring bus + coherency from a research project
  - All would be heavily modified by the end
- SIGGRAPH 2008 paper was the main public announcement

Agent Larrabee, Get Smart

Larry the hot-air beeloon

Predication masks

vaddps v0, v1, v2	All 16 lanes do operation & store to v0
vaddps v0{k1}, v1, v2	Some v0 values preserved, ALU lanes inactive

- Predication is crucial for efficient wide SIMD, used by all GPUs
  - Used for if{} else{} clauses, loops, etc
  - Deactivate unused ALU lanes to save power
  - Do not signal exceptions/flags for disabled lanes
  - Avoids extra instructions to merge results
- Novel (and contentious!) feature in the CPU world

Predication masks

vaddps v0{}, v1, v2	Implicit masking like GPUs?
vaddps v0{rax}, v1, v2	Use existing registers?
vaddps v0{k1}, v1, v2	Create new registers?

- Implicit registers have caused enough trouble in x86!
  - Direction flag, rounding mode settings, etc
- Wanted to use rax/rbx/rcx/rdx for elegance
  - But scalar and vector units are far apart – worried about timing constraints
- So we created k0-k7 and a scalar mini-ISA for them

Register lane swizzles

- Expectation was that there would be lots of SSE code to be ported
- SSE code tends to have lots of lane-swizzle instructions
- Put swizzles “for free” for one input of any maths instruction
  - vaddps v0, v1, {cdab}v2 – swaps adjacent lanes of v2

Register lane swizzles

- Expectation was that there would be lots of SSE code to be ported
- SSE code tends to have lots of lane-swizzle instructions
- Put swizzles "for free" for one input of any maths instruction
  - vaddps v0, v1, {cdab}v2   – swaps adjacent lanes of v2
- **BUT NOBODY EVER DID THIS**
  - People took one look at SMACNI/LRBNI and cheered
  - Threw away existing SSE code and rewrote from scratch
- In retrospect, free swizzles were a mistake – big area and timing cost
  - Removed by KNL, made into standalone shuffle instructions – a good thing

Memory swizzles

- Distinct from reg->reg swizzles
  - Even though uses similar encoding position
- Totally different bit of hardware!
  - Already need the shuffle network on the way to/from memory to support standard x86 byte/word/dword reads/writes
- Used for gather/scatter
- Used for compress/expand
- Used for single-element broadcast-on-load (very common!)
- These remain an excellent idea

Floating-point-Multiply-Add combos

- Ideally would have 3 sources and separate destination
  - But didn't have space to encode 4 registers
- So the destination register must be one of the sources
  - Compiler discovered it's very rare to need an extra move to work around this
- One source can also come from memory using load-op
- Including sign flips, twelve combos in total:
  - A = ± A*B ± [mem]
  - A = ± A*[mem] ± B
  - A = ± B*[mem] ± A
- Encoded as 12 different instructions
  - FMA is such a common instruction, easily worth the encoding space

madd233

- Scale+bias (aka 2D interpolation) is a very common operation
    - y = A + B * x
    - Where A and B are the same for all SIMD lanes, and are in memory
- Can only read memory once per instruction
    - So would normally need at least two instructions, e.g. load+madd
    - But A and B are usually next to each other in memory
- Load [A,B] as a single 64-bit value
    - Do a “double broadcast” out to the FMA ALU

vmadd233ps v0, v1, [rax]

v0 = [rax+0] + [rax+32] * v1

FP fixup

- Microcode is slow, so recip/sqrt/etc done with instruction sequences
  - Typically a table lookup followed by one or more Newton-Raphson iterations

```
vrcpresps        v2, v0          e0 = 1 - x * approx_rcp(x)
vrcprefineps     v1, v2, v0      r1 = approx_rcp(x) + approx_rcp(x) * e0
vmsubr23c1ps     v2, v1, v0      e1 = 1 - x * r1
vmadd231ps       v1, v1, v2      y = r1 + r1 * e1
```

- But – how do we deal with problem inputs?
  - 1/0 = inf, 1/inf = 0, 1/nan = nan
  - Above code cannot produce the right results for these special inputs
- Solution
  - Categorize input, e.g. “this is +0”
  - Look up in table
  - Fix up result according to table, e.g. “set to +inf”

FP fixup

vfixupps v1, v0, 0x018842

- v1 has result of code sequence, v0 is original input
- The immediate value is 21 bits: 7-entry table, 3 bits each
- For each lane of v0, categorize as one of:
    - 0: -inf   1: <-0   2: -0
    - 5: +inf   4: >+0   3: +0
    - 6: nan
- Index into immediate, table entry says we set the lane in v1 to one of:
    - no change   NaN
    - -0   -MAXFLT   -inf
    - +0   +MAXFLT   +inf

Ternlogop

- Every ISA needs logical operations – and/or/xor/nor/etc
- A lot of possible instructions
  - They all need documenting
  - And testing
  - And encoding space
  - And you still never have the one you need
  - Ugh, that all sounds like hard work
- I remembered a trick from FPGAs
  - Use a programmable lookup table instead of fixed logic
- Like vfixupps, what if the lookup table was part of the instruction?

Ternlogop

vternlog v0, v1, v2, #imm8

- #imm8 is an 8-entry table, 1 bit per entry
- Take each bit in v0, v1, v2
- Forms a 3-bit number, which is an index 0-7
- Look up the value of that bit in #imm8
- Write that bit into v0

- Can do all 2-input and 3-input logical operations
  - Including super-useful things like v0 = (v0&(~v2)) | (v1&v2)
- Satisfyingly elegant solution
- And solved the original problem of… laziness!

Ternlogop

Note the genesis of this instruction:

1. Recip/sqrt/exp/log are complex multi-clock sequences
2. Microcode is slow on P54C
3. Multi-instruction sequences don’t produce 0/inf/NaN properly
4. Invent “fixup” instruction with immediate-as-lookup-table trick
5. Ternlogop uses this same trick

Without the P54C uarch “problem”, ternlogop might not exist!

Scale+round+convert

- Very common in graphics to convert float<->int with a pow-of-2 scale
- x87 and SSE use an implicit rounding mode register – BAD!
- Create an instruction that explicitly specifies everything – GOOD!

vcvtps2pi v0, v1, rd, 24

- v0 = (int)floor (v1 * (float)(1<<24))
- ps2pi = packed single precision to packed integer
- rn/ru/rd/rz = round nearest/up/down/zero
- clamp if out of range

Bit extract/insert

- When dealing with textures, there is often some odd addressing
    - Morton Order: xyxyxyxyxyxyxy
    - Nested rectangles: xxxxyyyyxxxxyyyy
    - Hybrid Morton/rectangles: xxxxyyyyxyxyxy

bitinterleave11 rax, rbx

- rax = interleave lower bits of rax and rbx

insertfield rax, rbx, #rot, #start, #end

- Insert (rbx<<rot) from bit (start) to bit (end) into rax

rotatefield rax, rbx, #rot, #start, #end

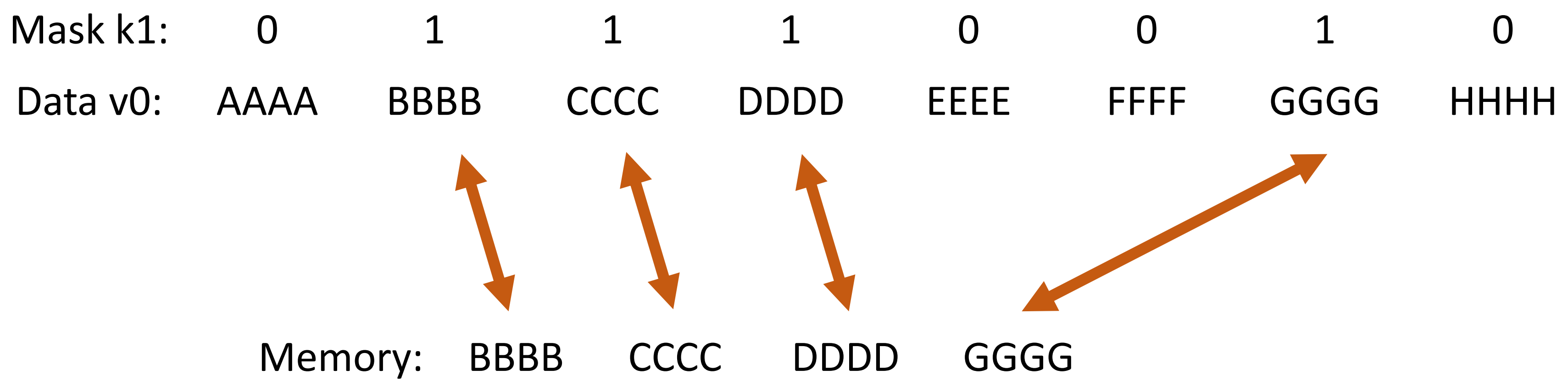
- Extract (rbx<<rot) bit (start) to bit (end) into rax

- The last two are equivalents of PowerPC instructions

Pack/unpack

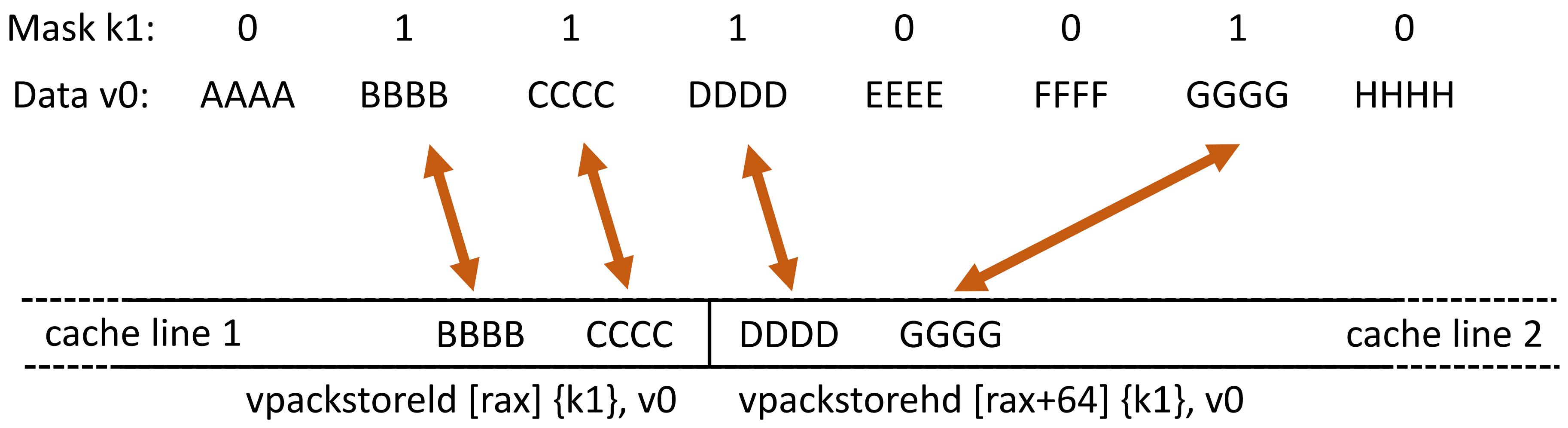
vpackstore [rax] {k1}, v0

- Takes elements from v0 enabled by k1, packs together and stores
- Also a loadunpack equivalent (later called “compress/uncompress”)
- Used to enqueue/dequeue sparse SIMD data


Mask k1: 0 1 1 1 0 0 1 0
Data v0: AAAA BBBB CCCC DDDD EEEE FFFF GGGG HHHH
Memory: BBBB CCCC DDDD GGGG

Pack/unpack

- Problem – the P54C can only do one cache line at a time
- Solution – “low” and “high” versions of the instruction
  - Each one can complete (or fault-then-complete) separately
- Using no write mask allows unaligned vector load/store


Mask k1: 0 1 1 1 0 0 1 0
Data v0: AAAA BBBB CCCC DDDD EEEE FFFF GGGG HHHH
cache line 1
BBBB
CCCC
DDDD
GGGG
cache line 2
vpackstoreld [rax] {k1}, v0
vpackstorehd [rax+64] {k1}, v0

Gather/scatter

vgather v0{k1}, [rax+v1]

- The fundamental vector-machine equivalent to scalar load/store
- Each lane of v0 gets loaded from a different address (rax+v1)
- Gather/scatter can access up to 16 cache lines & pages!
    - P54C doesn’t do that well
    - OS will FREAK if a single instruction hits more than 2 pages
    - Huge deadlock/livelock/forward-progress problems

Gather/scatter

vgather v0{k1}, [rax+v1]

- The fundamental vector-machine equivalent to scalar load/store
- Each lane of v0 gets loaded from a different address (rax+v1)
- Gather/scatter can access up to 16 cache lines & pages!
  - P54C doesn't do that well
  - OS will FREAK if a single instruction hits more than 2 pages
  - Huge deadlock/livelock/forward-progress problems
- How we going to solve this?
  - Brand new problem, never faced before, **everybody panic!**

Gather/scatter

vgather v0{k1}, [rax+v1]

- The fundamental vector-machine equivalent to scalar load/store
- Each lane of v0 gets loaded from a different address (rax+v1)
- Gather/scatter can access up to 16 cache lines & pages!
  - P54C doesn't do that well
  - OS will FREAK if a single instruction hits more than 2 pages
  - Huge deadlock/livelock/forward-progress problems
- How we going to solve this?
  - Brand new problem, never faced before, **everybody panic!**
  - Wait…

rep movsb

- Remember this old 8086 instruction?
- Single-instruction memcpy
- Copies (rcx) number of bytes from [rsi] to [rdi]
- …and doesn’t that hit an arbitrary number of cache lines?
  - So what’s their special magic? Can we steal it for gather/scatter?

rep movsb

- Irritating “side effect” of this instruction
  - rsi, rdi end up pointing past the end of the memory block
  - rcx ends up set to zero
  - Why did they do that? How is this useful to anyone?

rep movsb

- Irritating “side effect” of this instruction
  - rsi, rdi end up pointing past the end of the memory block
  - rcx ends up set to zero
  - Why did they do that? How is this useful to anyone?
- Think about what happens if it hits a mid-copy page fault
  - rsi/rdi point to the address that faulted
  - rcx has been decremented by the amount that successfully copied
  - OS handles fault (e.g. brings memory off disk)
  - …and just jumps back to the instruction
  - No “hidden state” required to resume the instruction where it faulted

rep movsb

- Irritating “side effect” of this instruction
  - rsi, rdi end up pointing past the end of the memory block
  - rcx ends up set to zero
  - Why did they do that? How is this useful to anyone?
- Think about what happens if it hits a mid-copy page fault
  - rsi/rdi point to the address that faulted
  - rcx has been decremented by the amount that successfully copied
  - OS handles fault (e.g. brings memory off disk)
  - …and just jumps back to the instruction
  - No “hidden state” required to resume the instruction where it faulted
  - Not just useful, but GENIUS!

Gather/scatter

vgather v0{k1}, [rax+v1]

- So back to gather/scatter
- We need some sort of “progress meter”
  - As it does its loads, it marks them as “done”
  - OS can handle a fault mid-process
  - Jump back to instruction
  - Instruction resumes where it left off, no hidden state

Gather/scatter

vgather v0{k1}, [rax+v1]

- So back to gather/scatter
- We need some sort of “progress meter”
  - As it does its loads, it marks them as “done”
  - OS can handle a fault mid-process
  - Jump back to instruction
  - Instruction resumes where it left off, no hidden state
- Solution – use the writemask “k1”
  - Clear bits in k1 as you go, indicating they are done
  - Instruction always ends with all bits in k1 clear

Gather/scatter

- Bravo! We added gather/scatter to ye olde P54C
- Genius work
- All home for tea & medals

Gather/scatter

- Bravo! We added gather/scatter to ye olde P54C
- Genius work
- All home for tea & medals
- A few minor problems…

Gather/scatter

- Bravo! We added gather/scatter to ye olde P54C
- Genius work
- All home for tea & medals
- A few minor problems…
  - Microcode/sequencing is slow on P54C
  - Have to figure out which of 16 addresses to do each cycle
  - Reading the **vector** registers is **after** TLB/fault/cache read
  - Not enough time to do vector calculations before the cache read
  - Reading 16 cache lines for every gather instruction is way too slow
  - What happens with a scatter if lanes go to the same address?

Gather/scatter

- As a result, gather/scatter took a huge amount of effort
    - Close interactions between design, architecture, compilers, OS SW
    - Massive effort on all four main architectures (KNF, KNC, KNL, Big Core)
    - Each new uarch faced new problems, new solutions
- KNF/KNC used “gather step” (KNC’s was significantly quicker)
- KNL, Big Core do it in microcode
    - They have high-speed microcode sequences, unlike KNF/KNC
- Big Core gets faster over time
    - Requires adding more AGU, TLB, cache ports
    - Already significantly faster than doing scalar load/stores

Gather/scatter

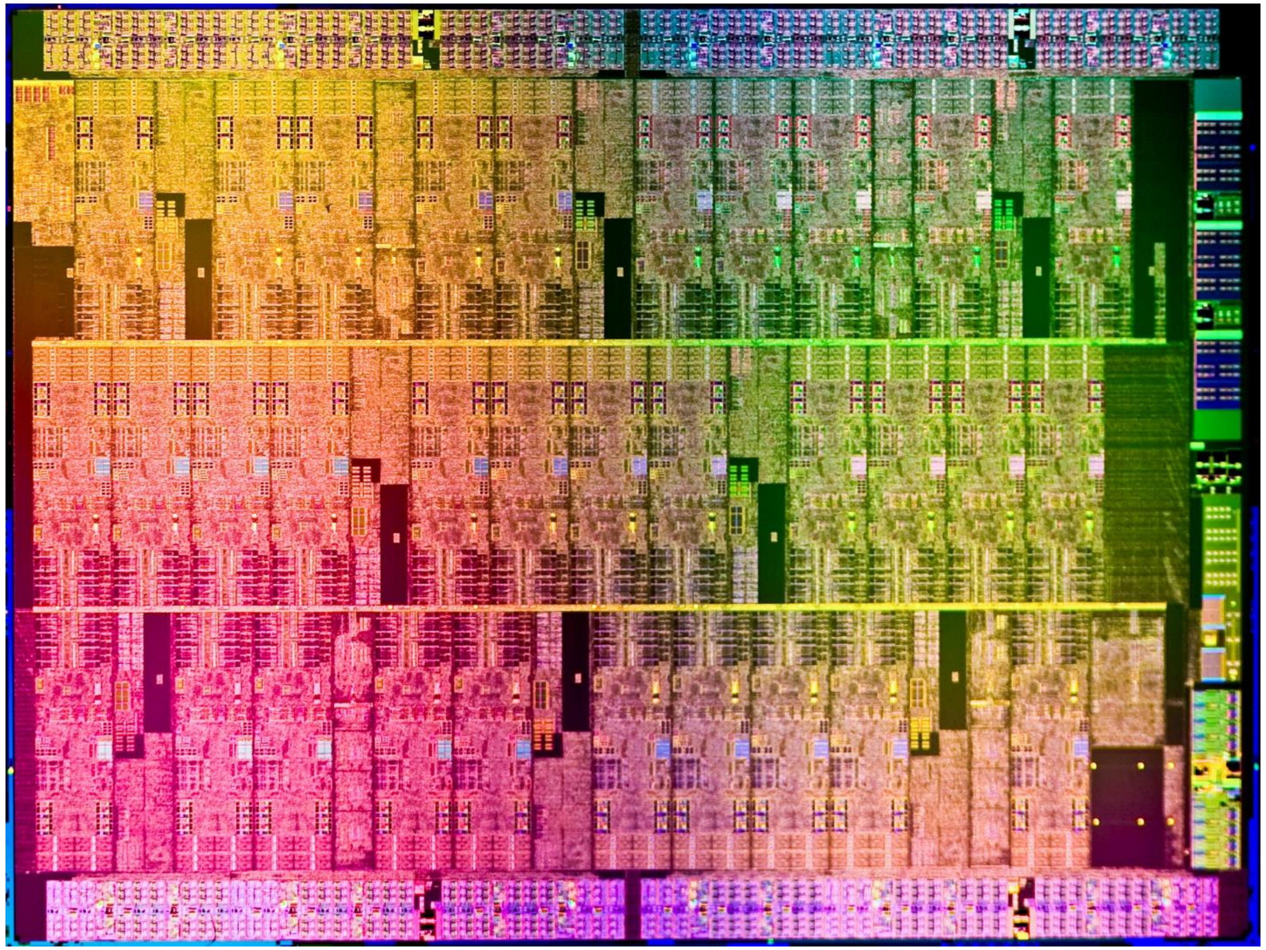
- For KNF + KNC the ugly solution was to run “gatherstep” in a loop:

```
loop_label:
   vgatherd v0{k1}, [rax+v1]
   vkortest k1, k1
   jnz loop_label
```

- Each time, vgatherd accesses at most a single cache line
  - Does address calc, TLB and cache check for first-enabled lane in v1
  - Will also fill in other lanes if they are in that same cache line
  - Very common in workloads - typically only run the loop 2-3 times
  - k1 gets cleared as lanes are finished, loop ends when k1 is all-zeros
- Unrolling the loop also works fine – executing extra gathers is benign

Larrabee 1 – Knights Ferry

- Knights Ferry
  - Shipped 2010, 45nm
  - 32 cores
    - Some disabled for yield
  - 8 texture units
  - HDMI/DVI scanout
  - 1.2GHz, 1.2TFLOPS
- “Aubrey Isle” PCIe card



Knights Ferry (KNF) - "Larrabee"
Aubrey Isle 32c/128t @1.2GHz
2GB GDDR5
45nm Process
c.2011
intel
intel

Larrabee 1 – Knights Ferry

- A-stepping: first working silicon
  - New architecture = a whole bunch of bugs
    - I$ broken, x87 was single-thread only, bit scan reverse broken
    - Fixed in “quick” B-stepping
  - L1$/L2$ interface had a fundamental flaw hurting memory performance
  - “Plan to throw one away; you will anyhow” – Fred Brooks
- D-stepping – the big fix
  - Significant re-engineering
  - Fix the L2$ speed path, misc other bugs
  - Add some new instructions (gather prefetch, delay)
- Not enough float64 performance, never made into a product
  - But Aubrey Isle used extensively as a SW development vehicle

Delay instruction

- When a thread mispredicts a branch or misses cache, it is put to sleep
- But we only discover this happens late in the pipeline
  - There will be other instructions from the same thread queued up
  - They need to be crushed into NOP instructions
  - But they still take time to execute & steal clocks from other threads
- Similar to existing PAUSE instruction
  - But delay also takes a number of clock cycles
- As soon as it is decoded, it stops picking instructions from that thread
  - Thread is then put to sleep for the given number of clocks
  - Pair with a mispredicted branch or a load/store that misses the cache a lot

Larrabee 2 – Knights Corner

- Knights Corner started immediately after KNF A-step
  - Focused more on HPC market
  - Name changed to “Xeon Phi” to reflect this
- Switch process 45nm -> 22nm
- Faster in every way
  - 4x float64 performance for HPC market
  - More gather/scatter performance
  - Bigger TLBs, faster L1$, faster L2$
  - Nearly double the core count, and more complex ring topology
- New “MVEX” encoding to be friendlier to Big Core

Larrabee 2 – Knights Corner

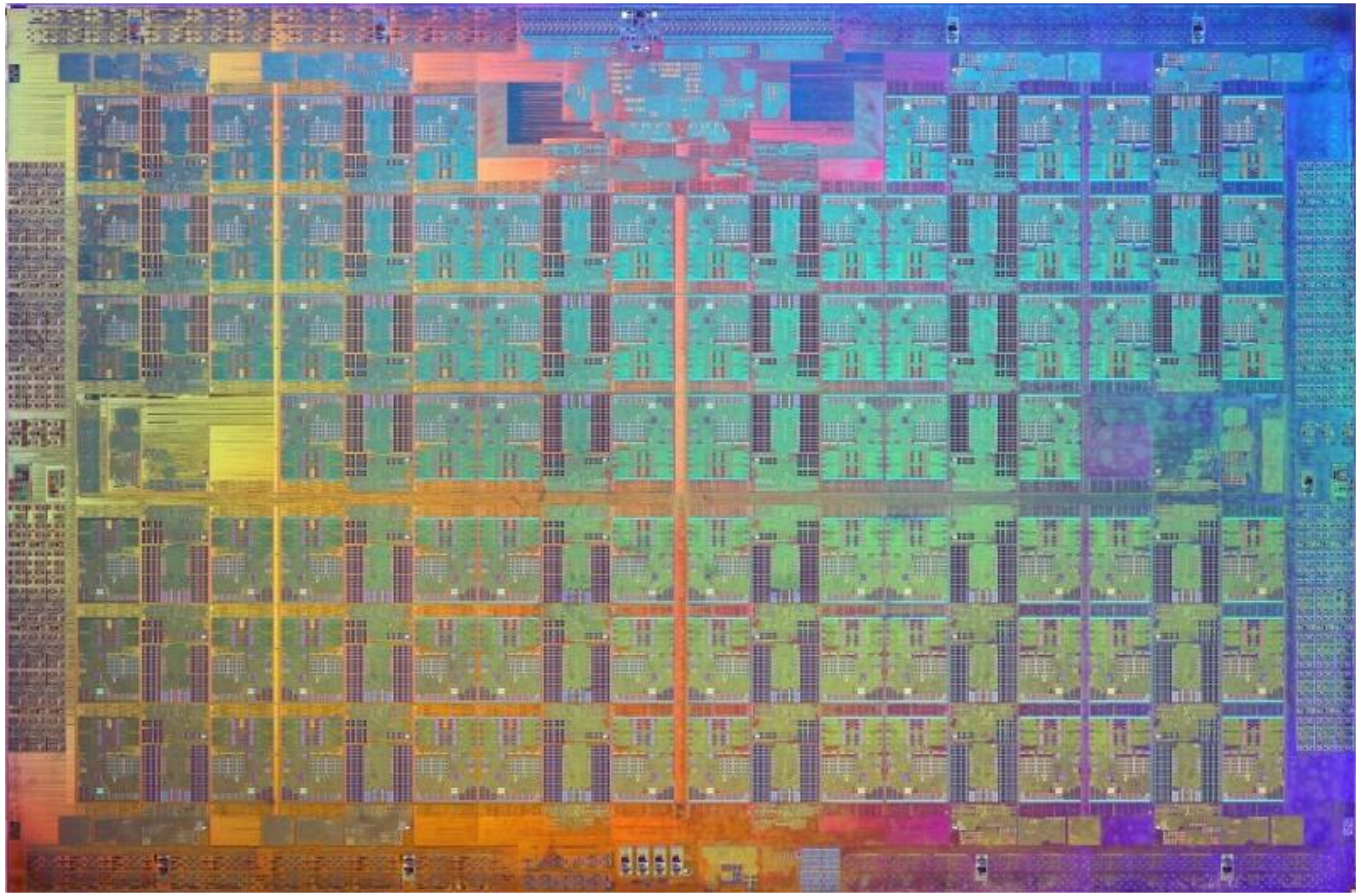
- Knights Corner
  - Shipped 2012, 22nm
  - 62 cores
    - Some disabled for yield
  - Only 8 texture units
  - No scanout
  - 1.2GHz, 2.4TFLOPS
- “Xeon Phi” PCIe card

Knights Landing

- Knights Landing goals:
  - 100% x86 compatibility and “bootable”
  - Out-of-order processing
  - 2x peak flops per core
- Replace P54C core with Silvermont “Atom” core
  - MMX and SSE already present
  - Standard modern debugging, profiling, etc
  - Motherboard chipset support
  - Higher frequency
  - “Bootable” – can be only CPU in the system, boots Windows etc
- New “EVEX” encoding – true AVX512 convergence with Big Core

Knights Landing

- Knights Landing
  - Shipped 2016, 14nm
  - 76 cores
    - Some disabled for yield
  - No texture units
  - No scanout
  - 1.5GHz, 7.3TFLOPS
- Motherboard!



Knights Mill

- Knights Mill
    - Shipped 2017, 14nm
    - Closely based on Knights Landing
- Reduced double-precision support, to make room for…
    - Doubled single-precision performance
    - Even higher performance for 16-bit formats
- Aimed at machine learning applications rather than HPC

Conflict

- Parallelize a loop with possible dependencies, e.g. histogram:

```
for ( int i = 0; i < 1024*1024*16; i++ )
{
    int index = data[i];
    int x = histo[index];
    x++;
    histo[index] = x;
}
```

Conflict

- Parallelize a loop with possible dependencies, e.g. histogram:

```
for ( int i = 0; i < 1024*1024*16; i++ )
{
    int index = data[i];        ← What if data[] is {1,5,6,7,1,8,9,5}?
    int x = histo[index];
    x++;
    histo[index] = x;
}
```

Conflict

- Parallelize a loop with possible dependencies, e.g. histogram:

```
for ( int i = 0; i < 1024*1024*16; i++ )
{
    int index = data[i];          ⟵  What if data[] is {1,5,6,7,1,8,9,5}?
    int x = histo[index];
    x++;
    histo[index] = x;             ⟵  histo[1] and histo[5] only +1, not +2!
}
```

Conflict

- The problem is doing the +1 multiple times in parallel
  - This is a "hidden" dependency that cannot be parallelized
- We need to spot when we have conflicting indices
  - Mask out all but the first conflicting lane
  - Keep doing the operation until all conflicts resolved
- Simple version:

```
vconflict k1, v0      v0={1,5,6,7,1,8,9,5}
                      k1={1,1,1,1,0,1,1,0}      Disable conflicting lanes
```

- Actual AVX512 implementation is slightly more complex
- This allows vectorization of a lot more types of loops

Conflict

- All-to-all comparison
  - Architecturally, this looks really complex
  - Dismissed for KNC – too expensive – emulated slowly with scatter/gather
- But then a smart designer called Dennis Bradford looked at it
  - Huh, this looks somewhat familiar…

Conflict

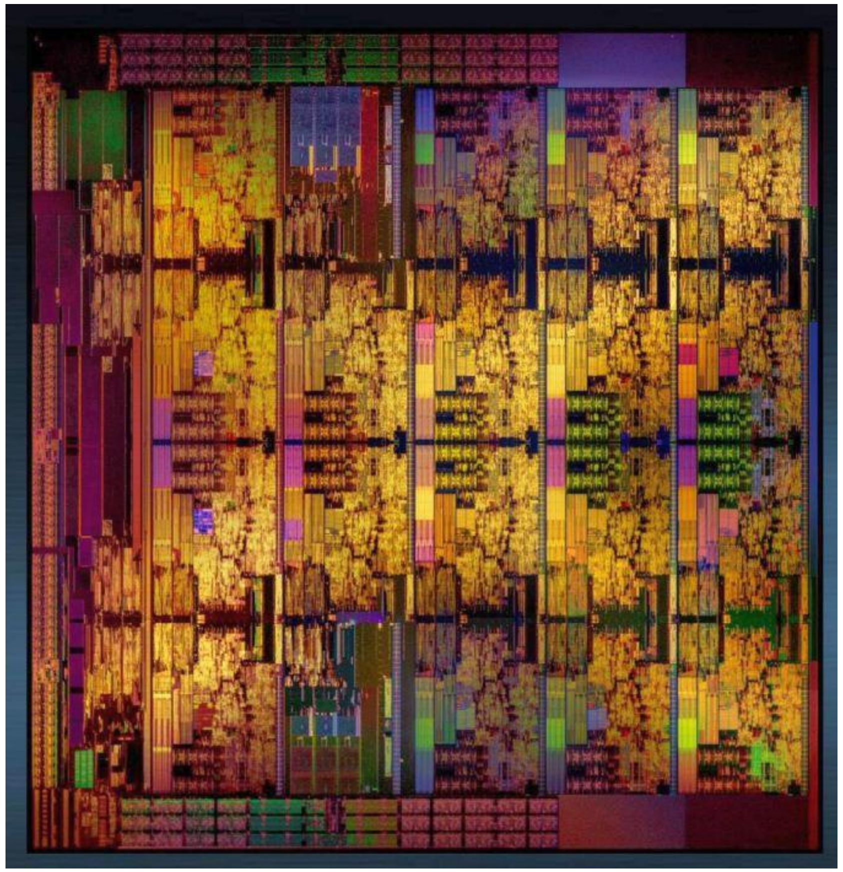
- All-to-all comparison
  - Architecturally, this looks really complex
  - Dismissed for KNC – too expensive – emulated slowly with scatter/gather
- But then a smart designer called Dennis Bradford looked at it
  - Huh, this looks somewhat familiar…
- In fact, we can re-use an existing structure with some minor changes
  - The two look totally different architecturally (i.e. in terms of math)
  - But at the design level (gates and wires) they look very similar
  - Both are “hard” but a very similar sort of “hard”
  - Dennis had already solved the problem once, which is why he recognized it
- vconflictd turned out to be very cheap to add to KNL

AVX512

- Finally achieved consensus on shared ISA!
    - So… many… meetings…
- Big cores struggled mightily with predication
    - Merging is more expensive when doing OOO register renaming
    - Much discussion over whether to only support zero-predication
    - Feedback from SW convinced them the effort was justified
- Still some areas of AVX512 that are optional
    - KNL/KNM have higher-precision recip/sqrt, more prefetch options
    - Big cores have more complete byte, word, DQ support
- And of course there will be new additions in the future

“Big Cores”

- Skylake Xeon (SKX)
  - Version of Skylake for servers
  - Shipped 2017
  - Variety of core counts and clock speeds
  - First big core implementation of AVX512
- Canon Lake
  - Shipped 2018
  - Variety of core counts and clock speeds
  - First laptop/desktop big core to support AVX512
- …and nearly everything since



Lessons – lean on the compiler

- Write the compiler first!
    - There’s a lot of smart ideas that a compiler cannot deal with
    - There’s some ugly ideas that it can use well, e.g. madd233
- Many tradeoffs are very marginal
    - Especially in “embarrassingly parallel” workloads
    - If a single instruction would cost 1% area, it needs to produce 1% speedup
    - Because the alternative is you add 1% more cores instead!
- Having a compiler gives you a big dataset
    - Intuition about instruction frequency can be misleading
- Iteration speed is vital
    - You have to wade through a lot of bad ideas to find the good ones
    - May have to try a lot of combos to discover instruction synergy

Lessons – every level of HW shapes an ISA

- Design
    - In-line swizzles (don’t!)
    - Conflict – cheaper than thought
- Micro-architecture (“uarch”)
    - Load-op and fat/thin pipe inherited from P54C
    - Slow microcode -> multi-instruction recip/sqrt/etc -> fixup instruction
- User-level architecture
    - FMA and ternary efficiency
    - Predication

- Questions?

- Backup slides
  - Interesting, but moved out of the main presentation for time reasons

Historical context

Because architecture is always created in a context, this talk is a historical journey, as well as a technical one.

Timeline of roughly when the ISA work was done (not when it shipped!):

2004: Pixomatic

2005: SMCANI – SMCA New Instructions

2008: LRBNI – Knight’s Ferry (KNF, MIC)

2010: LRBNI2 – Knight’s Corner (KNC, XeonPhi)

2013: AVX512 – Knight’s Landing (KNL, KNM, XeonPhi 2)

2015: AVX512 – Big Cores (SKX, CNL)

Predication masks

- Predication is crucial for efficient wide SIMD, used by all GPUs
  - Deactivate unused ALU lanes to save power
  - Do not signal exceptions/flags for disabled lanes
  - Avoids extra instructions to merge results

vaddps v0, v1, v2	All 16 lanes do operation & store to v0
vaddps v0{k1}, v1, v2	Some v0 lanes preserved, ALU lanes inactive

- Most GPUs have an implicit “current” write mask
- But implicit registers have caused enough trouble in x86!
  - Direction flag, rounding mode settings, etc
  - Decided to use explicit registers instead

Predication masks

vaddps v0{rax}, v1, v2 Use existing registers for predication?

vaddps v0{k1}, v1, v2 Invent new registers?

- Really wanted to use rax/rbx/rcx/rdx, but worries about timing
- New registers k0-k7 and new mini-ISA to manipulate them
  - In retrospect, we could have made rax-rdx work on P54C
  - But would have bitten us when making AVX512 on the Big Cores
  - So a bullet dodged by luck, even though it felt ugly at the time
- Only 3 bits of encoding space
  - Thus 8 registers
  - 000 = “no predication”, so k0 could not be used for predication

Conflict

```
for ( int i = 0; i < 1024*1024*16; i++ )      mov rcx, #1024*1024
{                                            loop_label:
   int index = data[i];                       vloadpi v0, [rax]
                                              add rax, 64
   int x = histo[index];                      vkmov k1, #FF
                                              vgather v1{k1}, [rbx+v0]
   x++;                                       vaddpi v1, #1
   histo[index] = x;                          vkmov k2, #FF
                                              vscatter [rbx+v0]{k2}, v1
}                                             dec rcx
                                              jnz loop_label
```

Conflict

```
  mov rcx, #1024*1024
loop_label:
  vloadpi v0, [rax]
  add rax, 64
  vkmov k1, #FF
  vgather v1{k1}, [rbx+v0]
  vaddpi v1, #1
  vkmov k2, #FF
  vscatter [rbx+v0]{k2}, v1
  dec rcx
  jnz loop_label
```

Problem:

what if histo[] = {10, 11, 12, 13} and v0 = {1, 2, 3, 1}?

v1 = {11, 12, 13, 11}

v1 = {12, 13, 14, 12}

histo[] = {10, 12, 13, 14}

Conflict

```
  mov rcx, #1024*1024
loop_label:
  vloadpi v0, [rax]
  add rax, 64
  vkmov k1, #FF
  vgather v1{k1}, [rbx+v0]
  vaddpi v1, #1
  vkmov k2, #FF
  vscatter [rbx+v0]{k2}, v1
  dec rcx
  jnz loop_label
```

Problem:

what if histo[] = {10, 11, 12, 13} and v0 = {**1**, 2, 3, **1**}?

v1 = {11, 12, 13, 11}
v1 = {12, 13, 14, 12}

histo[] = {10, 12, 13, 14}
…but the correct answer should be:
histo[] = {10, **13**, 13, 14}
…and the problem is the duplicated index “1” in v0